Blackboxing Performance Monitoring Units

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis No. 152

Systems Group, Department of Computer Science, ETH Zurich

Blackboxing Performance Monitoring Units

by

Denny Lin

Supervised by

Prof. Timothy Roscoe
Gerd Zellweger

March 2016–September 2016

inf | Informatik
Computer Science

Abstract

Due to the complexity of modern hardware, computer programs often cause complicated interactions when scheduled together. Users therefore frequently turn to specialized tools such as Performance Monitoring Units (PMUs) for diagnosing system issues. However, PMUs themselves are complex, so they require domain knowledge and a good understanding of the underlying hardware to use. Such obstacles prevent PMUs from seeing more widespread use.

In this thesis, a survey of PMUs investigates the various intricacies involved in selecting performance events and configuring performance counters. Major problems include the limited number of performance counters relative to the number of measurable performance events, abstruse definitions of performance events, and the processor-specific nature of PMUs.

Numerous grouping techniques which associate similar objects are explored with a view of reducing the event space. We also experiment with classification algorithms to automatically detect interference and remove the need for human interpretation of events and counter measurements. Moreover, proposals in this thesis are designed with portability, i.e., the applicability of methods to different machines, in mind.

All in all, we show the complexity of PMUs is a manageable problem and select procedures can be automated, which opens up the possibility of abstracting away PMU details from users.

Contents

1 Introduction

This thesis explores PMUs and presents an automated process for selecting performance events and assessing the validity of selected events in the context of interference detection. The proposed methods are intended to be applicable to any given machine.

1.1 Motivation

There is no doubt computer systems have become increasingly complex in recent years. We have witnessed the advent of multicore processors, and multiprocessor machines are now commonplace. To complicate matters, the ever-widening gap between processor and memory speeds has resulted in intricate memory hierarchies.

Since one fundamental goal of users and programmers is to fully utilize available hardware resources, these changes have inevitably increased their burden, for the underlying hardware has become more difficult to understand. Nevertheless, this problem is manageable as vendor documentation is widely available, and there are tools such as hwloc [6] which provide information pertaining to processor topologies including cache levels and sizes and NUMA nodes.

Thus the real challenge is not understanding the hardware itself, but the interactions between applications running on top of the hardware. One contributing factor to this problem is the aforementioned intricacies of hardware; another is the complexity of modern workloads themselves. First, modern workloads consist of heterogeneous tasks. For instance, enterprises often run a mix of batch jobs and latency-sensitive applications in data centers. Second, the additional parallelism in multicore processors has given rise to the number of multithreaded programs. Needless to say, understanding the interactions caused by a mix of heterogeneous, parallel applications within a complex system is not a trivial task.

While there are various categories of interactions, each with its own characteristics, we are primarily concerned with interactions of a specific nature, namely, those that cause performance degradations. There are many shared resources within systems (e.g., CPU caches, processor interconnects, and memory buses), and these frequently become points of contention. Due to the power wall, CPU manufacturers are using extra transistors gained from Moore's law to introduce

more parallelism in the form of additional cores instead of increasing clock speeds. Parallel applications have therefore gained popularity, but this trend has aggravated resource contention.

How to mitigate resource contention to limit the impact on performance is now an important question. Although it is possible to monitor applications and observe the consequences of their interactions, devising a method which is applicable to all applications and scenarios is nontrivial. An alternative approach is to monitor components in hardware, and others have shown PMUs can be employed to avoid resource contention. However, the usage of PMUs presents a whole new set of challenges.

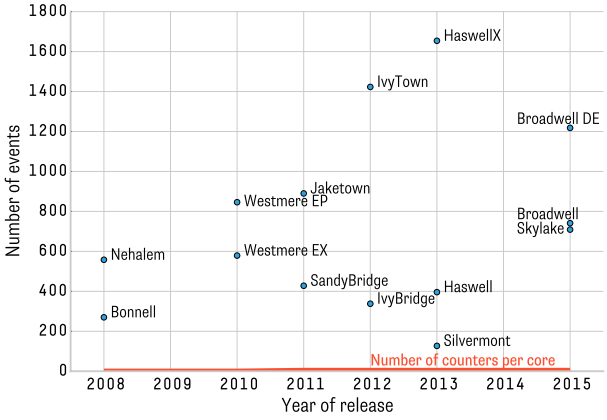


Figure 1.1: Number of counters and events of Intel microarchitectures.

Figure 1.1 shows the number of events and counters of Intel microarchitectures over time. The following observations can be made:

1. The number of available counters has remained relatively constant.
2. The number of countable events greatly exceeds the number of available counters.
3. Each microarchitecture supports different events.

The first observation necessitates the careful selection of events since we can only collect a handful of data points. However, there are typically hundreds of events, and users rarely possess the domain knowledge required to understand event definitions. As a result, the problem is not caused merely by the scarcity of counters, but also a consequence of the multitude of events. Furthermore, a good understanding of the underlying hardware is often essential for interpreting event counts, which raises the entry bar even higher. Another issue is performance events are processor-specific, which prevents the use of performance counters on a large scale.

In summary, we view PMUs as an invaluable tool, but significant hurdles must be cleared for them to become accessible to users. Specifically, the number of events which need to be measured must be small, but the selection of events is presently too cumbersome. In addition, portability is an unsolved problem despite

the development of various libraries. We aim to address these issues so that details of PMUs can be hidden from users.

1.2 Objectives

Survey of PMUs: Previous work has shown PMUs are invaluable tools with applications ranging from scheduling improvements to problem troubleshooting. However, they focus solely on the applications of PMUs and do not provide sufficient insight into PMUs themselves. Therefore, we begin by conducting a detailed survey of PMUs to uncover issues which hinder their adoption by users. This includes case studies of how PMUs are used in practice.

Event Space Reduction: It is important to note PMUs vary from processor to processor. Regardless, they all share one common trait: the number of measurable performance events greatly exceeds the number of available performance counters. As such, we apply grouping techniques which prune redundant events. This is to address the problem of having a limited number of counters.

Interference Detection: While PMUs have been shown to be useful in a number of contexts, we evaluate selected events based on their ability to detect interference. Event values are collected into samples which are supplied to a binary classifier. The accuracy of the classifier enables us to determine the effectiveness of the various grouping methods. This approach opens up the possibility of automatically selecting events for users.

Portability: Previous applications of PMUs are mostly machine-dependent. However, this must change if PMUs are to see more widespread use. Although the grouping methods and evaluation of event groupings in this thesis were applied on a specific machine, the process can be reproduced on another machine without any prior knowledge of performance events other than the list of supported events.

1.3 Outline

The remainder of this thesis explores PMUs and presents a framework for automatically selecting and evaluating events. Chapter 2 contains a brief overview of PMUs and examines related work. Chapter 3 presents a survey of PMUs. Chapter 4 explores methods for grouping performance events. Chapter 5 applies a classification technique to detect interference with samples consisting of event counts. The results are used to evaluate the effectiveness of different event groupings. Chapter 6 discusses possible directions for future work. Chapter 7 concludes. Benchmark configurations are documented in Appendix A, and the list of events considered for selection can be found in Appendix B.

2 Background

This chapter is dedicated to the background information required to understand subsequent chapters. We begin by introducing Performance Monitoring Units. The purpose of this section is to provide insight to the centerpiece of this thesis. Next, a detailed study of related work is presented to understand the possible applications of Performance Monitoring Units.

2.1 Preliminaries

Here we give an overview of Performance Monitoring Units and introduce associated terminology. Although Performance Monitoring Units are processor-specific, we primarily focus on abstract concepts applicable to all processors in this section. Related tools are also mentioned as they will be relevant in later chapters.

2.1.1 Performance Monitoring Units

A **Performance Monitoring Unit** (**PMU**) is a component present in virtually all modern processors. As the name suggests, PMUs can be used to obtain performance measurements within computers. Each PMU contains two main components:

- Performance events
- Performance counters

Performance Events

A **performance event** is a hardware event within the processor which can be measured by the PMU. Examples of events include the number of clock cycles, instructions executed, and last-level cache misses. Some events have **subevents** which allow fine-grained control. For instance, cache misses may be classified as data reads, data writes, instruction fetches, and hardware prefetches. Depending on the PMU, it may or may not be possible to select multiple subevents when measuring an event. Moreover, there are three categories of events: **core events**, **offcore events**, and **uncore events**. Core events are measured within cores whereas uncore events are measured outside of cores. Offcore events are in the middle,

and they often measure requests sent to processor components outside of cores. In this thesis, we consider offcore events to be a special type of core events.

Since PMUs are processor-specific, the supported events vary from processor to processor. Different vendors provide completely distinct events. The difference between microarchitectures from the same vendor is not as pronounced, but there are still minor variations. Generally, each PMU supports several hundred events.

Performance Counters

A **performance counter** is a special-purpose register which is capable of measuring a single event.[1] For example, we could have two counters which measure the number of cache misses and references and then derive the cache miss rate. Counter values are always integers. Each core has its own set of counters. Moreover, there are two types of counters: **fixed counters** and **general-purpose counters**. Fixed counters can only measure specific events while general-purpose registers are capable of measuring any supported event (with a few exceptions).

It is important to note the number of counters is PMU-dependent, and not all PMUs have fixed counters. Generally, each core has less than a dozen counters. If simultaneous multithreading (SMT) is enabled, general-purpose counters in a core are normally evenly divided among SMT threads whereas each SMT thread has its own set of fixed counters.

Instrumentation

Despite the processor-dependent nature of PMUs, the instrumentation is more or less the same for all machines. The following information is therefore applicable to all CPUs. Every event is represented by an event code, and subevents are selected by applying unit masks (umasks). Counters are managed through a set of **model specific registers** (**MSRs**). Typically, each general-purpose counter is associated with two MSRs:

- Event select MSR: This register controls the counter.
- Event counter MSR: This register stores the count of the selected event.

The bits of the event select MSR are used to configure the corresponding event counter MSR:[2]

- Counters are enabled or disabled by toggling a bit.
- Events are selected using event codes and umasks.
- Counters can be restricted to count only when the core is executing in kernel mode or user mode.

On x86 processors, the `WRMSR` instruction is used to write to event select MSRs while `RDMSR` is used to read event counter MSRs. Some vendors have special

[1] Since the use of a performance counter implies the measurement of an event, we will use the terms PMU and performance counters interchangeably.

[2] While event select MSRs provide additional functionality, the details have been omitted for brevity.

instructions for accessing PMU MSRs (e.g., counters can be read with `RDPMC` on Intel CPUs).

There may be additional restrictions on events and counters, e.g., some events can only be measured with specific counters. As the rules are PMU-specific, it is not possible to present a generalization here.

Instrumentation of fixed counters is essentially the same as general-purpose ones, except event select MSRs are replaced by fixed counter control MSRs, which lack the capability to select events.

2.1.2 Tools

Given that instrumenting counters by reading and writing to MSRs is rather cumbersome, a number of tools have appeared which facilitate the use of PMUs by providing high-level abstractions. These include operating system support and user libraries with advanced features. In addition, vendors have also created tools for performance analysis which utilize PMUs.

