Precise Event Sampling: In-depth Analysis and Sampling-based Profiling Tools for Data Locality

by

Muhammad Aditya Sasongko

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

in

Computer Science and Engineering

KOÇ ÜNİVERSİTESİ

February 2, 2022

Precise Event Sampling: In-depth Analysis and Sampling-based Profiling Tools for Data Locality

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a doctoral dissertation by

Muhammad Aditya Sasongko

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Assist. Prof. Didem Unat (Advisor)

_____

Prof. Öznur Özkasap

_____

Assoc. Prof. Aykut Erdem

_____

Assist. Prof. Ayşe Yılmazer

_____

Prof. Özcan Öztürk

Date: _____

To my mother, wife Riana, and daughter Sofia

ABSTRACT

Precise Event Sampling: In-depth Analysis and Sampling-based Profiling Tools for Data Locality
Muhammad Aditya Sasongko
Doctor of Philosophy in Computer Science and Engineering
February 2, 2022

Precise event sampling is a profiling feature in current commodity CPUs that allows sampling of hardware events and identifies the instructions that trigger the sampled events. It offers the ability to detect performance bottlenecks with low overhead as well as the locations of the bottlenecks in source code. There have been a number of profiling tools developed using this feature that detect various sources of performance bottlenecks. However, none of these tools detects inter-thread data movement nor measures data locality in multithreaded applications, which have become widely used due to the ubiquity of multicore architectures. Furthermore, though this hardware facility has been used in multiple profiling tools, there have been only few works that analyze it in terms of accuracy and time overhead. All of these works target only the facility in Intel architecture, and none of these works evaluates other aspects of precise event sampling such as memory overhead, stability, and functionality of the facility.

In this dissertation, we present threefold major contributions. First, we perform the most comprehensive and in-depth qualitative and quantitative analyses to date on PEBS and IBS, which are the precise event sampling facilities of two major vendors, Intel and AMD, respectively. Next, we show the potential for imaginative use of precise event sampling in developing low overhead yet accurate profiling tools for multicore and design two diagnostic tools with a particular focus on data movement as it constitutes the main source of inefficiencies. First of such tools is ComDetective that detects inter-thread communications, classifies them into true sharing or false sharing, and records them in the form of communication matrices. Second is ReuseTracker that measures data locality in private and shared caches of multithreaded applications. ComDetective and ReuseTracker leverage pre-

cise event sampling to profile multithreaded applications accurately and with low overheads compared to their state-of-the-art alternatives.

To analyze key differences between Intel PEBS and AMD IBS, we firstly developed a series of carefully designed microbenchmarks. Through our qualitative analysis and quantitative study using the microbenchmarks, we found that Intel PEBS samples hardware events more accurately and with higher stability in terms of the number of samples that it captures, while AMD IBS records richer set of information at each sample and incurs lower time overhead per sample. We also discovered that both PEBS and IBS are afflicted with bias when sampling the same event across multiple different instructions in a code. Moreover, we also show how our findings from the quantitative experiments using the microbenchmarks are relevant for a full-fledged profiling tool that runs on Intel and AMD machines.

We develop ComDetective, a profiling tool that captures inter-thread communications accurately and with low runtime and memory overheads. ComDetective employs precise event sampling to sample memory accesses and utilizes hardware debug registers to detect inter-thread communications. In addition to detecting communications, ComDetective can also classify them into true or false sharing. Its time and memory overheads are only 1.30× and 1.27×, respectively, for the 18 applications studied under 500K sampling interval. Using ComDetective, we generate insightful communication matrices from several microbenchmarks, PARSEC benchmark suite, and some CORAL applications and compare the produced matrices against the matrices of their MPI counterparts. Using ComDetective, we identify communication bottlenecks in a few codes and achieve up to 13% speedup from code refactoring those codes.

We also design ReuseTracker, which is a profiling technique that measures *reuse distance* – a widely used metric that measures data locality. Reuse distance is a measurement of data locality as it is the number of unique memory locations that are accessed between two consecutive accesses to a particular memory location (*use* and *reuse*). ReuseTracker leverages precise event sampling to capture *uses* and debug registers to detect *reuse* in measuring reuse distance. ReuseTracker can measure reuse distance in multithreaded applications by also considering cache-coherence effects with much lower overheads than existing tools. It introduces only 2.9× time and 2.8× memory overheads. It achieves 92% accuracy when verified against a carefully crafted configurable microbenchmark that can generate user-specified reuse

distance patterns. We demonstrate in two use cases how ReuseTracker can be used to guide code refactoring by detecting spatial reuses in shared caches that are also false sharing and how it can also be used to predict whether certain applications can benefit from adjacent cache line prefetch optimization.

We expect that the analysis, algorithms, and the tools presented in this dissertation will benefit hardware architects in designing new precise event sampling features and performance engineers in performance tuning of their software while also paving the way for a new generation of low-overhead profiling tools. Moreover, the outcomes of the dissertation can be used by the end-users (e.g., data analysts, engineers, compiler developer) to identify the performance issues and increase the data locality aspects of their software.

ÖZETÇE

Kesin Olay Örnekleme: Derinlemesine Analiz ve Örneklemeye Dayalı Veri Konumu için Profil Oluşturma Araçları
Muhammad Aditya Sasongko
Bilgisayar Bilimleri ve Mühendisliği, Doktora
13 Ocak 2022

Kesin olay örneklemesi, mevcut emtia işlemcilerde bulunan, donanımsal olayların örneklenmesinde ve bu olayların tetiklenmesine sebep olan komutların tanımlanmasını sağlayan bir profilleme özelliğidir. Bu özellik sayesinde, performans darboğazları düşük ek masraf ile saptanabilir ve bu darboğazların kaynak koddaki yerleri belirlenebilir. Çeşitli performans darboğazlarını belirlemek üzere birkaç profilleme aracı geliştirilmiştir. Ancak bu araçlardan hiçbiri, iş parçacıkları arasındaki veri hareketini belirleyemez, veya çok iş parçacıklı uygulamalarda veri yerelliğini ölçemez. Ek olarak, bu donanım özelliği birçok profilleme aracında kullanılmış olsa da, bu özelliğin doğruluğu ve zaman masrafını analiz eden çok az çalışma vardır. Tüm bu çalışmalar sadece Intel mimarisine yöneliktir; ve bunlardan hiçbiri bu donanım özelliğinin hafıza masrafı, stabilite, ve fonksiyonellik taraflarını değerlendirmemiştir.

Bu tezde, üç-yönlü büyük katkı öne sürülmüştür. İlk olarak, sırasıyla Intel ve AMD'nin kesin olay örneklemesi araçları olan PEBS ve IBS üzerinde derinlemesine nitel ve nicel analiz yapılmıştır. İkinci olarak, iş parçacıkları arasında haberleşmeyi saptayabilen ve bunları true-sharing ve false-sharing olarak sınıflandırıp haberleşme matrislerinde kaydedebilen bir profilleme aracı olan ComDetective öne sürülmüştür. Üçüncü olarak, çok iş parçacıklı uygulamalarda özel önbellek ve paylaşılan önbellek içerisinde veri yerelliğini ölçebilen bir profilleme aracı olan ReuseTracker öne sürülmüştür. ComDetective ve ReuseTracker, kesin olay örneklemesi kullanarak yüksek doğruluk oranı ve düşük ek masraf ile çok iş parçacıklı uygulamaları profilleyebilmektedir.

Intel PEBS ve AMD IBS arasındaki kilit farkları analiz edebilmek adına, ilk olarak bir dizi dikkatle tasarlanmış microbenchmark geliştirilmiştir. Bu microbenchmarklar ile yapılan nicel analiz ve nitel çalışmalar sonucunda, Intel PEBS'in do-

nanımsal olayları örneklem sayısı bakımından daha yüksek doğruluk ve stabilite ile örnekleyebildiği gözlemlenirken; AMD IBS'in ise örneklem başına daha düşük ek masraf ile bilgi bakımından daha kapsamlı örnekleme yaptığı görülmüştür. Ek olarak, PEBS ve IBS'in farklı komutlar üzerinde aynı donanım olayını örneklerken kötü yönde etkilendiği gözlemlenmiştir. Dahası, elde edilen deney sonuçlarımızın Intel ve AMD makinelerinde çalışabilecek tam teşekküllü bir profilleme aracı bağlamında ilişkilendirilmesi gösterilmiştir.

Intel PEBS ve AMD IBS'in incelenmesinden sonra, iş parçacıkları arasındaki haberleşmeyi yüksek doğruluk, düşük ek masraf düşük çalışma zamanı ile saptayabilen bir profilleme aracı olan ComDetective öne sürülmüştür. ComDetective kesin olay örneklemesi ile hafıza erişimlerine örnekleyerek ve donanımsal debug yazmaçlarını kullanarak iş parçacıkları arasındaki haberleşmeleri saptayabilmektedir. Haberleşmeyi saptamaya ek olarak, ComDetective bu haberleşmeleri true-sharing ve false-sharing olarak sınıflandırabilmektedir. 18 farklı uygulamada 500K örnekleme aralığı ile çalıştırıldığında, ComDetective'nin sırasıyla zaman ve hafıza ek masrafları sadece 1.30*times* ve 1.27*times* olarak ölçülmüştür. ComDetective kullanarak, birkaç microbenchmark, PARSEC benchmark koleksiyonu ve bazı CORAL uygulamaları için haberleşme matrisleri oluşturulmuş, ve bu matrisler MPI karşıtları ile karşılaştırılmıştır. Bu sayede bazı uygulamalarda haberleşme darboğazları keşfedilmiş olup, düzeltilmeleriyle beraber 13%'e kadar hızlanma başarılmıştır.

Ek olarak, bir veri yerelliği ölçütü olarak sıkça kullanılan yeniden-kullanım mesafesi'ni ölçebilen ReuseTracker öne sürülmüştür. Yeniden-kullanım mesafesi, herhangi bir hafıza adresine ard-arda yapılan iki erişim (kullanım ve yeniden-kullanım) arasında erişilen farklı adreslerin sayısıdır, ve dolayısıyla bir veri yerelliği ölçütüdür. ReuseTracker kesin olay örneklemesi ve de donanımsal debug yazmaçlarından faydalanarak yeniden-kullanım mesafesini ölçmektedir. Ek olarak, ReuseTracker önbellek-tutunum etkilerini göz önünde bulundurarak çok iş parçacıklı uygulamalarda, var olan diğer araçlara göre daha az ek masraf ile yeniden-kullanım mesafesini ölçebilmektedir. Öne sürülen bu araç, sadece 2.9× zaman ve 2.8× hafıza ek masrafına sebep olmaktadır. Kullanıcı tarafından belirlenebilen yeniden-kullanım mesafesine sebep olacak şekilde özel olarak yazılmış bir microbenchmark ile ölçlüdüğü üzere, ReuseTracker ortalama 92% doğruluk oranına sahiptir. Paylaşılmış önbelleklerde false-sharing olan mekansal yeniden-kullanım'ların saptanması, ve bazı uygulamaların komşu önbellek-satırı prefetch optimizasyonundan fayda sağlayabileceğine dair tah-

min yapılması olarak iki farklı senaryoda ReuseTracker'nin, kod düzenlemesinde nasıl rehber alınabileceği gösterilmiştir.

Bu tez içerisinde öne sürülen araçların ve analizlerin, donanım mimarlarının yeni kesin olay örnekleme özellikleri geliştirirken ve de performans mühendislerinin yazılım performansını ayarlarken faydalı olabileceği gibi; performans analiz ve donanım içerisindeki profilleme araçları alanında ileride olabilecek araştırmalar için yeni yollar açabileceği beklentimizdir.

ACKNOWLEDGMENTS

First, I would like to express my gratitude to my advisor Assist. Prof. Didem Unat. Without her advice, support and guidance, this work could not have been completed successfully.

I am also thankful to my thesis progress committee members, Prof. Öznur Özkasap from Koç University and Assist. Prof. Ayşe Yılmazer from Istanbul Technical University for their critical feedbacks that have helped me improve this work. I would also like to thank Assoc. Prof. Aykut Erdem from Koç University and Prof. Özcan Öztürk from Bilkent University for their willingness to be parts of my thesis jury committee.

I would like also to thank Milind Chabbi from Scalable Machine and Paul H. J. Kelly from Imperial College London for the technical discussions that we have had and their advices for the improvement of this work.

I am also grateful to have some of the sharpest and friendliest research colleagues at ParCoreLab that have inspired me with a lot of insightful ideas through various discussions that I have had with them. I am also thankful to them for the fun moments we have shared together.

Special thanks to Mr. Ufuk Yılmaz from Koç University Advanced Computing Center for helping me with running the HPC resources. And finally, I would like to extend my deepest gratitude to my family, especially my mother and wife for their emotional support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

CPU	Central Processing Unit
PMU	Performance Monitoring Unit
PEBS	Processor Event Based Sampling
IBS	Instruction Based Sampling
MRK	Marked Event Sampling
SPE	Statistical Profiling Extension
OS	Operating System
PMC	Performance monitoring counter
MSR	Model Specific Register
TLB	Translation Look-aside Buffer
PEBS	Processor Event Based Sampling
RAW	Read after Write
RAR	Read after Read
MRC	Miss Ratio Curve
AET	Average Eviction Time
RISC	Reduced Instruction Set Computer
ITLB	Instruction Translation Look-aside Buffer
MPI	Message Passing Interface
WP	Watchpoint
RFO	Read for Ownership
LRU	Least Recently Used
DRAM	Dynamic Random-Access Memory
ACP	Adjacent Cache Line Prefetch

Chapter 1

INTRODUCTION

Profiling tools are essential resources for performance tuning of any computer applications. Using profiling tools, programmers are able to identify bottlenecks, such as cache misses, long-latency memory accesses, or branch mispredictions, that cause performance slowdowns in the profiled programs. In the post-Moore era where multithreaded applications become ubiquitous, there is a higher variety of bottlenecks that prevent these applications from reaching their theoretical peak performance. Some examples of bottlenecks that may afflict multithreaded code are false sharing, inter-thread cache line transfers, and coherence misses.

A number of profiling tools have been developed to detect performance bottlenecks in multithreaded applications and associate these bottlenecks with the threads or locations in source code that cause them. Among these tools, there are techniques that leverage hardware simulators [14, 29, 104]. There are also techniques that insert profiling code to original program code, i.e. code instrumentation, using compiler techniques [82, 83] or binary instrumentation [34, 33, 103, 125]. These existing tools, though have been demonstrated to be able to locate some bottlenecks in the code that they profile, still suffer from several drawbacks. One drawback is that the tools that rely on cycle-accurate simulators and most tools that leverage code instrumentation suffer from huge time and memory overheads. Another drawback is that the simulator and code instrumentation-based tools might suffer from inaccuracy. Hardware simulators might not accurately capture the behavior of multithreaded programs when running on actual hardware due to the simplistic or idealized nature of the simulated machines. Some examples of how simulators might alter program behavior are when they simulate CPU cores with in-order ex-

ecution instead of out-of-order execution, or when the simulated CPU cores access shared data from memory in different order compared to CPU cores in real machines, which will result in different amount of inter-thread communication or false sharing detected by the simulators. Not only simulators, binary instrumentation-based techniques might also fail to capture the behavior of multithreaded code on actual hardware as these techniques can distort the parallel schedule of concurrent threads [92] unless a supplementary tool that can record and replay original program execution deterministically such as Intel's PinPlay is also utilized [90].

To avoid the overhead and inaccuracy problems incurred by simulator and code instrumentation-based techniques, one solution is to leverage performance monitoring units (PMUs), which are hardware features that exist in current commodity CPUs. PMUs can be programmed to count and sample hardware events, such as retired instructions, branch instructions, and memory accesses, and software events, such as page faults and context switches. As PMUs are hardware features, they incur much lower overheads than simulators or code instrumentation-based tools, and cause minimal perturbations to program execution when used for profiling. However, ordinary PMUs still have a flaw in the way that they cannot accurately attribute sampled events to the actual instructions that trigger those events. When one such PMU samples an event, the instruction address recorded is the value of instruction pointer at the time the sampling interrupt is handled, and not the address of the actual instruction that causes the event. To fix this flaw, PMUs are enhanced further with precise event sampling technology [53, 38, 108, 124], which allows sampled events to be accurately associated with the instructions that trigger them and the effective addresses that they access, in case the sampled events are memory accesses.

Precise event sampling is supported in a number of architectures. Intel provides this feature through Processor Event Based Sampling (PEBS) that is supported in Nehalem microarchitecture and its successors [53]. AMD supports this feature through Instruction Based Sampling (IBS) that is available in AMD Opteron (microarchitecture family 10h) and its successors [38]. IBM PowerPC architecture also provides this capability through Marked Event Sampling (MRK) [108] feature that

is available in IBM POWER5 and its successors. The most recent example of precise event sampling technology is ARM's Statistical Profiling Extension (SPE) which is available in Armv8.2 [124] and its successors.

A number of profiling techniques have leveraged precise event sampling for various purposes. Some of these techniques detect long-latency memory accesses and identify their locations in source code [71, 70]. There are also other techniques that capture remote memory accesses [72, 74], cache misses [93], false sharing [74, 23, 50], and profile data locality in single-threaded applications [119]. However, to the best of our knowledge, none of these hardware-based techniques is able to capture data movement and measure data locality in the context of multithreaded execution, which are common sources of bottlenecks in shared memory parallel programs that run on multicore machines. To address this need, we design a profiling algorithm that captures inter-thread communications in the form of communication matrices by also differentiating them into true and false sharing communications, and we also devise two other profiling algorithms that measure reuse distance, which is a widely used metric for data locality, in private and shared caches, respectively, in the context of multithreaded execution.

Though precise event sampling features have been used in a number of profiling tools, there have been very few works that perform in-depth analysis on these features. These works only focus on Intel PEBS, and only analyze its accuracy and time overhead without addressing other aspects such as memory overhead, stability of sample counts, and differences in functionality among precise event sampling features across different architectures. To address this research gap, in addition to proposing profiling techniques, we also perform in-depth qualitative and quantitative analyses of two most widely used precise event sampling facilities, i.e. Intel PEBS and AMD IBS, by analyzing their accuracy, time and memory overheads, stability, and functionality.

In this dissertation, we perform qualitative and quantitative analyses on Intel PEBS and AMD IBS, and present profiling tools that leverage Intel PEBS to capture inter-thread communications and measure reuse distance. To our knowledge, the

analysis on PEBS and IBS that we perform in this work is the most comprehensive study on precise event sampling to date. Furthermore, the algorithms utilized by our profiling tools are microarchitecture agnostic as they will work on any multicore with precise event sampling support and debug registers. To summarize, the major contributions of this dissertation are as follows.

- The most comprehensive qualitative and quantitative study to date on the precise event sampling facilities of two major vendors, i.e. Intel and AMD, using the microbenchmarks

- Imaginative uses of precise event sampling in the forms of two low-overhead, open-source tools and their underlying algorithms that profile data locality and communication using precise event sampling and debug registers

  - ComDetective, a communication detection tool and its algorithm that captures inter-thread communications and distinguishes them into false and true sharing communications

  - ReuseTracker, a reuse distance analysis tool and its two algorithms that measure reuse distance of multithreaded applications in private and shared caches, respectively

- An extensive set of microbenchmarks that evaluates accuracy, overheads, sampling biases, stability, and functionality of precise event sampling facilities, and another set of microbenchmarks that evaluates the accuracy of the developed precise event sampling-based tools, which can also be used to evaluate other tools that capture inter-thread communications and measure reuse distance

- Extensive experiments on an Intel Cascade Lake and an AMD Zen 2 machines using the microbenchmarks as well as large benchmarks from Parsec, Rodinia, CORAL, CORAL 2, and Synchrobench suites to measure the accuracy and overheads of the developed tools, and to gain insights into data locality and communication in the benchmarks

- Important use-case scenarios that demonstrate how the profiling tools can be used to guide optimizations on profiled applications

ComDetective and ReuseTracker have been published in [101] and [102] respectively. The repository of ComDetective is publicly available at `https://github.com/comdetective-tools`, and the repository of ReuseTracker is publicly available at `https://github.com/ParCoreLab/ReuseTracker`. The microbenchmarks developed and used in this work are listed in Appendix A.

Chapter 2

BACKGROUND

2.1 Hardware Performance Monitoring Unit (PMU)

CPU's PMU offers a programmable way to count hardware events such as loads, stores, CPU cycles, etc. PMUs can be configured to trigger an overflow interrupt once a threshold number of events elapse. A profiler, running in the address space of the monitored program, handles the interrupt and records and attributes the measurements to their corresponding communication types or objects. We refer to a PMU interrupt as a "sample." PMUs are per CPU core and virtualized by the operating system for each OS thread.

A subset of PMUs that is called *precise event sampling* can sample hardware events and accurately locate the instructions that trigger the events. Intel supports this capability through Processor Event Based Sampling (PEBS) [25] that is available in Intel Nehalem and its successors, while AMD processors allow precise event sampling through their Instruction-Based Sampling (IBS) [17] feature that is supported in AMD Opteron (microarchitecture family 10h) and its successors.

2.1.1 Intel PEBS

On Intel architecture, precise event sampling can utilize any programmable counters available in PMUs [55]. The PMUs in an Intel core consist of a number of components: global control registers, status registers, event select registers, and performance monitoring counters (PMCs). Global control registers are used to globally enable or disable event counters or precise event sampling ability of each counter. Status registers contain info about the capabilities supported by the PMUs or the overflow status of each event counter. Event select registers are used to choose the

hardware or software event to be monitored. A PMC counts the occurrences of a monitored event.

To enable precise event sampling, a programmable counter is enabled along with its PEBS capability by global control registers [55]. This counter is then configured to monitor a targeted hardware event by programming its event select register with the mask and number of the targeted event. The counter is also configured to have a counter overflow in every elapsing of a specified number of events, which is the *sampling interval*. When a counter overflows, PEBS hardware is armed to trap the next occurrence of the monitored event. When the next monitored event occurs, a mechanism called *PEBS assist* will copy the machine state to a memory buffer called *PEBS buffer*. The machine state and other data such as effective address and load latency collected by a PEBS assist are grouped into a data structure called a *PEBS record* in the PEBS buffer. When the number of PEBS records in the PEBS buffer has reached a predefined threshold, a hardware interrupt is triggered and handled by an interrupt handler that is a part of the OS. The interrupt handler reads all of the PEBS records in the PEBS buffer, clears the buffer, and sends an OS signal to a user process or thread that will collect the sampled data. On receiving the signal, the user process/thread retrieves the sampled data and uses it for profiling.

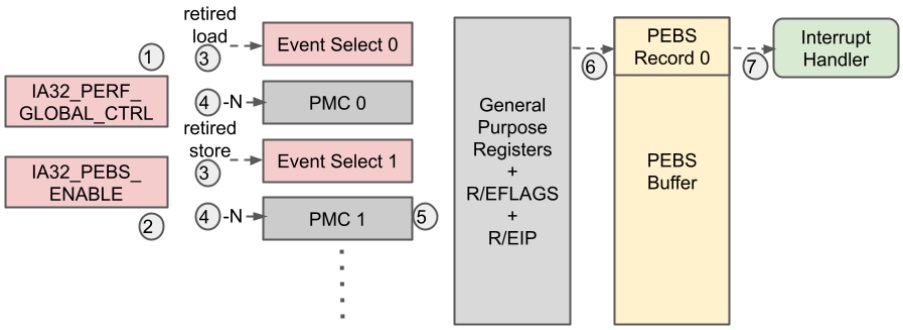


Figure 2.1: Execution scenario of Intel PEBS hardware when it monitors retired load and store micro-operations.

Figure 2.1 shows an example execution of PEBS when profiling retired load and store micro-operations. (1) The global control register `IA32_PERF_GLOBAL_CTRL`

enables PMC 0 and PMC 1 by setting its bits that correspond to the counters to 1. (2) Another global control register, i.e. `IA32_PEBS_ENABLE` MSR, enables PMC 0 and PMC 1 to capture architectural state using PEBS by setting the corresponding bits to 1. (3) The event select registers `IA32_PERFEVTSEL0` and `IA32_PERFEVTSEL1` are programmed to make PMC 0 and PMC 1 count retired load micro-operations and retired store micro-operations, respectively. (4) The configured PMCs are also preloaded with -N with N being the sampling interval, so that they overflow on elapsing N monitored events. (5) After the profiled program executes for a while, PMC 1's counter overflows after N stores occur. The PEBS hardware is *armed* to trap the next store access that happens in the same core. PMC 1 is preloaded with -N again. (6) Another store access occurs after the counter overflow. The armed PEBS hardware traps the access and a microcode records the architectural state in a PEBS record located in PEBS buffer in kernel space. (7) If the number of PEBS records has reached a specified threshold, which is 1 in this case, a hardware interrupt is triggered, and an interrupt handler is called to transfer the PEBS records to user space.

2.1.2 AMD IBS

Different from Intel, the precise event sampling in AMD, i.e. IBS, employs a hardware-based facility that is separate from the PMUs that are commonly used to count or imprecisely sample specific hardware or software events. This mechanism is based on the instruction sampling technique proposed in [31]. The IBS facility in each CPU core consists of a couple of components: two control registers, two internal counters, and a number of MSRs (Model Specific Registers) for sampled data [8].

This facility allows only two flavors of sampling: instruction fetch sampling and micro-operation sampling [38, 8]. Either one of the two control registers is programmed to control the IBS hardware depending on the sampling flavor that is chosen. If instruction fetch sampling is selected, the fetch control register is programmed. Otherwise, the execution control register is programmed. After the con-

trol register is programmed, the internal counter that belongs to the selected sampling flavor, i.e. either the fetch counter or the op counter, will count the monitored event in the CPU core. When an event sampling occurs, information related to the event is recorded in the MSRs of sampled data that belong to the chosen sampling flavor.

The counter for micro-operation sampling, i.e. op counter, can be programmed to count either clock cycles or dispatched micro-operations. If it is programmed to count clock cycles, it increments for each clock cycle, and when the counter reaches the specified sampling interval, a micro-operation is selected for sampling from the next dispatch line. On the other hand, if the counter is programmed to count dispatched micro-operations, it increments for every dispatched micro-operation, and when the counter reaches the sampling interval, a micro-operation is selected to be sampled in the next cycle. In this work, we only consider the op counter to be programmed for counting dispatched micro-operations, which is the default configuration of the IBS driver in [47]. We also choose this configuration as it generates predictable number of samples given a known number of micro-operations in a profiled application, which suits our accuracy verification method.

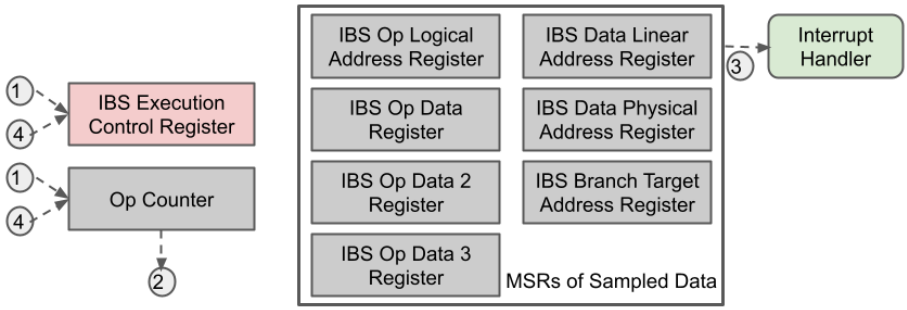


Figure 2.2: Execution scenario of AMD IBS hardware when sampling executed micro-operations.

Figure 2.2 shows an execution scenario of IBS when counting and sampling micro-operations. (1) IBS execution control register in a CPU core is programmed to make the IBS hardware count and sample executed micro-operations. The sampling interval is also written as a field in the control register. The op counter is set to a

pseudorandom 7-bit value in the range of 1 to 127. (2) After the profiled thread executes for a while, the value in the op counter equals the specified sampling interval. When the value of the counter equals the sampling interval, the next executed micro-operation will be tagged for sampling. (3) The tagged micro-operation retires. The execution info of the tagged micro-operation, which includes instruction pointer and effective address, is recorded in a number of MSRs. After the tagged operation retires, a hardware interrupt is triggered, and the interrupt handler copies the recorded data in the MSRs to a memory buffer in kernel space. (4) After copying sampled data, the interrupt handler configures the control register again to re-enable IBS, and the op counter is preloaded with another pseudorandom 7-bit value.

2.2 Hardware debug registers

Hardware debug registers [59, 85] trap CPU execution when a program counter reaches an instruction address (breakpoint) or when an instruction accesses a monitored address (watchpoint). When a thread sets up a debug register to trap a future memory access to certain address, we consider this thread to be *arming a watchpoint* on that address. One can configure debug registers with different addresses, widths, and types of memory accesses to be trapped (i.e. `W_TRAP` for stores and `RW_TRAP` for both loads and stores). Current processors from x86 architectures typically have four debug registers.

2.3 Linux perf_events

Linux provides an interface to configure PMUs and debug registers using the `perf_event_open` [67] system call and `ioctl` calls. The ability to configure debug registers has been available since Linux 2.6.33, and the ability to program multiple PMUs since Linux 2.6.39 [67]. The Linux kernel can deliver a signal to the specific thread that encounters a PMU interrupt or a debug register trap. The user code can (1) create a circular buffer into which the kernel appends the sampled data on each sample using `mmap` function and (2) collect the signal context on each watchpoint trap.

Chapter 3

RELATED WORK

3.1 Analysis on Precise Event Sampling Features

While there have been a few works that analyze the accuracy and overheads of PEBS, there is no in-depth study on the accuracy, overheads, stability, or functionality of IBS. To the best of our knowledge, none of the previous work on PEBS analyzes the stability and explore the functionality, such as execution mode detection, of PEBS either.

Larysch [64] evaluated the accuracy and overhead of PEBS in measuring memory bandwidth. The author performed their experiments using low sampling intervals, which are between 10 and 1000. Through the experiments, the author discovered that PEBS suffers from higher sample losses as sampling interval is decreased.

Nonell et al., [89] evaluated the accuracy and overhead of PEBS on applications running on a large numbers of CPU cores, which range from 2048 to 128k cores. By using the PEBS driver that they developed in a lightweight kernel, they could reach low overhead in capturing memory access patterns with high accuracy. Their driver could also maintain high accuracy of PEBS in capturing memory access patterns even under low sampling interval, which is up to 64.

Yi et al., [128] analyzed the accuracy of PEBS, and discovered that PEBS is prone to bias in event sampling due to shadowing. To eliminate the bias, they propose insertion of nop instructions after each monitored event. Gottschall et al., [43] proposed *Oracle profiler* as a golden reference for time-proportional attribution of event sampling. They found out that existing precise event sampling facilities such as Intel PEBS, AMD IBS, and ARM SPE are not time proportional in sampling instructions i.e. the number of samples taken from an instruction is not proportional

to the number of CPU cycles incurred by that instruction. They also discovered that PEBS suffers from instruction attribution problem when multiple instructions commit at the same cycle under ILP after a sample is taken.

Weaver and McKee [120] evaluated the variation or, as we refer to it, stability of event counting by PMUs in nine implementations of x86 architecture. They discovered that inter-machine variations could happen because certain instructions could be counted double in certain microarchitectures or a counter for a specific event in one CPU version could be more accurate than in older versions. They also found out that intra-machine variations of PMU-generated instruction count might also occur, which could be caused by virtual memory layout of profiled programs or OS activities such as page faults and timer interrupts. Weaver et al. [121] also evaluated PMUs in eleven different implementations of x86_64 architecture and discovered sources of variation in their counted events. They explored possible ways to work around these limitations in the machines to produce more deterministic counts.