Operating System Support

Linux has supported PMUs with perf_event through the `perf_event_open` system call (syscall) since version 2.6.3.1 [41].[3] Each measured event is represented as a file descriptor, and it is possible to group events. There are various options such as only monitoring specific processes or cores. It is also possible to periodically sample counters and write the values into a ring buffer. Furthermore, the syscall provides "generic" events for portability across different machines although this requires events to be hardcoded for each supported microarchitecture. Other events are accessible by supplying event codes and umasks. Control of counters is performed through `ioctl` calls.

On FreeBSD, pmc provides an interface to performance counters through the hwpmc kernel module [25, 26]. FreeBSD has event name aliases which are similar to generic events on Linux. Both counting and sampling events are also possible.

User Libraries

Two common libraries with PMU are support are libpfm [20] and PAPI [7, 23]. The former mainly aids users in the selection of events while the latter is a more comprehensive suite with extra functionality.

The libpfm library is normally coupled with perf_event. Instead of requiring users to pass in "raw" event codes and umasks to access events not in the limited set of predefined generic events, libpfm supplies a list of events compiled from vendor documentation. Hence most events have a self-evident name which simplifies event selection for users. Support for each new microarchitecture must still be added to libpfm, but end users need not be concerned with this process.

On the other hand, PAPI is more akin to an all-in-one package. The library was designed as a cross-platform interface, and users configure and access counters

[3] It was originally named `perf_counter_open`

through its API. It provides generic events under the name "standard events" and supports microarchitecture-specific events as "native events." Unlike perf_event and libpfm, it possesses the capacity to multiplex counters. Moreover, PAPI has low-level and high-level APIs. The latter only includes basic functionality.

Vendor Tools

CPU vendors have released PMU tools such as Intel PCM (Performance Counter Monitor) [12] and Intel VTune Amplifier [13]. Although Intel PCM's support is limited to Intel CPUs, it is capable of producing detailed performance metrics for various CPU components. Furthermore, Intel PCM has good support for multiple microarchitectures, and its source code is available online under the 3-clause BSD license.

2.2 Related Work

Previous work has demonstrated that PMUs have numerous applications ranging from performance modeling to power efficiency. It is possible to divide the use cases into those which improve performance of programs online and other applications (here performance refers to either throughput or latency). Both categories are discussed in detail although we are mainly concerned with the former.

2.2.1 Performance Applications of PMUs

Resource Contention

One way of dealing with resource contention in multicore systems is contention-aware scheduling. Another method is to employ cache partitioning to avoid interference. In both cases, previous work has shown good heuristics can be obtained from performance counter measurements.

Azimi et al. reduced cache contention and cache thrashing with cache partitioning [2]. Both objectives were achieved by using the cache miss rate as a heuristic. For the former, it helped determine the cache partition sizes, and in the latter, it provided an estimate of the *reuse distance* so that cache lines which "polluted" the cache by evicting useful cache lines were restricted to a special partition.

Knauerhase et al. made scheduling improvements by employing the number of cache misses per cycle as a heuristic [24]. In a nutshell, the scheduler distributed tasks evenly among caches using the heuristic to reduce interference. Zhuravlev et al. evaluated a similar scheduling policy, but used the cache miss rate as the heuristic instead [45]. A scheme was also devised to quantify factors causing performance degradation, and they concluded that factors such as memory controller contention and hardware prefetching were more dominant than cache contention. However, they noted the cache miss rate is highly correlated with the number of memory controller and prefetch requests, which explains its effectiveness as a heuristic.

Saini et al. assessed the performance benefits of enabling SMT by quantifying processor resource utilization using PMU measurements [39]. They concluded SMT is generally beneficial, but contention in the memory hierarchy can reduce its advantages.

Application Latency

Coscheduling batch jobs and latency-sensitive applications is desirable to maximize resource utilization, but this must not come at the cost of increased latency. While interference is a consequence of resource contention, we need not be concerned with the root cause in contrast to the previous use case. In other words, the contended resource is not important so long as the latency is kept under control.

Zhang et al. collected the cycles-per-instruction (CPI) metric from applications and used historical data to detect anomalies [44]. They identified "antagonists" which caused interference and throttled them. Lo et al. applied resource utilization to determine whether batch jobs and latency-sensitive applications can be scheduled together, for interference occurs when shared resources are saturated [28]. They relied on performance counters to calculate memory bandwidth usage for their resource utilization estimations.

2.2.2 Other Applications of PMUs

Compiler Optimizations

Techniques such as profile-guided optimization (PGO) attempt to find an optimal combination of compiler flags, and they have been applied to real-world programs such as Mozilla Firefox [34]. Cavazos et al. proposed an alternative approach using performance counters [8].

Debugging and Troubleshooting

Bare et al. made measurements of a small set of events and used support vector machines (SVMs), a machine learning technique, to detect anomalies in an e-commerce system [3]. In addition, Bhatia et al. developed a tool which collects system information including counter measurements [5]. The data can then be used for debugging by finding irregularities.

Performance Modeling

Ofenbeck et al. showed performance counter measurements can be used for the roofline model [43] when analyzing application performance by counting the number of FLOPS, memory operations, and clock cycles with performance events on Intel processors [38].

Power Consumption

Singh et al. estimated power consumption of a computer system with data gathered from performance counters [40]. Merkel et al. took the concept a step fur-

ther and developed a power-aware scheduler in addition to estimating power consumption [32].

2.3 Discussion

We can formulate a number of conclusions from the examination of PMUs and related work. This includes not only benefits but also limitations of performance counters.

First, there is no question concerning the usefulness of PMUs. In particular, counter measurements are good indicators of performance [2, 24, 45, 46]. Determining the root cause of interference is relevant in cases where scheduling is employed to avoid resource contention. However, it is not necessary when providing latency guarantees for programs [28, 44]. Moreover, the storage of historical data aids the detection of performance degradation [44].

Second, applications in previous work target a specific processor, which demonstrates a drawback in terms of portability—moving to a new machine would require reselecting the performance events. In cases where the same events exist on the new platform, one must still verify the validity of the events. The presented PMU tools are aware of this limitation, and in response attempt to support a limited set of generic events across all platforms [7]. This is one step to solving the problem even though it is not a complete solution.

Third, most studies make the assumption contention occurs either in caches or the memory bus and measure corresponding performance events such as the number of cache misses or memory accesses. Yet there is no proof such events are the best indicators; it is plausible there are other more suitable candidates.

Fourth, the number of performance counters limits the data obtainable by users. Recent Intel and AMD processors have 11 and 4 counters per core, respectively. In contrast, there are hundreds of measurable events. PAPI supports multiplexing of events, and there are proposals exploring different approaches. Azimi et al. utilized statistical sampling to multiplex counters to get values within 15% of non-multiplexed counts [1]. A similar study was undertaken by May [31]. Consequently, multiplexing is a valid solution, but we will investigate other alternatives which do not rely on statistical sampling to reduce the probability of error.

3 Survey of PMUs

In this chapter, we present a survey of PMUs and discuss their limitations in detail. Topics include the portability of performance events and the scarcity of performance counters. In contrast to the previous chapter, we mention architecture-specific details of PMUs. Case studies of PMUs are also conducted to understand typical workflows of users.

3.1 Vendor Documentation

Priority is given to the x86 architecture, so we only consider Intel and AMD processors in this thesis. Documentation of PMUs for Intel microarchitectures is located in the Intel 64 and IA-32 Architectures Software Developer's Manual (SDM) Volume 3 [11]. Each microarchitecture has its own section with a list of supported performance events. To illustrate, Figure 3.1 contains a truncated list of events for the Ivy Bridge microarchitecture obtained from the SDM. Each event has an event code, umasks for subevents, and a description. Certain events have additional restrictions or notes. Note that high-end CPU models often support extra events compared to basic models within the same microarchitecture.

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split Store-address uops dispatched to L1D.	

Figure 3.1: Snippet of event list for the Ivy Bridge microarchitecture.

Event select and fixed counter control MSRs are also documented although there is less variation between generations. A sample event select MSR from the SDM is shown in Figure 3.2. The various bits control the corresponding counter MSR. Bits 0–7 are for the event code, and bits 8–15 are for the subevent. Bits 16 and 17 specify whether execution in kernel mode and user mode should be measured, respectively. Bit 22 determines whether the counter is enabled or disabled.

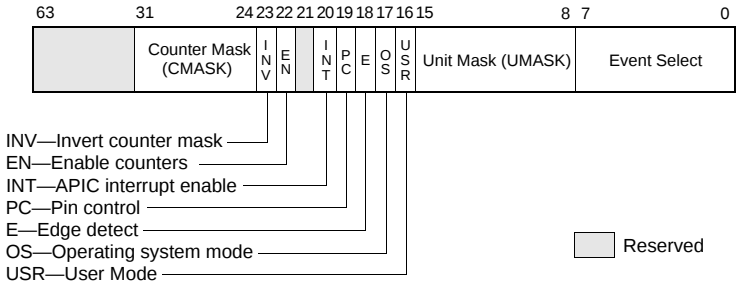


Figure 3.2: Layout of an Intel event select MSR.

In addition to the SDM, Intel also provides event data as JSON and TSV files for each microarchitecture [14].

Documentation for AMD is similar. Whereas Intel covers all microarchitectures in a single volume, AMD releases a separate BIOS and Kernel Developer's Guide (BKDG) for each processor family [18].[1] For instance, there is a dedicated document for family 10h processors [17]. The specification of events and PMUs MSRs is more or the less the same, so details have been omitted for brevity.

3.2 Tools

In this section, we study the actual use of some tools listed in Section 2.1.2, namely, perf_event in the Linux kernel and the libpfm and PAPI libraries.

3.2.1 perf_event

The function prototype of `perf_event_open` is as follows:

```
int perf_event_open(struct perf_event_attr *attr,
                    pid_t pid, int cpu, int group_fd,
                    unsigned long flags);
```

The syscall sets up an event to be measured by a counter and returns a file descriptor which references the event. We will cover the parameters beginning with `attr`.

Event selection: The selection of events is completed through the `type` and `config` fields in `attr`. We are primarily concerned with 3 types:

- `PERF_TYPE_HARDWARE`
- `PERF_TYPE_HW_CACHE`
- `PERF_TYPE_RAW`

The types `PERF_TYPE_HARDWARE` and `PERF_TYPE_HW_CACHE` provide "generic" events across different platforms. The latter contains events concerning caches (e.g., cache accesses, DTLB misses, etc.) while the remaining events belong to the

[1] Intel and AMD use different terminology. A microarchitecture is roughly equivalent to a family.

former (e.g., instructions executed, clock cycles, etc.). Other events not available as generalized events can be selected using `PERF_TYPE_RAW`. The event is set in `config`.

Reading counter values: There are two methods of reading event counts: *counting* and *sampling*.

- Counting reads event counts through the `read` syscall. The format is based on the `read_format` field. Events in the same group can be read with a single invocation using `PERF_FORMAT_GROUP`.
- Sampling writes event counts to an `mmap` ring buffer, and either the sampling frequency or period must be specified. The `sample_type` field defines the sample format. Values for events in the same group can also be read into a single sample using `PERF_SAMPLE_READ`.

Counter control and scope: Various other attributes are specified in `attr`. For instance, the `disabled` field determines if the counter is to be enabled immediately. Counters may be configured to exclude counting of events while executing in user mode and kernel mode using `exclude_user` and `exclude_kernel`, respectively. Monitored processes which are forked or spawn threads have the option of having children inherit their counters with the `inherit` field.