Akiyama and Hirofuchi [6] quantitatively analyzed the overhead of PEBS, and demonstrated how the quantified overhead can be used to predict the actual overhead of complex applications. They also evaluated the effect of sampling rate and PEBS buffer size on cache pollution and the performance of profiled applications. Xu et al., [126] identify the inaccuracies of sampling in PEBS and develop a mathematical model in software to rectify inaccuracies.

This work differs from the previous work first by benchmarking several behavioral aspects of precise event sampling through some carefully crafted microbenchmarks. Second, we evaluate the accuracy and overheads under more complex situations such as while monitoring multiple events in PEBS and monitoring in different modes. Third, we quantify on the stability of sample counts generated by precise event sampling.

3.2 Inter-Thread Communication

3.2.1 Simulator-based Approaches

Barrow-Williams et al. [14] generate communication patterns for SPLASH-2 and PARSEC benchmarks by collecting memory access traces using Virtutech simics simulator [79]. Thread table of the kernel running on the simulator is also accessed to keep track of all running threads. Similar to [14], Henrique Molina da Cruz et al. [29] also employ a simulator to generate memory access traces. The resulting memory traces are used as the basis to create memory sharing matrix. By considering the memory sharing matrix, thread affinity is implemented by taking memory hierarchy into account. Application threads are mapped according to the generated thread affinity by using Minas framework [88]. ComDetective differs from these techniques in the way that they generate thread communication pattern with the help of a hardware simulator, while we generate communication matrix by PMUs. This makes ComDetective practical to use and runs faster than the simulator-based techniques, especially for full application execution.

3.2.2 OS-based Approaches

Tam et al. [109] and Azimi et al. [12] obtain communication patterns from running parallel applications through PMUs. Unlike ComDetective, their technique requires kernel support. PMUs are accessed by the kernel and the communication pattern of a running application can be generated by the kernel. The PMUs that are accessed are pipeline stall cycle breakdown, L2/L3 remote cache access counters, and L1 cache miss data address sampler.

Cruz et al. [28] use Translation Look-aside Buffers (TLBs) to generate of communication matrix that records page level memory sharing. Two approaches were introduced that use software-managed TLB and hardware-managed TLB. For the software-managed TLB, a trap is sent to OS when TLB miss occurs. Before the missing page table entry is loaded, TLB content of each core is checked for the matches of the missing entry. The information on the matches is used to update

the communication matrix. For the hardware-managed TLB, kernel will check the content of TLBs periodically. Both approaches require OS support. In contrast, ComDetective uses user-space PMU sampling. Moreover, TLB-granularity monitoring is too coarse-grained because inter-thread communications happen at cache-line granularity.

3.2.3 Code Instrumentation-based Approaches

Diener et al. [33, 34] develop Numalize, which uses binary instrumentation [75] to intercept memory accesses and identify potential communications among threads by comparing the intercepted memory accesses. Two or three threads that perform accesses to a memory block consecutively are considered to communicate by the tool. We have compared ComDetective with Numalize in our experimental study. Numalize introduces more than 16× runtime overhead and almost 2000× memory overhead, whereas ComDetective introduces only 1.30× runtime overhead and 1.27× space overhead. Moreover, ComDetective does not dilate execution and produces more accurate communication matrices.

A more recent work [83, 82] performs code instrumentation with the help of the LLVM compiler. This instrumentation allows detection of RAW and RAR dependencies in the original code and outputs this information as communication and reuse matrices. Through communication reuse distance and communication reuse ratio derived from these outputs, the tool facilitates analysis of communication bottlenecks that arise from thread interactions in different code regions. However, this tool still suffers from significant slowdown (140×), and is limited to detection of memory accesses to similar addresses. Hence, to our knowledge, it cannot detect cache line transfers that are triggered by false sharing.

3.2.4 Profiling Memory Accesses

Concerning the use of Performance Monitoring Units (PMUs) by library or standalone tool to profile memory accesses or data movement, our work is not the first one that implements this idea. Lachaize et al. [61] introduced MemProf, which utilizes

kernel function calls to sample data from memory access events. This data is used to identify objects that are accessed remotely by any thread. Like ComDetective, MemProf also intercepts functions for thread creation, thread destruction, object creation, and object destruction to differentiate memory accesses belonging to different objects and different threads. Unat et al. [113] introduce a tool, ExaSAT, to analyze the movement of data objects using compiler analysis. Even though it has no runtime overhead, it cannot capture all the program objects or their references as it relies on static analysis. Chabbi et al. [23] employ PMUs and debug registers to detect false sharing but do not generalize it for inter-thread communication matrices; furthermore, their technique does not quantify communication volume even for false sharing. Even though these tools can count memory access events, they do not associate these events to threads and are not used in generating communication pattern among threads.

3.3 PMU-based Multi-Core Reuse Distance Analysis

Existing works related to PMU-based multi-core reuse distance analysis can be organized into two groups. The first group consists of the papers that proposed multi-core reuse distance analysis techniques that profile individual threads and shared caches. The second group introduced reuse distance analysis techniques that leverage PMUs and debug registers.

3.3.1 Modeling Individual Threads and Shared Caches

Ding and Chilimbi proposed an analytical model that predicts shared cache behavior indirectly by combining per-thread reuse distance, interleaving, and data sharing information [35]. This technique processes execution trace of a multithreaded code to generate a set of per-thread metrics. This set of metrics covers locality (i.e. reuse distance profile), data sharing, and interleaving information of each thread. By accounting for multiple sets of metrics from a subset of threads using the model, miss ratio of this subset of threads while running in a shared cache can be predicted. Another work by Jiang et al. [58] proposed a probabilistic model that can approx-

imate concurrent reuse distance from reuse distances of individual threads. The multithreaded applications targeted by this model are those that perform similar computations in all threads and have uniform amount of shared data across their thread groups. These techniques in [35] and [58] require processing of execution traces that might incur large storage overhead in the host machines. Furthermore, the computation of reuse distance profiles from the traces can also require huge performance overhead depending on the size of the traces.

Schuff et al. introduced a different approach in [104]. Their technique performs on-the-fly reuse distance computations by using a modified hardware simulator. This technique computes reuse distances in both private and shared caches by also ensuring the coherence in the private caches. Furthermore, it also models the interleaving of memory accesses by all threads in the shared cache. This technique was improved in [103] by having all threads run in parallel while being profiled. Moreover, sampling technique was also introduced to reduce overhead by computing reuse distances only on randomly selected memory references. A shortcoming in these approaches is that they require simulator or binary instrumentation to intercept every single memory access. The use of simulator introduces huge performance and memory consumption overhead. Furthermore, the use of binary instrumentation might distort the parallel schedules of task-parallel and dataflow applications as reported in [92].

Chandra et al. [32] introduced one of the first models that predict L2 cache misses when multiple threads run on a shared L2 cache. Eklov et al. [42] also proposed another model that estimates miss ratio and CPI of co-scheduled applications that run on a shared cache. These models work by firstly approximating the reuse distance profile of the applications' interleaved accesses in the shared cache. After that, the models calculate the shared cache miss ratio based on the approximated profile. One key difference of these works from ours is that they target co-scheduled applications that do not share data among them.

Pericas et al. [92] introduced a low-overhead method to generate execution traces of multithreaded code and compute reuse distances of shared caches from these traces. In generating shared cache reuse distance histograms, it also captures cache

coherence-based invalidations across caches. To minimize overhead, this method reduces trace sizes by operating at the granularity of compute kernels. By operating at coarse-grained level, this method can accurately capture distant reuses while losing information on near reuses.

Maeda et al. [78] introduced a technique to profile reuse distances in every level of a multi-level cache hierarchy. Since this technique still requires memory address trace as an input, it still needs other tools to generate the trace. As a result, this technique is still exposed to the drawbacks of these tools, such as huge overhead in simulators or distortion of parallel schedule due to binary instrumentation.

Another work that leverages memory traces to profile reuse distances in private and shared caches was proposed by Barai et al. [13]. In this work, they utilized a compiler-assisted technique to generate a basic block-labeled memory trace from a single sequential execution of a profiled application. This trace is then used by a probabilistic analytical method to predict the reuse distance profiles of the application in private and shared caches when the application runs in parallel.

Hu et al. [51] proposed a model that reduces time and space costs in constructing cache miss ratio curves (MRCs) by analyzing only reuse-time distribution. They used average eviction time (AET) as a parameter to detect reuses that lead to cache misses. Their model could be extended to predict cache misses in a shared cache when multiple threads run on it. However, in estimating cache misses in private and shared caches, their model does not assume data sharing among threads that might lead to coherence misses in private caches and shared caches.

Ji et al. [56] developed a probability model that computes L2 reuse distance profile and predicts cache misses in L2 without requiring extra simulations or trace generations. Though able to give cache miss prediction, this model still needs inputs in the form of L1 reuse distance histograms. These inputs might have to be generated by simulator or binary instrumentation-based tools.

There are also recent works in [100], [118], [65], and [66] that proposed analytical models to profile shared cache behaviors by processing L1 cache reuse distance profiles. Similar to previously discussed works, these works also depend on other

tools to generate per-thread reuse distance histograms that they need.

3.3.2 Leveraging PMUs and Debug Registers

To the best of our knowledge, there have been only two techniques in literature that utilize hardware counters and watchpoint mechanism to compute reuse distances. The first technique was presented by Berg and Hagersten in [16]. In that paper, they proposed StatCache, a profiling tool that calculates time reuse distances by using hardware counters and watchpoints and deploys a statistical model to predict cache miss ratio based on the collected time reuse distance profile. This tool could predict the miss ratio of a fully associative cache with random replacement policy. The second technique that leverages hardware counters and watchpoints was introduced by Wang et al. [119]. This technique, which is named RDX, utilizes PMUs and debug registers in Intel machines to compute time reuse distances, and then, convert them to stack reuse distances. Both techniques in [16] and [119] differ from our work in the way that they do not account for inter-thread interactions in multithreaded code and they do not model shared caches.

Chapter 4

COMPARISONS OF PRECISE EVENT SAMPLING FEATURES IN AMD AND INTEL ARCHITECTURES

4.1 Introduction

Precise event sampling is a powerful profiling feature supported by Performance Monitoring Units (PMUs) in modern CPUs. This technology has been incorporated in a number of profiling tools that identify performance bottlenecks in parallel applications [52, 54, 80, 77, 101, 70, 101, 102, 73, 97, 74, 69, 122, 23, 98]. For instance, some of such tools are used to detect long latency memory accesses, false sharing and inter-core cache line transfers. In addition to identifying the bottlenecks, precise event sampling also offers the possibility to pinpoint the source code lines and data objects causing the bottlenecks through its ability to sample instruction pointers and effective addresses of the operations. Compared to alternative technologies such as cycle-accurate hardware simulators [79, 19] and binary instrumentation [21, 75], techniques that leverage precise event sampling incur much lower time and memory overheads as they employ existing hardware features to capture real hardware events without introducing additional software layer.

A number of architectures support hardware-based precise event sampling. Intel supports this capability through Processor Event Based Sampling (PEBS) [53] that is available in Intel Nehalem and its successors and many researchers develop several tools for PEBS [52, 54, 40, 69, 23, 101]. Similarly, AMD processors allow event sampling through their Instruction-Based Sampling (IBS) [38] feature that is supported in AMD Opteron (microarchitecture family 10h) and its successors. A number of tools have been developed using this sampling facility for AMD [10, 39, 7, 84, 62, 70, 74]. The support for event sampling in IBM PowerPC

architecture is provided through Marked Event Sampling (MRK) [108] feature that is available in IBM POWER5 and its successors. This capability is also recently supported in ARM architecture through Statistical Profiling Extension (SPE) feature that was introduced in Armv8.2 [124].

Despite the fact that event sampling feature is commonly used for developing profiling tools, there exists no rigorous study that benchmarks this capability in the microarchitecture. In this dissertation, we analyze and compare the precise event sampling facilities of two major vendors namely, Intel and AMD, in depth through extensive benchmarks. To support precise event sampling, Intel PEBS and AMD IBS adopt drastically different designs resulting in different characteristics at hardware level that affect the accuracy, stability, overhead and functionality of the sampling facility. While the outcomes of this work can be used by the profiling tool developers to better understand the behaviours of their tools, hardware designers can leverage the findings to design better PMUs not only for x86-based systems but also for ARM [124, 11] and emerging RISC-V processors [95, 37].

We firstly present qualitative differences between the two precise event sampling schemes in terms of usable counters, type of precise events, sampled data, and execution mode. Based on the observations on the qualitative characteristics, we developed a number of microbenchmarks that can assess the effects of the observed qualitative characteristics. Through these benchmarks, we then quantitatively compare the two schemes in terms of accuracy, time and memory overheads in sampling individual and multiple events. We also evaluate the stability and sampling bias of both schemes, and analyze the ability in attributing samples to the instructions that trigger the samples and the execution modes of those instructions, i.e. kernel mode or user mode. Lastly, to demonstrate how the microarchitectural characteristics that we identified in our qualitative and quantitative analyses impact profiling tools, we study a full-fledged open-source tool, namely ComDetective [101, 1], that employs precise event sampling to detect inter-thread communication.

Our findings based on the quantitative and qualitative study are as follows. (1) PEBS offers a large set of specific hardware events such as branches, memory loads,

etc to select from, while IBS has only two *flavors* of sampling: instruction fetch sampling and micro-operation execution sampling. One impact of this difference is that PEBS is more accurate than IBS in capturing proportional number of samples from specific events, such as loads or branches. (2) Though IBS supports fewer sampling choices than PEBS, it offers richer information in each sample, which shows, for example, the origin of accessed data in memory hierarchy and the status of TLB accesses. As a consequence, PEBS would have to monitor multiple events simultaneously in order to generate the similar level of information as in one IBS run. (3) PEBS shares the same counters with other non-PEBS PMU events, while IBS has its own internal counters. As a result, the number of different events that PEBS can monitor without multiplexing is limited to the number of available PMU counters per logical core. Multiplexing in PEBS suffers from sample loss, leading to reduced accuracy. (4) While PEBS is more stable than IBS in capturing sample counts, it suffers from larger time overhead in processing each individual sample. However, IBS' time overhead in profiling a benchmark can be higher than PEBS if PEBS monitors only specific events such as memory loads and stores, which are subsets of all retired instructions/micro-operations. Both PEBS and IBS exhibit similar memory overheads. (5) IBS is very sensitive to the sampling interval and its accuracy significantly drops at high sampling frequency. PEBS has high accuracy regardless of sampling rate. (6) Both PEBS and IBS are equally biased in sampling an event from multiple different instructions. (7) PEBS can be programmed to count events that execute only in user mode, only in kernel mode, or in any of the two modes, while an IBS counter always increments on any fetch or micro-operation without discriminating its execution mode.

In summary, the contributions in this dissertation are.

- Presenting the most comprehensive study to date on precise event sampling supported by two major vendors

- Detailed qualitative and quantitative comparisons of microarchitectural characteristics between Intel PEBS and AMD IBS and demonstrating their accu-

racy, stability, bias, functionality and overheads

- A suite of synthetic benchmarks that can be used for extending this study to other vendors and architectures

- Providing invaluable information both to the hardware designers and tool developers through our findings that would help understand and improve their designs

4.2 Qualitative Comparison

This section presents qualitative differences between the two precise event sampling schemes, Intel PEBS and AMD IBS, and Table 4.1 summarizes these differences.

Aspect	***Intel PEBS**	**AMD IBS**
Usable Counters per thread	4 general-purpose performance counters	2 internal counters (1 for each sampling flavor)
Event Type	62 subevents of 12 events	2 sampling flavors
Sampled Data	general purpose registers, RFLAGS register, RIP register, applicable counter, data linear address, data src encoding/store status, latency value, timestamp counter eventingIP, TX abort info	16 attributes in each sampled data for inst. fetch sampling 44 attributes in each sampled data for micro-operation sampling
Execution Mode	User or kernel or both modes	Both user and kernel modes

Table 4.1: Qualitative comparison of Intel PEBS and AMD IBS. *This information is valid for Cascade Lake microarchitecture [55].

4.2.1 Usable Counters

Observation 1 *PEBS can use up to 4 counters and monitor up to 4 events without multiplexing in microarchitectures. Starting Ice Lake, PEBS can monitor up to 7, while IBS has two dedicated counters, one for each sampling flavor.*

Observation 2 *If the op counter of IBS is programmed to count dispatched micro-operations, it is always preloaded with a pseudorandom 7-bit value after its sampling interrupt is handled.*

PEBS shares the PMCs that are also used by other non-precise PMU events. All microarchitectures before Ice Lake (launched in 2019) allow PEBS to utilize any of the four general-purpose performance counters in each logical core. In Ice Lake, PEBS can also use the three fixed-function performance counters in addition to the general-purpose ones. When the number of events that it monitors is higher than the number of counters, software overcomes this limitation by context switching more events on limited counters. When this oversubscription happens, the approximated counter values are inaccurate, and might cause the events to lose some counter overflows.

Different from PEBS that shares counters with other PMU events, IBS counts instruction fetches and executed micro-operations using its own internal counters that are separate from other PMCs in each AMD CPU core. It has two internal counters, one for each sampling flavor. Since these counters are not multiplexed, IBS does not miss an overflow.

Another difference between PEBS and IBS counters is that the IBS counters are randomized after each sampling interrupt is handled. For the fetch counter, randomization is optional, and it is enabled when certain bit in the control register is set to 1. On the other hand, the op counter is always randomized with a pseudorandom 7-bit value when it is programmed to count dispatched micro-operations. This randomization might cause the number of hardware events detected by IBS to vary across different runs.

4.2.2 Type of Precise Events

Observation 3 *PEBS has many choices of precise events to monitor, while IBS has only two sampling flavors to choose from. As a result, a PEBS counter can be programmed to count only specific hardware event and trigger sampling only of that event, while an IBS counter can only count instruction fetches or micro-operations indiscriminately and trigger sampling of any event that might or might not be of interest.*

PEBS has the ability to monitor hardware events at finer-grained than IBS. For example, in Cascade Lake, in total, there are 62 possible hardware sub-events, each of which is identified by a combination of an event number and a unit mask, that can be monitored using PEBS. When monitoring certain sub-event such as memory load, the used hardware counter will increment only for each occurrence of memory load, and it will trigger sampling of only memory load.

In contrast to PEBS, IBS has only two possible sampling flavors to choose from: instruction fetch sampling and micro-operation execution sampling. In instruction fetch sampling, IBS counts instruction fetches and samples from them, while in micro-operation execution sampling, IBS counts dispatched micro-operations and samples from retired ones. Consequently, an IBS counter can overflow on any instruction fetch or micro-operation, and the sampled fetch or micro-operation does not have to be the event that is targeted by a profiling code. For instance, a profiling code that aims to profile memory accesses might encounter non-memory access samples during profiling.

4.2.3 Sampled Data

Observation 4 *IBS generates rich set of attributes for each sampled fetch or micro-operation that record the hardware events during the execution in CPU pipeline. These hardware events might correspond to multiple different precise events in PEBS. Thus, PEBS might have to monitor multiple events simultaneously in order to get the same level of information.*

Though IBS has only two sampling flavors, there are a number of attributes included in each sampled data. IBS generates 16 attributes in each sampled data for instruction fetch and 44 attributes in each sampled data for executed micro-operation. These attributes can help identify the hardware events that are triggered by each sampled instruction fetch or micro-operation, for example, whether a sampled micro-operation triggers a memory load, an L1 cache miss, or an L1 DTLB miss.

In each sample, PEBS generates a PEBS record that contains information related to the sampled event, such as the instruction pointer, the architectural state of the logical core after the event is retired, and the effective address in case the sampled event is a memory access. In addition to this general information, PEBS also offers more detailed information on the origin of accessed data in memory hierarchy if load latency sampling facility is enabled.

There is some common information that is available in the data sampled by both PEBS and IBS, such as memory access latency and whether a fetched instruction or data misses in L1 or L2 cache. However, there are also some attributes that are included in IBS' sampled data but not in PEBS, and vice versa. One example is the width of accessed memory region, which is an attribute in IBS but not in PEBS. To make up for this, a profiler that uses PEBS will have to utilize a supplementary library such as Intel XED [24]. Moreover, PEBS has to monitor multiple events simultaneously in order to get the same amount of information offered by one sample of IBS. For instance, to profile memory loads, memory stores, and branch instructions, PEBS has to monitor those three events separately, while IBS can profile these in a single micro-operation sample.

4.2.4 Execution Mode

Observation 5 *While Intel PMUs can be programmed to count events that execute only in user mode, only in kernel mode, or in any mode indiscriminately, IBS counters can only count instruction fetches and micro-operations without regarding their execution mode.*

In each event select register (`IA32_PERFEVTSELx`) of Intel PMUs, there are bits that determine whether the controlled counter should count events that execute in user mode or kernel mode. As a result, it is possible to program PEBS to sample events that execute only in user mode, only in kernel mode, or any of the two modes. In contrast, IBS does not have this ability supported at hardware level. Consequently, IBS always triggers an interrupt regardless of the execution mode of the event that it samples, and the identification of the execution mode has to be done by the interrupt handler by checking the `user_mode(regs)` macro provided by the Linux kernel. This approach might be less accurate than the method used by PEBS, which can dedicate a counter to count events that execute only in certain mode. The inaccuracy of this approach can affect events that occur very close to mode switches between user mode and kernel mode. For example, a tagged event that executes in user mode can be counted as an event from kernel space if it occurs just before the execution mode switches from user mode to kernel mode.

4.3 Quantitative Comparison

In this section, we quantitatively analyze PEBS and IBS. Through carefully designed benchmarks, we quantify the accuracy of both PEBS and IBS under different scenarios: (i) accuracy in monitoring a single event, (ii) accuracy in monitoring multiple events, (iii) accuracy under different sampling intervals, and (iv) the stability of the accuracy across multiple runs. We also study the sampling bias and the functionality of both PEBS and IBS to attribute samples to their instructions and we evaluate the time/memory overheads. Lastly, we study the functionality supported by PEBS and IBS to detect samples from kernel/user mode execution.

The experiments presented in this section were carried out on an Intel Xeon Gold 6258R CPU and an AMD EPYC 7352 Zen 2 CPU. Table 4.2 displays the specifications of the machines. The microbenchmarks used in the experimental study were written using `asm` statement [3] as assembly instructions included in C code and compiled using `gcc-10.3.0` compiler on the Intel machine. The same binary is also used on AMD to ensure that the same software level optimizations are applied.

Specification	**AMD**	**Intel**
CPU Model	AMD EPYC 7352 CPU	Intel Xeon Gold 6258R CPU
Microarch Family	Zen 2 (17h)	Cascade Lake
#Sockets	2	2
#Cores/Socket	24	28
#SMT [111]	2-way	1-way
Private Caches	L1i, L1d, L2	L1i, L1d, L2
Cache Sizes	L1i: 32KB, L1d: 32KB, L2: 512KB, L3: 16MB	L1i: 32KB, L1d: 32KB, L2: 1MB, L3: 39MB
Linux Kernel Version	Linux 5.11.0-36	Linux 5.11.0-36

Table 4.2: Specs of the AMD and the Intel Machines

Unless otherwise stated, the default compiler optimization flag is -O0. We chose this optimization flag because we did not want any compiler optimizations to modify our microbenchmarks.

We programmed PEBS using `perf_event_open` system call, and programmed IBS using the IBS driver available in [47]. We chose `perf_event_open` to interface with PEBS as it is the most widely used method that is utilized by Linux perf tool [48, 4] and a number of other profiling tools [5, 74, 119, 69, 23, 101]. We used the IBS driver in [47] to program IBS because it is the only possible way to interface with IBS in our AMD machine as `perf_event_open` requires certain versions of BIOS that are not commonly available as default in commodity AMD machines [45, 46]. To minimize overheads at user space, we implemented a dummy signal handler to handle OS signals triggered by sampling interrupts for both PEBS and IBS. The signal handler is dummy as it does not read the data sampled by PEBS or IBS. The default sampling interval in all experiments is set to 100K. The experimental results are averaged over 5 runs.

4.3.1 Accuracy

In this experiment, we aim to evaluate the accuracy of PEBS and IBS in capturing samples from a benchmark with known number of monitored events. We define accuracy as the closeness of the number of samples captured by PEBS or IBS to the number of expected samples given a sampling interval and a known number of events in the benchmark.

Hypothesis

Based on Observations 2 and 3, we expect PEBS to have better accuracy than both sampling flavors of IBS in capturing samples from any hardware event.

Methodology

To evaluate the accuracy of PEBS and IBS, we ran both sampling facilities on a microbenchmark with known numbers of loads and instructions. We programmed PEBS to sample the precise event version of retired instruction, i.e. `INST_RETIRED:PREC_DIST`, and retired load, i.e. `MEM_INST_RETIRED.ALL_LOADS`, in separate runs. We configured IBS to run micro-operation execution sampling, i.e. `IBS op`, and instruction fetch sampling, i.e. `IBS fetch`, in separate runs.

To evaluate the accuracy of both sampling facilities, firstly, we compared the number of retired instruction samples captured by PEBS, the number of executed micro-operations sampled by IBS op, and the number of instruction fetches sampled by IBS fetch against their ground truths. The ground truth for PEBS' retired instruction sampling is the number of instructions in the microbenchmark divided by the sampling interval, the ground truth for IBS fetch is the number of instruction fetches that hit in L1 ITLB as counted by `perf` divided by the sampling interval, and the ground truth for IBS op is the number of retired micro-operations counted by `perf` divided by the sampling interval.

After evaluating the accuracy of those sampling facilities in monitoring their most general events, i.e. instructions, instruction fetches, and micro-operations, we

evaluated their accuracy in sampling a subset event, which is *load* operation in our case. For this evaluation, the numbers of load samples detected by PEBS' retired load monitoring and IBS op are compared against the expected load sample count of the microbenchmark. IBS op is used for this comparison because it can capture loads among its sampled micro-operations.

For this experiment we devised a microbenchmark, called *Load-Ratio* as it can be configured to have different load ratios to all instructions. One of such configurations, *1/4 Load*, is shown in Listing 4.1. In each iteration of `loop0` there is exactly one load, i.e. `movl (%rax), %ebx`, out of four instructions. Therefore, the ratio of load to any instruction in the configured benchmark is 1/4. By knowing the number of loads and instructions in the benchmark, given the sampling interval we can easily calculate the number of expected load and instruction samples. For example, in the *1/4 Load* case, there is one load out of four micro-operations in each iteration, and the loop iterates 10 billions times. Thus, in total, there are 10 billions loads and 40 billions instructions in a single run of the benchmark. As we set up PEBS to monitor retired load and retired instruction with sampling interval 100K, we can expect 400K instruction samples to be generated, and among these, the number of load samples should be 100K.

```
movq $10000000000, %rcx
movl $1, %ebx
loop0:
movl (%rax), %ebx
subq $1, %rcx
cmpq $0, %rcx
jne loop0
```

Listing 4.1: Code for *Load-Ratio* benchmark with 1/4 load ratio

Results

Figure 4.1 compares the accuracy of PEBS when monitoring retired instruction against both sampling flavors of IBS. In the figure, PEBS, IBS op, and IBS fetch

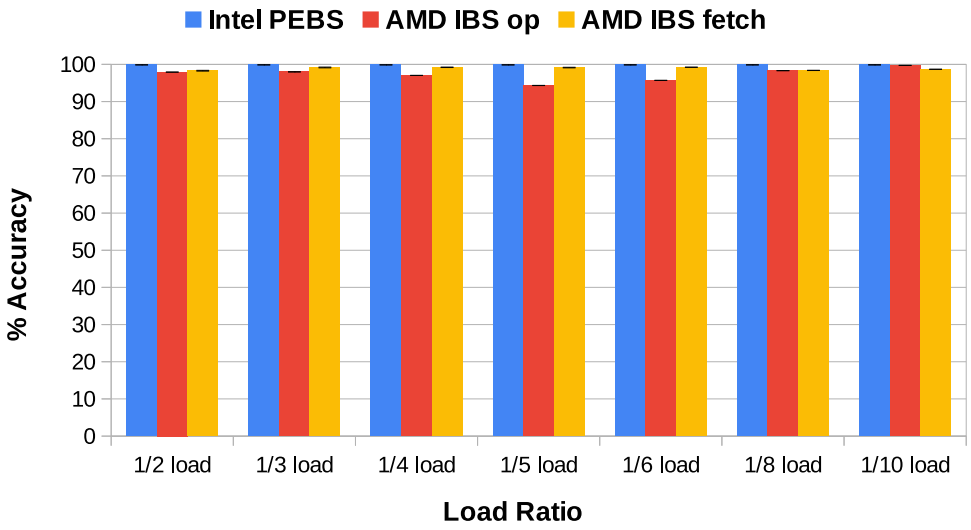


Figure 4.1: Comparison of accuracy of PEBS monitoring retired instruction, IBS monitoring micro-operation execution (IBS op), and IBS monitoring instruction fetch on the *Load-Ratio* benchmark

display high accuracy, though IBS op still shows slightly lower accuracy than the rest due to its randomization of counter after each sampling interrupt. Figure 4.2 shows the accuracy of PEBS and IBS when capturing load samples from the microbenchmark. The plotted results are produced by PEBS that monitors retired load and IBS op. From the figure, it can be seen that PEBS achieves higher accuracy as its sample counts are very close to the expected counts, while IBS deviates more from the ground truth. Unlike the results in Figure 4.1, IBS displays lower accuracy in Figure 4.2. It shows that, though IBS can capture micro-operation and instruction fetch samples accurately, its detection of subset event, e.g. how many of the detected micro-operation samples are load samples, is less accurate.

Findings

PEBS always shows high accuracy in sampling any event, while IBS is accurate only in sampling its most general events, i.e. micro-operation execution and instruction fetches. When detecting a subset event, such as load operation, among its samples, IBS displays lower accuracy.

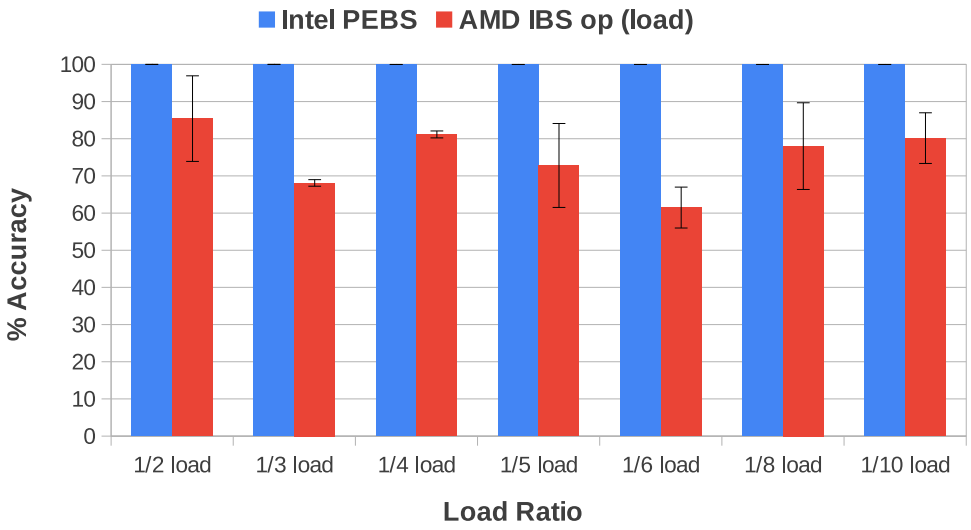


Figure 4.2: Accuracy of PEBS and IBS in capturing retired load samples from the *Load-Ratio* benchmark

4.3.2 Sensitivity to Sampling Rate and Stability

Next, we evaluate the accuracy under different sampling intervals and the stability of both schemes. Stability here refers to variation of accuracy across multiple runs.

Hypothesis

Due to Observations 2 and 3, we expect PEBS to have higher accuracy than IBS under any sampling interval. Based on those observations, we also expect PEBS to be more stable, i.e. having more consistent accuracy, when profiling the same benchmark across different runs.

Methodology

To evaluate the accuracy across different sampling intervals, we use the *1/4 Load* configuration of the *Load-Ratio* benchmark, and we alter the load operation in each loop iteration into a locked load operation. The reason for this is that we invoke *ioctl* function calls to disable and re-enable the counter of PEBS in the dummy signal handler, and the load operations in the *ioctl* function code can significantly

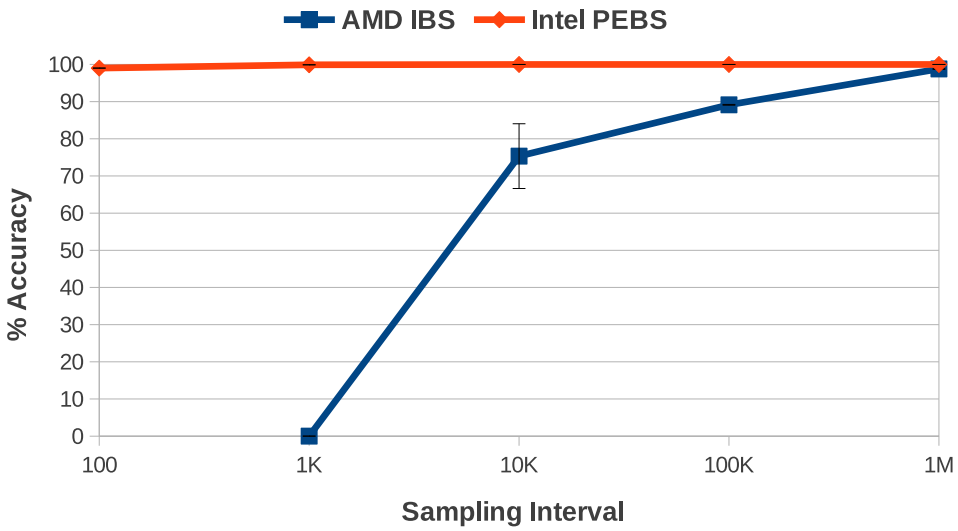


Figure 4.3: Accuracy of PEBS and IBS in sampling locked load operations under different sampling intervals

increase the number of loads counted by the counter when the sampling interval is very low, e.g. 100. As locked load operations do not exist in the code of *ioctl*, we introduce locked load operations in the benchmark, and monitor them using PEBS and IBS to prevent any inaccuracy resulting from our monitoring tool that might happen at very low sampling intervals. We refer to this modified benchmark as the *Locked-Load* benchmark.

We also evaluate the stability of PEBS and IBS by measuring the standard error of sample counts detected across multiple runs in the accuracy benchmarks.

Results

Figure 4.3 shows the accuracy of PEBS and IBS when sampling locked load operations from the *Locked-Load* benchmark under different sampling intervals. While PEBS maintains high accuracy across different sampling intervals, the accuracy of IBS is high only at 1M sampling interval and degrades dramatically at shorter intervals. As the IBS profiler terminates prematurely when the sampling interval is 100, we do not show any result from that case.

In Figures 4.1, 4.2, and 4.3, the measured standard errors are presented as error bars. PEBS displays high stability, i.e. low variation of accuracy, regardless of the event and the number of events that it monitors. In contrast, Figures 4.2 and 4.3 show higher variation of accuracy for IBS in detecting sub-events of executed micro-operations, e.g. the number of load and locked load samples among detected samples. The maximum standard error of IBS op is 11.66% in Figure 4.2 and 8.71% in Figure 4.3, which is at 10K sampling interval, while the maximum standard error of PEBS is nearly zero. However, IBS is quite stable in capturing the cumulative counts of micro-operation and instruction fetch samples when their sub-event types are not considered as shown in Figure 4.1.

Findings

In capturing samples of subset events, such as load, locked load, PEBS has high accuracy under different sampling intervals, while IBS exhibits high accuracy only at large sampling interval, and its accuracy decreases substantially when the sampling interval become short. PEBS displays high stability across multiple program executions, and IBS is relatively stable in capturing cumulative counts of micro-operation and instruction fetch samples. However, its stability is much lower in capturing counts of subset event samples.

4.3.3 Bias and Instruction Attribution

In this experiment, we evaluate the bias of PEBS and IBS in sampling the same event from multiple different locations in a benchmark, and the accuracy of their ability in attributing samples to the instructions that trigger them.

Hypothesis

We expect PEBS and IBS to have no bias in sampling from multiple different instructions that perform the same monitored event. We also expect PEBS and IBS to accurately attribute the sampled events to the actual instructions that trigger those events.

Methodology

```
movq $10000000000, %rcx
movl $1, %ebx
loop0:
movl (%rax), %ebx // load 1
movl (%rax), %ebx // load 2
movl (%rax), %ebx // load 3
movl (%rax), %ebx // load 4
subq $1, %rcx // subq
cmpq $0, %rcx // cmpq
jne loop0 // jne
```

Listing 4.2: Code for the sampling bias and instruction attribution

We evaluate the sampling bias and the instruction attribution of PEBS and IBS by programming them to sample retired loads from a synthetic benchmark, *Bias-Bench*, shown in Listing 4.2. If there is no sampling bias, the portion of samples attributed to each load instruction should be 25%. Furthermore, if all samples can be associated with their triggering instructions accurately, there should not be any sample associated with non-load instructions, i.e. *subq*, *cmpq*, and *jne*. We use the tools from HPCToolkit [5] to attribute the sampled instruction pointers to the source code lines in the benchmark code.

Results

Table 4.3 presents the percentage of samples attributed to each instruction in the *Bias-Bench* benchmark. As PEBS does not associate any load samples with the non-load instructions, we can infer that PEBS accurately attributes the load samples to the instructions that actually trigger them. In contrast, IBS does not associate any of the samples with the *load 1* instruction, and it associates 29.62% of the load samples with the *subq* instruction. From these results, we can infer that IBS actually records the instruction pointers of the next instructions that execute after the actual instructions that trigger sampling interrupts. Each instruction pointer recorded by IBS is likely to be the content of the *rip* register, which is the address of the next instruction to be executed, at the time the micro-operation tagged by IBS retires.

Concerning the sampling bias, both PEBS and IBS do not detect load samples

Instruction	**Expected**	**Intel PEBS load**	**AMD IBS op**
load 1	25%	58.6%	0%
load 2	25%	9.04%	3.58%
load 3	25%	18.34%	62.14%
load 4	25%	14.02%	4.64%
subq	0%	0%	29.62%

Table 4.3: Percentage of samples attributed to each instruction in the *Bias-Bench* benchmark.

equally across the load instructions. For PEBS, most samples are associated with the *load 1* instruction. For IBS op, most load samples are attributed to the *load 3* instruction, which means they were triggered by the *load 2* instruction.

Findings

Both PEBS and IBS are equally biased in sampling an event from multiple different instructions. From the synthetic benchmark that we use in this experiment, more than 50% of the samples are captured only from 1 out of 4 instructions that execute in a loop.

While PEBS could accurately attribute samples to the instructions that trigger them, the instruction pointers recorded by IBS op are not actually the addresses of the triggering instructions. Those instruction pointers are the addresses of the next instructions that execute after the triggering instructions.

4.3.4 Time Overhead

We evaluate the time overheads of PEBS and IBS by running them on simple and more complex benchmarks.

Hypothesis

Based on Observation 3, we expect PEBS that monitors a specific hardware event, e.g., memory load or store, to incur lower time overhead than IBS as it will most likely encounter fewer sampling interrupts than IBS. However, if PEBS monitors retired instructions, we expect PEBS to incur similar overhead to IBS because it will encounter approximately the same number of sampling interrupts as IBS.

Methodology

We programmed PEBS and IBS to sample from the *Load-Ratio* benchmark and the Rodinia benchmark suite [25] as representative real workloads. The Rodinia applications were compiled using the gcc compiler on the Intel machine with optimization flag -O2. We configured PEBS to monitor retired load and retired instruction in separate runs. In the AMD machines, we programmed IBS to run IBS op and IBS fetch also in separate runs. We ran each benchmark on a single thread to avoid microarchitectural factors such as cache line contention that might become a confounding variable that affects time overhead.

Results

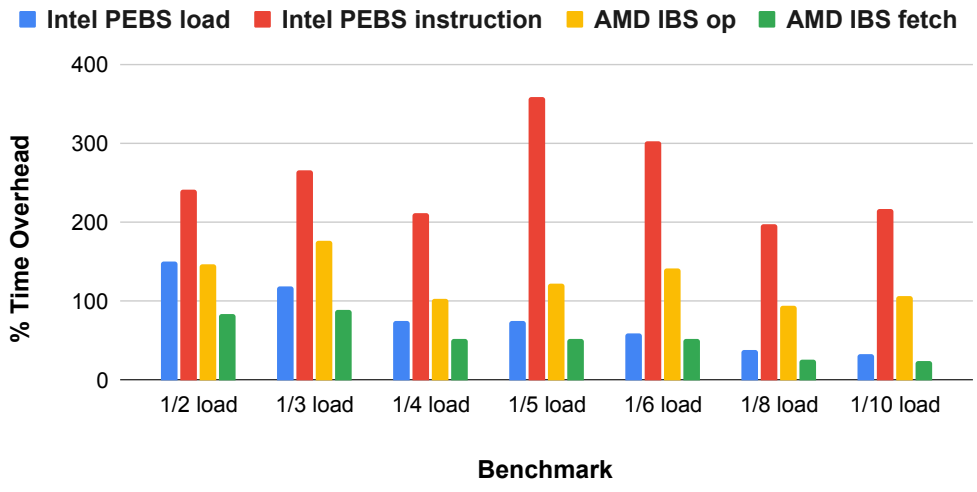


Figure 4.4: Comparison of time overheads on *Load-Ratio* benchmarks

Figure 4.4 presents the time overheads of PEBS and IBS for all chosen events and sampling flavors when monitoring the *Load-Ratio* benchmark. One observation from the figure is that IBS fetch almost always incurs the lowest overhead. It has lower time overhead than retired instruction monitoring by PEBS and IBS op because an instruction fetch actually fetches multiple instructions that lie in the same fetch block from an L1 instruction cache. Therefore, the number of instruction fetches is always fewer than both the numbers of executed instructions and micro-operations. However, interestingly, IBS fetch incurs lower time overhead than PEBS that monitors retired load in all cases. This result deviates from our expectation. The reason for this could be the higher overhead in PEBS imposed by the combination of the sampling mechanism in hardware, the microcode that writes to PEBS buffer, and the kernel code that handles the interrupts. The same reason is also what makes PEBS' retired instruction monitoring have higher time overhead than both sampling flavors of IBS.

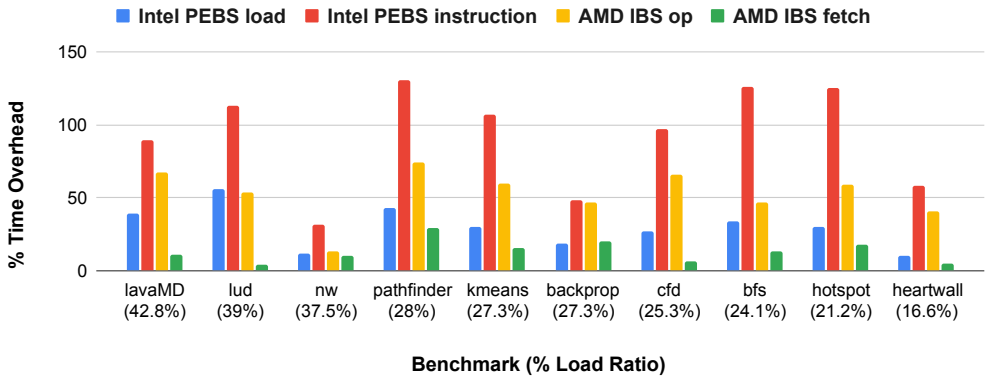


Figure 4.5: Comparison of time overheads on Rodinia benchmarks. The X-axis shows the benchmark name and its load ratio.

Figure 4.5 shows the time overheads on more complex workloads when monitoring 10 Rodinia benchmarks. PEBS that monitors retired load incurs higher time overhead than IBS fetch for nearly all benchmarks except *backprop*. These results contradict the expectation in our hypothesis but align well with the *Load-Ratio* benchmark results. Furthermore, PEBS that monitors retired instruction always exhibits higher

time overhead than both sampling flavors of IBS.

When comparing Figure 4.4 and 4.5, the time overheads of PEBS' retired instruction monitoring and IBS op are lower on the Rodinia benchmarks than on the *Load-Ratio* benchmark. This result can be attributed to the higher variety of instruction mix in Rodinia that may include types of instructions that have higher latencies than the load, arithmetic, and branch instructions in the *Load-Ratio* benchmark. The extra latency incurred by these instructions makes the portion of time spent in handling sampling interrupts lower within the entire execution time of the profiled program.

Findings

PEBS incurs higher time overhead than IBS both for simple and complex benchmarks, which contradicts our expectation. The higher time overhead of PEBS can be attributed to the following factors. (i) On each sample, PEBS records the entire CPU architectural state, whereas IBS records only a few specific events of interest. (ii) On each sample, the microcode of PEBS has to copy the entire architectural state into PEBS buffer in main memory, which is more expensive than IBS, which involves only reading a few model-specific registers by the interrupt handler.

4.3.5 Memory Overhead

In this experiment, we evaluate the memory overheads of PEBS and IBS. Measured memory overhead is the maximum resident set size of a process in main memory during the process' lifetime while being monitored by PEBS or IBS.

Hypothesis

As we use a dummy signal handler to handle sampling signals, we expect low overheads in terms of maximum resident set size in main memory from both PEBS and IBS. The memory overheads of PEBS and IBS should also be approximately the same.

Methodology

We evaluate the memory overheads by having PEBS and IBS monitor 10 Rodinia benchmarks. Using PEBS, we monitored retired load and retired instruction in separate runs, and we also ran IBS op and IBS fetch separately. We chose not to use the *Load-Ratio* benchmark as the memory footprint of the benchmark is too small and when monitoring that microbenchmark, the memory overhead of any profiling code that uses PEBS or IBS would appear to be much larger than it usually is when profiling real workloads.

Results

Figure 4.6 displays the experiment results from the 10 benchmarks. As can be seen from the figure, PEBS and IBS display nearly the same memory overheads on all of the tested benchmarks. It shows that memory overhead in terms of maximum resident set size in main memory is not a function of sample count when each sampling interrupt is handled by a dummy signal handler. The figure also shows that the memory overheads incurred by the mechanisms in hardware and OS kernel code that handle each sampling interrupt in PEBS and IBS are nearly the same.

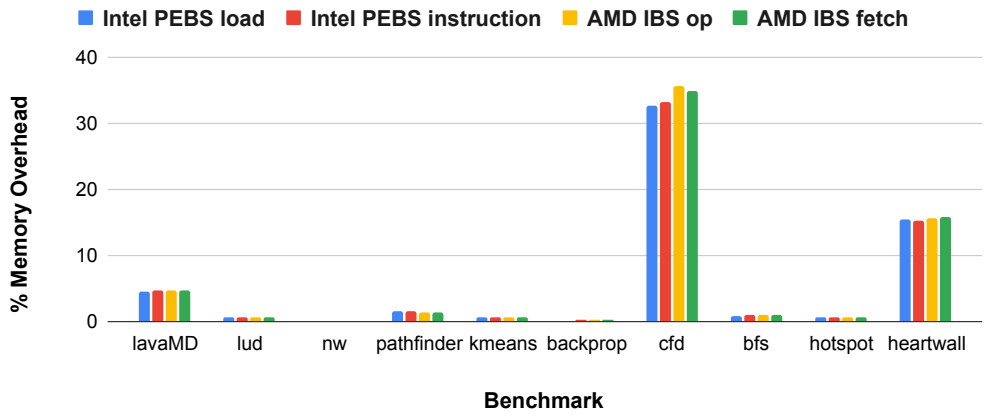


Figure 4.6: Comparison of memory overheads on Rodinia benchmarks

Findings

PEBS and IBS incur nearly the same amount of memory overhead. This result confirms the expectation in our hypothesis.

4.3.6 Multiple Event Monitoring

As a consequence of Observation 4, PEBS might have to monitor multiple events simultaneously to capture the same amount of information that can be captured by IBS in one run. Next, we compare the accuracy and overhead of PEBS that monitors multiple events against IBS. The accuracy here refers to how close the detected sample counts to the expected.

Hypothesis

Based on Observations 1, 2, and 3, we expect PEBS to have better accuracy than IBS as long as the number of monitored events is less than or equal to the number of general-purpose counters. In case the number of monitored events is higher than the number of general-purpose counters, PEBS will lose samples, thus its accuracy would drop.

```
void foo1() {
        return;
}
int main () {
        int val = 0;
        __asm__ __volatile__ (
                "movq $100000000, %%rcx\n\t"
                "loop0:\n\t"
                "call foo1\n\t"
                "lock\n\t"
                "addl $1, %0\n\t"
                "lock\n\t"
                "addl $1, %0\n\t"
                "movl %0, %%ebx\n\t"
                "subq $1, %%rcx\n\t"
                "cmpq $0, %%rcx\n\t"
                "je loop1\n\t"
                "cmpq $0, %%rcx\n\t"
                "jne loop0\n\t"
```

```
                "loop1:\n\t"
                : "=m" (val)
                :
                : "memory", "%eax", "%ebx", "%ecx"
            );
        return 0;
}
```

Listing 4.3: Code for multiple event monitoring benchmark

Methodology

To compare the accuracy of PEBS monitoring multiple events with IBS, we developed a microbenchmark, called *Mult-Event* benchmark, that has known numbers of *load, store, branch, taken branch, return,* and *locked load instructions*. The code of the microbenchmark is shown Listing 4.3. We programmed PEBS to monitor one, four, five and six of these events in separate runs to observe the effect of monitoring more events than the number of available general-purpose counters on accuracy.

In addition to evaluating accuracy, we also used PEBS to monitor different individual events and different numbers of events on a larger benchmark to see how multiple event monitoring affects the profiling time and memory overheads.

Results

Figure 4.7 shows the accuracy of PEBS when monitoring multiple events simultaneously on the *Mult-Event* benchmark. We compare the accuracy of PEBS when monitoring 1 event (each event is monitored alone), 4 events (only load, store, branch, and taken branch are monitored together), 5 events (all of them except return), and 6 events (all of them) against the accuracy of the micro-operation sampling of IBS. Because there are only 4 general-purpose counters that can be used by PEBS in each logical core, it is shown that PEBS loses higher percentage of samples and undercounts when the number of events that it monitors is higher than the number of available counters, i.e. when there are 5 or 6 events monitored. These results confirm our hypothesis.

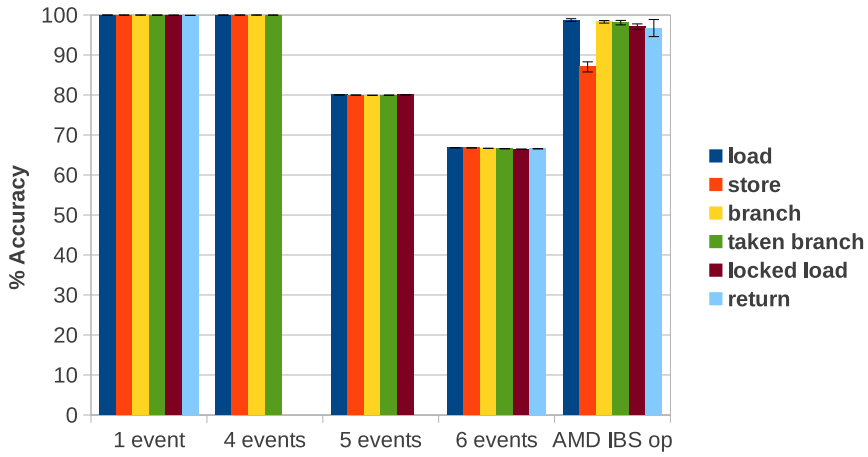


Figure 4.7: Comparison of PEBS accuracy in monitoring different numbers of events against IBS op. PEBS monitors multiple events simultaneously except for 1 event case, where it monitors each event in a separate run. IBS can capture all events at once in its microoperation sampling.

We also evaluate the effect of multiple event monitoring on overheads. Table 4.4 presents the compared overheads of PEBS across different individual events and different event counts when monitoring the *heartwall* benchmark from Rodinia. To evaluate the overheads when monitoring different event counts, we programmed PEBS to monitor 4 different sets of events in addition to monitoring individual events, i.e. retired instruction, retired L1 load miss, and retired load, across different runs.

Since we ran the experiment using a dummy signal handler to handle each sampling interrupt, the memory overheads are nearly the same across the different cases. The monitoring of retired instruction is shown to have he highest time overhead, which is expected as retired instruction is the most frequent of all events monitored in this experiment. When PEBS monitors other events, we can observe lower time overheads and we can still see that more frequent events, such as retired load, incurs higher time overhead than less frequent events, such as retired L1 load miss. Furthermore, as we increase the number of monitored events, the time overhead also increases until we reach the point where number of monitored events equals the number of available general-purpose counters in each logical core, which is in *set*

Monitored Event/ Event Set	**Event Count**	**Time Overhead**	**Memory Overhead**
IBS op	-	1.4x	1.15x
PEBS retired instruction	1	1.58x	1.16x
PEBS retired l1 load miss	1	1.01x	1.16x
PEBS retired load	1	1.1x	1.16x
PEBS set 1 (retired load + retired store)	2	1.13x	1.16x
PEBS set 2 (set 1 + retired L1 load hit + retired conditional branch)	4	1.35x	1.16x
PEBS set 3 (set 2 + retired taken near branch + retired mispredicted conditional branch + retired L1 load miss + retired L2 load hit)	8	1.33x	1.16x
PEBS set 4 (set 3 + retired mispredicted taken near branch + retired L2 load miss + retired L3 load hit + retired far branch + retired L3 load miss + 3 L3 load hit via cross-snooping (none, miss, hit)	16	1.27x	1.16x

Table 4.4: Overheads of PEBS monitoring multiple events and IBS op on the *heartwall* benchmark from Rodinia suite.

2. When we increase the number of events further, the time overhead stagnates as there are fewer sampling interrupts when multiplexing is applied at the cost of lower accuracy.

Findings

If the number of events that are simultaneously monitored is higher than the general purpose counters in PEBS, accuracy of PEBS drops. However, its time overhead is affected by the type of the event monitored rather than the number of events monitored because of the event multiplexing.

4.3.7 Kernel Mode vs User Mode Identification

In this experiment, we evaluate the methods utilized by PEBS and IBS to identify the execution mode of the sampled events.

Hypothesis

Based on Observation 5, we expect the execution mode of the sample detected by PEBS to be more accurate than by IBS.

Methodology

To evaluate the accuracy of execution mode detection methods in PEBS and IBS, we developed a microbenchmark comprising a code that runs in user space and a simple Linux kernel module. This microbenchmark is referred to as *Exec-Mode* benchmark. The user-space code and the kernel module of the *Exec-Mode* benchmark run together to cause repetitive execution mode switching during the execution of the microbenchmark. The code of the user-space code and the relevant piece of code in the kernel module are shown in Listing 4.4 and 4.5. To repeatedly switch from user mode to kernel mode, the user-space code calls the *ioctl(fd, TEST_CMD)* function call in every loop iteration. In the assembly code in Listing 4.4, the system call number of *ioctl*, which is 0x10, is passed as a parameter for the *syscall* instruction in the *%eax* register, the value of *fd* is already stored in the *%edi* register earlier before

the shown code, and `TEST_CMD`, which is a macro for 0x67, is placed in the *%esi* register as the third parameter for *syscall*. Upon handling the *ioctl* function call, the code in Listing 4.5 executes in the kernel mode. Using this *Exec-Mode* benchmark, we can expect that 1 billion locked load operations occur in kernel space and no such operation in user space.

```
movq $1000000000, %r8
loop0:
movl $0x10, %eax
movl $0x67, %esi
syscall
subq $1, %r8
cmpq $0, %r8
jne loop0
```

Listing 4.4: Code in user space

```
case TEST_CMD:
        lock
        addl $1, (%rax)
break;
```

Listing 4.5: Code in kernel space

We ran this experiment by installing the kernel module and running the microbenchmark while being monitored by PEBS and IBS. If the user mode detection method is accurate, we expect no locked load sample to be detected in user space.

Results and Findings

Based on the 5 runs of the benchmark, the output from PEBS always shows no detection of locked load sample in user mode, while the output from IBS shows that 41 locked load samples on average out of 10K expected samples in the user space. These results show that PEBS can detect execution mode precisely, while misattribution of execution mode might occur to IBS samples.

4.4 Full-Fledged Profiling Tool

To compare the precise event sampling capabilities of Intel and AMD for a full-fledged profiling tool, we use an open-source tool, ComDetective [101], which monitors the inter-thread communication within an application. The main idea of ComDetective is to use PMU samples and debug register traps to detect cache line transfers between threads. We performed experiments to compare the accuracy, overheads, and stability of the communication analyzer under PEBS and IBS. The

sampling interval that we use in each experiment is 500K, which is the default sampling interval in the experimental study reported in [101].

Accuracy

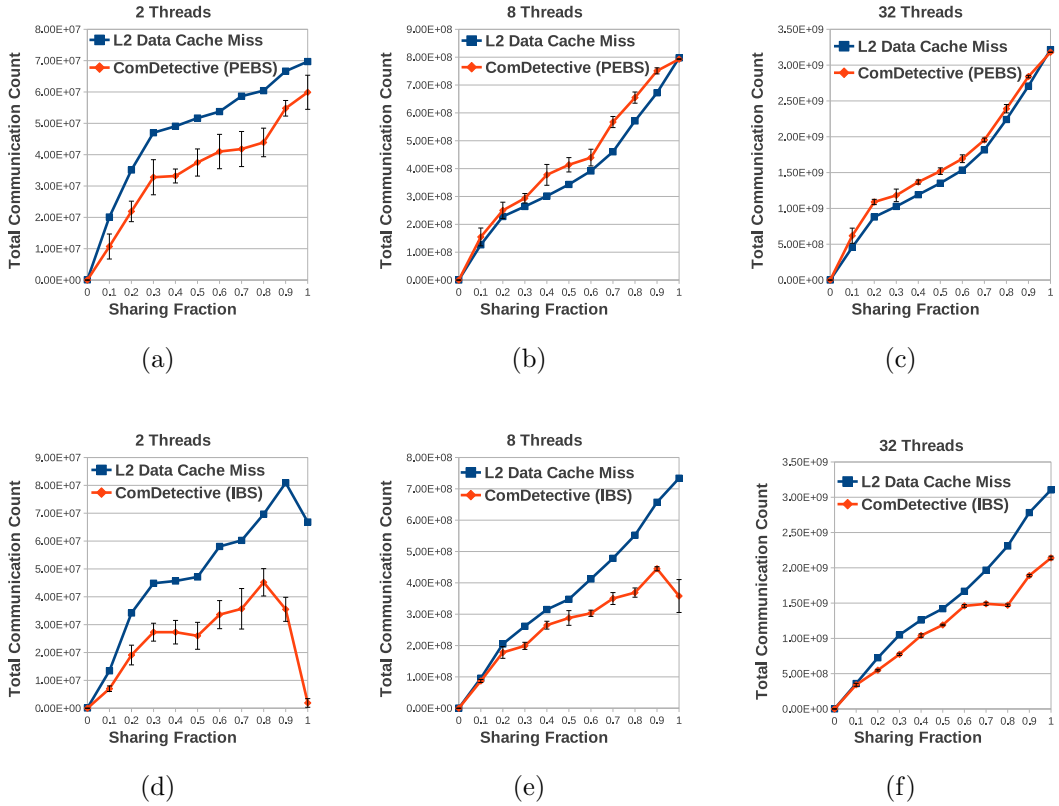


Figure 4.8: Total communication counts under different sharing fractions in the Intel (Figures 4.8a, 4.8b, 4.8c) and AMD (Figures 4.8d, 4.8e, 4.8f) machines.

To compare the accuracy, we ran ComDetective on a microbenchmark named *Write-Volume* benchmark that we developed. In this benchmark, all threads perform an atomic write operation to either a shared variable or a private variable in a loop that iterates 100M times. The number of accesses to the shared variable by each thread is controlled by a parameter called *sharing fraction*. For example, if the sharing fraction is 0.7, each thread writes to the shared variable approximately 70M times. The ground truth for total communication count in this experiment is the L2 data cache misses counted by *perf* since each thread is mapped to

have its own L2 cache and the number of cache line transfers is equal to the L2 data cache misses. Figure 4.8 displays the total communication counts detected by ComDetective and the ground truths under different sharing fractions when multiple threads are mapped evenly across different sockets. Consistent with our results in Section 4.3.1, ComDetective exhibits higher accuracy with PEBS than AMD as the gaps between the total communication counts and the ground truths are closer in Figures 4.8a, 4.8b, and 4.8c than in Figures 4.8d, 4.8e, and 4.8f.

Time and Memory Overheads

To evaluate the time and memory overheads of ComDetective when working with PEBS and IBS, we ran ComDetective on 10 Rodinia benchmarks in the Intel and AMD machines. Each benchmark ran with 32 threads, and all threads were bound to CPU cores with compact mapping.

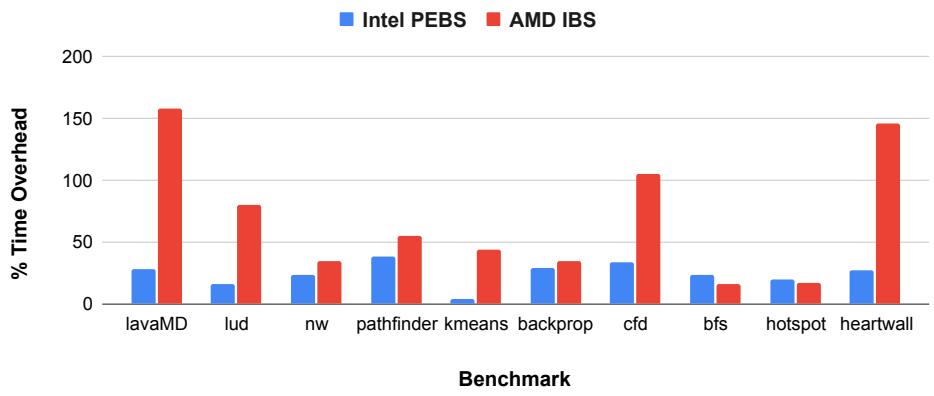


Figure 4.9: Time Overheads of ComDetective when running with PEBS and IBS on 10 Rodinia benchmarks

Figure 4.9 compares the time overheads of ComDetective incurred in both machines. Consistent with the results of our experiment on the *Load-Ratio* benchmark presented in Table 4.4, ComDetective incurs lower time overheads on nearly all benchmarks when running with PEBS that monitors only retired load and retired store. The only exceptions are *bfs* and *hotspot*, which generate more memory access samples in the Intel machine than in the AMD machine. Since both benchmarks

produce more memory access samples in the Intel machine, ComDetective performs more operations that are intended to detect inter-thread communications such as reading/updating a global data structure and arming debug registers, which incur extra overheads.

While running on the Rodinia benchmarks, ComDetective also incurs nearly the same memory overheads in the Intel and AMD machines. The results of this experiment confirm our results from Figure 4.6 and Table 4.4.

Stability

As shown by the standard error bars in Figure 4.8, ComDetective displays the same level of stability when running with PEBS and IBS in nearly all cases. Lower stability, i.e. high standard error, is shown when ComDetective runs on 2 threads. As thread count increases, the stability also gets higher. However, exceptions can be seen on the results from IBS when the sharing fraction is 1 and the thread counts are 8. In that case, IBS exhibits lower stability than PEBS as indicated by the larger error bar.