Monitored processes/threads and cores: The counter can be restricted to monitor a specific process/thread by passing in the PID/TID in `pid`. A value of 0 monitors the calling process/thread while a value of -1 monitors all processes/threads. Similarly, the counter only monitors a given CPU if its identifier is passed in as `cpu`.[2] A value of -1 monitors the specified process/thread on any core. If a PID/TID is passed in, the counter value follows the process/thread wherever it is scheduled to run (subject to CPU restrictions), i.e., the counter value is saved and restored if the process/thread migrates to a different core. Hence the returned file descriptor only references a *logical* counter as the underlying hardware counter may change. If a CPU ID is specified, a counter is allocated on the core to measure the event (subject to process/thread restrictions).

Grouping counters: Counters may be grouped together. Counters in the same group can be collectively enabled/disabled, and their values can also be read with a single function call. The only restriction is counters belonging to the same group must be on the same core, and they are to be scheduled together. Passing in `group_fd` = -1 creates a new group with the returned file descriptor as the *group leader*. Specifying the group leader of an existing group adds the new counter to that group.

[2]Here CPU is defined as a logical core. Each logical core has its own unique identifier.

Miscellaneous options: The parameters `attr` and `flags` contain many other options. However, the remaining functionality is not relevant for this thesis, so they have been omitted. Refer to the man page for more details.

Counter operations: Operations are performed with the `ioctl` syscall on file descriptors. The following are used to enable, disable, and reset counter values to zero:

- `PERF_EVENT_IOC_ENABLE`
- `PERF_EVENT_IOC_DISABLE`
- `PERF_EVENT_IOC_RESET`

If `PERF_IOC_FLAG_GROUP` is set, then the operations above are applied to all counters in the same group.

Limitations: The `perf_event_open` syscall has a number of limitations.

- The configuration `pid` = -1 and `cpu` = -1 is an invalid combination. The syscall sets up exactly one *logical* counter to measure the specified event, and it maps to at most one hardware counter at any given time. Monitoring all processes on all cores would violate this.
- The `inherit` bit has no effect on existing children, for it would be impossible to apply this change retroactively.
- The options `PERF_FORMAT_GROUP` and `PERF_SAMPLE_READ` are incompatible with `inherit` since reading counter values of children could potentially require reading counters on different cores, causing synchronization problems.
- Ring buffers for sampling must pinned to memory, so users may have to increase `RLIMIT_MEMLOCK`.
- The syscall is an abstraction layer, so users cannot select the underlying hardware counter. This restricts the control users have over multiplexing counters.

3.2.2 libpfm

The purpose of libpfm to is allow users to select events using self-evident names instead of event codes and subevent umasks, so it is not responsible for configuring PMU MSRs. In practice, libpfm is most commonly used to prepare the `attr` argument for `perf_event_open` by setting `type` to `PERF_TYPE_RAW` and filling in the appropriate value in `config` according to the given event name. The function prototype is shown below:

```
int pfm_get_os_event_encoding(const char *event, int dfl_plm,
                              pfm_os_t os, void *attr)
```

The first argument is reserved for the event name string. The function can limit the scope of counters in `attr` to specific protection rings with `dfl_plm`. Although there are multiple levels, only `PFM_PLM0` and `PFM_PLM3` are used. The

former corresponds to kernel mode while the latter is normally user mode. For use with perf_event, `os` is set to `PFM_OS_PERF_EVENT`. In this case, users may ignore `dfl_plm` and set the appropriate flags in `attr` before passing it to the syscall.

The library provides a list of event names which it translates to event codes for each microarchitecture. While the lists are not complete, they are comprehensive enough to cover frequently used events.

3.2.3 PAPI

PAPI is another library dedicated to cross-platform support of PMUs. Whereas perf_event provides the functionality to access PMUs and libpfm allows users to select events, PAPI covers both domains.

PAPI groups events into event sets which are similar to groups in perf_event. The following functions are responsible for event set management:

```
int PAPI_create_eventset(int *EventSet)
int PAPI_cleanup_eventset(int *EventSet)
int PAPI_destroy_eventset(int *EventSet)
int PAPI_add_named_event(int EventSet, char *EventName)
int PAPI_set_multiplex(int EventSet)
```

PAPI supports both generic and microarchitecture-specific events under the names standardized and native events, respectively, and it is possible to select events with names instead of event codes and umasks. Event sets may also be multiplexed.

Performance counters are managed with the following functions:

```
int PAPI_reset(int EventSet)
int PAPI_start(int EventSet)
int PAPI_stop(int EventSet)
int PAPI_read(int EventSet, long long *values)
```

3.3 Portability

The issue of portability is clearly visible since both perf_event and PAPI attempt to address the problem by introducing generic events. However, we note this approach has three shortcomings:

1. The set of generic events is very limited. For instance, perf_event only defines 17 generic events as of Linux 4.7 while each PMU typically supports hundreds of events.
2. Not all generic events are available on all platforms, i.e., there are generic events which cannot be mapped to suitable events on some processors. If users are restricted to generic events which are always available, they are presented with an even smaller set, further limiting the usefulness of PMUs.
3. Although PMUs may have events with identical or similar names, it does not mean they are equivalent. For example, Intel does not include prefetches in last-level cache misses or references, but other vendors may. Therefore,

generic events give users some degree of convenience, but do not completely solve the problem.

3.4 Performance Overhead

The low performance overhead has been mentioned as an incentive for using PMUs to obtain system diagnostics [1, 24, 44]. We perform microbenchmarks to verify such claims and to provide concrete numbers on the cost of reading counter values.

The benchmarks were performed on *Babybel*. For all benchmarks in this thesis, refer to Appendix A for details such as the hardware configuration, operating system, etc. Event values of event groups were read with bulk operations for perf_event and PAPI 5.4.3, and the group size was varied. Each benchmark performed a total of 10,000,000 read operations, and we calculated the average time.

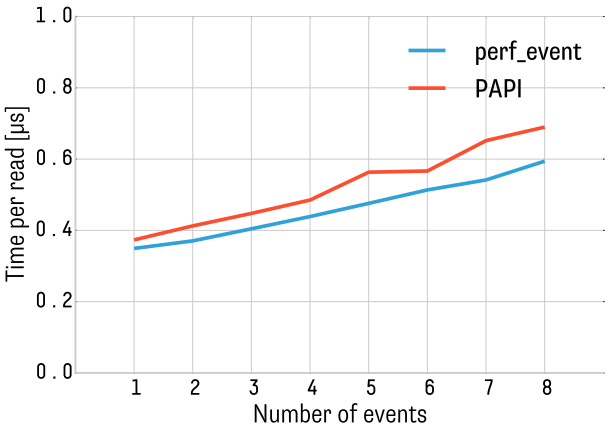


Figure 3.3: Cost of counter reads.

From Figure 3.3, we observe reads for larger group sizes take longer although the increase is not linear with respect to the number of events. Therefore, it is possible to reduce the overhead by leveraging event groups. Even if each event were read individually, the cost to read a single event is approximately 0.4 μs, which is negligible. This corroborates earlier claims.

Finally, we also experimented with different events to determine the effect of selected events on read times. However, there was no observable variance, so the choice of events should be irrelevant.

3.5 Case Study: Scheduling

Now that we have examined various aspects of PMUs, we turn to case studies to understand how they are used in practice. The remainder of this chapter consists of two case studies. The objective of the first case study is to characterize system behaviour. The second case study looks at different types of interference on two separate machines.

Apart from the underlying hardware, thread placements are another major factor which influence application performance. Therefore, it is imperative for us to investigate how they are determined by schedulers and the ramifications of different thread placements.

3.5.1 Preliminaries

We performed benchmarks using the applications PR, HD, and SSSP on *Gottardo* with the following partitioning schemes:

- *shared cores*
- *whole sockets*
- *default*

Every application was configured to spawn 32 threads (the machine has 64 logical cores). Refer to Appendix A for more details concerning benchmarks.

The logical cores of the 4 processors are numbered 1–16, 17–32, 33–48, and 49–64. Logical cores 1 and 9 are SMT threads on the same physical core. The same holds for 2 and 10, 3 and 11, etc. The numbering for remaining processors is analogous.

For *shared cores*, the first application was assigned cores 1–8, 17–24, 33–40, and 49–56 while the second application received the remaining cores. For *whole sockets*, the first application was allocated cores 1–32 while the second application received cores 33–64.

First, we ran each application alone, i.e., the cores reserved for the second application were idle. Next, they were run pairwise to observe the effects of interference, e.g., PR and PR, PR and HD, PR and SSSP, etc. This process was repeated for every partitioning scheme.

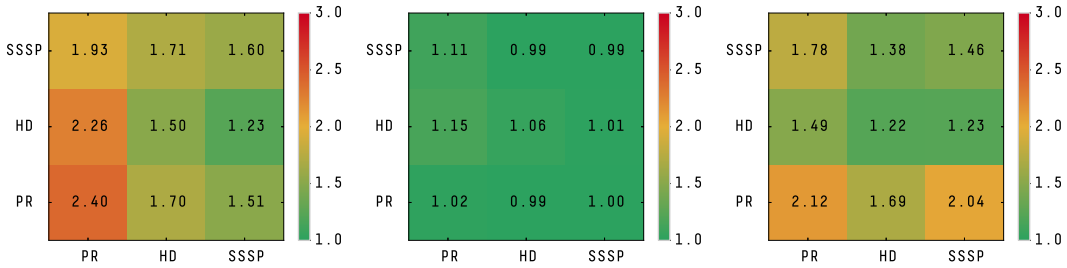
3.5.2 Results

We define the **interference factor** of a given application and partitioning scheme to be the execution time of the interference case divided by the execution time of the base case. As mentioned in Appendix A.4, if an application was run for multiple iterations, we use the execution time of the first iteration only. E.g., if the execution times with and without interference are 60 s and 30 s, respectively, the interference factor is $60/30 = 2$. Clearly, lower factors are desirable, and a factor of 1 is optimal as it signals the absence of performance degradation.

Observe that *whole sockets* exhibits the least interference while *shared cores* has the most in Figure 3.4. *Default* is somewhere in between. This reflects our choice of thread placements. The applications do not share any resources within individual CPUs with the *whole sockets* scheme. In contrast, *shared cores* results in applications sharing memory controllers, L1 to L3 caches, physical cores, etc. Since we selected two extremes, *default* naturally lies in the middle.

Determining the optimal thread placement for a given set of constraints is difficult. To illustrate, if we assume SSSP is a latency-sensitive task running alongside PR, then *whole sockets* is the best strategy as it causes the least interference for SSSP.

However, if we want to attain the highest throughput possible for PR, the other two schemes are preferable.[3]



(a) *shared cores* (b) *whole sockets* (c) *default*

Figure 3.4: Interference heatmaps for each partitioning scheme.

Scheduling History Reconstruction

As part of this thesis, we developed a novel algorithm which reconstructs the thread placement history with performance counter samples. This is useful for understanding how the Linux scheduler allocates cores. In cases where there is perceived interference, we can see which resources are shared between applications. Obviously, this is only meaningful for *default* as other schemes pin threads to cores. The instrumentation of PMUs is described in Appendix A.4. Since we only make use of sample metadata and not event counts, we are not concerned with which events are actually measured.