Chapter 5

COMDETECTIVE: INTER-THREAD COMMUNICATION ANALYSIS

5.1 Introduction

Inter-thread communication is an important performance indicator in shared-memory multi-core systems [112]. Thread communication information offers valuable insights: it divulges, to an extent, the inner workings of the program without having to examine the code meticulously; it can be used for identifying possible sources of communication-related performance overhead in parallel applications [23, 107]; it can also be used for verifying the multicore hardware design. Therefore, identifying which groups of threads communicate in what volume and their quantitative comparison against expectations offer avenues to tune software for high performance.

Several techniques exist to capture communication patterns in multi-threaded applications [14, 29, 34, 33, 109, 12, 28]. Though the proposed techniques succeed in generating communication patterns (often called as communication matrix), they come with several limitations. Simulator-based methods (e.g., [14] [29]) (a) make simplistic assumptions about CPU features (e.g., an in-order core), cache protocols and memory hierarchies, (b) introduce $\sim 10,000\times$ runtime slowdown, and (c) generate enormous volume of execution traces that grow linearly with execution time; hence, they are a misfit for evaluating a complex, long-running application in its entirety. Furthermore, to extract communication patterns from simulators, post-mortem analysis of execution traces is needed, which adds additional effort to the user.

Approaches in [109][12][28] use either a modified operating system kernel or hardware extensions to mitigate overheads. The communication pattern that they gen-

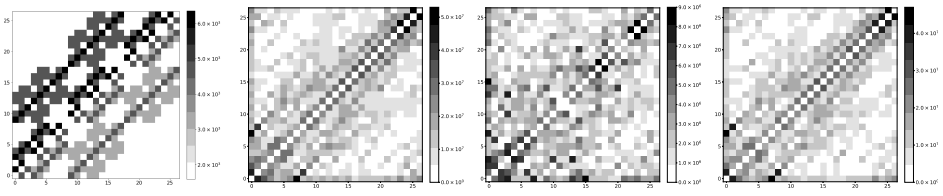
erate, however, might contain *false communication*[1]—a situation where a cache line that is already evicted by a core is accessed by another core. Such false communication is reported when the accesses to the same cache line by different cores are separated in time. Prior approaches using binary instrumentation techniques, such as [34][33], detect communications only by retaining the thread ids of previous accesses but disregard the timestamps of those accesses. Hence, these schemes also suffer from false communication. An additional source of inaccuracy in binary instrumentation is the time dilation caused by fine-grained instrumentation—the time gap between consecutive accesses by the same core to the same cache line is widened due to the online analysis overheads, which allows other threads to interleave, which in turn results in overestimating communication compared to uninstrumented execution. For example, Numalize [33], one such tool that we use for comparison in our experimental study, dilates execution, changes the execution behavior, and as a result, overestimates total communication count. Other works by Mazaheri et. al [82][83] instrument program code by using a compiler-assisted tool. The code instrumentation enables detection of read-after-write (RAW) and read-after-read (RAR) dependencies among threads in the program and generates true communication (RAW) and reuse (RAR) matrices as outputs. However, their method still introduces large overhead, on average 140× slowdown.

In this work, we propose ComDetective, a communication matrix extraction tool that avoids the drawbacks of the prior art. The key premise of ComDetective is to observe the execution with minimal perturbation. ComDetective resorts to the data offered by hardware Performance Monitoring Units (PMUs) and debug registers as a means of measuring inter-thread communication. Hardware PMUs enable extracting the effective addresses involved in loads and stores in sampling fashion. Additionally, debug registers enable monitoring memory access to a designated address by a thread, without introducing any overhead in the intervening

[1]False communication should not be confused with false sharing. False sharing results in communication at the hardware level that was not intended by the programmer, while false communication does not lead to inter-core communication.

window of execution. By employing both PMUs and debug registers, we are able to detect memory accesses performed by different threads on shared cache lines in a short time window while not becoming a severe victim of false communication, unlike other approaches.

Besides being lightweight, ComDetective differentiates communication as true vs. false sharing, where true refers to the actual communication intended by the programmer due to the shared objects and false refers to the false sharing between two threads due to the cache line sharing. Two-dimensional matrices that are generated by tools such as Numalize[34][33] do not differentiate different types of communication. Figure 5.1 shows a motivating example, where we present the communication matrices for the multi-threaded implementation of LULESH [60] and compare it against the MPI implementation. The MPI matrix is generated using EZTrace [110] and requires post-mortem analysis. Meanwhile executing the application with ComDetective took only 136 sec with 1.48× runtime overhead. In addition, ComDetective can optionally attribute communication to each object in the application. To the best of our knowledge, there exists no other tool for multi-threaded applications that delivers these features while maintaining a low overhead. Our contributions can be summarized as follows:



(a) LULESH - MPI (b) LULESH (c) LULESH True Sharing (d) LULESH False Sharing

Figure 5.1: Communication matrices of LULESH taken from an Intel Broadwell machine (Left to Right: MPI, ComDetective: All, True and False Sharing). Darker color indicates more communication.

- ComDetective, a communication detection algorithm and its lightweight

tool for multi-threaded applications with the feature to distinguish false vs. true sharing communication

- A thorough evaluation of accuracy, sensitivity, and overhead of ComDetective, and tool's comparison with ground truth and prior work

- Insightful communication matrices of PARSEC benchmark suite and six CORAL applications (AMG, LULESH, MiniFE, PENNANT, Quicksilver, and VPIC), and comparison with MPI communication matrices for the CORAL applications

- Independent of code size, only 30% runtime and 27% memory overheads on the 18 applications studied, making it a practical tool for production use.

The ComDetective tool is available at `https://github.com/comdetective-tools`.

5.2 Background

Inter-thread communication: We define communication among threads as the transfer of cache lines across different CPU cores due to cache coherence protocol in a shared-memory system. An example is a transfer of cache line from a thread running on a core that has a cache line with 'modified' status, according to MESI protocol, to another thread running on a different core that has the same cache line in the 'invalid' status. Such communication or cache line transfer can also happen from a core that has a cache line with 'exclusive', 'modified', or 'shared' status to another core that does not have that cache line in its local caches.

This kind of communications can occur due to either *true sharing* or *false sharing*. True sharing happens when two different threads communicate or transfer a cache line as both of them access the same variable located in the cache line. False sharing ensues when two threads communicate on a cache line, yet they do not access the same variables, but these variables happen to reside on the same cache line. While true sharing is an inevitable communication for cooperating threads

in parallel programs, false sharing can be considered as an overhead since the two threads do not actually need to communicate as they access different variables.

Communication Matrix: Communication matrix is defined as a matrix that counts instances of communications between each pair of threads in a multi-threaded application. The $(i, j)^{\text{th}}$ entry in the matrix represents the number of communication instances between thread i and thread j. The communication matrix is symmetric (both parties are involved in communication) and has zero along the diagonal (a thread does not communicate with itself). The cells only count the number of cache line-granularity data transfers; they do not account other transactions that may be involved by the underlying implementation of the coherence protocol.

5.3 Design of ComDetective

In generating communication matrices, ComDetective leverages PMUs and debug registers to detect inter-thread data movement on a sampling basis. If communication is frequent, the same addresses appear in the samples taken on communicating threads; by comparing the addresses seen in closely taken samples on different threads, one can potentially detect communication. If communication is infrequent, however, the probability of seeing the same address in two samples taken by two different threads becomes rare. Hence, ComDetective leverages debug registers to identify infrequent communications. A thread sets a watchpoint for itself to monitor an address recently accessed by another thread. If and when the thread accesses such address in the near future, the debug register traps and thus detects communication.

In ComDetective, each application thread uses PMU to sample its memory access (load and store) events. When a threshold number of events of a certain type (load or store) happen, the corresponding PMU counter overflows. The thread, say T_1, encountering an overflow extracts the effective address involved in the instruction at the time of the overflow (aka sample) and tries to publish the address on to a global data structure, `BulletinBoard`, that other threads can readily access. When another thread, say T_2, encounters its PMU overflow, it looks up the `BulletinBoard`

for an address conflicting with its sampled address located on the same cache line. If such an entry is found in `BulletinBoard` and the two accesses are by different threads, then communication is detected between the two threads. If, however, no conflicting entry is found, it may mean the sampled address may be a private address (which is common when the fraction of sharing is less) or the thread may access the location in the near future. In this situation, T_2 picks an *unexpired* address $\mathcal{M}$ posted in `BulletinBoard` and arms its CPU's debug registers to monitor all or as many as possible addresses that fall on the same cache line $\mathcal{L}$ shared by $\mathcal{M}$. A subsequent access by T_2, anywhere on $\mathcal{L}$, is a communication between T_2 and the thread that published $\mathcal{M}$. This communication will be detected by trapping of the watchpoints in T_2. Once communication is detected, the corresponding communication matrices are updated. The communication is reported if and only if at least one store operation is involved.

ComDetective maintains `BulletinBoard` as a concurrent hash table. The sampled address, rounded down to the nearest cache line address, serves as the key to the `BulletinBoard`; the value for each entry in the `BulletinBoard` is the following tuple: Memory address $\mathcal{M}$ accessed at the point of PMU sample, access length δ, ID of the publishing thread, timestamp of the publishing. Only addresses involved in store operations are inserted into the `BulletinBoard`, but PMU address samples generated for both loads and stores are looked-up in the `BulletinBoard` to detect communication. This arrangement detects both write-after-write and read-after-write sharing; note that any repeating write-after-read sharing in one thread will be captured as a read-after-write sharing in another (the reader) thread.

5.3.1 Communication Detection Algorithm

The main components of ComDetective and one possible workflow scenario are displayed in Figure 5.2. Next, we explain the algorithm used in ComDetective.

Setup: Every thread configures its PMU to monitor its memory store and load events. Each of these threads is interrupted on elapsing a specified number of events.

Algorithm 1 Communication Detection

```
1: global ConcurrentMap BulletinBoard
2: thread_local Timestamp t_prev = 0
3:
4: procedure PMUSampleHandler(Address M_1, AccessLen δ_1, Timestamp ts_1, ThreadID T_1, AccessType A_1)
5:     L_1 = getCacheline (M_1)
6:     entry = BulletinBoard.AtomicGet (key=L_1)                    ▷ Is L_1 in hash?
7:     if entry == NULL then                                        ▷ Matching cache line is not found in hash
8:         TryArmWatchpoint(T_1)
9:     else
10:        < M_2, δ_2, ts_2, T_2 > = getEntryAttributes (entry)
11:        if T_1 != T_2 and ts_2 > t_prev then                     ▷ A new sample from a different thread
12:            if [M_1, M_1 + δ_1) overlaps with [M_2, M_2 + δ_2) then
13:                Record true sharing
14:            else
15:                Record false sharing
16:            end if
17:            t_prev = ts_2
18:        else
19:            TryArmWatchpoint (T_1)
20:        end if
21:    end if
22:    if (A_1 is not STORE) or (entry != NULL and M_2 has not expired) then
23:        return
24:    end if
25:                                  ▷ A_1 is a store and the current entry has expired, then publish M_1
26:    BulletinBoard.TryAtomicPut(key = L_1, value = <M_1, δ_1, ts_1, T_1>)
27: end procedure
28:
29: procedure TryArmWatchpoint(ThreadID T)
30:    if current WPs in T are old then
31:        Disarm any previously armed WPs
32:        Set WPs on an unexpired address from BulletinBoard that is not from T
33:    end if
34: end procedure
```

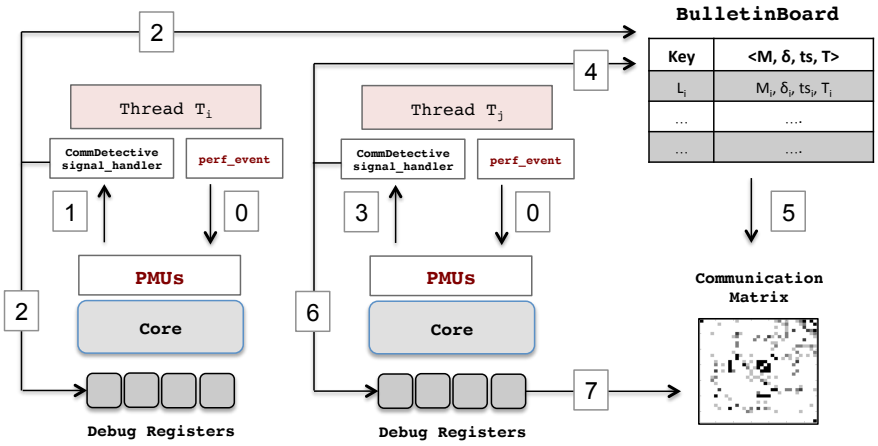


Figure 5.2: One possible execution scenario: 0) Every thread configures its PMU to sample its stores and loads. 1) Thread T_i's PMU counter overflows on a store. 2) T_i publishes the sampled address to BulletinBoard if no such entry exists and tries to arm its watchpoints with an address in the BulletinBoard (if any). 3) Thread T_j' PMU counter overflows on a load. 4) T_j looks up BulletinBoard for a matching cache line. 5) If found, communication is reported. 6) Otherwise, T_j tries to arm watchpoints. 7) T_j accesses an address on which it set a watchpoint, the debug register traps, communication is reported.

On A PMU Sample: When a PMU counter overflows, the thread T_1 that encounters the overflow, tries to publish the address M_1 that it sampled to `BulletinBoard` and calls PMUSampleHandler presented in Algorithm 1. In Line 6, the thread queries the `BulletinBoard` by using the base address of the cache line L_1 containing M_1. If no entry is found, it tries to arm its watchpoints (WPs) (Line 13). If the previously armed WPs are old, the thread T_1 selects an unexpired address M_3 in the `BulletinBoard` and arms its debug registers to monitor the cache line that M_3 belongs to (Line 26-31). Since WPs of a thread belong to the same cache line, they are either all expired or all recent. On x86 with four 8-byte length debug register, ComDetective can monitor only 32 bytes out of the 64 bytes of a cache line. Hence, ComDetective randomly chooses four chunks of the 64-byte cache line to monitor.

In case the entry is already filled by a cache line L_2 from a previous sample and the cachelines are the same, then Line 11 checks the IDs of the publisher thread

T_2 and the sampling thread T_1. If thread IDs are different, then communication is detected between T_1 and T_2 (Line 12-16). The communication could be a true sharing or false sharing. If the sampled access region $[M_1, M_1 + \delta_1)$ overlaps with the access region published in `BulletinBoard` $[M_2, M_2 + \delta_2)$ we treat it as a true sharing event and treat it as false sharing event otherwise. We defer the details of how the volume of communication is computed to Section 5.3.2.

In order not to overcount communications associated with the same published address between two threads, we keep t_{prev} per thread, which is set when a communication is detected for that thread. Line 17 sets t_{prev} to the timestamp of the publisher thread, ensuring that we do not overcount the cache line transfer between two threads. If no communication is recorded for T_1, T_1 tries to arm its WPs (Line 19) using an unexpired addressed published by some other thread into the `BulletinBoard`, as described previously.

If either the sample is for a memory load operation or the previously published entry by the same thread is not expired yet, the thread simply returns and resumes its execution. Otherwise, the thread T_1 publishes the sampled address along with other attributes associated with the cache line L_1, such as the timestamp of sampling, memory access length, and thread ID (Line 26). Atomic operations that perform load and store are treated as store.

On watchpoint trap: When a thread T_i experiences a trap in one of the debug registers, T_i is considered to communicate the thread T_j—the thread that had published an address in the `BulletinBoard` whose cache line T_i is monitoring via its debug registers.

After watchpoint trap: After handling the watchpoint trap, the trapping thread disables all debug register armed to monitor the same cache line. This is justified because the subsequent accesses to the same cache line are *expected* to be served locally without generating any communication. If the cache line were modified by another core in the meantime, it will not be detectable and it is indeed not necessary in the coarse-grained sampling scheme. Watchpoints are re-armed with newer published addresses upon next PMU counter overflow, as explained previously.

On program termination: The profiled data need not leave the matrix symmetric. For example, the reported communication may be more in the thread $\langle T_i, T_j \rangle$ pair compared to the thread $\langle T_j, T_i \rangle$ pair. However, since both parties are equally involved in a communication event, we update every $\langle T_i, T_j \rangle$ pair to be the sum of both $\langle T_i, T_j \rangle$ and $\langle T_j, T_i \rangle$, thus making the matrix symmetric.

Expiration period: For practical considerations, each thread treats the timestamp of a `BulletinBoard` entry as "recent" (aka "unexpired") if it was published between its current sample and its previous sample (i.e., one sample period), and "old" (aka "expired") otherwise. This scheme allows each published address or watchpoint to survive long enough to be observed by all threads working at the same rate and yet be naturally evicted by a newer address. A published address is deemed expired, if it survived for more than two store events from the same thread. Load events are not used for determining the expiration period of a published address, since only stores can ever be published into the `BulletinBoard`. The expiration period of watchpoints includes loads as well because watchpoints can be armed by samples generated by loads or stores.

5.3.2 Quantifying Communication Volume

There are two sources leading to underestimation in communication volume: sparsity of PMU samples and limited number of debug registers to monitor an entire cache line. For instance, four debug registers can cover 32 bytes of the total 64 bytes of an x86-64 cache line. To address the first problem, on each communication detection or trap, instead of recording just one communication event, ComDetective scales up the quantity by the *sampling_period*. In case a communication is detected in a sample and without using debug registers, we update the $Matrix[T_i, T_j]$ cell as: $Matrix[T_i, T_j] += sampling\_period$.

To address the second problem, we use the probability theory. If D number of debug registers can monitor M bytes of memory each, they can monitor a total of $D \times M$ bytes. If the CPU cache line is L bytes long, where $L > (D \times M)$, then the probability of trapping on an address involved in a communication after sampling

it is $p = (D \times M)/L$. If K traps are detected, in expectation, we can scale it up by $1/p$ to get an estimated number of events, i.e., K/p. Taking both effects into account, on each watchpoint trap, we update the $Matrix[T_i, T_j]$ cell as:

$$Matrix[T_i, T_j] += \frac{sampling\_period \times L}{(D \times M)}$$

5.3.3 Implementation

We implement ComDetective atop the open-source HPCToolkit performance analysis tools suite [5]. ComDetective's profiler loads the monitoring library into the target application's address space at link time for statically linked executables or at runtime using `LD_PRELOAD` [87] for dynamically linked executables. As the target application executes, the profiler in ComDetective manages PMUs and debug registers to record communication pairs. On Intel processors, we use `MEM_UOPS_RETIRED:ALL_STORES` and `MEM_UOPS_RETIRED:ALL_LOADS` to sample memory access events. These events offer the effective memory address accessed in a sample along with the program counter. On a PMU sample, the profiler walks the sampled thread's call stack via an online binary analysis. It, then, attributes the measurements to the sampled call path.

Monitoring stack addresses in the target application is tricky, because the frames of ComDetective's sample/trap handler can overwrite the stack location and cause undesired debug register trap. We avoid this problem by establishing a separate signal-handler stack frame for both PMU signal handler and watchpoint exception handler using the Linux `sigaltstack` facility [68]. The `sigaltstack` facility allows each thread in a process to define an alternate signal stack in a user-designated memory region. We use alternate stack to handle PMU and watchpoint signals. All other signals continue to use the default stack unless specified otherwise by the application.

ComDetective optionally allows mapping each communication event to runtime objects in the program. It uses ADAMANT[26] to extract static and dynamic object information. Static objects are detected by parsing the binary file and the dynamic objects are detected by intercepting allocation routines such as `malloc`

and `free`. All stack objects of a given thread are grouped into a single object, while dynamic objects that have the same call stack are grouped into an object.

5.4 Experimental Study

This section evaluates the accuracy, sensitivity, and overheads of ComDetective and presents insightful communication matrices for the selected CORAL and PARSEC benchmarks. Our evaluation systems are a 2-socket Intel Xeon E5-2640 v4 Broadwell CPU and a 2-socket AMD EPYC 7352 Zen 2 (17h) CPU.

In the Intel machine, there are ten cores per socket with 2-way simultaneous multi-threading. Each core has its own local L1i, L1d, and L2 caches, while all cores in a socket share a common L3 cache. We use `Linux 4.15.0-rc4+` and `GNU-5.4` toolchain.

The AMD machine has 24 cores per socket also with 2-way simultaneous multi-threading. Each core has its own private L1i, L1d, and L2 caches, and shares L3 cache with other cores in the same socket. We run `Linux 5.11.0-36` and `GNU-10.3.0` toolchain in this machine.

Unless otherwise stated, the default sampling interval in all experiments is 500K for both reads and writes in the Intel machine and 50K for executed micro-operations in the AMD machine. Furthermore, the default hash table size in `BulletinBoard` is 127.

5.4.1 Accuracy Verification

We evaluate the accuracy of ComDetective with four microbenchmarks we have developed. These benchmarks assess the accuracy against the known ground truth by varying the parameters such as communication volume, false sharing fraction, communicating thread subgroups, and read-to-write ratios.

Write-Volume .

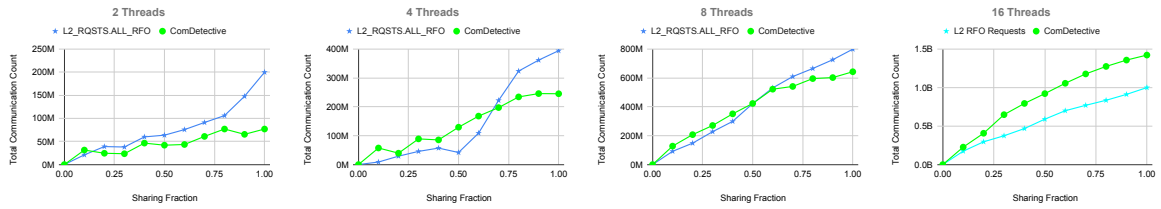
In this benchmark, each thread performs only a single store operation (atomic write) in each iteration of a loop as shown in Listing 5.1. Each thread randomly either accesses its private data or common shared data. The ratio of accesses to shared vs. private data is controlled via the `SHARING_FRACTION`. For example, if the sharing fraction is specified as 20%, then approximately 20% of the time over the entire execution, thread writes into the shared data and writes to its private data in the remaining 80% of the time. There is no false sharing in this benchmark. The source of ground truth for this benchmark is the sum of L2_RQSTS.ALL_RFO hardware performance event obtained from each thread in the absence of other cache sharing effects (which there is none in the benchmark). An RFO event happens when a core tries to gain ownership of a cache line for updating it. If it is not possible to count RFO events in certain machines, e.g. in AMD machines, another possible ground truth for this benchmark that can be measured in those machines is the number of L2 data cache misses. The reason for this is that the number of RFO events in the benchmark is always close to the total number of L2 data cache misses that it encounters.

```
#pragma omp parallel shared(sharedData) private(privateData) \
 num_threads(nThreads)
{
  for(int i = 0 ; i < N_ITER; i++) {
    int rNum  = rand_r(); // thread private
    if (rNum < SHARING_FRACTION) {
      sharedData = rNum;
    } else {
      privateData = rNum;
}}}
```

Listing 5.1: Write-Volume Benchmark

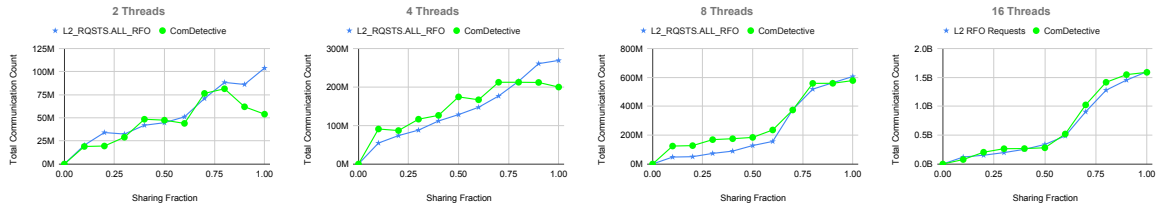
Figures 5.3, 5.4, 5.5, and 5.6 display the results with different number of threads for the *Write-Volume* benchmark running on the Intel and the AMD machines, where the x-axis is the sharing fraction and y-axis is the total communication volume. Figures 5.3 and 5.5 show thread mapping to the same socket (compact), while Figures 5.4 and 5.6 display results from thread mapping to two different sockets

(scatter). As expected, the communication volume increases as the sharing fraction increases or thread count increases. Notice, however, that the ground truths in most cases do not follow straight lines and also in most cases, ComDetective is very accurate in capturing this trend. The nonlinear growth of communication is because when the same cache line is repeatedly accessed by the same core, even if there is a pending request from another core, the request from the core that holds the line is unfairly favored. While such optimizations are not unexpected from a CPU design perspective, they are unintuitive for a programmer and make it harder for them to envision the communication pattern and volume in their programs without the help of tools such as ComDetective. Another unintuitive behavior is that mapping threads to different sockets results in less communication than when they are mapped to the same socket and ComDetective can identify this phenomenon. We have also performed similar experiments with `atomic_add` and `compare_and_swap` and observed similar behaviors.



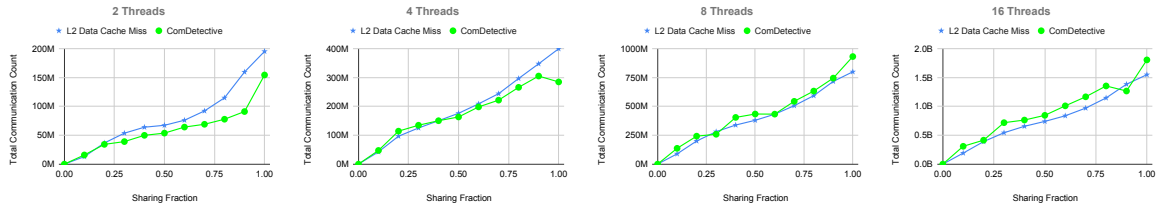
(a) (b) (c) (d)

Figure 5.3: Total communication counts for across different sharing fractions with threads mapped to a single socket (compact) in the Intel machine.



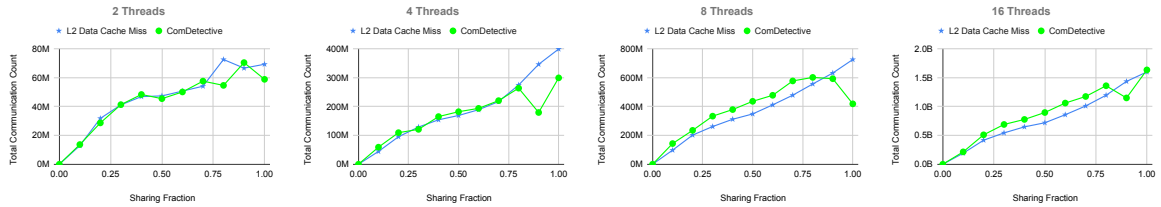
(a) (b) (c) (d)

Figure 5.4: Total communication counts for different sharing fractions with threads mapped evenly to two sockets (scatter) the Intel machine.



(a) (b) (c) (d)

Figure 5.5: Total communication counts for across different sharing fractions with threads mapped to a single socket (compact) in the AMD machine.



(a) (b) (c) (d)

Figure 5.6: Total communication counts for different sharing fractions with threads mapped evenly to two sockets (scatter) the AMD machine.

The gaps of undercounting and overcounting in certain cases is an artifact of sampling that relies on probability theory in estimating total number of communications between any two threads. As described in Sec 5.3.2, we use sampling period to es-

timate the number of communication events that might have been missed between samples. Because of this reason, certain degree of undercounting and overcounting with respect to the ground truth is inevitable.

In Figures 5.3, 5.4, and 5.5, ComDetective underestimates the number of communications when the thread count is small and the sharing fraction is high (˜100%). This undercounting can be attributed to signal handling. When a thread (say T_1) takes a PMU sample or watchpoint trap, T_1's execution gets diverted to handling the signal. During signal handling, T_1 will not generate any cache line communication with its peer thread (say T_2). During this time, T_2 progresses unhindered and continues performing memory access operations across its loop iterations. The act of monitoring reduces communication and hence it appears as undercounting with respect to the unmodified original execution. Note however that this level of extreme sharing without any computation as in our synthetic benchmark shown in Listing 5.1 is as a pathological case for ComDetective and unlikely in real-world code.

The right most plot in Figure 5.3 presents the communication volume for 16 threads running on 10-core socket, where some of the physical cores are oversubscribed with more than one thread. From the figure, it appears that ComDetective overestimates the communication. However, RFO events are no longer the ground truth in this case. This is because `L2_RQSTS.ALL_RFO` counts RFO events between physical cores at L2 caches; and L2 is shared by logical cores. As a result, communication happening between the threads mapped to the same physical core does not result in an RFO event. The RFO counts of threads sharing a physical core are combined if they communicate with other physical cores. Consequently, one would expect that the RFO counts should be lower than the actual communication count when cores are oversubscribed. Indeed, ComDetective gives higher counts than the counts of `L2_RQSTS.ALL_RFO` events.

We compare ComDetective with the prior art in Figure 5.7, which plots the communication volume captured by Numalize [33], ComDetective, and the ground truth when two threads are mapped to the same or different sockets using

atomic add benchmark running on the Intel machine. Numalize hugely overestimates the volume possibly because it does not maintain the timestamp of accesses, records many false communications, and ignore data from the underlying hardware.

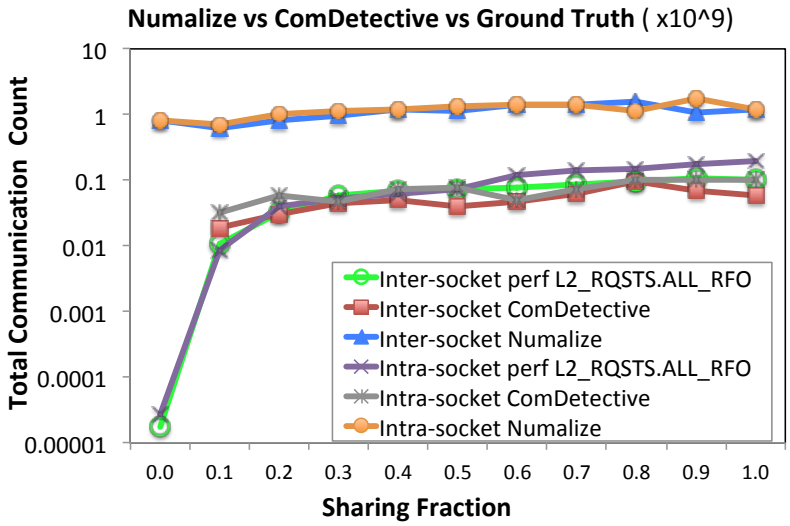


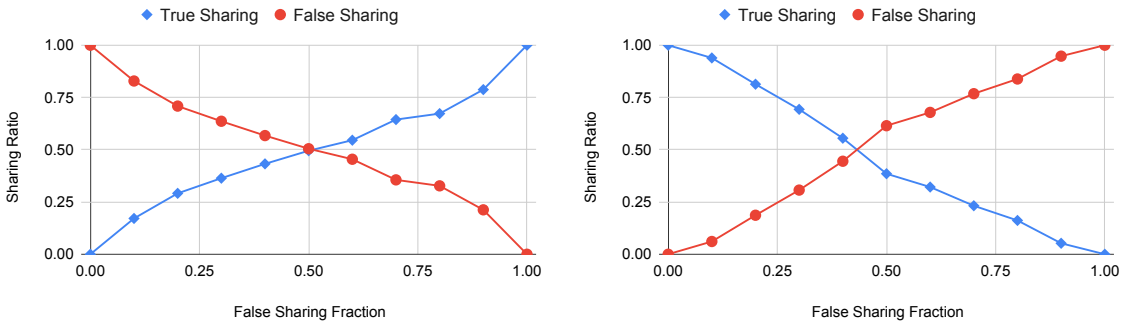
Figure 5.7: Comparison between total communication counts captured by Numalize[33], ComDetective, and the real RFO counts in the Intel machine

***False-Sharing* .**

Unlike *Write-Volume* , which has no false sharing, this benchmark introduces a controllable amount of false sharing as shown in Listing 5.2. Also for coverage, instead of an atomic write, it performs atomic add operation. This benchmark is valuable to assess the statistical nature of randomly selecting parts of a cache line to observe using limited number of debug registers. The ratio of false sharing to the entire communications captured is expected to match the fraction of false sharing specified by the user. Figure 5.8 shows the true and false sharing counts for eight threads with varying false sharing fractions. As expected, the false sharing count increases linearly as false sharing fraction increases. Furthermore, the ratio of false sharing count to total communication count is very close to the specified false sharing fraction for each data point.

```
#pragma omp parallel shared(trueSharingData, falseSharingData) \
  private(privateData) num_threads(nThreads)
{
 int tid = omp_get_thread_num();
 atomic<uint64_t> * falseShared = &(falseSharingData[tid]);
  for(int i = 0 ; i < N_ITER; i++) {
    int rNum  = rand_r(); // thread private
    if (rNum < FALSE_SHARING_FRACTION) {
      *falseShared += rNum;
    } else {
      trueSharingData += rNum;
}}}
```

Listing 5.2: False Sharing Benchmark



(a) Intel (b) AMD

Figure 5.8: Comparing true sharing vs. false sharing counts across different sharing fractions using 8 threads

Read-Write .

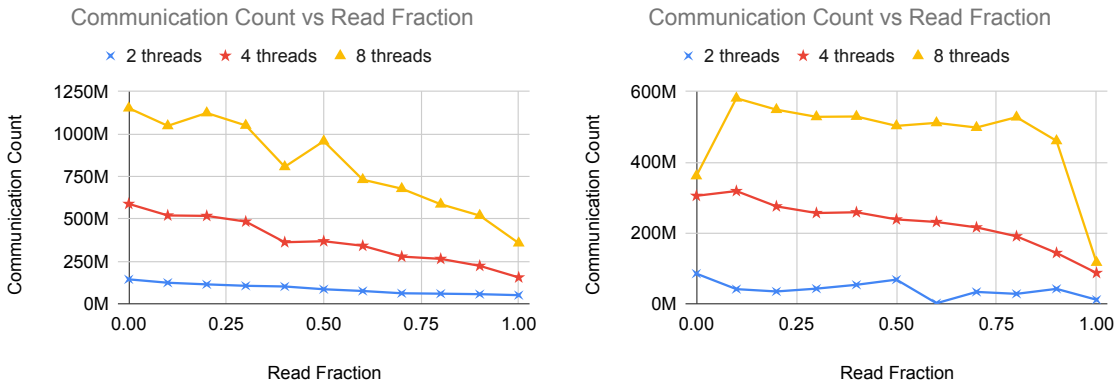
Since only store operations are inserted into the `BulletinBoard`, it is important to assess the quality of results for benchmarks that involve a mix of loads and stores. The benchmark is configured so that one thread always and only performs a write operation in each iteration in a shared location, while the remaining threads might perform either a write or a read operation on the same shared data depending on the specified read fraction. The usage of the read fraction to control the amount of read operations is illustrated in Listing 5.3. For the compiler not to eliminate the

loads, the loads are implemented with asm volatile.

```
#pragma omp parallel shared(sharedData) private(privateData) \
 num_threads(nThreads)
{
  for(int i = 0 ; i < N_ITER; i++) {
    int rNum  = rand_r(); // thread private
    if (rNum < READ_FRACTION) {
      rNum = sharedData;
    } else {
      sharedData = rNum;
}}}
```

Listing 5.3: Read-Write Benchmark. Reading from shared data vs. writing to shared data

Figure 5.9 captures the total detected communication count as a function of read fraction at different thread counts (2, 4, and 8). The communication volume is naturally higher when there are more number of readers. As read fraction increases, more and more reads hit in the local cache before the newly written value by the writer are visible. Thus, increasing the reading fraction linearly decreases the communication volume. One unexpected pattern that we can observe from the results from AMD is that there is a steep increase in communication volume from read fraction 0.0 to read fraction 0.1 when the thread count is 8. The cause of this is a flaw in IBS that labels fewer effective address samples as valid addresses when the number of threads is high and all threads only write to a shared cache line, i.e. when the read fraction is 0.0, than when there are interleaved reads to the shared cache line, i.e. when the read fraction is more than 0.0 and less than 1.0. Since we only consider effective address samples that are labeled as valid addresses in detecting inter-thread communications, this reduction of valid address labels at read fraction 0.0 also results in lower communication count. It is also worth noting in the results from Intel that the drop in communication is more steep with increasing reading fraction for larger number of threads than for a fewer number of threads.



(a) Intel (b) AMD

Figure 5.9: Total communication counts under different fraction of read operations detected by ComDetective

Point-to-Point Communication .

In this benchmark, threads are grouped in pairs and the shared variables are per pair instead of a single shared variable for all threads. This benchmark evaluates the accuracy of point-to-point communication (every cell of the communication matrix). To make a pair of threads communicate, they both need to have similar values of index variables (`shared_data_index`), which point to a same shared array element that they write into as shown in Listing 5.4.

```
#pragma omp parallel shared(sharedDataArray) private(privateData) \
 num_threads(nThreads)
{
 int tid = omp_get_thread_num();
 int shared_data_index = getSharedDataIndex(tid);
 int sharing_fraction =  getSharingFraction(shared_data_index);
 atomic<uint64_t> * sharedData = \
                       &(sharedDataArray[shared_data_index]);
  for(int i = 0 ; i < N_ITER; i++) {
    int rNum  = rand_r(); // thread private
    if (rNum < sharing_fraction) {
      *sharedData = rNum;
    } else {
      privateData = rNum;
```

```
}}}
```

Listing 5.4: Point-to-point Communication Benchmark. Communication happens between threads that have the same `shared_data_index` value

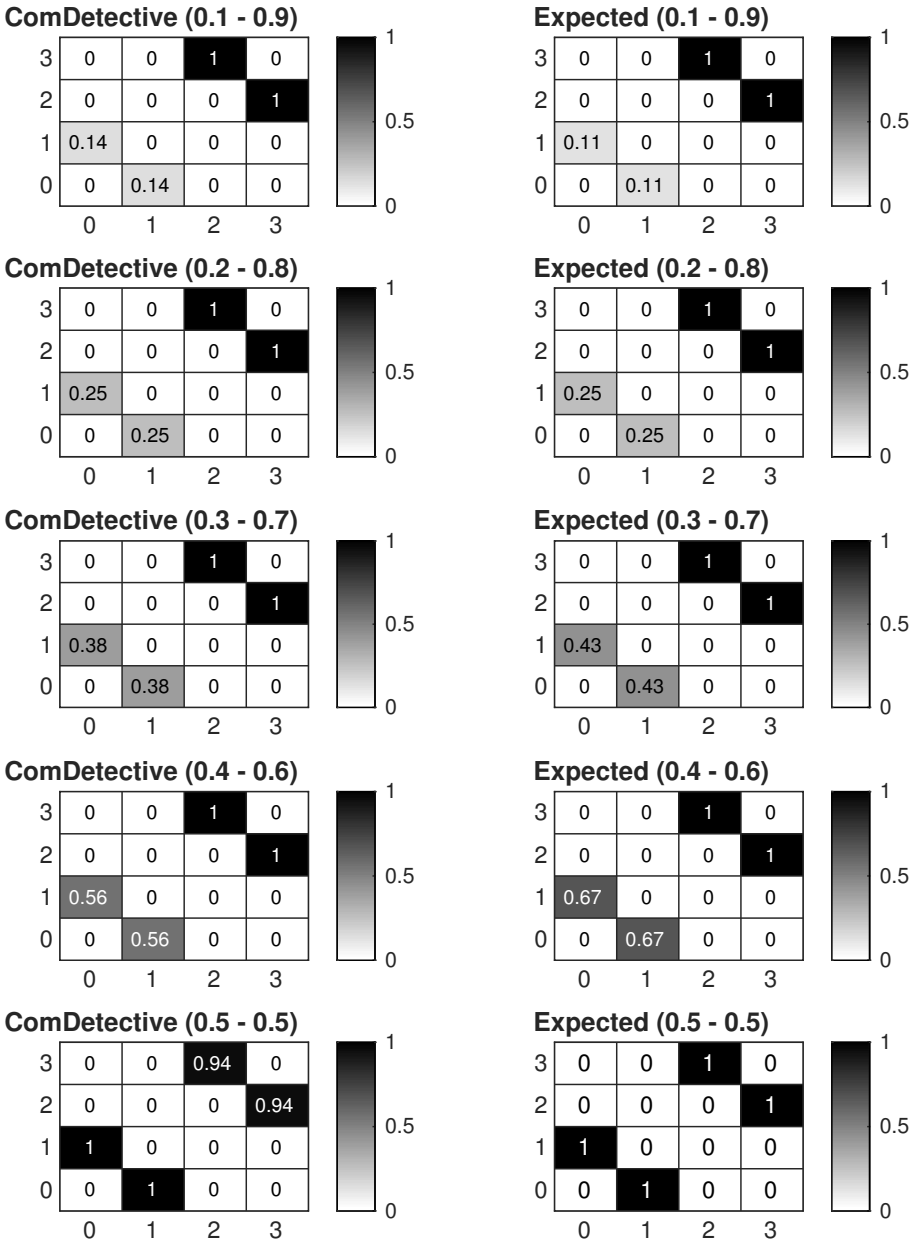


Figure 5.10: Communication matrices for point-to-point communications having different sharing fractions in the Intel machine. Thread 0 only communicates with thread 1, thread 2 only communicates with thread 3. Sharing fractions for each pair are shown on the top of the maps.

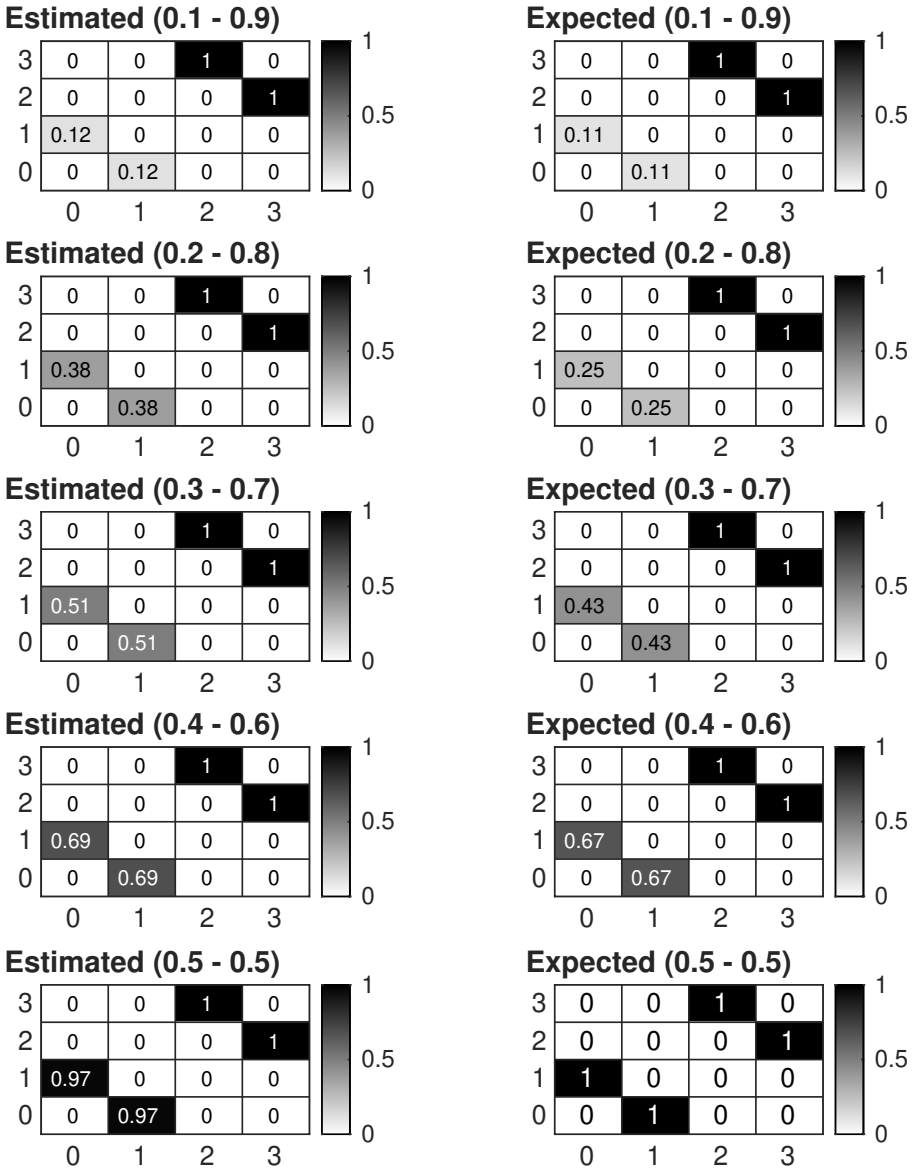


Figure 5.11: Communication matrices for point-to-point communications having different sharing fractions in the AMD machine.

Figures 5.10 and 5.11 show the results for two groups performing only write operations; thread 0 communicates only with thread 1, and thread 2 only communicates with thread 3. Both figures show the communication matrices as heat maps; the observed communication is on the left side and the expected results are on the right side. The number in each matrix cell displays the *normalized* communication count in that cell, which is computed by dividing each cell by the cell with the highest

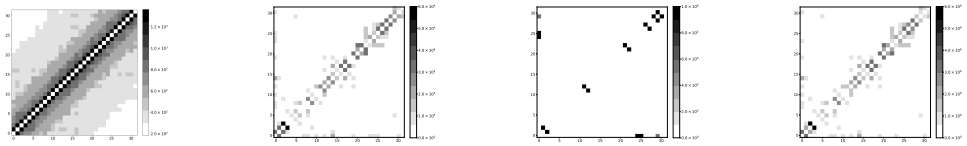
count in its matrix. It is evident that heat maps produced by ComDetective are close to the expected heat maps.

5.4.2 Communication in CORAL Benchmarks

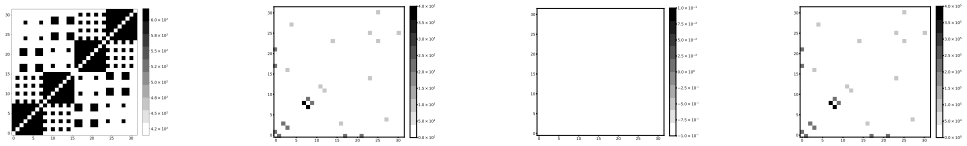
In this section, we present insightful communication matrices for the selected CORAL and CORAL-2 benchmarks, namely AMG [127, 9], LULESH [76], miniFE [86], PENNANT [91], Quicksilver [94], and VPIC [20, 117] as heatmaps in Figure 5.12, where darker color indicates more cache line transfers between pairs. The matrices are core-indexed not thread-indexed as ComDetective can covert the thread IDs to core IDs using the `sched_getcpu()` system call if needed. The threads in each benchmark are bound to the cores with compact mapping strategy but evenly distributed to two sockets.

We compare the inter-thread communication matrices generated by ComDetective with the inter-process communication matrices generated by EZTrace [110]. EZTrace is a generic trace generation framework and it collects the necessary information by intercepting function calls and recording events during execution using the FxT library [30] and then performs a post-mortem analysis on the recorded events. The MPI and OpenMP variants of all six applications are based on the same source distributions with optional flags to turn on/off the OpenMP/MPI compilation in their makefiles. As a result, there are no significant algorithmic differences in their implementations. The MPI matrices report the total number of messages exchanged between processes, not the message size. All applications use 32 threads for OpenMP and 32 ranks for MPI except for LULESH which uses 27 threads (or ranks) since it needs a cubic number. For the hybrid implementations of MPI, we set the thread count per rank to 1.

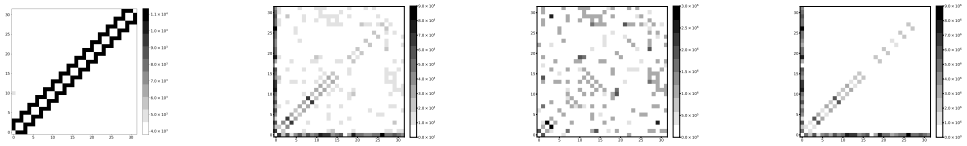
In general, ComDetective offers insights into communication patterns in these applications. For example, the following patterns emerge from our matrices: 1) L-shape pattern in the lower left corners (e.g. *LULESH, PENNANT*), which indicates that all threads heavily communicate with the master thread (a central bottleneck), 2) nearest neighborhood communication pattern, where threads mostly



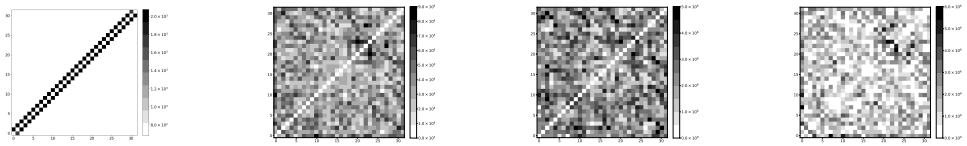
(a) AMG - MPI (b) AMG (c) AMG True Sharing (d) AMG False Sharing



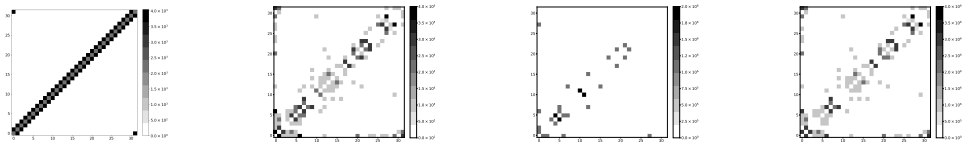
(e) MiniFE - MPI (f) MiniFE (g) MiniFE True Sharing (h) MiniFE False Sharing



(i) PENNANT - MPI (j) PENNANT (k) PENNANT True Sharing (l) PENNANT False Sharing



(m) Quicksilver - MPI (n) Quicksilver (o) Quicksilver True Sharing (p) Quicksilver False Sharing



(q) VPIC - MPI (r) VPIC (s) VPIC True Sharing (t) VPIC False Sharing

Figure 5.12: Communication matrices of CORAL benchmarks. Darker color indicates more communication.

communicate with adjacent threads (e.g. *AMG, MiniFE, VPIC*), and 3) group communications (e.g. *Quicksilver, LULESH*). Although the inter-thread communication matrices are generally more populated than the inter-process communication matrices, in most cases, they logically resemble their MPI counterparts except for MiniFE and Quicksilver. Quicksilver uses a mesh in its computation and the user defines mesh elements per dimension. If the decomposition geometry is not explicitly specified by the user for the MPI ranks, the MPI communication matrix (not shown) becomes very similar to ComDetective's matrix. However, following the suggested decomposition by the Quicksilver developers [94] we decompose the mesh in only one dimension, resulting in nearest neighborhood communication for MPI. It is not possible for a user to perform similar type of decomposition for threads in a configuration file, resulting in more neighbors to communicate.

	Execution Time (sec)		Data Movement (GB)	
	MPI	OpenMP	MPI (Msg Size)	OpenMP (Cache Lines)
AMG	35.19	39.22	6.22	7.33
MiniFE	111.82	142.25	3.24	1.46
Quicksilver	19.04	23.45	32.74	106.13

Table 5.1: Running time and data movement comparison of OpenMP and MPI implementations for AMG, MiniFE and Quicksilver using 32 threads

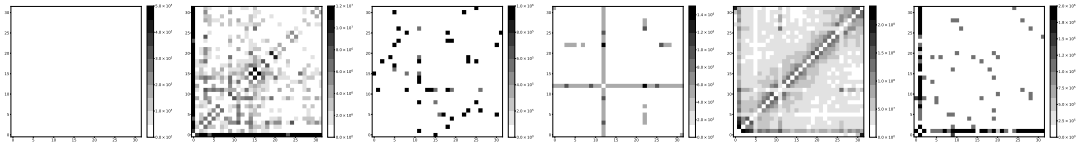
The total communication counts captured by the communication matrices might help explain the performance difference between OpenMP/MPI versions and scalability of benchmarks. Table 5.1 presents the execution time of the *AMG, MiniFE* and *Quicksilver* applications. The table also shows the resulting data movement for each benchmark, where data movement for the multi-threaded applications is calculated based on the total number of cache line transfers in Gbytes with the help of ComDetective. Similarly, for MPI, we computed the total message size exchanged including peer-to-peer and collective communications with the help of EZTrace. In all three applications, MPI outperforms OpenMP. This result, perhaps, can be at-

tributed to the fact that the MPI implementations lead to less data movement than their OpenMP counterparts. For example, the multi-threaded versions of *AMG* and *Quicksilver* perform respectively 11% and 23% more data movement than the multi-process versions. The exception for this is *MiniFE*, in which the communication count of its OpenMP implementation is lower than its MPI counterpart. However, while the MPI version exchanges 0.5M messages for its data movement, the OpenMP version of *MiniFE* leads to 24.5M cache line transfers during its execution, which explains the performance gap.

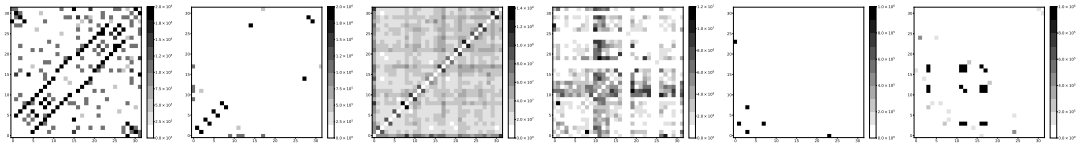
Figure 5.12 also splits the inter-thread communication matrices into two matrices one each for true and false sharing. Due to the space limitation, we discuss the false sharing matrices for only MiniFE, which solves kernels of finite-element applications. It generates a sparse linear-system from the steady-state conduction equation on a brick-shaped problem domain of linear 8-node hex elements and then solves the linear-system using a conjugate-gradient algorithm. ComDetective shows that the communication is among the adjacent threads (other than with the thread id 0) and dominated by false sharing. False sharing occurs `sum_in_symm_elem_matrix` and `sum_into_vector` functions, where adjacent elements in a vector falling into a single cache line are accessed by different threads. While padding each scalar forming the elements of a vector can eliminate such false sharing, it can also have the deleterious effect of bloating the memory.

5.4.3 Communication in PARSEC Benchmarks

Figure 5.13 shows the PARSEC matrices created by ComDetective. Our matrices differ from the ones previously studied by [14], [27] and [33]. In general, ours are sparser. This can be explained by the fact that our approach takes into account the cache coherency protocol. Since we use expiration period to discard false communications among threads, which might happen due to the huge time gap between memory accesses by two supposedly communicating threads, our tool records much fewer false positives than the techniques previously used. In fact, ComDetective identifies no communication for *Blackscholes* and very infrequent communication for



(a) Blacksc- holes (b) Bodytrack (c) Canneal (d) Dedup (e) Facesim (f) Ferret



(g) Blacksc- holes (h) Bodytrack (i) Canneal (j) Dedup (k) Facesim (l) Ferret

Figure 5.13: Communication matrices of PARSEC benchmark suites. Darker color indicates more communication.

Vips and *Freqmine*. *Blackscholes* and *Vips* indeed exhibit very low communication, which is also pointed out by the PARSEC authors [18]. For example, *Blackscholes*, which is a financial analysis benchmark, splits the price options among threads where each thread can process the options independently from each other. Communication can potentially occur at the boundaries of the partitions if boundaries share a cache line. However, it is very unlikely for threads to access the boundaries around the same time because these accesses are far separated in time. The PARSEC authors note that *Freqmine* has a high amount of sharing; however it has a very large working set size too, which implies that accesses are served from memory, not from cache. Moreover, the work in [14] fails to identify any meaningful communication patterns for *Bodytrack, Dedup, Facesim, Ferret, Streamcluster* and *Swaptions*, on the other hand, ComDetective successfully detects these patterns.

5.4.4 Use-Case: Data Structure Optimization

ComDetective can optionally map detected communications, either true or false sharing, to the data objects that experience them at the expense of slightly increased

overhead. Object-level attribution and quantification offers actionable feedback to the developers for object-specific optimizations or code modifications for performance tuning. To demonstrate this feature, we analyzed PARSEC's *fluidanimate* and *streamcluster* to identify their data objects that suffer from false sharing the most. After identifying and analyzing these objects, we modified some of their data structures to reduce false sharing and improve the applications' performance.

For *fluidanimate*, false sharing is caused by several dynamically allocated objects and a global variable named `barrier`. Due to the size of the dynamically allocated objects, applying padding among object elements might result in memory bloat. Therefore, we modified only the data structure of `barrier`. The variable `barrier` is a struct that has `pthread_cond_t` as an attribute. Since the attributes of `pthread_cond_t` are read and written by multiple threads in the `pthread_cond_wait` function, we introduced padding among the attributes of `pthread_cond_t` in the pthread library. After this modification, we achieved 13% speedup in *fluidanimate*.

For *streamcluster*, most of its false sharing is due to inter-thread synchronization by using `pthread_mutex_t` data structure. By introducing padding to the mutex attributes in the pthread library and no changes in *streamcluster* itself, we achieved 6% speedup.

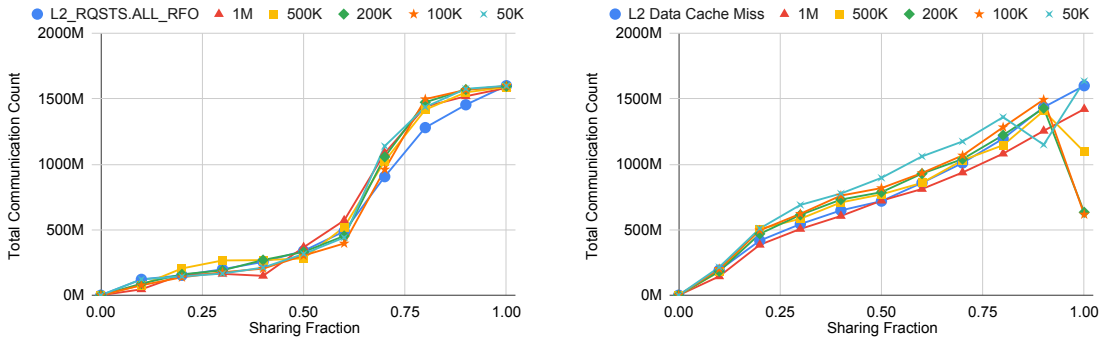
5.4.5 Sensitivity and Overhead Analysis

BulletinBoard Size:

To test the sensitivity of the ComDetective under different hash table sizes, we use the *Write-Volume* benchmark but vary the size of `BulletinBoard.` Using 16 threads, we observe no difference in total communication counts detected by ComDetective under hash table sizes of 5, 17, 31, 61 and 127. Furthermore, we evaluate the performance overhead at different hash table sizes using LULESH [60]. Increasing the hash table size does not materially affect the runtime overhead. For that reason, we use 127 as the hash table size for all experiments.

Sampling Interval:

We measure the sensitivity of the tool against sampling interval in terms of both the accuracy and overhead using the *Write-Volume* benchmark with 16 threads. Figure 5.14 shows the total communication counts under different sharing fractions and sampling intervals from 50K up to 1M. The detected total communication count does not deviate much from the ground truth across all sampling intervals. However, as can be seen in Figure 5.14b, there are noticeable gaps between the detected communication counts and the ground truth when the sharing fraction is 1.0 in the AMD machine for all sampling intervals other than 50K. This is the reason why we chose 50K to be the default sampling interval for our experiments in the AMD machine. In general, we expect that in an application where communication is infrequent, a large sampling interval would result in highly sparse communication matrices or no communication would be detected in the worst case. In such cases, a small sampling interval should be chosen at the expense of increasing overhead.



(a) Intel (b) AMD

Figure 5.14: Total communication counts detected by ComDetective under different sampling intervals compared with the ground truths when 16 threads are mapped to 2 sockets

Overhead:

In the Intel machine, we evaluate the runtime and memory overheads of ComDetective by running it on six CORAL benchmarks and twelve PARSEC benchmarks. Ta-

Sampling Interval	Runtime Overhead			Memory Footprint Overhead		
	AMG	LULESH	MiniFE	AMG	LULESH	MiniFE
100K	1.07×	2.12×	1.16×	1.00×	1.76×	1.00×
500K	1.10×	1.48×	1.10×	1.00×	1.62×	1.00×
1M	1.07×	1.33×	1.06×	1.00×	1.58×	1.00×
2M	1.08 ×	1.20×	1.03×	1.00×	1.51×	1.00×
	PARSEC + CORAL			PARSEC + CORAL		
500K	1.30×			1.27×		

Table 5.2: Runtime and space overhead of ComDetective under different sampling intervals for applications using 32 threads (LULESH 27 threads)

ble 5.2 displays the performance overhead of ComDetective under different sampling intervals for three of the CORAL benchmarks, i.e. AMG, LULESH and MiniFE. As seen from the table, the tool has a low space overhead, which allows it to be used in practice for large-scale applications. The runtime overhead drops significantly when the sampling interval is increased from 100K to 500K for LULESH and the overhead is even lower for the other two applications. Since ComDetective maintains good accuracy with reasonable performance overhead on average at a sampling interval of 500K, we chose 500K as the default sampling interval for all experiments. For the twelve PARSEC benchmarks, the runtime overhead ranges from 1.03× (*streamcluster*) to 2.10× (*x264*) with an average of 1.32×. For the six CORAL benchmarks, the runtime overhead ranges from 1.02× (PENNANT) to 2.17× (VPIC) with an average of 1.27×.

In the AMD machine, we evaluate the overheads by running ComDetective on four PARSEC benchmarks, i.e. *bodytrack*, *fluidanimate*, *streamcluster*, and *swaptions*, with a sampling interval of 50K. The average runtime overhead over these benchmarks is 5.02×, and the average memory overhead is 3.43×.

Debug Registers:

x86 processors have four debug registers, and ComDetective uses all four for arming watchpoints. We study the impact of the number of debug registers (1, 2, 3 and 4) on the total communication counts detected by ComDetective for 16 threads using the *Write-Volume* benchmark running on both the Intel and the AMD machines. We observed that the number of debug registers has a negligible impact on the accuracy of ComDetective. This is because when we quantify the communication volume, we scale the volume based on the number of debug registers as discussed in Section 5.3.2.

Chapter 6

REUSETRACKER: REUSE DISTANCE ANALYSIS

6.1 Introduction

Data locality remains an important concern in shared-memory multicore architectures and it is often a more important concern than computation in terms of both energy consumption and performance [112]. Because of its importance, data locality optimizations have become a central focus of application tuning and tool development [114]. One metric widely used to measure data locality is *reuse distance*. Reuse distance is an architecture-independent metric that is defined as the number of unique memory locations that are accessed between two consecutive accesses to a particular memory location (*use* and *reuse*). For example, consider a sequence of accessed memory locations: a_1, b_1, c_1, b_2, d_1, a_2. In this example, the reuse distance of a is three because there are three unique locations accessed between the two consecutive accesses to a, namely b, c, and d. Reuse distance shows the likelihood of a cache hit for a memory access in a typical least-recently used (LRU) cache. If the reuse distance of a memory access is larger than the cache size, the latter access (*reuse*) is likely to cause a cache miss.