Each sample contains the PID, TID, and CPU ID[4] in addition to the event and event count. Therefore, it is possible to track threads as they migrate between cores. The reconstruction algorithm is presented in Algorithm 1. The input is a list of samples sorted by timestamp in ascending order; the output is a list of periods. The general idea is to keep track of a set of *active periods* with an invariant: each thread or core has at most one active period at any point in time. We apply the following rules to generate the list of periods:

1. If the current core has an active period belonging to another thread, it is terminated, and a new one is created for the current thread.
2. If the current thread is continuing execution execution on the same core, the active period is extended.
3. If the current thread was migrated from another core, the previous active period is terminated, and a new one is created.

We display the scheduling histories of the benchmark involving PR and SSSP as an example in Figure 3.5. Each thread is shown in a distinct colour, and we use the numbering of CPU cores described above. Clearly, the Linux scheduler attempts to group threads belonging to the same process on the same processor. However,

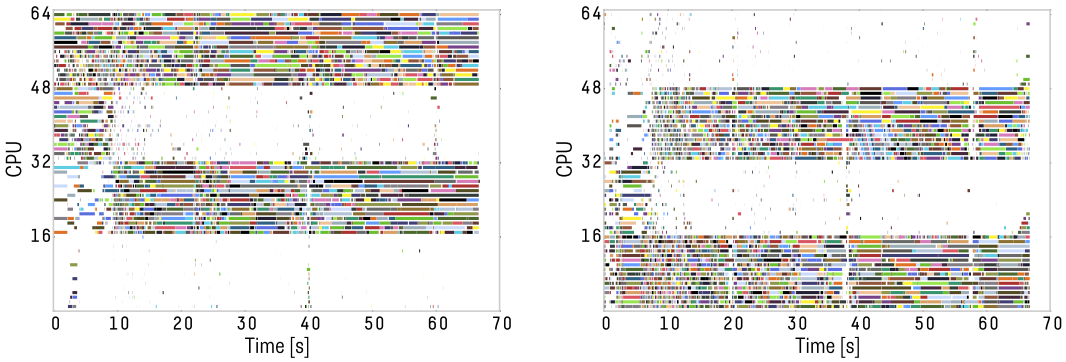
[3] A lower interference factor is not indicative of a shorter execution time. It is possible for a scheme to experience more interference, but still have a shorter execution time.

[4] CPU IDs reference logical cores instead of whole processors.

Algorithm 1 Scheduling history reconstruction

```
1:  function Reconstruct-Scheduling-History(samples)
2:      periods = list()
3:      active_periods = dict()  // Key = TID, value = period
4:      last_tids = dict()       // Key = CPU, value = TID
5:      for all sample ∈ samples
6:          // If another thread was running on current CPU, end its active period.
7:          prev_tid = get(last_tids, sample.cpu)
8:          if prev_tid ≠ null and prev_tid ≠ sample.tid
9:              add(periods, get(active_periods, prev_tid))
10:             remove(active_periods, prev_tid)
11:         // Get active period of current thread if it exists.
12:         period = get(active_periods, sample.tid)
13:         if period ≠ null and period.cpu == sample.cpu
14:             // Thread was running on same CPU, so extend period.
15:             period.end = sample.time
16:         else
17:             // If the active period is on another CPU, end it and clear CPU state.
18:             if period ≠ null  // Condition period.cpu == sample.cpu implied
19:                 add(periods, period)
20:                 remove(last_tids, period.cpu)
21:             // Create new active period.
22:             new_period = Create-Period(sample.cpu, sample.tid, sample.time)
23:             set(active_periods, (sample.tid, new_period))
24:         set(last_tids, (sample.cpu, sample.tid))
25:     return periods
```

there is some unnecessary overhead as threads from the same task are frequently migrated between cores. In this particular case, *default* behaves similarly to *whole sockets*. However, we have also seen scenarios where it is not as clear cut and threads belonging to different processes are scheduled on the same socket. The cost of thread migration is not merely limited to context switches. For example, machines having multiple NUMA nodes may initially allocate memory on the first NUMA node a thread is scheduled on. Then subsequent migrations run the risk of introducing additional memory latency.



(a) PR (b) SSSP

Figure 3.5: Scheduling histories of PR and SSSP.

Basically, the Linux scheduler tries to strike a balance between latency and throughput, yet it is still difficult to impose specific constraints users may need. This is complicated by the fact that finding an optimal thread placement is non-trivial. As it has been shown scheduler bugs can lead to severe performance issues [29], the reconstruction of scheduling histories using performance counters has potential use cases since users can gain additional insight into the system.

Algorithm 2 Scheduling history reconstruction (auxiliary function)

```
1: function CREATE-PERIOD(cpu, tid, time)
2:     period.cpu = cpu
3:     period.tid = tid
4:     period.begin = time
5:     period.end = time
6:     return period
```

3.6 Case Study: Interference

Now that it has been shown applications can have complex interactions, we want to establish that PMUs are capable of diagnosing performance issues. In this section, we present case studies involving cache and memory interference.

3.6.1 Cache Interference

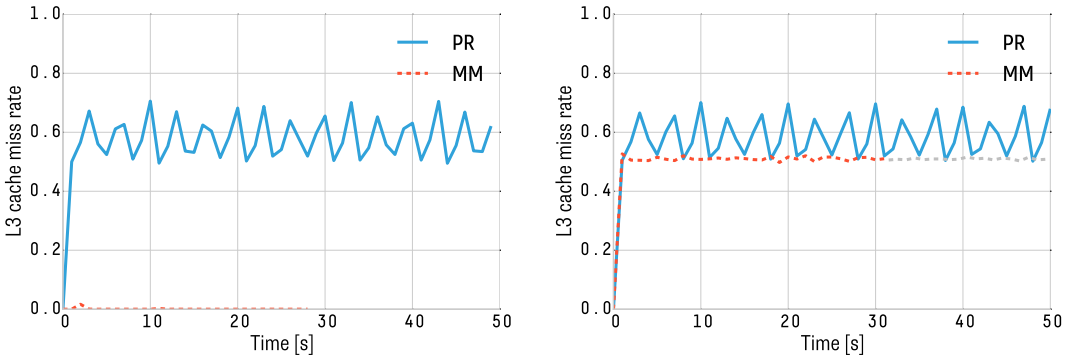
In this case study we paired an application with good locality of reference and an application with poor locality to demonstrate contention on the last-level cache (LLC).

We ran MM with PR on *Babybel* (in this case study MM refers to MM1). PR was configured to use 8 threads. The PR threads and the MM process were pinned to separate cores on the same processor. *Babybel* has 2 processors with 10 cores each, so one processor had 9 cores in use while the remaining core and all cores on the other processor were idle. The following two events were used to calculate the L3 cache miss rate [11]:

- `LONGEST_LAT_CACHE.REFERENCE`
- `LONGEST_LAT_CACHE.MISS`

The L3 cache miss rates of the applications running alone are shown in Figure 3.6a. MM and PR have average cache miss rates of 0.0008 and 0.57, respectively. The low miss rate for the former can be explained by the fact the input and output matrices are both approximately 2 MB each and fit into the processor's L3 cache (25 MB). In contrast, PR has a much higher cache miss rate which explains consumption of memory bandwidth in the previous case study.

The results of the applications running side by side are presented in Figure 3.6b. The cache miss rate of MM dramatically increases to 0.57 whereas the rate for PR remains steady at 0.58. However, the elevated cache miss rate resulted only in an 8.9% increase in execution time for MM. Because the overall execution



(a) PR and MM running alone.

(b) PR and MM running simultaneously.

Figure 3.6: L3 cache miss rates of PR and MM instances.

time of MM is shorter than PR, we restarted MM after the program finished in Figure 3.6b. The continued execution is indicated by the grey line.

At a glance, the increases in the cache miss rate and execution time appear to be disproportionate; however, examination of the L2 cache miss rate clarifies this—it is relatively low at 0.07. As only a small number of requests reach L3 cache, an increase in L3 cache misses has limited effect on the execution time.

Unlike the L3 cache, there is no event which counts all the different types of L2 references and misses collectively, so they had to be measured separately. The following events were required to calculate the L2 cache miss rate [11]:

- `L2_RQSTS.ALL_DEMAND_DATA_RD`
- `L2_RQSTS.DEMAND_DATA_RD_HIT`
- `L2_RQSTS.ALL_RFO`
- `L2_RQSTS.RFO_MISS`
- `L2_RQSTS.ALL_CODE_RD`
- `L2_RQSTS.CODE_RD_MISS`

As mentioned before, Intel processors only have 8 general-purpose performance counters per core. Consequently, the data for the L2 cache was collected in a separate run with identical configuration.

3.6.2 Memory Interference

Memory interference is caused by applications with working sets unable to fit inside the CPU cache and/or poor locality of reference. Typically, such applications have higher L2 and L3 miss rates which result in an increased number of main memory accesses. This in turn can lead to contention on memory controllers and/or interconnect paths.

To show the effects of memory interference, we measured the memory bandwidth of PR on *Appenzeller*. Memory bandwidth was calculated using a formula obtained from an AMD white paper [19]. The event "DRAM Accesses" was used to count memory accesses [18].

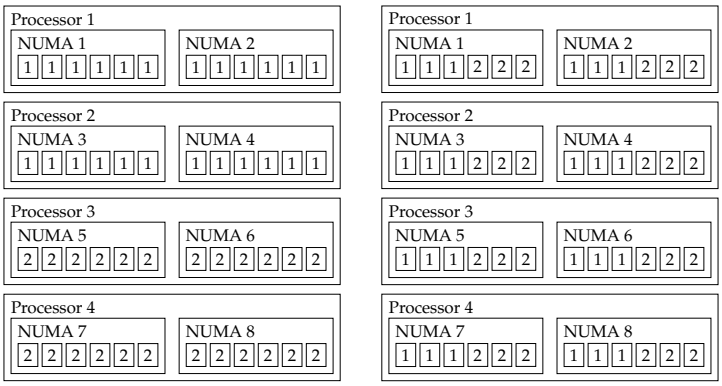
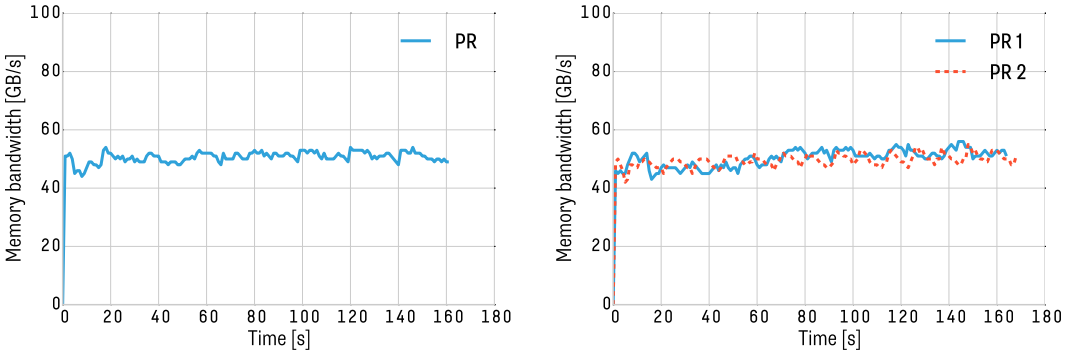


Figure 3.7: Thread placement strategies 1 (left) and 2 (right).