Reuse distance is well-studied for single-threaded programs [81, 36, 125, 129, 17, 15, 41, 57, 105, 22] and a handful of tools are available [2, 125, 119]. The techniques introduced in [35, 58, 78] employ analytical models that digest and analyze execution traces to predict shared cache behavior. The execution traces needed by these models are generated through *hardware simulation* or *binary instrumentation*. Hardware simulators typically incur significant performance and memory overheads and hence are limited to studying only a small part of a full application. Similarly high overhead exists in binary instrumentation methods. Loca, a reuse distance analysis tool [2,

125] implemented using PIN binary-instrumentation tool [96] incurs 49× runtime and 40× memory overheads, which makes the tool impractical to use in real-life applications. In addition to the overhead problem, authors in [92] show that binary instrumentation can potentially distort the parallel schedule among threads of the profiled program.

Significant overheads and distortion of parallel schedule can also occur in the methods proposed in [42, 56, 100, 65, 118, 104, 103] as they rely on either hardware simulators or binary instrumentation. A technique that avoids these flaws was proposed in [92]. This technique reduces trace size by generating traces at coarser granularity instead of at every load/store access. However, as it operates at coarser granularity, it might miss reuses that are very near in distance while can still capture distant reuses. Another shortcoming of existing reuse distance profilers is that a limited number of them are publicly available. To the best of our knowledge, only Loca [2, 125] and RDX [119] are open source. Yet, both of these tools are limited to measuring reuse distances of a single thread and discard interactions among threads in multi-threaded programs. Considering the wide-spread use of multicore architectures, having a fast yet accurate reuse distance analysis tool for multi-threaded applications is the need of the hour.

This work proposes ReuseTracker — an open-source, low-overhead reuse distance analysis tool for multi-threaded applications. To make ReuseTracker fast and memory efficient, we leverage hardware counters of *performance monitoring units* (PMUs) and debug registers, which are commonly available in current commodity CPUs. In ReuseTracker's design, PMUs are utilized to periodically sample memory accesses (*uses*) in each profiled thread, while debug registers are used to trap the next accesses (*reuses*) to the sampled memory regions. While the prior PMU-based tools [119, 16] that profile reuse distance have focused only on reuses in a private cache of a core and single-threaded applications, our tool can analyze multi-threaded codes by considering cache line invalidations in detecting reuses in individual threads. Moreover, our tool detects *reuses* across different threads in profiling reuse distance in shared caches. This capability has several applications,

one of which being deciding the cache sizes in multicore CPUs.

To account for cache-coherence effects when profiling reuse distance in threads, ReuseTracker employs a novel algorithm that monitors loads and stores in each thread. A thread that samples a particular memory access arms one of its debug registers and the debug registers of other cores to trap any future accesses to the sampled accessed address. When a trap happens, if it pertains to the same core that had sampled the initial access, reuse is detected. If the trap pertains to a store access in another core, cache line invalidation is detected, reporting no reuse. When detecting *reuse* in a shared cache, when a trap happens in another core that shares the same cache with the sampling core, a *reuse* is detected. A cache line invalidation is detected when the trap is due to a store access in another core located on a different socket.

Evaluating multithreaded reuse distance, whether for threads or shared caches, turns out to be surprisingly complicated because one may not know the ground truth reuse distance; the act of measuring the ground truth via instrumentation can perturb the parallel schedule and produce incorrect results unless an additional tool such as Intel's PinPlay [90] that can record and replay application execution deterministically is used. Hence, we developed a benchmark that can generate a variety of reuse distance histograms to aid in validating ReuseTracker. Our set of benchmarks is useful not only for evaluating this work but can also serve to evaluate future tools in this area of research.

Table 6.1 compares ReuseTracker against five similar tools that perform online reuse detection. ReuseTracker — to the best of our knowledge — is the first tool that accurately profiles reuse distances in threads and shared caches for multi-threaded applications while introducing significantly lower overhead compared to instrumentation-based or simulation-based tools. More specifically, it introduces 2.9× performance and 2.8× memory overheads. ReuseTracker relies on existing hardware features in commodity CPUs and works on fully-optimized binaries, which makes it appropriate for monitoring full applications in production. Furthermore, ReuseTracker is more accurate compared to other available tools as it profiles

Attributes	Schuff et al. [104]†	Schuff et al. [103]†	StatCache [16]†	loca [125]‡	RDX [119]‡	ReuseTracker‡
Time Overhead	High (>100x)	30x	<1.4×	49×	2.03x	2.9x
Space Overhead	Not reported	Not reported	Not reported	40×	2.8×	2.8×
Intra-thread Profiling Accuracy	Close to 100%	96%	*High but not quantified	Close to 100%	90%	92%
Primary Method	Cycle accurate simulator	Binary instrumentation	PMU	Binary instrumentation	PMU	PMU
Profiling Private Cache	YES	YES	YES	YES	YES	YES
Profiling Shared Cache	YES	YES	-	-	-	YES
Cache Line Invalidation	YES	YES	-	-	-	YES
Open Source	-	-	-	YES	YES	YES

Table 6.1: Comparison of ReuseTracker against other techniques that perform online detection of *reuses*. Overheads of RDX are measured using a sampling interval of 100K. †The reported overheads and accuracy are from the original paper. ‡The reported overheads and accuracy are measured in this work. *StatCache's accuracy is high in terms of predicting miss ratio.

multi-threaded code based on real hardware events without dilating the original execution. It achieves 92% accuracy when verified against a configurable-synthetic benchmark that we developed.

In our experimental study, we also demonstrate several use cases of REUSETRACKER. We show that the reuse distance profiles produced by REUSETRACKER can help guide code refactoring by programmers to reduce false sharing. Through this code refactoring, we could improve the performance of the benchmarks by up to 87%. Furthermore, we also show that our tool can help predict whether profiled applications can gain or lose performance when adjacent cache line prefetch (ACP) is enabled.

In summary, our contributions are listed as follows:

- Formal definitions of reuse in threads and shared caches as *microarchitecture-independent* events in multi-core systems.

- Low-overhead reuse distance analysis algorithms that profile reuse distance in threads and shared caches for multi-threaded programs.

- A synthetic benchmark that can be configured to produce a variety of reuse distance histograms.

- Open source implementation of aforementioned techniques in a tool called REUSETRACKER, as well as extensive evaluation of this tool using a number of benchmarks and applications.

- Demonstration of use-case scenarios using PARSEC [18], Rodinia [25], and Synchrobench [44] benchmark suites where REUSETRACKER guides code refactoring in these benchmarks by detecting spatial reuses in shared cache that are also false sharing and predicts whether some benchmarks in these suites can benefit from adjacent cacheline prefetch (ACP).

The repository of REUSETRACKER is publicly available at `https://github.com/ParCoreLab/ReuseTracker`.

6.2 Background and Terminology

In this section we first provide the background on reuse in single-threaded applications; we then expand the definition to multi-threaded applications; we finally provide a background of a few basic building-block hardware and software facilities needed to develop our tool.

6.2.1 Single-threaded Reuse Distance

Time Reuse Distance and Reuse Distance: *Time reuse distance* is defined as the number of memory accesses between the current access (*reuse*) and the previous access to the same element (*use*). If the accessed element is in a unit of cache line, *reuse* can be classified into two, *temporal reuse* and *spatial reuse*. A temporal reuse happens when both *use* and *reuse* access exactly the same address, while a spatial reuse occurs when its *use* and *reuse* access different addresses that are located in the same cache line.

The term *reuse distance* used in this work refers to *LRU stack distance* as defined in [81]. The difference between *time reuse distance* and *reuse distance* is described in the following example. Consider the sequence of memory accesses: a_1, b_1, c_1, b_2, d_1, d_2 a_2. In this example, the time reuse distance of a is five because there are five *total* memory accesses between the first access to a (a_1) and the second access to a (a_2). On the other hand, the reuse distance of a is three since there are only three *unique* accesses between a_1 and a_2, i.e. b, c, and d.

Time reuse histograms can be accurately translated into stack reuse histograms [106, 119]. REUSETRACKER captures time-reuse during its online profiling and exploits the same conversion techniques during a post-processing step to finally present the stack reuse histograms.

6.2.2 Multi-threaded Reuse Distance

We now define reuse as a *microarchitecture-independent* event in the context of cache-coherent multi-core systems. For the ease of prose, in the rest of this chapter,

we treat OS threads being pinned to CPU cores and cores are not oversubscribed hence we use the terms "threads" and "cores" interchangeably.

Definition 6.2.1 (Intra-thread reuse) *If two consecutive accesses (loads or stores) happen to the same memory location in the same thread without an intervening store to the same location in another thread, then the access pair represents an instance of reuse.*

The time reuse distance is the number of memory accesses elapsed on the *same thread* between the use and reuse events. As we define intra-thread reuse as a microarchitecture-independent event, we disregard microarchitectural configurations such as private cache size, shared cache size, and cache inclusion policy. Therefore a *reuse* in a thread, no matter how long it is from its *use*, is always counted as a *reuse* even if it hits in a shared cache or DRAM memory in a real machine provided that it still happens in the same thread as the *use*.

Definition 6.2.2 (Invalidation for intra-thread reuse) *If two consecutive memory accesses happen to the same memory location by two different threads where the second access is a store operation, the second access represents an instance of invalidation with respect to the first access.*

An invalidation makes the previous access ineligible for reuse with the immediately next access to the same location. Reuse and invalidation are mutually exclusive events. A pair of consecutive memory accesses to the same location can also neither be a reuse nor be an invalidation; for example, a read by one core followed by a read of the same address by another core is neither a reuse nor an invalidation.

Table 6.2 summarizes reuse, no-reuse, and invalidation in all possible read/write access and same/different core scenarios. $R \rightarrow W$ indicates that a write access happens following the read access to the same location. The Read or Write may be accessing data at any level in the memory hierarchy.

Definition 6.2.3 (Reuse in shared cache) *If two consecutive memory accesses happen to the same memory location from two different cores that share the same*

access order	same core	different cores (any socket)
$R \rightarrow R$	reuse	no reuse
$R \rightarrow W$	reuse	invalidation
$W \rightarrow W$	reuse	invalidation
$W \rightarrow R$	reuse	no reuse

Table 6.2: Intra-thread reuse. Read (R) or Write (W) may be accessing data at any level in the memory hierarchy.

	same core	different cores (same socket)	different cores (different socket)
$R \rightarrow R$	no reuse	reuse	no reuse
$R \rightarrow W$	no reuse	reuse	invalidation
$W \rightarrow W$	no reuse	reuse	invalidation
$W \rightarrow R$	no reuse	reuse	no reuse

Table 6.3: Reuse in shared cache.

cache on a multi-core machine, then the access pair represents an instance of reuse in the shared cache.

Definition 6.2.3 covers a situation where one thread fetched data into the shared cache of a multi-core system and a subsequent access by another core can reuse the same data without having to refetch it from the main memory. We are aware of the possibility that an intra-thread reuse might also hit in a shared cache or DRAM. However, we omit that situation from this definition as it has been covered by our definition of intra-thread reuse, and thus, will be detected by the intra-thread profiling. On multi-socket systems, cores span beyond a single socket. In such cases, writes happening on another socket to the same address can create an invalidation at shared cache level. This is analogous to write accesses on another thread in the intra-thread reuse.

Definition 6.2.4 (Invalidation in shared cache) *If two consecutive memory accesses happen to the same memory location by cores on different sockets, where the second access is a store operation, the second access represents an instance of shared cache invalidation with respect to the first access.*

Table 6.3 summarizes reuse, no-reuse, and invalidation in all possible read/write access and same/different core scenarios. We describe a few cells in more detail.

$R \rightarrow R$: The first access is a read that may access data at any level in the memory hierarchy. (a) If the subsequent read happens on the same core, it is already captured as a reuse in the intra-thread profiling and thus not a case of shared-cache reuse. (b) If the subsequent read happens on another core of the same socket, it offers an opportunity to exploit the data fetched by one core into the shared cache to be reused by another core (c) Finally, if the subsequent read happens on another socket, there is no reuse opportunity but it does not invalidate the data on the first socket.[1]

$R \rightarrow W$: This is similar to $R \rightarrow R$ except that the second access is a write and the write on a different socket causes an invalidation.

$W \rightarrow W$: The first access is a write. (a) If the subsequent write happens on the same core, it is already captured as a reuse in the intra-thread profiling and hence it is not counted as a case of shared-cache reuse. (b) If a subsequent write happens on another core of the same socket, it is guaranteed to miss in its private cache; as before, it offers an opportunity for data reuse at the shared-cache level. (c) Finally, if the subsequent write happens on another socket, it will invalidate the data on the first socket.

$W \rightarrow R$: This is similar to $W \rightarrow W$ except that the subsequent read on another socket does not cause an invalidation on the first socket. A following R/W on the same socket can create a reuse pair.

[1]We note that a different thread pinning can expose the reuse opportunity but we consider it orthogonal to our current work.

Since we treat *reuses* and *invalidations* as microarchitecture-independent events, we do not consider microarchitectural configurations such as cache inclusion policy, private cache size, or shared cache size in our definition. Therefore, any memory access that we count as a reuse in shared cache does not necessarily hit in a shared cache in an actual machine. Nevertheless, we can still derive metrics such as L2 or L3 cache miss rates from the reuse distance profiles that follow this definition if the microarchitectural configurations of the machines are known. The reuse distance profiles based on this definition can also recommend suitable L3 cache sizes that optimize the applications' runtime.

Duality property: Shared and intra-thread reuses are the dual of one another. A pair of accesses can either be a reuse in a single thread or in the shared cache, but not both. If an access pair in the same core is counted towards intra-thread reuse, it is not counted towards shared-cache reuse. If an access pair in the same socket is not counted towards intra-thread reuse, it is counted towards shared-cache reuse on the same socket. A subsequent remote write operation always invalidates both intra-thread and shared cache reuses. A subsequent remote read operation never causes an invalidation but does not contribute towards reuses neither in a thread nor in a shared cache.

From the hardware perspective, reuse happens at the cache line granularity or sometimes larger than cache line granularities (e.g., Adjacent Cache-Line Prefetch [49]).

6.3 Methodology

In this section, we provide a high-level sketch of our approach to measuring reuse distance in multi-threaded applications. Our tool — ReuseTracker — adopts a sampling philosophy. One need not observe every instance of reuse or invalidation but only a few events in an unbiased manner to produce reuse histograms[2]. We rely on the statistical significance of reuse or invalidation events to guide us in detecting

[2]The same philosophy has been at the core of RDX [119], but it is limited to single-threaded reuse distance measurement.

reuse and invalidation and eventually in computing multi-core reuse distance.

In order to accomplish the sampling-based monitoring, we need two building block components:

1. The ability to sample memory addresses accessed by each thread in the monitored program.

2. The ability to identify whether the immediate next memory access to a sampled address happens on the same thread (reuse) or on a different thread (invalidation if the access is a memory write).

In other words, we need the ability to sample pairs of consecutive accesses to the same memory location, irrespective of which thread performs those accesses.

We achieve (1) by using hardware performance monitoring units. As previously described, when programming in sampling mode, on reaching a threshold number of events i.e. *sampling period*, the PMU can pause the CPU and deliver an interrupt which can include a packet of information including the memory address being accessed by the thread at the time of counter overflow. This sample is treated as a *use* whose *reuse* or *invalidation* is to be detected later.

We achieve (2) by using hardware debug registers to trap execution when a thread accesses a designated memory address. A sampling thread can arm a watchpoint for itself and for other threads in the process so that any thread accessing a designated address will cause a trap; based on whether the same thread traps or a different thread traps (on a different core or a different socket), we can conclude reuse vs. invalidation. For example, if a watchpoint traps in the sampling thread, before it traps in any other thread, an intra-thread reuse is detected and the *time reuse distance* between the *use* and *reuse* is the total loads and stores elapsed on the same thread; if a watchpoint traps in a different thread on the same socket, a shared cache reuse is detected and the *time reuse distance* between the *use* and *reuse* is the total loads and stores elapsed between the two events on all threads on the same socket.

In practice, one may not be able to sample an address and instantaneously convey the sampled address to all other threads so that they can monitor their access to it.

Hence, we mildly relax the definitions as follows.

Relaxed Definition of Reuse. Rather than holding up to the ground truth definition of reuse in Definition 6.2.1 (6.2.3), we relax it to mean that *no intervening store happened on another core (another socket) between two consecutively observed accesses to the same memory location by the same thread.* This relaxation accommodates the possibility of the profiler missing an invalidation event in a short window between when the tool samples an address accessed in one thread and informs all other threads to monitor their own accesses to the same address. Thus, if an instance of reuse is detected in this scheme, there is a high probability that a reuse happened; but we could have missed an invalidation with a small probability.

We do not relax the definition of invalidation because a thread that produces an address sample can be immediately paused before it can execute the next instruction, thanks to the precise performance events in modern PMUs. Hence, any invalidation detected by a sampling-based profiler is a true invalidation event.

6.4 Design and Implementation

ReuseTracker utilizes two slightly different algorithms to profile reuse distance in individual threads and in shared caches. Next, we present the intra-thread profiling algorithm followed by the one for shared caches.

6.4.1 Intra-thread profiling

Figure 6.1 shows the main components of the Intra-thread profiling algorithm and one of the possible execution scenarios. In this algorithm, ReuseTracker samples memory store and load events. It handles PMU samples of these events and watchpoint traps with algorithms shown in Algorithm 2 and 3, respectively. In both algorithms, there is a common global data structure, `GlobalWP`, which is an array of a struct of length equal to the number of available debug registers per core (e.g., four on an x86 machine). The i^{th} element in `GlobalWP` holds the information about the watchpoints currently set in the i^{th} debug register of every thread in the monitored process.

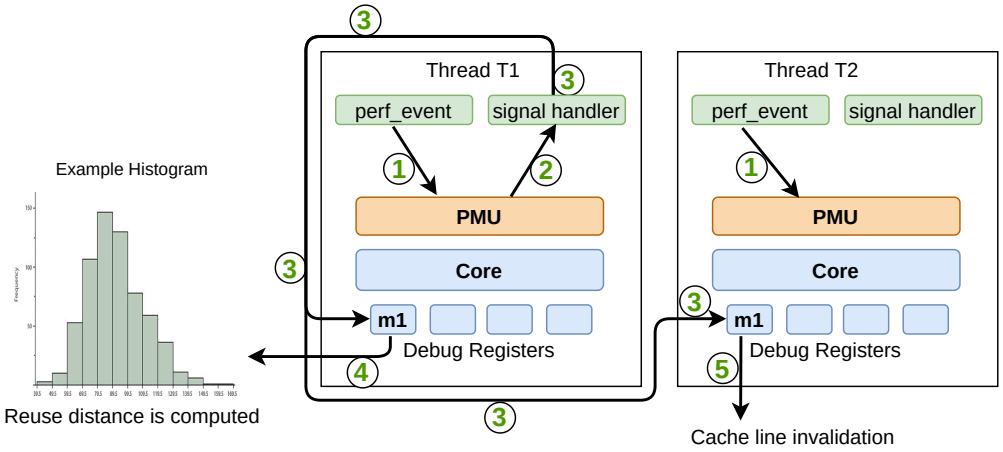


Figure 6.1: One possible execution scenario when profiling intra-thread reuse distance: (1) Every thread sets its PMUs to sample its stores and loads. (2) Thread T_1's PMU counter overflows on a store to address m_1. (3) T_1 arms its watchpoint with type `RW_TRAP` and watchpoints of other threads (e.g., the one in T_2) with type `W_TRAP` and with address m_1 in debug registers. (4) T_1 accesses address m_1 again before any other thread, the watchpoint traps, time reuse distance is computed. (5) Cache line invalidation happens if T_2 stores to address m_1 before T_1 accesses m_1.

On a PMU Sample: On reaching a threshold number of loads or stores an interrupt is delivered to the thread reaching the threshold. Assume M_1 is the address accessed at the time of interrupt by thread, say T_1. This signal-handling function, `PMUSampleHandler` shown in Algorithm 2, takes the sampled address, sampling thread ID, and the total value of load and store PMU counters at the time of sample as inputs. We iterate over all watchpoint slots looking for an available slot (line 4); we look for the unused slots first; the search ends if we find one. If we do not find an unused (inactive) slot, the search continues into in-use (active) slots. The check `AllowReplacement` on line 5 is the reservoir sampling logic, which we explain later. If the slot is in-use, it returns true or false probabilistically. If the slot is inactive, `AllowReplacement` will return true because slot.samples is reset to 1.0 when a watchpoint traps in Line 11 of Algorithm 3. In case of a successful search, the address will be armed on all watchpoints on all cores. However, the search may end without arming any watchpoint, in which case the sampled address will be dropped.

Special attention needs to be placed on the *mode* in which the watchpoints are

set up. A subsequent *load or store* access to the sampled address by *the same thread*, indicates reuse (see Table 6.2); hence, the sampling thread, T_1, arms the watchpoints in its own debug registers in read-write `WP_RW` mode. A subsequent *store* to the sampled address by *another thread*, indicates invalidation; hence, the sampling thread arms the watchpoints in debug registers of other threads in write-only `WP_WRITE` mode.

Algorithm 2 PMU Sample Handler

```
1: global GlobalWP[NUM_DEBUG_REGS]
2:
3: procedure PMUSampleHandler(Address M, ThreadID T, PMUCounterValue P)
4:     for each slot ∈ { GlobalWP } ordered-by inactive to active do
5:         if AllowReplacement(slot) then
6:             slot.active = true
7:             for each T_i ∈ { all threads in the program } do
8:                 if T_i == T then
9:                     mode = WP_RW
10:                else
11:                    mode = WP_WRITE
12:                end if
13:                ArmWatchpoint (M, T_i, P, slot, mode) return
14:            end for
15:        end if
16:    end for
17: end procedure
18:
19: procedure AllowReplacement(WPInfo slot)
20:    r = GenerateRandomNumber(0.0, 1.0)
21:    ret = (r ≤ 1.0 / slot.samples) ? true : false
22:    slot.samples++
23:    return ret
24: end procedure
25:
26: procedure ArmWatchpoint(Address M, ThreadID T, PMUCounterValue P, WPInfo slot, WPMode t)
27:    WP.address = M
28:    WP.PMUValue = P
29:    WP.mode = t
30:    Set a WP in the debug register watchpoint slot of thread T
31: end procedure
```

Algorithm 3 WATCHPOINT TRAP HANDLER

```
1: procedure WPTrapHandler(ThreadID T, PMUCounterValue P, WPInfo slot)
2:
3:     if GlobalWP[slot].active then                ▷ Check if no thread has trapped on this address
4:         if T == GlobalWP[slot].ownerTID then
5:             P_0 = GlobalWP[slot].GetPMUCounterValue()
6:             Record time reuse distance P - P_0                ▷ Reuse distance is recorded
7:         else
8:             Record invalidation                          ▷ WP type must be WP_WRITE
9:         end if
10:        GlobalWP[slot].active = false
11:        GlobalWP[slot].samples = 1.0
12:    end if
13: end procedure
```

On watchpoint trap: When a watchpoint traps in a thread, say T_2, due to a memory access to an address, say M_1, REUSETRACKER handles the trap with an algorithm shown in Algorithm 3. In Line 3, T_2 checks if the watchpoint *slot* that corresponds to the trapping debug register is still *active* (meaning never trapped in any threads). If it is still active, T_2 checks if the thread ID of the watchpoint *slot* matches its own thread ID. If it matches, time reuse distance is computed as *use* and *reuse* occur on the same thread. In this case, the time reuse distance is the number of loads and stores elapsed from the *use* to the *reuse* point, which is read from PMU counters. If the two accesses are from two different threads (the latter one guaranteed to be a write access), an invalidation is detected. After detecting either a *reuse* or an invalidation, in Line 10, T_2 marks the watchpoint *slot* as *inactive*; subsequent traps, if any, on the same slot will be ignored so that at most one *reuse* or invalidation is detected per sampled address.

After detecting reuse or invalidation, the watchpoint slot is immediately released for use by a subsequent sample. As a special case, if every armed watchpoint traps before the next PMU address sample, the reservoir sampling ensures all sampled addresses to be monitored.

Reservoir Sampling Strategy: On a PMU sample, a previously armed watchpoint may have already trapped (either because of reuse or invalidation) or not yet trapped. REUSETRACKER needs to balance between retaining a previously armed

watchpoint to potentially detect a reuse separated by some other memory accesses and a new address that may trap sooner; neither of these is predictable without time traveling into the future of execution.

We employ reservoir sampling to give equal probability to retain an already monitored address for a longer time vs. beginning to monitor a new address, disregarding when the sample happens. This gives a fair opportunity to short vs. long distance reuses. The probability of an older, already armed watchpoint to be replaced with a new sampled address decays following the harmonic progression [123] over time with each new PMU sample; meaning, the longer a watchpoint stays armed, the higher the probability for it to stay even longer. However, a newly sampled address always has a probability to replace an older sample in such a way that any sampled address has equal probability to be observed irrespective of when the address was sampled. AllowReplacement on Line 19 in Algorithm 1 probabilistically answers this question, whether the sampling thread is allowed to arm watchpoint in a slot in all threads including replacing any old existing un-trapped watchpoints. We refer the reader to [116] for theoretical guarantees of reservoir sampling [122, 119].

6.4.2 Shared cache profiling

To detect *reuses* in the shared cache, ReuseTracker again samples load and store accesses, and employs debug registers to trap accesses to the sampled addresses that happen in the same shared cache.

In this algorithm, when a thread samples a load or a store access, this sampled event is considered as a *use*. After extracting the effective address of the event, the sampling thread arms watchpoints on the address in all threads. Watchpoints that trap load or store accesses are created in cores that share the same socket (and hence the same shared cache) with the sampling thread. In cores that do not share the same socket, watchpoints that trap only store accesses are set up. When the next trap happens in a watchpoint in a core on the same socket, a *reuse* in the shared cache is detected, otherwise an invalidation is detected if the trap happens in a core on another socket.

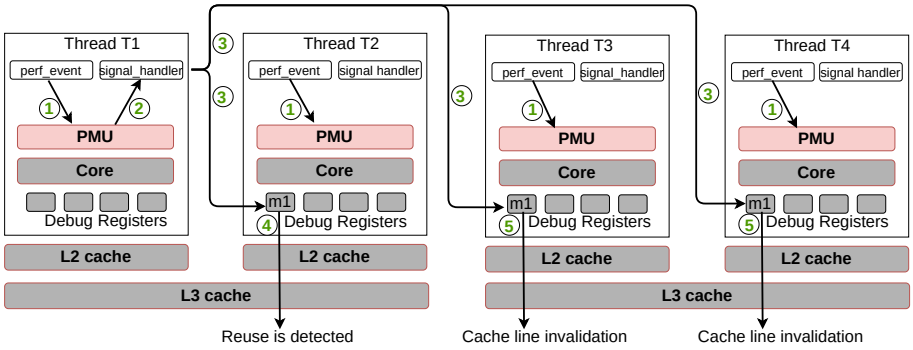


Figure 6.2: One possible execution scenario in profiling reuse distance in L3 cache: (1) Every thread sets its PMUs to sample its load and store accesses. (2) Thread T_1's PMU counter overflows on a store or a load on address m_1. (3) T_1 arms the watchpoints on other cores that share the same L3 cache with itself with type `RW_TRAP` and with type `W_TRAP` on cores that do not share the same L3 cache. (4) T_2 accesses address m_1 again before any other thread, the debug register traps, time reuse distance in L3 is computed. (5) Cache line invalidation in L3 level occurs if T_3 or T_4 stores to address m_1 before T_2 accesses m_1.

Figure 6.2 shows the main components and one possible execution scenario of the SHARED CACHE PROFILING algorithm. Next, we explain the steps how REUSETRACKER handles a PMU sample when profiling reuse distance in shared cache.

On a PMU sample: When a thread, say T_1, samples a load or a store access to an address, say M_1, a signal is delivered by the Linux kernel to T_1. This signal is handled by our signal-handling function, which is very similar to the intra-thread profiling algorithm except for few differences. If T_1 is allowed to arm watchpoints globally, then T_1 collects the sum of the PMU counter values of load and store events from all cores that share the same socket as the sampled core. T_1 sets the watchpoint types and arms watchpoints that trap load or store accesses in all of the other threads that share the same socket with itself. The aim of these watchpoints is to detect a reuse in the shared cache. To detect cache line invalidation from the shared cache, T_1 also arms watchpoints that trap only store accesses in cores of other sockets.

On watchpoint trap: When a watchpoint trap happens in a thread, say T_2,

due to an access to an address, say M_1, the trap is handled by ReuseTracker. Firstly, T_2 checks if the watchpoint slot is still active to see if no trap has occurred on this slot globally. On an active slot, T_2 checks whether it shares the same socket with the thread that arms its watchpoint. If so, a shared cache *reuse* is detected and time reuse distance between the trap in T_2 and the sample owner is computed. However, if T_2 and the sample owner are on different sockets, an invalidation is detected. After detecting a *reuse* or an invalidation, the reservoir sampling probability is reset, and the watchpoint slot is marked as inactive.

Measuring Time Reuse Distance in Shared Cache Profiling: To measure time reuse distance when a *reuse* in a shared cache is detected, we leverage the PMU counter for loads and stores. The time reuse distance is measured as follows.

1. At the sampling point when the watchpoints are armed, the PMU sample handler function memorizes the total count $C_1 = \sum_{i=1}^{ncores\ on\ skt} Loads(i) + Stores(i)$ of load and store events from all cores that share the same socket where the sample occurs. C_1 becomes an attribute in each watchpoint that can be accessed by the watchpoint trap handler.

2. When a *reuse* is detected, and the trap handler records the total count, say $C_2 = \sum_{i=1}^{ncores\ on\ skt} Loads(i) + Stores(i)$ of loads and stores again from all cores sharing the same socket again. $C_2 - C_1$ is recorded as time reuse distance.

6.4.3 Implementation

ReuseTracker is built on top of the open-source HPCToolkit tools suite [5]. It intercepts thread creation and termination using LD_PRELOAD for dynamically-linked executables [87]. On each thread creation, we configure the Linux `perf_events` for the newly created thread to monitor relevant PMU events.

To profile reuse distance in individual threads and in shared caches, we sample MEM_UOPS_RETIRED:ALL_STORES and MEM_UOPS_RETIRED:ALL_LOADS events, respectively for sampling stores and loads. All of these events allow the signal-handling function to record the accessed effective address and the

program counter when an event is sampled. By recording effective address, REUSETRACKER's signal-handling function is able to detect a memory access *use* and its *reuse*.

In addition to recording effective address and calculating time reuse distance, the signal-handling and the watchpoint trap-handling functions also record program counters at the moments of PMU samples and watchpoint traps. Using the program counters, the profiling functions can trace the sampled and trapped threads' call stacks through an online binary analysis. This online binary analysis is performed by HPCToolkit every time a PMU sample or a watchpoint trap is handled to retrieve the procedure frames that are parts of the call stack that becomes the execution context of the detected PMU sample or the watchpoint trap. After tracing the call stack of a sampled or a trapped memory access, the detected *use* or *reuse* can be attributed to the call stack, which makes it easier for programmers to spot the locations of each detected use-reuse pairs in the source code.

Concurrency Control: `GlobalWP` is a shared data structure and accessed concurrently by different PMU sample handlers and watchpoint traps. We use a two-counter-based transactional memory mechanism proposed by Lamport [63] that allows multiple readers and a single writer to concurrently access the same slot of `GlobalWP`; accessing different slots of `GlobalWP` by different threads is obviously conflict free. We ensure mutual exclusion among multiple writers to the same slot via a test-and-test-and-set lock [99].

Workaround for Limited `GlobalWP` Slot Count: We acknowledge the possibility of having much higher thread count than there are `GlobalWP` slots. If profiled threads exhibit uniform behavior, a low number of `GlobalWP` slots, which is 4 in x86 architecture, is sufficient. In case there are more than 4 different behaviors at the same time, REUSETRACKER might still capture all of these behaviors in one run as multiple threads can context switch to be monitored by a `GlobalWP` slot. However, in an extreme condition where the thread count in an application is much higher than the number of `GlobalWP` slots and the behaviors of the threads are very diverse to the point that each thread might have its own distinct behavior, there is a

possibility that ReuseTracker might not capture all of these behaviors as it can only monitor 4 behaviors at the same time. To work around this limitation, a user of our tool might have to profile the application a number of times to get a more complete insight into the reuse behavior of the application.