We ran two benchmarks: (1) one instance of PR running alone and (2) two instances of PR running simultaneously. Each instance of PR was configured to use 24 threads (the machine has 48 cores). The threads were pinned to cores during execution to allocate memory controllers and avoid interference by the Linux scheduler. We ran each benchmark with two different thread placement strategies. The first strategy assigned 2 entire processors to each PR instance, which translates to 4 whole NUMA nodes. For the second strategy, each PR instance was allocated half of every NUMA node. A graphical depiction is given in Figure 3.7



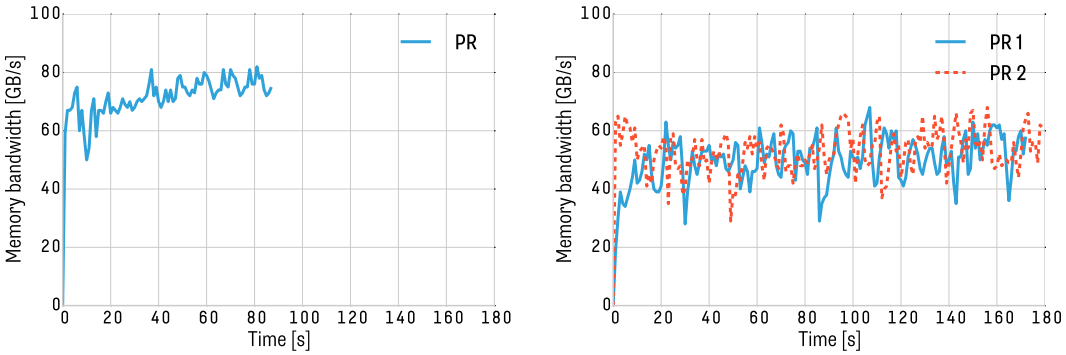
(a) A single PR instance.

(b) Two PR instances.

Figure 3.8: Memory bandwidth of PR instances distributed on 4 NUMA nodes.

In the benchmarks for the first strategy, the execution time for a single PR instance running alone is around 162 seconds, and the measured memory bandwidth is approximately 51 GB/s (c.f. Figure 3.8a). If two PR instances are executed together, the execution time and memory bandwidth remain roughly constant. The two instances do not interfere with each other since the threads of instances are assigned to separate sockets.

For the second strategy, each PR instance received half of every NUMA node, i.e., each instance gets 3 out of 6 cores on all NUMA nodes. We observe the execution time of a single PR instance is nearly halved (c.f. Figure 3.9a). We attribute this to the fact that we now have the ability to make use of twice the number of



(a) A single PR instance.

(b) Two PR instances.

Figure 3.9: Memory bandwidth of PR instances distributed on 8 NUMA nodes.

memory controllers in the system. We see that the memory bandwidth is roughly 21 GB/s higher than in Figure 3.8a. However, once we execute two PR instances simultaneously on the machine, we see that the execution time goes up to approximately 176 seconds and the memory bandwidth decreases to around 52 GB/s. Assuming the two instances share memory controllers fairly, each instance receives the equivalent of $0.5 \times 8 = 4$ memory controllers. The execution time and memory bandwidth attained by instances therefore appears to be dependent on the number of available memory controllers.

3.7 Discussion

From the scheduling case study, we verify the claim that running tasks on modern systems result in complex interactions, and finding an optimal thread placement for a given set of constraints is a difficult problem. The case study on interference then goes on to show PMUs are able to make inroads to a solution by gathering data on hardware components.

It should be noted both case studies are representative of typical use cases of performance counters. One normally proceeds with the following steps:

1. Formulate an objective.
2. Observe the behaviour of the system.
3. Experiment with various events.
4. Adjust the system.

Using the memory interference case study as an example, the objective was to determine the reason why the two thread placement strategies resulted in different performance characteristics. Shared resources in the memory hierarchy are often contended, so we might hypothesize either CPU caches or memory controllers are the bottleneck. Then trial and error is required to find a suitable event capable of diagnosing the problem. The conclusion is to allocate the highest number of NUMA nodes possible without having to coschedule PR instances on the same NUMA nodes.

While this approach is valid and feasible, it suffers from two limitations: (1) the selected events are specific to this machine, and (2) the selection of events is a manual process which is labour-intensive and requires domain knowledge. The first limitation is evident in all applications of performance counters. Previous applications of performance counters also experienced this very same problem. Portability itself is not a showstopper, but the second limitation makes it impossible for this approach to scale. The requirement to understand both the hardware and implications of various events also raises the entry bar. Moreover, event selection is inherently time-consuming, so it causes users to automatically turn to a small set of candidate events such as cache misses or memory accesses. While these may be relevant events for performance issues, they are not necessarily the best events. The failure to consider a wider range of events opens up the possibility of passing over more suitable candidates.

We therefore propose a new framework which is required to be (1) portable across *all* architectures and (2) capable of *automatically* selecting relevant events for users. Although we have seen numerous applications of PMUs, we will restrict ourselves to detecting and classifying interference for now. The steps for our proposal are as follows:

1. Define list of performance events.
2. Run set of applications and collect measurements for each event.
3. Determine which events are indicative of interference.
4. Classify the types of interference indicated by the selected events.

Step 1 is straightforward as vendors provide a list of events for each CPU. We select as many events as possible, but uncore events are excluded since all processes on the same socket share the same count for an uncore event. Therefore, it is impossible to tie event counts to a specific process.

Step 2 requires an application set containing a mix of programs which covering common workloads of the majority of users. This is difficult as the applications must also exhibit contention of various shared resources when run together. We begin with a small set with the future plans to expand it.

Step 3 will involve a binary classifier which determines whether certain event sets serve as effective heuristics for detecting performance degradation.

Step 4 is reserved for future work.

However, since CPUs possess a limited number of performance counters with respect to the number of measurable events, we must first group similar events before proceeding to interference detection. Consequently, we will discuss our application set and possible approaches for event grouping in the next chapter.

4 Event Space Reduction

Recall the wide gap between the number of countable events and available counters in Figure 1.1. The ability to only measure a limited number events leads to two immediate questions:

1. How many events are needed to detect interference?
2. Which events are needed to measure to detect interference?

Ideally, we would like to see a change in event counts in the presence of interference, and previous studies which used performance counter measurements as heuristics prove this is possible. However, the minimum number of events necessary to detect interference is an open question. Our first step is then to find the intrinsic dimensionality of the event space. Next, we must reduce the event space to allow users to gather information without measuring every single event. This involves grouping correlated events together, and we evaluate several grouping schemes.

4.1 Preliminaries

As our solution is designed with a view to solve portability issues, we do not give answers specific to any platform. Instead, we present a framework which is capable of reducing the event space on any machine by running a predefined set of applications known as the *application set*. We will begin by describing it in detail.

4.1.1 Applications

The selection of applications must be sufficiently diverse to cover most workloads end users run. Furthermore, it must be capable of causing various types of interference. In other words, it moves the burden of experimenting with various events under different workloads away from end users. Due to these reasons, it is a challenge to add new applications. We begin with a small set of applications with plans to expand it at a later date. Currently, the application set consists of the following applications:

- PR

- MM
- SORT

PR instances are configured to spawn 8 threads, and all three variations of MM are used, i.e., MM1, MM2, and MM3.

4.1.2 Benchmarks

A variety of benchmarks were run with the selected applications on *Babybel*. The complete list is shown in Table 4.1. First, each application was run independently. Next, applications were paired together to generate interference. Benchmarks involving paired applications were run with three different partitioning schemes:

- *shared cores*
- *shared sockets*
- *whole sockets*

The table explicitly documents the cores assigned to each application as a set, e.g., for the benchmark MM + PR (*shared sockets*), MM and PR are assigned the core sets {1} and {2–9}, respectively. Hence MM runs on core 1 whereas PR is allocated cores 2–9. Recall *Babybel* has 2 processors with 10 cores each. We label the cores in the first processor 1–10 and the cores in the second 11–20.

Notice that PR is run independently twice. This is because the partitioning of cores for *shared sockets* is different from the other two schemes, so a different baseline is necessary. Normally, the scheme column is left empty if applications are running alone. In addition, MM should be expanded into MM1, MM2, and MM3, e.g., the row MM + SORT (*shared cores*) should be read as three rows:

- MM1 + SORT (*shared cores*)
- MM2 + SORT (*shared cores*)
- MM3 + SORT (*shared cores*)

Application(s)	Scheme	Core Set(s)
MM		{1}
SORT		{1}
PR		{1–8}
PR	*shared cores*	{1–4, 11–14}
MM + SORT	*shared cores*	({1}, {1})
MM + SORT	*shared sockets*	({1}, {2})
MM + SORT	*whole sockets*	({1}, {11})
MM + PR	*shared cores*	({1}, {1–8})
MM + PR	*shared sockets*	({1}, {2–9})
MM + PR	*whole sockets*	({1}, {11–18})
SORT + PR	*shared cores*	({1}, {1–8})
SORT + PR	*shared sockets*	({1}, {2–9})
SORT + PR	*whole sockets*	({1}, {11–18})
PR + PR	*shared cores*	({1–8}, {1–8})
PR + PR	*shared sockets*	({1–4, 11–14}, {5–8, 15–18})
PR + PR	*whole sockets*	({1–8}, {11–18})

Table 4.1: List of benchmarks.

4.1.3 Event Selection and Measurement

The libpfm library provides events for the Intel Xeon E5-2670 v2 processor through the PMU module *ivb_ep*, and as many events as possible were selected. The list of 190 selected events can be found in Appendix B.

Uncore events were excluded since their counts cannot be traced back to a specific process/thread. In more concrete terms, suppose there is an uncore event measuring the number of memory bus transactions. The event count includes transactions for all processes running on the socket as we cannot restrict it to a specific process or core.

The processors in *Babybel* only have 8 general-purpose counters per core, so it is impossible to measure 190 events simultaneously. As a result, each benchmark was run multiple times to collect measurements for all events. Iterations for the same benchmark had identical configuration, with the exception of monitored events. The sample standard deviation of execution times was calculated for each execution. We define the *relative standard deviation* to be σ/μ, where σ denotes the standard deviation and μ is the mean. The maximum maximum standard deviation across all executions is 0.96%. Hence all the iterations of the same benchmark were similar in terms of execution time.

4.1.4 Data Representation

Data mining techniques frequently require input to be represented as matrices. This part describes how we represent sample counts for events as a matrix. Each application is understood to be a single process which may contain multiple threads, and we always use a time slice of 1 second in this thesis.

Suppose we measured n events $e_1, \ldots, e_n$ for a process p on a machine with m cores (note that the threads of p may execute on multiple cores simultaneously), and the execution time was t seconds. The execution time can be divided into time slices of equal length. The delta of an event is defined to be the increase in the event count during a specific time slice (deltas are always nonnegative integers). Since we have separate counters for each core, they must be aggregated for every time slice. Let c_{ijk} denote the increase of e_i's event count during the jth time slice on core k. Thus the delta of e_i for the jth time slice is

$$c_{ij} = \sum_{k=1}^{m} c_{ijk}.$$

Then the deltas of all events for the jth time slice can be represented as a column vector

$$\boldsymbol{x}_j = \begin{bmatrix} c_{1,j} \\ \vdots \\ c_{n,j} \end{bmatrix}.$$

By defining column j of matrix X to be the vector of deltas for time slice j, we obtain

$$X = \begin{bmatrix} \boldsymbol{x}_1 & \ldots & \boldsymbol{x}_t \end{bmatrix}.$$

In other words, row i of X is simply a vector containing e_i's deltas in chronological order.