6.5 Evaluation

This section evaluates the accuracy of ReuseTracker and demonstrates its functionality in analyzing reuse distances in PARSEC, Rodinia, and Synchrobench benchmarks. Additionally, this section also evaluates the performance and memory overheads of ReuseTracker.

The experimental study is carried out in a 2-socket Intel Xeon Gold 6148 Skylake CPU and a 2-socket AMD EPYC 7352 Zen 2 (17h) CPU. In the Intel machine, each socket has twenty cores, each core has its own L1d, L1i, and L2 caches, and each socket has one shared L3 cache. The machine runs `Linux 5.5.2 kernel`, and we use `gcc 8.3.1` compiler. In the AMD machine, there are 24 cores per socket. Each core has its local L1i, L1d, and L2 caches, and shares L3 cache with other cores in the same socket. The AMD machine runs `Linux 5.11.0-36` and `gcc-10.3.0` compiler.

Unless otherwise stated, the default sampling interval for both load and store events in the Intel machine is 100K, the default sampling interval for executed micro-operations in the AMD machine is 50K, and the used debug register count in each core is 4. In each experiment, the threads are distributed evenly across the two sockets, and the threads in each socket are bound to CPU cores with *compact* mapping[3] strategy by default.

[3]Compact mapping assigns the thread $t+1$ to a free thread context as close as possible to the thread context where the thread t was placed.

6.5.1 RIBench *Benchmark*

To evaluate the accuracy of ReuseTracker, we develop a synthetic benchmark with controllable reuse and invalidation counts shown in Listing 6.1. We refer to this benchmark as *RIBench*. Using the *RIBench*, one can configure the amount of reuses for different reuse distances as well as the amount of invalidations that happen during execution.

```
#pragma omp parallel shared(shared_array) \
 private(private_array1, private_array2, private_array3, \
 private_array4, private_array5)
{
for(int i = 0; i < outer; i++)
    for(int j = 0; j < a; j++)
        for(int k = 0; k < inv; k++)
            Store to shared_array[k]
        for(int k = 0; k < a1; k = k+2)
            Load from private_array1[k]
            Store to private_array1[k+1]
    for(int j = 0; j < b; j++)
        for(int k = 0; k < inv; k++)
            Store to shared_array[k];
        for(int k = 0; k < b1; k = k+2)
            Load from private_array2[k]
            Store to private_array2[k+1]
    for(int j = 0; j < c; j++)
        for(int k = 0; k < inv; k++)
            Store to shared_array[k]
        for(int k = 0; k < c1; k = k+2)
            Load from private_array3[k]
            Store to private_array3[k+1]
    for(int j = 0; j < d; j++)
        for(int k = 0; k < inv; k++)
            Store to shared_array[k];
        for(int k = 0; k < d1; k = k+2)
            Load from private_array4[k]
            Store to private_array4[k+1]
    for(int j = 0; j < e; j++)
        for(int k = 0; k < inv; k++)
            Store to shared_array[k]
        for(int k = 0; k < e1; k = k+2)
            Load from private_array5[k]
            Store to private_array5[k+1]
```

```
}
```

Listing 6.1: Pseudo-code for Reuse-Invalidation Benchmark, *RIBench*

In the *RIBench*, there are several configurable parameters; *outer*, a, a_1, b, b_1, c, c_1, d, d_1, e, e_1, and inv. These parameters can be configured to generate expected reuse distances and number of reuses for each reuse distance. There are five different reuse distances that can be generated. These reuse distances are a_1+inv, b_1+inv, c_1+inv, d_1+inv, and e_1+inv. When we want to configure each thread to have five different non-zero reuse distances without being interrupted by any invalidation, we set up the values of a_1, b_1, c_1, d_1, and e_1 to be five different non-zero values as these five variables determine the number of iterations to access five different private arrays, while the value of inv variable is set to be zero. Then, in case we want to introduce cache line invalidations, we set the value of inv variable to be a non-zero value since this variable determines the number of iterations that access a shared array. The other parameters, i.e. *outer*, a, b, c, d, and e, can be configured to determine the reuse count of each reuse distance.

The computation of expected reuse count for each reuse distance is displayed in Table 6.4. By changing the parameters, it is possible to produce a variety of reuse distance histogram patterns. For example, if $e_1 > d_1 > c_1 > b_1 > a_1$, $inv = 0$, and $e * e_1 < d * d_1 < c * c_1 < b * b_1 < a * a_1$, the reuse distance histogram will exhibit a decreasing pattern. Another example is when $e_1 > d_1 > c_1 > b_1 > a_1$, $inv = 0$, and $e * e_1 > d * d_1 > c * c_1 > b * b_1 > a * a_1$, then the reuse distance histogram will form an increasing pattern.

Reuse Distance	**Reuse Count**
inv+a1	*num_threads*outer*a*a1*
inv+b1	*num_threads*outer*b*b1*
inv+c1	*num_threads*outer*c*c1*
inv+d1	*num_threads*outer*d*d1*
inv+e1	*num_threads*outer*e*e1*

Table 6.4: Reuse distance and reuse count of *RIBench*

In addition to controlling the shapes of patterns in generated reuse distance histograms, it is also possible to control the amount of cache line invalidations by setting up the value of *inv* parameter when there are more than one thread. This parameter determines the iteration count of a loop that performs store access to a shared array `shared_array`. The number of cache line invalidations can be increased by increasing the value of *inv* variable. Therefore, say, for reuse distance $inv+a1$, instead of having reuse count $num\_threads*outer*a*(a1+inv)$, its reuse count becomes $num\_threads*outer*a*a1$, as each access to `shared_array` is expected to lead to a cache line invalidation instead of a reuse.

To evaluate the accuracy of REUSETRACKER, we compute the expected reuse distance histogram of the *RIBench* with a given set of parameters, and this histogram is compared with the histogram produced by REUSETRACKER. Given two histograms H and $\hat{H}$, the accuracy, S, is calculated as below [106][119]:

$$S = 1 - \frac{\sum_{i=1}^{n} \left| B_i - \hat{B}_i \right|}{2}$$

B_i and $\hat{B}_i$ are fractions of reuse-pairs in H and $\hat{H}$, respectively, that fall into the i^{th} bin, and n is the number of bins in each histogram. The value of S metric is in the range $[0, 1.0]$ with 1.0 meaning H and $\hat{H}$ are perfectly similar.

6.5.2 Accuracy without Invalidation

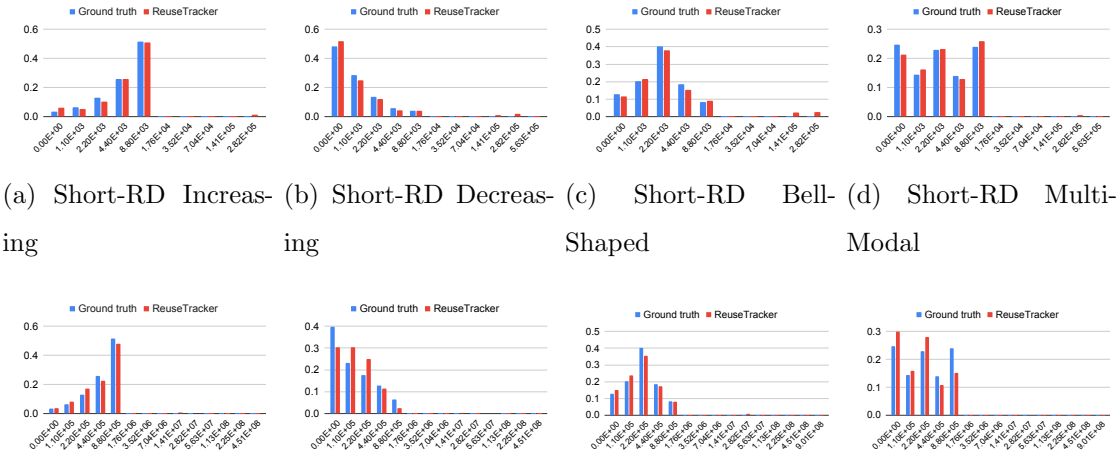
The simplest case is that of intra-thread reuse distance when there is no invalidation (this is analogous to single-threaded reuse distance). For this base case, we configured the parameters of the *RIBench* to generate four different reuse distance patterns, *increasing*, *decreasing*, *bell-shaped*, and *multi-modal*. We also created two cases for each pattern, short reuse distance (*short-RD*) and long reuse distance (*long-RD*) cases. In the *short-RD* cases, the reuse distance is lower than the sampling period, while in the *long-RD* cases, the reuse distance is higher than or equal to the sampling period. Therefore, in total there are eight different test cases. Table 6.5 shows the values of the parameters in each test case. As can be seen in the table, $a1$, $b1$, $c1$, $d1$, and $e1$ that determine the reuse distances in each thread are always less than the sampling interval, which is 100K, for the *short-RD* cases, while they are always equal to or higher than the sampling interval for the *long-RD* cases.

Test Case	**Parameter Values**
Short-RD Increasing	outer = 10, a = 20, a1 = 1000, b = 20, b1 = 2000, c = 20, c1 = 4000, d = 20, d1 = 8000, e = 20, e1 = 16000, inv = 0
Short-RD Decreasing	outer = 10, a = 200, a1 = 1000, b = 60, b1 = 2000, c = 15, c1 = 4000, d = 4, d1 = 8000, e = 2, e1 = 16000, inv = 0
Short-RD Bell-Shaped	outer = 10, a = 50, a1 = 1000, b = 40, b1 = 2000, c = 40, c1 = 4000, d = 10, d1 = 8000, e = 3, e1 = 16000, inv = 0
Short-RD Multi-Modal	outer = 10, a = 100, a1 = 1000, b = 30, b1 = 2000, c = 24, c1 = 4000, d = 8, d1 = 8000, e = 7, e1 = 16000, inv = 0
Long-RD Increasing	outer = 10, a = 10, a1 = 100000, b = 10, b1 = 200000, c = 10, c1 = 400000, d = 10, d1 = 800000, e = 10, e1 = 1600000, inv = 0
Long-RD Decreasing	outer = 10, a = 100, a1 = 100000, b = 30, b1 = 200000, c = 12, c1 = 400000, d = 5, d1 = 800000, e = 2, e1 = 1600000, inv = 0
Long-RD Bell-Shaped	outer = 10, a = 50, a1 = 100000, b = 40, b1 = 200000, c = 40, c1 = 400000, d = 10, d1 = 800000, e = 3, e1 = 1600000, inv = 0
Long-RD Multi-Modal	outer = 10, a = 100, a1 = 100000, b = 30, b1 = 200000, c = 24, c1 = 400000, d = 8, d1 = 800000, e = 7, e1 = 1600000, inv = 0

Table 6.5: Parameter values of *RIBench* when assessing accuracy without cache line invalidation

The results produced by ReuseTracker running these test cases with 32 threads are presented in Figures 6.3 and 6.4. As can be seen in the figures, the

reuse distance histograms generated by REUSETRACKER are close to the ground truth. In the Intel machine, the average accuracy for short reuse distance cases is 94%, and the accuracy for long reuse distance cases is 90%, while in the AMD machine, the average accuracies are 95.4% and 94.7% respectively. In short reuse distance cases, all reuses can be trapped without having to rely on reservoir sampling. Thus, REUSETRACKER exhibits better accuracy in these cases than in the long reuse distance ones.



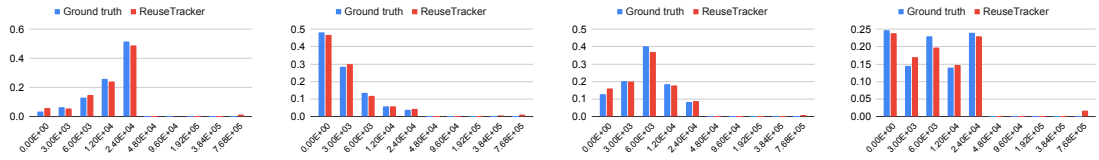
(a) Short-RD Increasing (b) Short-RD Decreasing (c) Short-RD Bell-Shaped (d) Short-RD Multi-Modal

(e) Long-RD Increasing (f) Long-RD Decreasing (g) Long-RD Bell-Shaped (h) Long-RD Multi-Modal

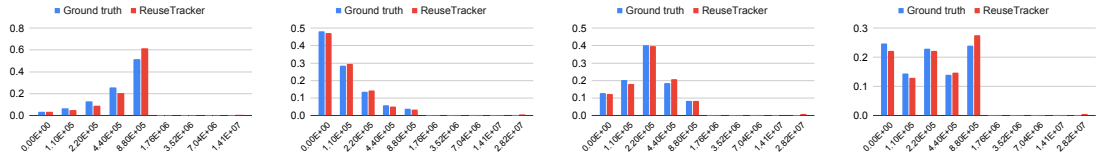
Figure 6.3: Reuse distance histograms of *RIBench* without cache line invalidation in the Intel machine. X-axis shows the reuse distance ranges in logarithm-scale. Y-axis displays the fraction of reuse-pairs that belong to specific reuse distance ranges.

Among the long reuse distance cases running in the Intel machine, we can see that the *Long-RD Decreasing* (Figure 6.3f) and the *Long-RD Multi-Modal* (Figure 6.3h) cases have the lowest accuracies. These results can be attributed to the probabilistic nature of reservoir sampling and the fact that the *Long-RD Decreasing* and the *Long-RD Multi-Modal* cases have fewer sample counts taken from the loops in lines 18-35 of Listing 6.1, which access large arrays, than the other two *Long-RD* cases.

s



(a) Short-RD Increasing (b) Short-RD Decreasing (c) Short-RD Bell-Shaped (d) Short-RD Multi-Modal



(e) Long-RD Increasing (f) Long-RD Decreasing (g) Long-RD Bell-Shaped (h) Long-RD Multi-Modal

Figure 6.4: Reuse distance histograms of *RIBench* without cache line invalidation in the AMD machine.

6.5.3 Accuracy with Invalidation

Next, we introduce invalidations in the setup and assess the intra-thread reuse profiling algorithm with cache line invalidations. This setup has twelve test cases that were derived from the *Bell-Shaped* and *Decreasing* configurations given in Table 6.5. Each configuration was modified to generate three different reuse distance histogram patterns that have different amount of cache line invalidations. Table 6.6 displays the modified configurations for the *Bell-Shaped* cases, while Table 6.7 shows the modified configurations for the *Decreasing* cases. Among these configurations, *Case 3* leads to more cache line invalidations than *Case 2*, which leads to more cache line invalidations than *Case 1*. The reason for this is that there are more loop iterations that perform store accesses to `shared_array` in *Case 3* than in *Case 2* and there are more store accesses to the shared array in *Case 2* than in *Case 1*. These store accesses to the shared array is also the main difference between this experiment and the experiment without invalidation in Section 6.5.2. In the experiment without invalidation, the parameter *inv*, which defines number of store accesses to the shared array, is always zero.

Test Case	**Parameter Values**
Short-RD Bell-Shaped Case 1	outer = 10, a = 50, a1 = 1000, b = 40, b1 = 2000, c = 40, c1 = 4000, d = 10, d1 = 8000, e = 3, e1 = 16000, inv = 0
Short-RD Bell-Shaped Case 2	outer = 10, a = 50, a1 = 500, b = 40, b1 = 1500, c = 40, c1 = 3500, d = 10, d1 = 7500, e = 3, e1 = 15500, inv = 1000
Short-RD Bell-Shaped Case 3	outer = 10, a = 50, a1 = 0, b = 40, b1 = 1000, c = 40, c1 = 3000, d = 10, d1 = 7000, e = 3, e1 = 15000, inv = 2000
Long-RD Bell-Shaped Case 1	outer = 10, a = 50, a1 = 100000, b = 40, b1 = 200000, c = 40, c1 = 400000, d = 10, d1 = 800000, e = 3, e1 = 1600000, inv = 0
Long-RD Bell-Shaped Case 2	outer = 10, a = 50, a1 = 50000, b = 40, b1 = 150000, c = 40, c1 = 350000, d = 10, d1 = 750000, e = 3, e1 = 1550000, inv = 100000
Long-RD Bell-Shaped Case 3	outer = 10, a = 50, a1 = 0, b = 40, b1 = 100000, c = 40, c1 = 300000, d = 10, d1 = 700000, e = 3, e1 = 1500000, inv = 200000

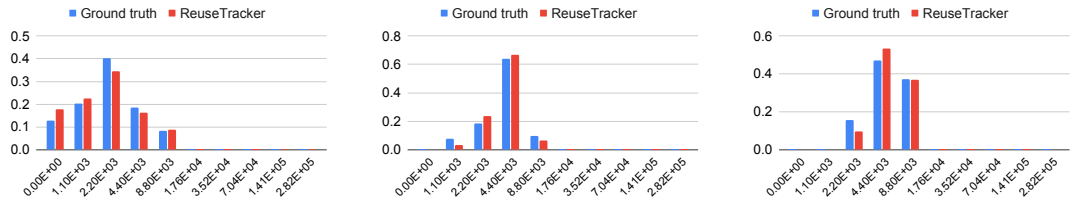
Table 6.6: Parameter values of *RIBench* when assessing accuracy with cache line invalidations on *bell-shaped* pattern

Test Case	**Parameter Values**
Short-RD Decreasing Case 1	outer = 10, a = 200, a1 = 1000, b = 60, b1 = 2000, c = 15, c1 = 4000, d = 4, d1 = 8000, e = 2, e1 = 16000, inv = 0
Short-RD Decreasing Case 2	outer = 10, a = 200, a1 = 500, b = 60, b1 = 1500, c = 15, c1 = 3500, d = 4, d1 = 7500, e = 2, e1 = 15500, inv = 1000
Short-RD Decreasing Case 3	outer = 10, a = 200, a1 = 0, b = 60, b1 = 1000, c = 15, c1 = 3000, d = 4, d1 = 7000, e = 2, e1 = 15000, inv = 2000
Long-RD Decreasing Case 1	outer = 10, a = 200, a1 = 100000, b = 60, b1 = 200000, c = 15, c1 = 400000, d = 4, d1 = 800000, e = 2, e1 = 1600000, inv = 0
Long-RD Decreasing Case 2	outer = 10, a = 200, a1 = 50000, b = 60, b1 = 150000, c = 15, c1 = 350000, d = 4, d1 = 750000, e = 2, e1 = 1550000, inv = 100000
Long-RD Decreasing Case 3	outer = 10, a = 200, a1 = 0, b = 60, b1 = 100000, c = 15, c1 = 300000, d = 4, d1 = 700000, e = 2, e1 = 1500000, inv = 200000

Table 6.7: Parameter values of *RIBench* when assessing accuracy with cache line invalidations on *decreasing* pattern

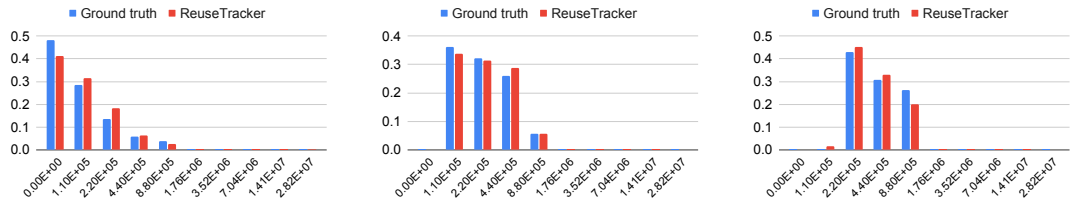
Figures 6.5, 6.6, 6.7, and 6.8 show the reuse distance histograms generated by REUSETRACKER running on 32 threads compared to the ground truth. As shown in both figures, our results are close to the ground truth with an average accuracy 89% over short reuse distance cases, and 93% over long reuse distance cases. The lower accuracy in short reuse distance cases is the result of our relaxed definition of reuse. Within a short interval between when a thread samples a memory access and when the thread arms watchpoints in all threads, a cache line invalidation to the sampled address might occur undetected. If the interval between the watchpoint

arming and the *reuse* of the sampled address is very short, which is the case for the short reuse distance cases, the undetected invalidation that happens between the sample and the watchpoint arming might be the only opportunity to detect cache line invalidation, which is missed. As a result, a false detection of reuse will occur, which still satisfies our relaxed definition of reuse but violates the ground truth definition in Definition 6.2.1.



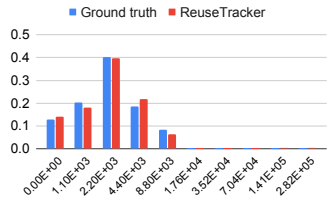
(a) Long-RD Bell-Shaped Case 1 (b) Long-RD Bell-Shaped Case 2 (c) Long-RD Bell-Shaped Case 3

Figure 6.5: Reuse distance histograms of *RIBench* with cache line invalidations on *bell-shaped* pattern in the Intel machine.

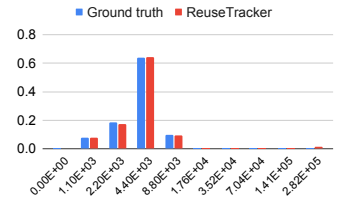


(a) Long-RD Decreasing Case 1 (b) Long-RD Decreasing Case 2 (c) Long-RD Decreasing Case 3

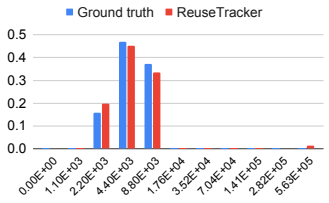
Figure 6.6: Reuse distance histograms of *RIBench* with cache line invalidations on *decreasing* pattern in the Intel machine.



(a) Long-RD Bell-Shaped Case 1

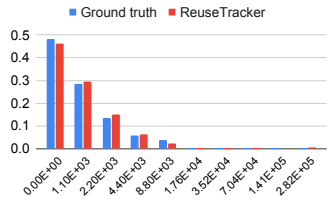


(b) Long-RD Bell-Shaped Case 2

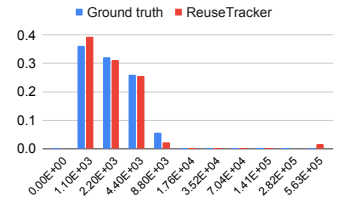


(c) Long-RD Bell-Shaped Case 3

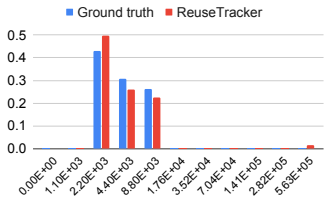
Figure 6.7: Reuse distance histograms of *RIBench* with cache line invalidations on *bell-shaped* pattern in the AMD machine.



(a) Long-RD Decreasing Case 1



(b) Long-RD Decreasing Case 2



(c) Long-RD Decreasing Case 3

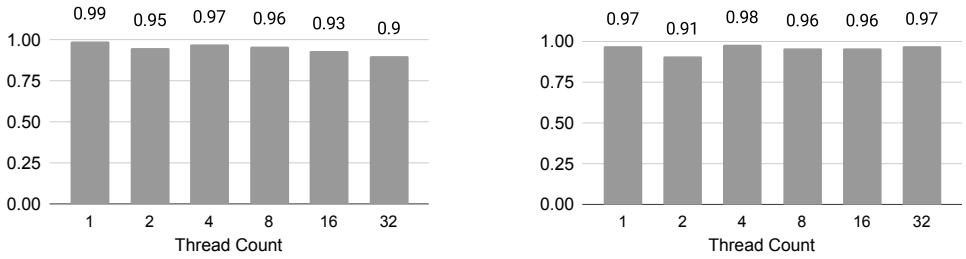
Figure 6.8: Reuse distance histograms of *RIBench* with cache line invalidations on *decreasing* pattern in the AMD machine.

One pattern that can be observed from the ground truths in Figures 6.5, 6.6, 6.7, and 6.8 is that the offsets of the histograms shift to the right as we go from *Case 1* to *3*. The reason for the offset shift is the sum of *inv+a1*, which is the shortest reuse distance in all cases, that keeps increasing from *Case 1* to *2* and from *Case 2* to *3*. Another pattern that can be observed from the ground truths and the results is that the histograms become narrower as we go from *Case 1* to *3*. This is the result of the logarithmic scaling of the bin sizes whose range increases along the X-axis. Therefore, as the histogram shifts to the right due to increases in reuse distances, the capacity of each bin becomes larger, and hence the number of filled bins becomes smaller. The increase in all reuse distances and the logarithmic scaling are also the reasons for the mode shift of the histograms in Figures 6.5 and 6.7 as the mode that

is in the third bin in *Case 1* becomes "pushed" to the fourth bin in *Case 2* and grouped together with the reuses that are already in the fourth bin in *Case 1*. Such transformation also happens from *Case 2* to *Case 3* as some reuses that belong to the fourth bin in *Case 2* are "pushed" to the fifth bin in *Case 3* and makes the height of the fifth bin taller.

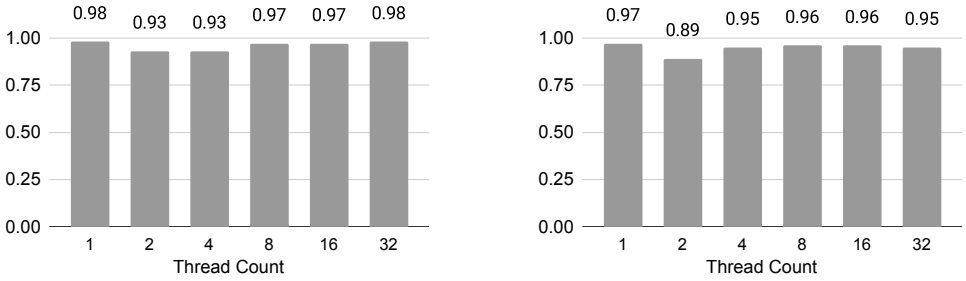
6.5.4 Accuracy under Different Thread Counts

To evaluate ReuseTracker's accuracy under different thread counts, we ran it on the *RIBench* with *Long-RD Bell-Shaped Case 2* and *Long-RD Decreasing Case 2* configurations. We chose these configurations as they include invalidations and reservoir sampling in their executions and, among all *Long RD* cases that have invalidations, they have the most complex patterns since they have more filled bins and more variation of tall and short bins in their patterns. We performed the experiment under six different thread counts; 1, 2, 4, 8, 16, and 32. The accuracy results are presented in Figures 6.9 and 6.10. The accuracy of ReuseTracker is consistently high under different thread counts with an average of 96% in the Intel machine and an average of 95% in the AMD machine.



(a) Long-RD Bell-Shaped Case 2 (b) Long-RD Decreasing Case 2

Figure 6.9: Accuracy of ReuseTracker running the *RIBench* under different thread counts in the Intel machine. X-axis displays the thread counts, and Y-axis shows the accuracy for each thread count.



(a) Long-RD Bell-Shaped Case 2

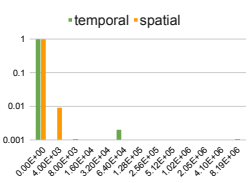
(b) Long-RD Decreasing Case 2

Figure 6.10: Accuracy of REUSETRACKER running the *RIBench* under different thread counts in the AMD machine.

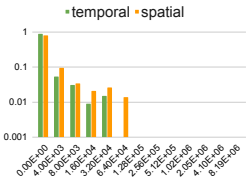
6.5.5 Reuse Distances of PARSEC Benchmarks

In this section, we present and discuss the reuse distance histograms for the PARSEC benchmark suite [18]. For the sake of brevity, we will only discuss four of the benchmarks in detail.

Figure 6.11 shows the histograms of *blackscholes* and *bodytrack* generated from intra-thread reuse distance profiling for 32 threads processing *native* input size. Figure 6.11 shows that most reuses in the selected benchmarks are short in distance. *blackscholes*, which computes prices of a portfolio of European options, has a huge portion of short-distanced reuses. These short-distanced reuses mostly occur on the *prices* array, a data structure that records options' prices. Reuses happen in the *bs_thread*, the *BlkSchlsEqEuroNoDiv*, and the *CNDF* functions. Longer-distanced reuses are detected in the *main* function on accesses to local variables which become memory accesses due to register spilling.

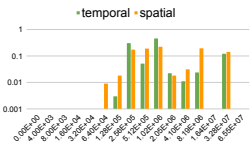


(a) blackscholes

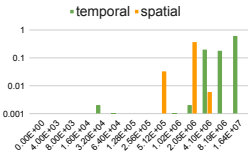


(b) bodytrack

Figure 6.11: Histograms of intra-thread reuse distance of *blackscholes* and *bodytrack* from PARSEC. X-axis shows the reuse distance ranges in logarithm-scale. Y-axis displays in logarithm-scale the fraction of reuse-pairs that belong to specific reuse distance ranges.



(a) streamcluster



(b) freqmine

Figure 6.12: Histograms of reuse distance in L3 cache of *streamcluster* and *freqmine* from PARSEC. X-axis shows the reuse distance ranges in logarithm-scale. Y-axis displays in linear scale the fraction of reuse-pairs that belong to specific reuse distance ranges.

In *bodytrack*, a lot of reuses of various distances happen within the *TrackingModel::LogLikelihood* function and in other functions called by it. *TrackingModel::LogLikelihood* is a function that computes the likelihood of each observed particle in tracking an object in an image. This computation is needed in resampling particles with the best fitness values from image data to help analyze interesting regions in the image in more detail.

Figure 6.12 displays the histograms of *streamcluster* and *freqmine* generated from shared cache reuse distance profiling. *streamcluster* is a benchmark that solves an online clustering problem by grouping streamed data points into their nearest centers. In *streamcluster*, some of the detected temporal and spatial reuses happen with distances less than or equal to 8M. Most of these use-reuse pairs happen on *Point*-typed shared array elements in the *pgain*, *dist*, and *shuffle* functions. Aside from accesses to *Point*-typed data structures, there are also detected short-distanced reuses due to locking mechanism in the *pthread_barrier_wait* function. For reuses that are longer than 8M in distance, they also occur because of accesses to shared *Point* data structures in the *dist* function. These reuses can be long in distance because *dist* is a function that measures distances between data points when making new

cluster centers and consecutive accesses to the same data points can happen across different *dist* function calls separated by wide time gaps.

freqmine is a data mining benchmark that performs the Frequent Pattern-growth (FP-growth) method for frequent itemset mining (FIMI) problem. In this benchmark, reuses of various distances are detected between the *FP_tree::database_tiling* and *FP_tree::scan2_DB* functions due to memory accesses to a shared tree data structure. Furthermore, long-distanced reuses are also detected in the *FP_tree::database_tiling* function on accesses to the *item_order* array. These long-distanced reuses occur due to accesses to the shared array in creating a prefix tree whose branches represent frequent itemsets.

6.5.6 Use Case: False Sharing Removal

Spatial reuses in shared cache detected by ReuseTracker include false sharing as the use-reuse pair belong to different threads that access different memory regions located in the same cache line. Therefore shared cache spatial reuse distance profiles produced by ReuseTracker can assist in guiding code refactoring that reduces false sharing in the profiled code. To demonstrate this capability, we leverage ReuseTracker to profile some benchmarks from Synchrobench[44], namely *ESTM-rbtree*, *MUTEX-hashtable*, *MUTEX-lazy-list*, *SPIN-hashtable*, and *SPIN-lazy-list*, and use the generated profiles to guide code modifications that improve their performances. To detect only false sharing, we generate a shared cache reuse distance profile that includes only read-after-write (RAW) and write-after-write (WAW) use-reuse pairs. The reason for this is these kinds of use-reuse pairs trigger inter-core cache line transfers caused by false sharing for certain.