In addition, data from several processes may be concatenated together. Assume we have matrices X_p and X_q for processes p and q, respectively. Then define

$$X_{pq} = \begin{bmatrix} X_p & X_q \end{bmatrix}$$

to be the combined matrix. Clearly, this can be done for an arbitrary number of processes.

For the remainder of this chapter, matrix X is defined to contain data for all executions of applications, i.e.,

$$X = \begin{bmatrix} X_{p_1} & \dots & X_{p_N} \end{bmatrix},$$

where N is the number of executions. We also exclude rows which only contain 0, i.e., events whose deltas are all 0. There are a number of possible explanations for this phenomenon. There may be a software or hardware bug affecting specific events, or the events were simply not triggered.

There are 164 remaining events out of 190, and hence $X \in \mathbb{Z}_{\geq 0}^{164 \times N}$, where $\mathbb{Z}_{\geq 0}$ denotes the set of nonnegative integers. The excluded events are listed in Appendix B. Note that any conclusions obtained from operations on X should be valid for all applications as X contains data from every execution in the application set.

We take the sum of counts of all threads for each application because we are primarily concerned with the overall performance and not the progress of individual threads. For example, in cases where only a few threads are affected by interference, we need not be overly concerned. However, carrying out aggregation of event counts implies we lose fine-grained control, so there are also disadvantages of this approach. An alternative is to analyze each thread on its own.

4.2 Dimensionality Analysis

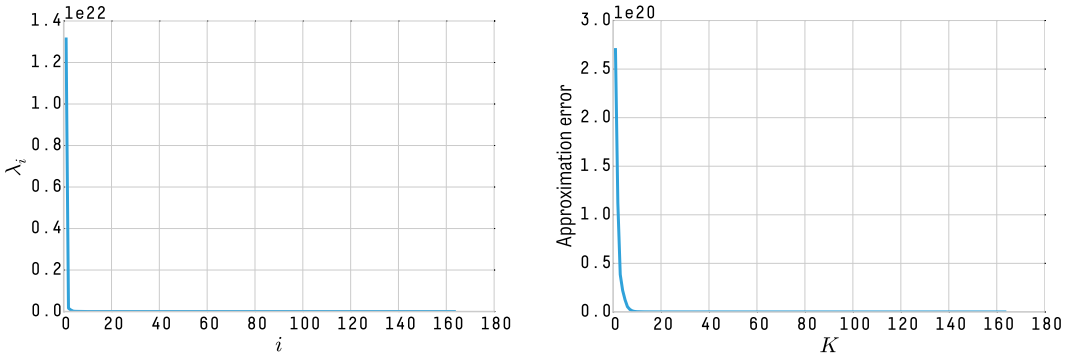
In this section, we apply principal component analysis (PCA) to find the *intrinsic dimensionality*, i.e., the number of variables determining the data points. PCA projects each column $\boldsymbol{x}_i \in \mathbb{R}^D$ of $X \in \mathbb{R}^{D \times N}$ onto a subspace to obtain $\tilde{\boldsymbol{x}}_i \in \mathbb{R}^K$, and the objective is to select the smallest possible $K \leq D$ such that each column retains its "features" [22]. Here $D = 164$ since there are 164 events in X.

Recall we have multiple runs of each application. Suppose p' and p are executions of an application with and without interference, respectively. Define

$$X_{pp'} = \begin{bmatrix} X_p & X_{p'} \end{bmatrix}.$$

After reducing the event space, we need to be able to differentiate whether vectors originate from p or p' to detect interference, and selecting a higher K obviously increases this probability. However, if K is small, then not all 164 events need to be measured to obtain sufficient data. Moreover, K may be used to determine the number of groups when performing event grouping.

4.2.1 Subspace Dimensionality and Approximation Error



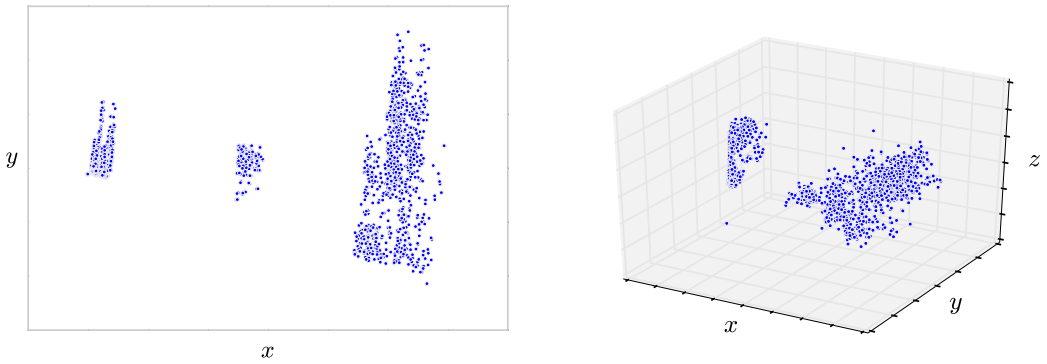
(a) Eigenvalues (b) Approximation error

Figure 4.1: Eigenvalues and approximation error of PCA.

As usual, matrix X was mean-centered before applying PCA by obtaining the eigendecomposition of the covariance matrix with functions from NumPy [35, 36, 37]. The results are shown in Figure 4.1. We are primarily concerned with how the approximation error changes according to K. There is a visible "knee" between $K = 1$ and $K = 10$ since the error decreased dramatically in this range, which implies the data points (i.e., the columns of X) can be orthogonally projected onto a low-dimensionality subspace. Assuming we get the maximum error when $K = 1$, further inspection reveals selecting $K = 4$ returns an error of less than 10% (8.16%). Increasing K to 7 reduces the error to below 1% (0.95%).

4.2.2 Projection of Data Points onto Subspaces

In order to visualize the data, the points are projected onto $\mathbb{R}^2$ and $\mathbb{R}^3$ in Figure 4.2. We see the points retain their "structure" when K is reduced from 3 to 2. This suggests $K = 3$ may be a valid choice for the intrinsic dimensionality of the event space.



(a) $\mathbb{R}^2$ (b) $\mathbb{R}^3$

Figure 4.2: Projection of data points onto subspaces.

However, we need to verify two additional conditions. First, points originating from the same execution should be clustered together. All of the applications in the application set only have one phase, so in theory the event deltas should be similar for all time slices. One minor exception is be SORT as it may behave differently as it makes progress. Second, different executions should be distinguishable. Executions exhibiting interference normally have irregular counter values. Next, we plot each application in a separate graph.

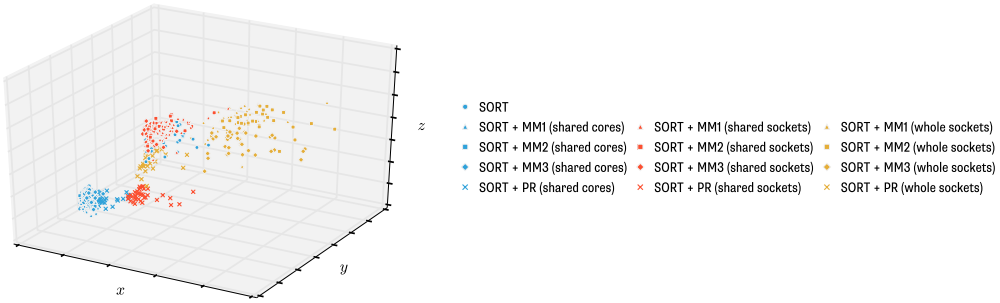


Figure 4.3: Data points of SORT.

Figure 4.3 exclusively contains data points belonging to executions of SORT. Each execution can be identified by a unique shape and colour. For instance, blue circles denote an execution of SORT without interference, and red squares denote an execution of with MM2 causing interference with the *shared sockets* partitioning scheme. Observe that data points from the same execution are frequently close to each other although there are some outliers, and certain executions exhibit more variance. In addition, executions experiencing more severe degradations are easily distinguishable—executions with the *shared cores* scheme form a cluster which is further removed from other executions.

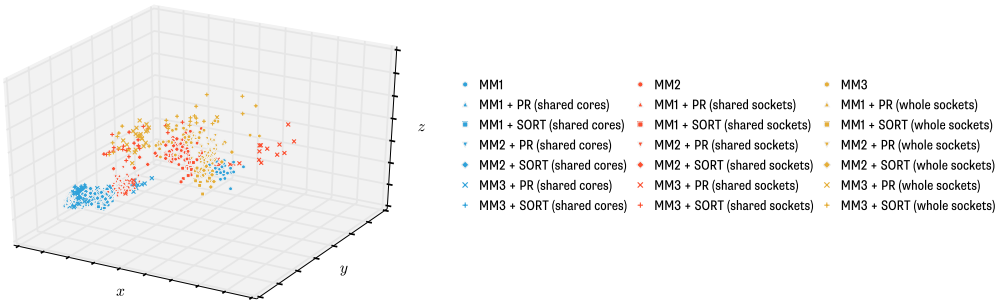


Figure 4.4: Data points of MM.

The same process is done for MM and PR in Figures 4.4 and 4.5, respectively. The MM application is also a single-threaded process like SORT. However, we experimented with three input matrices of different sizes as they act on the LLC differently. Smaller matrices which fit inside the cache have very low miss rates while larger ones have higher miss rates. Regardless of MM's effect on the LLC,

we can draw similar conclusions to SORT since executions experiencing more interference form an distinct cluster.

In contrast, the reaction of PR to interference is a bit more varied since it is a multithreaded program. When PR experiences interference caused by a single-threaded program (e.g., SORT and MM), the performance degradation is not as pronounced. We also see a serious slowdown when two PR instances are coscheduled with the *shared cores* scheme. Therefore, it is possible to conclude there are varying *degrees* of interference—some severe and others less so—which can be detected and visualized through performance counter measurements.

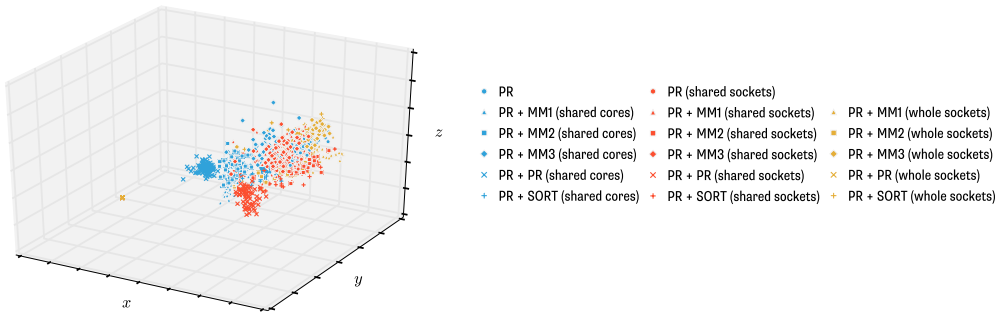


Figure 4.5: Data points of PR.