In *ESTM-rbtree*, we identify false sharing in the `TMlookup` function that involves `struct node` and `struct thread_data` data types. To remove the false sharing, we inserted padding manually into `struct node` and `struct thread_data`. After this modification, *ESTM-rbtree*'s performance is improved by 87%. In *MUTEX-hashtable*, *MUTEX-lazy-list*, *SPIN-hashtable*, and *SPIN-lazy-list*, false sharing is found in the `parse_delete` function or in other functions called from it. In all

of these benchmarks, the detected false sharing involves `struct node_l` data type. Paddings in this data structure improves the performance of these benchmarks by 6%, 46%, 39% and 58% for *MUTEX-hashtable*, *MUTEX-lazy-list*, *SPIN-hashtable*, and *SPIN-lazy-list*, respectively.

6.5.7 Use Case: Adjacent Cache Line Prefetch

By analyzing the intra-thread and shared cache reuse distance histograms of a multi-threaded code, we demonstrate that it is possible to decide whether the code will benefit from adjacent cache line prefetch (ACP) [49] or not. ACP, a feature in Intel microarchitectures, allows prefetching a cache line that is adjacent to the currently accessed cache line. This feature can improve an application performance if the application threads have a good spatial locality where each thread actually accesses the prefetched cache lines before they are evicted. In addition to locality in individual threads, another factor that affects application performance when using ACP is inter-thread communication. Excessive communication, such as false sharing, may hinder the benefit of spatial locality that ACP offers. Due to communication, prefetched cache lines could be invalidated before they are accessed by the prefetching threads. To account for locality-affected access latencies and inter-thread communication, we build a model to predict whether an application can benefit from ACP or not.

We consider an application to have better performance with ACP if its performance overhead due to main memory access is higher than its overhead due to false sharing. Let G be a binary predictor metric, where

$$G = T_m - T_{fs},$$

such that T_m is the total latency of accesses to main memory, and T_{fs} is the total false sharing latency. T_m in this model is computed as follow.

$$T_m = R_{dram} * L_{dram},$$

where R_{dram} is the number of detected spatial reuses in the intra-thread reuse distance histogram that access DRAM, and L_{dram} is the latency of access to DRAM. In the model, T_{fs} is calculated as follow.

$$T_{fs} = R_{fs} * L_{fs},$$

where R_{fs} is the number of detected spatial reuses in the shared cache reuse distance histogram with reuse distances that are short enough to be in the range of false sharing communication, and L_{fs} is the latency of inter-core cache line transfer. We obtained the values of L_{dram} and L_{fs} by running Intel®Memory Latency Checker (Intel MLC)[115] on our machine.

Benchmarks	**Predictor Metric**	**Actual Speedup**
bodytrack	$-1.1 * 10^9$ (G <0)	-5.41%
streamcluster	$-1.2 * 10^8$ (G <0)	-8.84%
swaptions	0 (G >= 0)	9.74%
vips	0 (G >= 0)	12.02%
backprop	$6.9 * 10^{10}$ (G >= 0)	8.21%
bfs	$7.6 * 10^{10}$ (G >= 0)	15.54%
ESTM-specfriendly-tree	$-1.6 * 10^{10}$ (G <0)	-25.00%
lockfree-fraser-skiplist	$-1.5 * 10^{12}$ (G <0)	-12.10%
MUTEX-hashtable	$-3.1 * 10^{11}$ (G <0)	-7.25%
MUTEX-skiplist	$-9.6 * 10^{10}$ (G <0)	-15.34%
SPIN-hashtable	$-5.3 * 10^{10}$ (G <0)	-11.66%
SPIN-hoh-list	$-1.8 * 10^{10}$ (G <0)	-11.24%

Table 6.8: Prediction of execution outcomes when ACP is activated

We expect an application to get a performance speedup when running with ACP if $G \geq 0$. By using this metric, we predict whether an application will gain or lose performance when ACP is activated in the Intel Xeon Gold 6148 Skylake. The

benchmarks that we use in this experiment are 10 benchmarks from PARSEC [18] (*bodytrack, streamcluster, swaptions, vips, blackscholes, dedup, facesim, ferret, fluidanimate,* and *freqmine*), 7 benchmarks from Rodinia [25] (*backprop, bfs, hotspot, kmeans, leukocyte, needle,* and *srad*), and all 14 benchmarks from Synchrobench [44]. After running these benchmarks with ACP enabled, we noticed that only 12 of them display performance gains or losses that are higher than 5%. Our model accurately predicts the execution outcomes of these 12 benchmarks as shown in Table 6.8.

6.5.8 Overhead Analysis

To evaluate the runtime and memory overheads of ReuseTracker in the Intel machine, we run it on ten PARSEC benchmarks to profile their intra-thread reuse distance with 100K sampling interval. For these benchmarks, the average runtime overhead is 2.9×, and the average memory overhead is 2.8×, which are much lower than the overheads of existing simulator and binary instrumentation-based tools as shown in Table 6.1. However, the overheads of ReuseTracker are slightly higher than the overheads of other PMU-based techniques like StatCache [16] and RDX [119]. This is because each sampling thread in ReuseTracker arms not only its own debug registers but also debug registers of other cores. The comparison between ReuseTracker and other similar techniques are shown in Table 6.1. The overheads of RDX and loca were obtained by running them with the same ten PARSEC benchmarks, and RDX with 100K sampling interval. The overheads of RDX reported here are higher than the ones reported in its paper because in [119] authors used a sampling interval of 5M. In addition to measuring overheads while profiling reuse distance in individual threads, we also measure overheads for shared cache profiling. The average runtime overhead for reuse distance profiling in shared cache is 2.1×, and the average memory overhead is 2.4×, which are even lower than the overheads of intra-thread profiling. These lower overheads might be caused by lower number of watchpoint traps that are handled at shared cache level than at intra-thread level.

To evaluate the overheads in the AMD machine, we run ReuseTracker on six

PARSEC benchmarks, i.e. *bodytrack*, *fluidanimate*, *freqmine*, *streamcluster*, *swaptions*, and *vips*, with a sampling interval of 50K. The average runtime overhead that we observe is 4.92×, and the average memory overhead is 3.84×.

Chapter 7

CONCLUSION AND FUTURE WORK

Precise event sampling is a low-overhead profiling technology in current commodity CPUs. It allows hardware or software event sampling by also attributing the sampled events to the instructions that trigger the events. This capability has been supported in several different architectures, and has been used in a number of profiling techniques. However, none of the existing techniques captures inter-thread communications and measures reuse distance in multithreaded applications, which are essential information to guide performance tuning on shared memory parallel applications. Furthermore, there have also been only few works that analyze the characteristics of the precise event sampling facilities in commodity CPUs. All of these works focus only on the facility in the Intel architecture, i.e. Intel PEBS. Furthermore, these works only analyze the accuracy and time overhead of PEBS, and do not address other aspects such as memory overhead, stability, and functionality of PEBS.

To address the mentioned research gaps, in this dissertation, we firstly present comprehensive qualitative and quantitative analyses on the precise event sampling facilities of Intel and AMD architectures. After that, we propose two precise event sampling-based tools that capture inter-thread communications and measure reuse distance of multithreaded applications, respectively.

We extensively analyze two precise event sampling facilities from Intel and AMD architectures. In our qualitative analysis, we present their differences in terms of usable counters, types of events that can be sampled, types of data that is available in each sample, and their abilities to identify the execution mode of each sample. Then, we quantitatively analyze the accuracy, stability, sampling bias, overheads, and functionalities of each sampling facility. We also relate how the qualitative

differences that we identified affect some of those aspects that we quantitatively study. We believe our findings can greatly help tool developers understand the behaviour of their profiling tools and guide hardware designers to better design precise event sampling facility of future CPUs.

After analyzing the precise event sampling facilities, we propose ComDetective, a communication matrix generation tool that leverages PMUs and debug registers to detect inter-thread data movement on a sampling basis and avoids the drawbacks of prior work by being more accurate and introducing low time and memory overheads. We present the algorithm used by ComDetective and its implementation details, then evaluate the accuracy, performance, and utility of the tool, by carrying out extensive experiments. Tuning code based on the insights gained from ComDetective delivered up to 13% speedup. Programmers can generate insightful communication matrices, differentiate true and false sharing, associate communication to objects, and pinpoint high inter-thread communication in their applications with the help of ComDetective.

We also present ReuseTracker, a reuse distance analysis tool that profiles reuse distance in individual threads and in shared caches of multi-threaded applications with low overheads by leveraging PMUs and debug registers. We proposed two different algorithms to profile reuse distance in individual threads and shared caches respectively. To verify the accuracy of the intra-thread profiling algorithm, we developed a synthetic benchmark that can be configured to generate a variety of reuse distance histogram patterns. We also demonstrated how ReuseTracker can be used to guide performance optimization by removing false sharing via code refactoring and how it can predict whether multi-threaded applications gain or lose performance when adjacent cache line prefetch (ACP) is enabled. By using ReuseTracker, programmers will be able to profile data locality in thread and shared caches with low overheads, and use the generated information to tune application performance [102],[101].

We plan to extend our work in two ways. Firstly, we will also perform in-depth qualitative and quantitative analyses on Statistical Profiling Extension (SPE),

which is the precise event sampling facility of ARM. Secondly, we will extend ComDetective and ReuseTracker to run on ARM processors by modifying them to interface with SPE in sampling memory accesses from profiled applications.

BIBLIOGRAPHY

[1] "ComDetective: A tool for inter-thread/inter-core communication analysis based on HPCToolkits." https://github.com/comdetective-tools/hpctoolkit.

[2] "dcompiler/loca: Program locality analysis tools," https://github.com/dcompiler/loca, accessed: 20 July 2020.

[3] "Extended Asm - Assembler Instructions with C Expression Operands," https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html.

[4] "perf: Linux profiling with performance counters ," https://perf.wiki.kernel.org/index.php/Main_Page, June 2020.

[5] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency Computation: Practice Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[6] S. Akiyama and T. Hirofuchi, "Quantitative evaluation of intel pebs overhead for online system-noise analysis," in *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*, ser. ROSS '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3095770.3095773

[7] AMD, "AMD uProf," https://developer.amd.com/amd-uprof/, Advanced Micro Devices, Inc., accessed: 2021-05-14.

[8] ——, *AMD64 Technology. AMD64 Architecture Programmer's Manual Volume 2: System Programming. Publication No. 24593 Revision 3.36*, Advanced Micro Devices, Inc., October 2020.

[9] AMG, "Parallel Algebraic Multigrid Solver," https://github.com/LLNL/AMG, 2017.

[10] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?" *ACM Trans. Comput. Syst.*, vol. 15, no. 4, p. 357–390, Nov. 1997. [Online]. Available: https://doi.org/10.1145/265924.265925

[11] ARM, *Arm Neoverse TM N1 Core. Version r3p1*, ARM, February 2019.

[12] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, "Enhancing operating system support for multicore processors by using hardware performance monitoring," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 56–65, 2009.

[13] A. Barai, G. Chennupati, N. Santhi, A.-H. Badawy, Y. Arafa, and S. Eidenbenz, "Ppt-sasmm: Scalable analytical shared memory model: Predicting the performance of multicore caches from a single-threaded execution trace," in *The International Symposium on Memory Systems*, ser. MEMSYS 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 341–351. [Online]. Available: https://doi.org/10.1145/3422575.3422806

[14] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of splash-2 and parsec," in *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*, 2009.

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

[16] 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*, ser. SIGMETRICS'05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 169–180. [Online]. Available: https://doi.org/10.1145/1064212.1064232

[17] K. Beyls and E. D'Hollander, "Reuse distance as a metric for cache behavior." in *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, IASTED, Anaheim, California, USA, 2001*, 2001, pp. 617–622.

[18] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2008, pp. 72–81.

[19] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[20] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, "0.374 pflop/s trillion-particle kinetic modeling of laser plasma interaction on roadrunner," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 63:1–63:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413435

[21] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the International Symposium*

on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, ser. CGO '03. USA: IEEE Computer Society, 2003, p. 265–275.

[22] C. Cascaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proceedings of the 17th Annual International Conference on Supercomputing*, ser. ICS '03. New York, NY, USA: Association for Computing Machinery, 2003, pp. 150–159. [Online]. Available: https://doi.org/10.1145/782814.782836

[23] M. Chabbi, S. Wen, and X. Liu, "Featherlight on-the-fly false-sharing detection," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 152–167. [Online]. Available: https://doi.org/10.1145/3178487.3178499

[24] M. J. Charney, "Intel X86 Encoder Decoder Software Library," https://software.intel.com/content/www/us/en/develop/articles/xed-x86-encoder-decoder-software-library.html, Jul 2015.

[25] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. USA: IEEE Computer Society, 2009, p. 44–54. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[26] P. Cicotti and L. Carrington, "Adamant: Tools to capture, analyze, and manage data movement," in *The International Conference on Computational Science, 2016. ICCS 2016.*, 2016.

[27] E. H. M. Cruz, M. Diener, L. L. Pilla, and P. O. A. Navaux, "Eagermap: A task mapping algorithm to improve communication and

load balancing in clusters of multicore systems," *ACM Trans. Parallel Comput.*, vol. 5, no. 4, pp. 17:1–17:24, Mar. 2019. [Online]. Available: http://doi.acm.org/10.1145/3309711

[28] E. H. Cruz, M. Diener, and P. O. Navaux, "Using the translation lookaside buffer to map threads in parallel applications based on shared memory," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[29] E. H. M. da Cruz, M. A. Z. Alves, A. Carissimi, P. O. A. Navaux, C. P. Ribeiro, and J.-F. Mehaut, "Using memory access traces to map threads and data on hierarchical multi-core platforms," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, 2011.

[30] V. Danjean, R. Namyst, and P.-A. Wacrenier, "An efficient multi-level trace toolkit for multi-threaded applications," in *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 166–175.

[31] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos, "Profileme: hardware support for instruction-level profiling on out-of-order processors," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, 1997, pp. 292–302.

[32] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 340–351.

[33] M. Diener, E. H. M. Cruz, M. A. Z. Alves, and P. O. A. Navaux, "Communication in shared memory: Concepts, definitions, and efficient detection," in

2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2016.

[34] M. Diener, E. H. Cruz, L. L. Pilla, F. Dupros, and P. O. Navaux, "Characterizing communication and page usage of parallel applications for thread and data mapping," *Performance Evaluation*, vol. 88-89, pp. 18–36, 2015.

[35] C. Ding and T. Chilimbi, "A composable model for analyzing locality of multi-threaded programs," Tech. Rep. MSR-TR-2009-107, August 2009. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-composable-model-for-analyzing-locality-of-multi-threaded-programs/

[36] C. Ding and Y. Zhong, "Reuse distance analysis," Tech. Rep., 2001.

[37] J. M. Domingos, P. Tomas, and L. Sousa., "Supporting risc-v performance counters through performance analysis tools for linux (perf)," in *CARRV 2021: Fifth Workshop on Computer Architecture Research with RISC-V*, 2021.

[38] P. J. Drongowski, "Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors," https://pdfs.semanticscholar.org/5219/4b43b8385ce39b2b08ecd409c753e0efafe5.pdf, November 2007.

[39] ——, "An introduction to analysis and optimization with amd codeanalyst™ performance analyzer," Advanced Micro Devices, Inc., Tech. Rep., 2008.

[40] A. Eizenberg, S. Hu, G. Pokam, and J. Devietti, "Remix: Online detection and repair of cache contention for the jvm," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 251–265. [Online]. Available: https://doi.org/10.1145/2908080.2908090

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

[42] D. Eklov, D. Black-Schaffer, and E. Hagersten, "Fast modeling of shared caches in multicore systems," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, ser. HiPEAC'11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 147–157. [Online]. Available: https://doi.org/10.1145/1944862.1944885

[43] B. Gottschall, L. Eeckhout, and M. Jahre, "Tip: Time-proportional instruction profiling," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 15–27. [Online]. Available: https://doi.org/10.1145/3466752.3480058

[44] V. Gramoli, "More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–10. [Online]. Available: https://doi.org/10.1145/2688500.2688501

[45] J. L. Greathouse, "Re: Error : IBS profiling is disabled in your BIOS ," https://community.amd.com/t5/general-discussions/error-ibs-profiling-is-disabled-in-your-bios/td-p/55043, AMD Community.

[46] ——, "Re: IBS not available on EPYC 7451 ?" https://community.amd.com/t5/server-gurus-discussions/ibs-not-available-on-epyc-7451/m-p/258228, AMD Community.

[47] ——, "AMD Research Instruction Based Sampling Toolkit," https://github.com/jlgreathouse/AMD_IBS_Toolkit, Jul 2017.

[48] B. Gregg, "perf Examples," http://www.brendangregg.com/perf.html, Jul 2020.

[49] R. Hegde, "Optimizing Application Performance on Intel Core Microarchitecture Using Hardware-Implemented Prefetchers," https://software.intel.com/content/www/us/en/develop/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware-imple
html, 2015.

[50] C. Helm and K. Taura, "Perfmemplus: A tool for automatic discovery of memory performance problems," in *High Performance Computing*, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds. Cham: Springer International Publishing, 2019, pp. 209–226.

[51] X. Hu, X. Wang, L. Zhou, Y. Luo, Z. Wang, C. Ding, and C. Ye, "Fast miss ratio curve modeling for storage cache," *ACM Trans. Storage*, vol. 14, no. 2, Apr. 2018. [Online]. Available: https://doi.org/10.1145/3185751

[52] Intel, *Intel Performance Tuning Utility 3.2 Update*. Intel Corporation, 2008.

[53] ——, "Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide," https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf, 2010.

[54] ——, "Avoiding and Identifying False Sharing Among Threads," https://software.intel.com/content/www/us/en/develop/articles/avoiding-and-identifying-false-sharing-among-threads.html, Intel Corporation, 2011.

[55] ——, *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B: System Programming Guide, Part 2. Order Number 253669*, Intel Corporation, May 2020.

[56] K. Ji, M. Ling, and L. Liu, "A probability model of calculating l2 cache misses," in *Proceedings of the 2018 International Conference on Computer Science, Electronics and Communication Engineering (CSECE 2018)*. Atlantis Press, 2018/02, pp. 329–332. [Online]. Available: https://doi.org/10.2991/csece-18.2018.71

[57] K. Ji, M. Ling, Y. Zhang, and L. Shi, "An artificial neural network model of lru-cache misses on out-of-order embedded processors," *Microprocessors and Microsystems*, vol. 50, pp. 66 – 79, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933117301126

[58] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen, "Is reuse distance applicable to data locality analysis on chip multiprocessors?" in *Compiler Construction*, R. Gupta, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 264–282.

[59] M. S. Johnson, "Some Requirements for Architectural Support of Software Debugging," in *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS I. New York, NY, USA: ACM, 1982, pp. 140–148. [Online]. Available: http://doi.acm.org/10.1145/800050.801837

[60] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[61] R. Lachaize, B. Lepers, and V. Quema, "Memprof: a memory profiler for NUMA multicore systems," in *USENIX ATC'12 Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012, p. 5.

[62] R. Lachaize, B. Lepers, and V. Quéma, "Memprof: A memory profiler for numa multicore systems," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 5.

[63] L. Lamport, "Concurrent reading and writing," *Commun. ACM*, vol. 20, no. 11, p. 806–811, Nov. 1977. [Online]. Available: https://doi.org/10.1145/359863.359878

[64] F. Larysch, "Fine-grained estimation of memory bandwidth utilization," Master's thesis, Karlsruhe Institute of Technology (KIT), Germany, 2016.

[65] M. Ling, J. Ge, and G. Wang, "Fast modeling l2 cache reuse distance histograms using combined locality information from software traces," *Journal of Systems Architecture*, vol. 108, p. 101745, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762120300394

[66] M. Ling, X. Lu, G. Wang, and J. Ge, "Analytical modeling the multi-core shared cache behavior with considerations of data-sharing and coherence," *IEEE Access*, vol. 9, pp. 17 728–17 743, 2021.

[67] Linux, "perf_event_open - Linux man page," https://linux.die.net/man/2/perf_event_open, 2012.

[68] ——, "SIGALTSTACK," http://man7.org/linux/man-pages/man2/sigaltstack.2.html, 2018.

[69] T. Liu and X. Liu, "Cheetah: Detecting false sharing efficiently and effectively," in *Proceedings of the 2016 International Symposium on Code*

Generation and Optimization, ser. CGO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–11. [Online]. Available: https://doi.org/10.1145/2854038.2854039

[70] X. Liu and J. Mellor-Crummey, "Pinpointing data locality problems using data-centric analysis," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. USA: IEEE Computer Society, 2011, p. 171–180.

[71] ——, "A data-centric profiler for parallel programs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2503210.2503297

[72] ——, "A tool to analyze the performance of multithreaded programs on numa architectures," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–272. [Online]. Available: https://doi.org/10.1145/2555243.2555271

[73] ——, "A tool to analyze the performance of multithreaded programs on numa architectures," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–272. [Online]. Available: https://doi.org/10.1145/2555243.2555271

[74] X. Liu and B. Wu, "Scaanalyzer: A tool to identify memory scalability bottlenecks in parallel programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing

Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807648

[75] 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*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[76] LULESH 2.0, "Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)," https://github.com/LLNL/LULESH.

[77] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti, "Laser: Light, accurate sharing detection and repair," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 261–273.

[78] R. K. V. Maeda, Q. Cai, J. Xu, Z. Wang, and Z. Tian, "Fast and accurate exploration of multi-level caches using hierarchical reuse distance," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 145–156.

[79] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[80] J. Mario, "C2C - False Sharing Detection in Linux Perf," https://joemario.github.io/blog/2016/09/01/c2c-blog/, 2016.

[81] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[82] A. Mazaheri, F. Wolf, and A. Jannesari, "Characterizing loop-level communication patterns in shared memory applications," in *Proceedings of the 2015 44th International Conference on Parallel Processing*, ser. ICPP 2015, 2015.

[83] ——, "Unveiling thread communication bottlenecks using hardware-independent metrics," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: ACM, 2018, pp. 6:1–6:10. [Online]. Available: http://doi.acm.org/10.1145/3225058.3225142

[84] C. McCurdy and J. Vetter, "Memphis: Finding and fixing numa-related performance problems on multi-core platforms," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010, pp. 87–96.

[85] R. E. McLear, D. M. Scheibelhut, and E. Tammaru, "Guidelines for Creating a Debuggable Processor," in *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS I. New York, NY, USA: ACM, 1982, pp. 100–106. [Online]. Available: http://doi.acm.org/10.1145/800050.801833

[86] miniFE, "MiniFE Finite Element Mini-Application," https://github.com/Mantevo/miniFE.

[87] G. Nakhimovsky, "Debugging and Performance Tuning with Library Interposers," http://dsc.sun.com/solaris/articles/lib_interposers.html, Jul 2001.

[88] D. S. Nikolopoulos, E. Ayguadé, and C. D. Polychronopoulos, "Runtime vs. manual data distribution for architecture-agnostic shared-memory programming models," *International Journal of Parallel Programming*, vol. 30, no. 4, pp. 225–255, 2002.

[89] A. R. Nonell, B. Gerofi, L. Bautista-Gomez, D. Martinet, V. B. Querol, and Y. Ishikawa, "On the applicability of pebs based online memory access

tracking for heterogeneous memory management at scale," in *Proceedings of the Workshop on Memory Centric High Performance Computing*, ser. MCHPC'18. New York, NY, USA: Association for Computing Machinery, 2018, p. 50–57. [Online]. Available: https://doi.org/10.1145/3286475.3286477

[90] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 2–11. [Online]. Available: https://doi.org/10.1145/1772954.1772958

[91] PENNANT, "Unstructured mesh hydrodynamics for advanced architectures," https://github.com/lanl/PENNANT, 2016.

[92] M. Pericas, K. Taura, and S. Matsuoka, "Scalable analysis of multicore data reuse and sharing," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 353–362. [Online]. Available: https://doi.org/10.1145/2597652.2597674

[93] A. Pesterev, N. Zeldovich, and R. T. Morris, "Locating cache performance bottlenecks using data profiling," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 335–348. [Online]. Available: https://doi.org/10.1145/1755913.1755947

[94] Quicksilver, "A proxy app for the Monte Carlo Transport Code, Mercury," https://github.com/LLNL/Quicksilver.

[95] C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez, and A. Cristal, "A risc-v simulator and benchmark suite for designing and

evaluating vector architectures," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, nov 2020. [Online]. Available: https://doi.org/10.1145/3422667

[96] V. Reddi, A. Settle, D. A. Connors, and R. Cohn, "Pin: a binary instrumentation tool for computer architecture research and education," in *WCAE '04*, 2004.

[97] P. Roy and X. Liu, "Structslim: A lightweight profiler to guide structure splitting," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 36–46. [Online]. Available: https://doi.org/10.1145/2854038.2854053

[98] P. Roy, S. L. Song, S. Krishnamoorthy, and X. Liu, "Lightweight detection of cache conflicts," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 200–213. [Online]. Available: https://doi.org/10.1145/3168819

[99] L. Rudolph and Z. Segall, "Dynamic decentralized cache schemes for mimd parallel processors," in *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ser. ISCA '84. New York, NY, USA: Association for Computing Machinery, 1984, p. 340–347. [Online]. Available: https://doi.org/10.1145/800015.808203

[100] J. M. Sabarimuthu and T. G. Venkatesh, "Analytical derivation of concurrent reuse distance profile for multi-threaded application running on chip multiprocessor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1704–1721, 2019.

[101] M. A. Sasongko, M. Chabbi, P. Akhtar, and D. Unat, "Comdetective: A lightweight communication detection tool for threads," in *Proceedings of*

the International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356214

[102] M. A. Sasongko, M. Chabbi, M. B. Marzijarani, and D. Unat, "Reusetracker: Fast yet accurate multicore reuse distance analyzer," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 1, dec 2021. [Online]. Available: https://doi.org/10.1145/3484199

[103] D. L. Schuff, M. Kulkarni, and V. S. Pai, "Accelerating multicore reuse distance analysis with sampling and parallelization," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 53–63.

[104] D. L. Schuff, B. S. Parsons, and V. S. Pai, "Multicore-aware reuse distance analysis," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010, pp. 1–8.

[105] R. Sen and D. A. Wood, "Reuse-based online models for caches," *SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, pp. 279–292, Jun. 2013. [Online]. Available: https://doi.org/10.1145/2494232.2465756

[106] X. Shen, J. Shaw, and B. Meeker, "Accurate approximation of locality from time distance histograms," Tech. Rep., 2006.

[107] P. N. Soomro, M. A. Sasongko, and D. Unat, "Bindme: A thread binding library with advanced mapping algorithms," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 21, 2018. [Online]. Available: https://doi.org/10.1002/cpe.4692

[108] M. Srinivas, B. Sinharoy, R. J. Eickemeyer, R. Raghavan, S. Kunkel, T. Chen, W. Maron, D. Flemming, A. Blanchard, P. Seshadri, J. W. Kellington, A. Mer-

icas, A. E. Petruski, V. R. Indukuru, and S. Reyes, "IBM POWER7 performance modeling, verification, and evaluation," *IBM JRD*, vol. 55, no. 3, pp. 4:1–4:19, May-June 2011.

[109] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007, pp. 47–58.

[110] F. Trahay, F. Rue, M. Faverge, Y. Ishikawa, R. Namyst, and J. Dongarra, "Eztrace: A generic framework for performance analysis," in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2011, pp. 618–619.

[111] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: Association for Computing Machinery, 1995, pp. 392–403. [Online]. Available: https://doi.org/10.1145/223982.224449

[112] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericas, "Trends in data locality abstractions for hpc systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 3007–3020, Oct 2017.

[113] D. Unat, C. Chan, W. Zhang, S. Williams, J. Bachan, J. Bell, and J. Shalf, "Exasat: An exascale co-design tool for performance modeling," *The International Journal of High Performance Computing Applications*, vol. 29, no. 2, pp. 209–232, 2015. [Online]. Available: https://doi.org/10.1177/1094342014568690

[114] D. Unat, T. Nguyen, W. Zhang, M. N. Farooqi, B. Bastem, G. Michelogiannakis, A. Almgren, and J. Shalf, "Tida: High-level programming abstractions for data locality management," in *High Performance Computing*, J. M. Kunkel, P. Balaji, and J. Dongarra, Eds. Cham: Springer International Publishing, 2016, pp. 116–135.

[115] K. Viswanathan, "Intel® Memory Latency Checker v3.9," https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html, 2013.

[116] J. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, pp. 37–57, 03 1985.

[117] VPIC, "Vector Particle-In-Cell (VPIC) Project," https://github.com/lanl/vpic.

[118] G. Wang, J. Ge, Y. Yan, and M. Ling, "A data-sharing aware and scalable cache miss rates model for multi-core processors with multi-level cache hierarchies," in *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, 2019, pp. 267–274.

[119] Q. Wang, X. Liu, and M. Chabbi, "Featherlight reuse-distance measurement," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, feb 2019, pp. 440–453. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HPCA.2019.00056

[120] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?" in *2008 IEEE International Symposium on Workload Characterization*, 2008, pp. 141–150.

[121] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *2013 IEEE*

International Symposium on Performance Analysis of Systems and Software (ISPASS), 2013, pp. 215–224.

[122] S. Wen, X. Liu, J. Byrne, and M. Chabbi, "Watching for software inefficiencies with witch," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 332–347. [Online]. Available: https://doi.org/10.1145/3173162.3177159

[123] "Harmonic progression," https://en.wikipedia.org/wiki/Harmonic_progression_(mathematics), Wikipedia, accessed: 12 January 2021.

[124] M. Williams, "Statistical Profiling Extension for ARMv8-A," https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/statistical-profiling-extension-for-armv8-a, Jan 2017.

[125] X. Xiang, C. Ding, H. Luo, and B. Bao, "Hotl: A higher order theory of locality," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 343–356, Mar. 2013. [Online]. Available: https://doi.org/10.1145/2490301.2451153

[126] H. Xu, Q. Wang, S. Song, L. K. John, and X. Liu, "Can we trust profiling results? understanding and fixing the inaccuracy in modern profilers," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 284–295. [Online]. Available: https://doi.org/10.1145/3330345.3330371

[127] U. M. Yang, "Parallel algebraic multigrid methods high performance preconditioner," *Numerical Solution of Partial Differential Equations on Parallel Computers, LNCS 51*, pp. 209–233, 2006.

[128] J. Yi, B. Dong, M. Dong, and H. Chen, "On the precision of precise event based sampling," in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 98–105. [Online]. Available: https://doi.org/10.1145/3409963.3410490

[129] Y. Zhong, X. Shen, and C. Ding, "Program locality analysis using reuse distance," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 6, Aug. 2009. [Online]. Available: https://doi.org/10.1145/1552309.1552310

Appendix A

LIST OF MICROBENCHMARKS

Benchmark Name	**Description**
Load-Ratio	Evaluates the accuracy of precise event sampling facilities in capturing expected sample count
Locked-Load	Evaluates the accuracy of precise event sampling facilities in capturing samples at low sampling intervals
Bias-Bench	Evaluates sampling bias and instruction attribution accuracy of precise event sampling facilities
Mult-Event	Evaluates precise event sampling facilities when monitoring different event numbers
Exec-Mode	Evaluates the accuracy of precise event sampling facilities in associating samples with the execution modes of the instructions that trigger them
Write-Volume	Evaluates the accuracy of inter-thread communication analyzers in capturing total communication counts of profiled applications
False-Sharing	Evaluates the accuracy of communication analyzers in differentiating true sharing and false sharing
Read-Write	Evaluates the effect of mixing read and write accesses to shared data on total communication counts
Point-to-Point Communication	Evaluates the accuracy of communication analyzers in capturing point-to-point communications
RIBench	Evaluates the accuracy of reuse distance analyzers in capturing different reuse distance patterns