Clearly, if these results can be obtained from a 3D subspace, then all of the above (and possibly more) should be achievable with subspaces containing more dimensions. It is important to note the chosen value for K, i.e., the intrinsic dimensionality or number of principal components, does not directly translate to the number of events which should be measured. However, it does suggest only a fraction of events are required to preserve information from the complete set of events. Finally, we wish to emphasize the generality of the dimensionality analysis again. PCA was performed on data points originating from all applications in the application set as well as executions with and without interference. Thus the selected K is not specific to any application, nor is it limited to a specific type of interference. This generality will continue to apply to all following results.

4.3 Event Similarity

By applying PCA, it is possible to show the intrinsic dimensionality of the event space is less than 10. However, PCA only tries to preserve information when mapping data points to a subspace and does not suggest the dimensions which are important, i.e., PCA does not determine the events which should be measured. In other words, K provides an estimate of *how much* the event space may be reduced, but we still need to explore *how* to perform the reduction.

Naturally, clustering is one way of proceeding, and there is certainly no shortage of clustering algorithms; one classic example is k-means clustering. The core of such algorithms involves defining *similarity* between objects, e.g., the Euclidean

distance between points is often used with k-means. In this section, we will discuss (1) how similarity is formally applied to the matrix X, (2) different notions of similarity, and (3) evaluate their suitability for X.

Recall each row in X maps to an event. More specifically, the deltas of an event form a row, where each delta corresponds to one time slice. Thus, every event can be represented by the entire row, which is simply a vector in $\mathbb{R}^N$. Suppose there are two events e_i and e_j, and their corresponding vectors are $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. Then we define e_i and e_j to be *similar* if and only if $\boldsymbol{x}$ and $\boldsymbol{y}$ are "close." Since similarity is defined as the "distance" between two vectors, there are numerous mathematical tools at our disposal.

4.3.1 Euclidean Distance

The Euclidean distance is one of the most commonly used definitions of distance. Let $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$. Write $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, \ldots, y_n)$. The **Euclidean distance** is defined as

$$d(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}.$$

A smaller value indicates the vectors are "closer."

Application of the Euclidean distance to events is straightforward as $\boldsymbol{x}$ and $\boldsymbol{y}$ are simply replaced by event vectors, and d reflects the "closeness" of the corresponding events. However, there are two caveats. First, the *curse of dimensionality* comes into play as the dimension increases [4]. In short, a high dimensionality causes points to become sparse, making finding similar events problematic. Second, the Euclidean distance does not take the magnitude of deltas into account. To illustrate, there may be two different events where one event vector is a constant multiple of the other. Some would consider them to be similar, but if the constant is too large, the Euclidean distance would not be able to discover this.

4.3.2 Correlation Coefficient

The correlation coefficient expresses the correlation between two variables. Let X and Y be variables. Suppose $\{x_1, \ldots, x_n\}$ and $\{y_1, \ldots, y_n\}$ are samples of X and Y, respectively. The **correlation coefficient** is defined as

$$r(X, Y) = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^{n} (y_i - \bar{y})^2}},$$

where $\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$ and $\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i$. The value always lies in $[-1, 1]$, and $r > 0$ indicates positive correlation while $r < 0$ signifies negative correlation. We are not currently concerned with orientation, so a value further away from 0 indicates the variables are more similar.

The correlation coefficient is applied by substituting events for variables and the corresponding event vectors for samples. The closer $|r|$ and 1 are, the "closer" the variables. In contrast to the Euclidean distance, the correlation coefficient is oblivious to the magnitude of deltas. Thus if the cache miss rate is stable, the cache accesses and cache misses will be correlated. Since there may also be cases where the magnitude is relevant, this particular characteristic can be both an asset and a handicap.

Figure 4.6 is a heatmap displaying the correlation between events in a matrix. The majority of correlated events have positive correlation. Those with negative correlation are often only weakly correlated. Furthermore, the white strips show there are a number of events which are mostly uncorrelated with others. Also notice the heatmap is symmetric across the diagonal since the order of the events is irrelevant when calculating the correlation coefficient. An interesting aspect which will be explored later is the orientation of correlations, but take note most negative correlations are weak, i.e., the absolute values of the coefficients are close to 0.

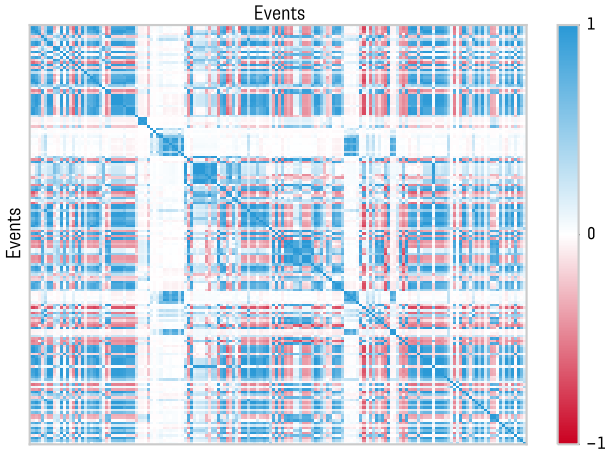


Figure 4.6: Heatmap of events displaying correlation.

4.4 Event Correlation

One possible application of correlation coefficients is the formulation of an equivalence relation. We define **correlation** to be a binary relation over the set of events. Events e_i and e_j are **correlated** if and only if $r(e_i, e_j) \geq c$, where $c \in [-1, 1]$ is a constant named the **correlation cutoff**. Correlated events may be denoted by $e_i \sim e_j$. For now, assume $c = 0.9$.

The next logical step would be to test whether correlation satisfies the three conditions of an equivalence relation: reflexivity, symmetry, and transitivity. Correlation is reflexive since $r(e_i, e_i) = 1 \geq c$, and $r(e_i, e_j) = r(e_j, e_i)$ implies it is also symmetric. Unfortunately, transitivity does not hold. There exist events e_i, e_j, and e_k such that $e_i \sim e_j$ and$e_j \sim e_k$ but $e_i \not\sim e_k$. The unordered pair $\{e_i, e_k\}$ is termed a **transitivity violation**. For $c = 0.9$, there are 540 violations. The number of viola-

tions is dependent on c; higher values lead to fewer violations while lower values have more.

4.4.1 Graph Representation

Observe the correlation relation has a graph representation. Let V be the set of events and $\{e_i, e_j\} \in E$ if and only if $e_i \sim e_j$. Then $G = (V, E)$ is an undirected graph. This opens up the possibility of applying graph algorithms.

The structure of the graph itself is also worth examining, especially the degree of nodes. Given an event e_i, the number of events correlated with e_i is simply the degree of the corresponding node in G and is denoted by $\deg(e_i)$.

Table 4.2 lists the degree of each event. The size refers to the number of events with the given degree. Refer to Appendix B to translate the event numbers in groups into event names. It is quite noticeable there are 40 events which are completely uncorrelated with other events. It remains to be seen whether these events are predictive of interference although they appear to be less commonly used events. On the other hand, a sizeable proportion of the remaining events are correlated with a few dozen events.

Degree	Size	Events
40	1	166
39	1	158
37	1	87
36	7	7, 8, 11, 23, 77, 83, 135
35	19	3, 28, 29, 30, 31, 65, 84, 90, 94, 107, 153, 154, 155, 157, 159, 172, 174, 182, 185
34	4	16, 18, 86, 108
33	1	187
24	1	32
19	1	13
18	8	64, 75, 93, 97, 99, 103, 104, 105
17	2	95, 131
16	4	98, 129, 145, 165
15	3	27, 124, 147
14	4	26, 96, 149, 152
13	2	1, 22
12	7	2, 17, 33, 34, 35, 148, 150
11	1	91
10	1	113
9	1	156
8	3	63, 68, 73
7	6	49, 67, 69, 70, 74, 120
6	6	50, 52, 56, 58, 118, 179
5	5	5, 114, 115, 130, 177
4	3	100, 101, 128
3	11	6, 21, 71, 72, 76, 79, 82, 85, 102, 106, 171
2	7	9, 12, 20, 37, 38, 39, 117
1	14	4, 10, 14, 15, 78, 126, 127, 132, 141, 142, 146, 160, 178, 186
0	40	19, 24, 25, 36, 40, 41, 42, 43, 44, 51, 57, 80, 81, 88, 89, 92, 109, 110, 111, 112, 116, 119, 121, 125, 144, 151, 161, 162, 163, 164, 169, 170, 173, 175, 176, 180, 181, 184, 188, 190

Degree	Size	Events

Table 4.2: Degree of events.

4.5 Event Grouping

In this section, we present applications of well-known algorithms which utilize previous notions of similarity to group events together. Both the Euclidean distance and the correlation coefficient are used. The first three grouping methods apply graph algorithms to the graph representation whereas the remaining two methods are based on k-means clustering.

4.5.1 Connected Components

A good starting point is to associate correlated events, i.e., if $e_i \sim e_j$, then they *may* be placed in the same group. Now the question is whether we require transitivity to hold in the same group. Let S be an event group, and fix $e_i \in S$. Do we require $e_i \sim e_j$ for all $e_j \in S$? Or is it sufficient if there exists $e_j \in S$ such that $e_i \sim e_j$? Both are plausible methods which will be surveyed.

We relax the transitivity requirement within groups for now and stipulate if $e_i \sim e_j$, then e_i and e_j belong in the same group. Using the graph representation, the problem can be solved with the connected components algorithm. The groups are shown in Table 4.3. Each group is assigned a group number and displayed as a set. In addition, groups of the same size are shown in the same row. For event names, see Appendix B.

Uncorrelated events are underlined in the table, so it is evident all 40 of them each form a singleton. There are only 10 non-singleton groups which cover the remainder of the 122 events, which is a good reduction. However, selecting a representative event for each group is not straightforward as transitivity within groups is not guaranteed. We could select the event with the largest degree in every group, or the event whose sum of correlation coefficients with elements in the same group is greatest.

We have also experimented with ignoring the orientation of correlations, i.e., taking the absolute value of correlation coefficients. However, negative correlations are generally weak, so this made little or no difference in the resulting groups. This holds for other grouping methods which employ the correlation coefficient.

Group No(s).	Size	Group(s)
1	56	{1, 2, 3, 4, 7, 8, 11, 13, 16, 18, 20, 21, 22, 23, 28, 29, 30, 31, 32, 33, 34, 35, 65, 76, 77, 83, 84, 86, 87, 90, 91, 94, 96, 107, 108, 113, 114, 115, 135, 148, 149, 150, 152, 153, 154, 155, 157, 158, 159, 160, 166, 172, 174, 182, 185, 187}
2	30	{5, 12, 17, 26, 27, 64, 75, 93, 95, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 124, 129, 130, 131, 145, 147, 165, 177, 178, 179, 186}
3	13	{63, 67, 68, 69, 70, 71, 72, 73, 74, 78, 79, 82, 156}
4	8	{49, 50, 52, 56, 58, 117, 118, 120}
5	5	{6, 85, 126, 128, 171}

Group No(s).	Size	Group(s)
6–7	3	{9, 10, 14}, {37, 38, 39}
8–10	2	{15, 127}, {132, 146}, {141, 142}
11–50	1	{19}, {24}, {25}, {36}, {40}, {41}, {42}, {43}, {44}, {51}, {57}, {80}, {81}, {88}, {89}, {92}, {109}, {110}, {111}, {112}, {116}, {119}, {121}, {125}, {144}, {151}, {161}, {162}, {163}, {164}, {169}, {170}, {173}, {175}, {176}, {180}, {181}, {184}, {188}, {190}

Table 4.3: Event groups generated by CC.

Figure 4.7 shows the event heatmap, but the events are rearranged so that event groups appear as clusters. Each group is denoted by a grey bounding box, and the largest group is in the top left corner. The clusters are generally blue thus showing events in the same group are mostly correlated. There are a few lighter patches in clusters caused by not enforcing the transitivity of correlation in groups. The singletons are also visible in the bottom right corner.

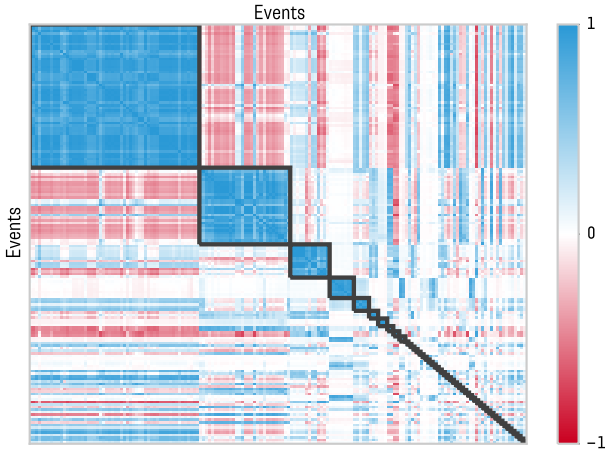


Figure 4.7: Event heatmap with CC groups.

4.5.2 Minimum Clique Cover

Now consider imposing the restriction that any two events in the same group must be correlated. Then the problem is essentially partitioning the vertices into cliques. Contrary to CC, any event in a group should be representative of all remaining events in the group, which implies the number of events which need to be measured to infer information about the entire event space is equivalent to the number of groups. Finding a minimum clique cover minimizes the number of groups.

Table 4.4 presents a Min-Clique grouping. Unlike CC, this method does not have a unique solution. Groups are smaller in comparison due to a stronger constraint. There are now 30 non-singleton groups instead of 10. The main benefit of this method is any event is characteristic of the entire group it belongs to. However, we still have an insufficient number of counters to measure all groups simultaneously (recall recent Intel processors have 3 fixed counters and 8 general-purpose counters). We must also take into account the clique cover problem is

NP-complete, but there are approximation algorithms available. One such example is to perform greedy vertex colouring of the complement graph.

Group No(s).	Size	Group(s)
1	18	{3, 7, 8, 11, 13, 84, 87, 90, 107, 135, 153, 154, 155, 157, 159, 172, 182, 185}
2	12	{17, 26, 27, 64, 75, 93, 97, 99, 103, 104, 105, 124}
3–4	7	{16, 18, 23, 32, 83, 86, 187}, {28, 29, 30, 31, 65, 108, 174}
5	6	{33, 34, 35, 114, 115, 148}
6–7	5	{50, 52, 56, 58, 118}, {98, 129, 145, 165, 179}
8–12	4	{63, 68, 71, 73}, {1, 21, 22, 91}, {2, 150, 158, 166}, {67, 69, 70, 74}, {100, 101, 102, 106}
13–19	3	{6, 85, 171}, {20, 149, 152}, {37, 38, 39}, {72, 79, 156}, {49, 117, 120}, {76, 94, 113}, {95, 131, 147}
20–30	2	{9, 10}, {4, 96}, {78, 82}, {15, 127}, {126, 128}, {12, 130}, {132, 146}, {141, 142}, {77, 160}, {177, 178}, {5, 186}
31–71	1	{14}, {19}, {24}, {25}, {36}, {40}, {41}, {42}, {43}, {44}, {51}, {57}, {80}, {81}, {88}, {89}, {92}, {109}, {110}, {111}, {112}, {116}, {119}, {121}, {125}, {144}, {151}, {161}, {162}, {163}, {164}, {169}, {170}, {173}, {175}, {176}, {180}, {181}, {184}, {188}, {190}

Table 4.4: Event groups generated by Min-Clique.

The rearranged heatmap can be found in Figure 4.8. Apart from the noticeable decrease in cluster sizes and increased number of clusters, the colours of clusters are more consistent compared to CC.

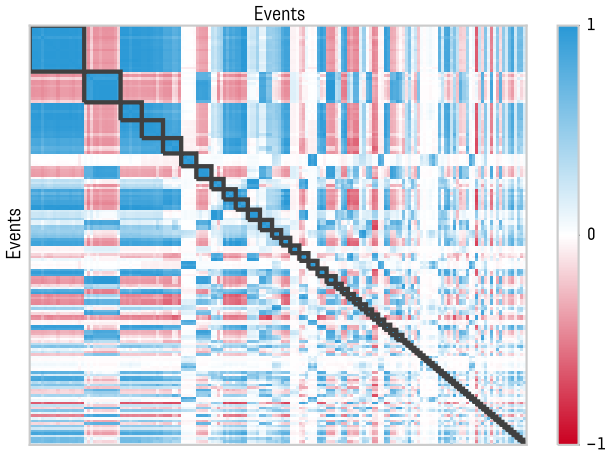


Figure 4.8: Event heatmap with Min-Clique groups.

4.5.3 Greedy Maximum Clique Cover

Instead of minimizing the number of groups, an alternative approach is to maximize the number of events covered without multiplexing counters. In other words, we would like to maximize the size of the n largest groups, where n is the number of counters. Assuming we continue to require any two events in the same group be correlated, then a greedy algorithm would be to find a maximum clique (i.e., a clique with maximum size), remove it from the graph, and repeat until there are

no remaining vertices.

The result is shown in Table 4.5. With 8 general-purpose counters, we are able to cover 86 out of 164 events without multiplexing. This is a 34% increase from 64 events compared to Min-Clique. The number of groups, however, is similar. The events have been shuffled so that more are placed in larger groups. Since maximum cliques are not always unique, there may be other valid groupings. Furthermore, finding a maximum clique is also NP-complete, but approximation algorithms exist.

Group No(s).	Size	Group(s)
1	33	{3, 7, 8, 11, 16, 18, 23, 28, 29, 30, 31, 65, 77, 83, 84, 86, 87, 90, 94, 107, 135, 153, 154, 155, 157, 158, 159, 166, 172, 174, 182, 185, 187}
2	15	{64, 75, 93, 95, 97, 98, 99, 103, 104, 105, 129, 131, 145, 147, 165}
3	11	{1, 2, 22, 33, 34, 35, 91, 148, 149, 150, 152}
4	8	{63, 67, 68, 69, 70, 73, 74, 156}
5	7	{49, 50, 52, 56, 58, 118, 120}
6–8	4	{5, 27, 124, 130}, {6, 85, 128, 171}, {100, 101, 102, 106}
9–10	3	{37, 38, 39}, {72, 79, 82}
11–19	2	{4, 96}, {9, 10}, {15, 127}, {17, 26}, {76, 113}, {114, 115}, {132, 146}, {141, 142}, {177, 178}
20–73	1	{12}, {13}, {14}, {19}, {20}, {21}, {24}, {25}, {32}, {36}, {40}, {41}, {42}, {43}, {44}, {51}, {57}, {71}, {78}, {80}, {81}, {88}, {89}, {92}, {108}, {109}, {110}, {111}, {112}, {116}, {117}, {119}, {121}, {125}, {126}, {144}, {151}, {160}, {161}, {162}, {163}, {164}, {169}, {170}, {173}, {175}, {176}, {179}, {180}, {181}, {184}, {186}, {188}, {190}

Table 4.5: Event groups generated by Greedy-Max-Clique.

Figure 4.9 is the corresponding heatmap for this method. The colours of clusters are similar to Min-Clique as the same condition is enforced for Greedy-Max-Clique.

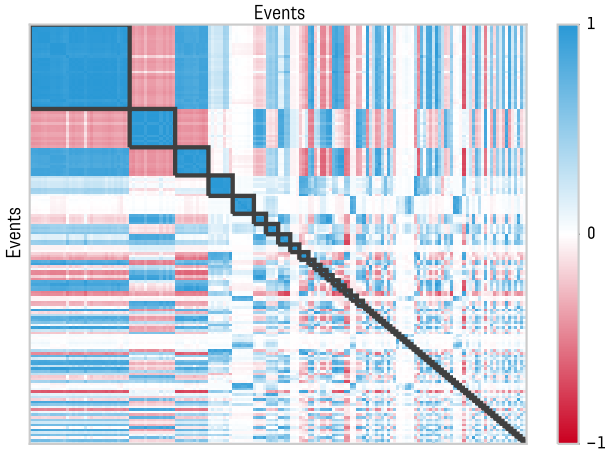


Figure 4.9: Event heatmap with Greedy-Max-Clique groups.

4.5.4 *k*-means Clustering with Euclidean Distance

The number of groups generated by previous graph algorithms cannot be predetermined even though a higher c generally leads to more groups while a lower value results in fewer groups. In cases where we wish to specify an exact number, k-means clustering is an established way of grouping objects [30] into k sets. We implemented the standard k-means algorithm with the following specifications:

- Initialization: We choose k points randomly to serve as the initial centroids.
- Termination: There is no threshold for termination. The process stops only when assignments are stable.
- Distance function: A custom distance function may be used.

Random initialization runs the risk of having a poor selection of initial centroids. Consequently, k-means is repeated 50 times, and we keep the best result. Optimality evaluated with the same function which computes the distance between points and centroids. We aim to minimize the sum of distances from every point to the centroid it is assigned. Obviously, the randomness of initialization implies k-means does not have a unique solution.

The value of k greatly influences the grouping outcome. While the purpose of this section is not to decide on the ideal value of k, we experiment with $k = 10$ and $k = 75$ to assess k-means itself. A value of 10 was chosen since it is approximately equal to the number of available counters whereas 75 is similar to number of groups returned by methods based on graph algorithms.

The results of k-means with the Euclidean distance for k = 10 and 75 are in Tables 4.6 and 4.6, respectively. For $k = 10$, the largest group contains over two-thirds of all events, including the majority of uncorrelated events. Such a presence indicates that the Euclidean distance and the correlation coefficient have very distinct notions of similarity. The remaining groups are relatively small. In fact, the difference is an order of magnitude, so $k = 10$ may be too small.

Group No(s).	Size	Group(s)
1	115	{3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 23, 24, 25, 26, 27, 28, 29, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 49, 50, 51, 52, 56, 57, 58, 65, 77, 80, 81, 83, 84, 85, 86, 92, 94, 95, 96, 97, 98, 99, 103, 104, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 124, 125, 126, 127, 128, 129, 130, 135, 141, 142, 144, 145, 146, 148, 157, 159, 160, 161, 162, 163, 164, 165, 169, 170, 171, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 184, 185, 188, 190}
2	13	{64, 75, 76, 93, 100, 101, 102, 105, 106, 131, 147, 151, 186}
3–4	7	{63, 67, 68, 69, 70, 82, 156}, {71, 72, 73, 74, 78, 79, 132}
5	6	{1, 2, 20, 21, 22, 91}
6	5	{89, 153, 154, 155, 172}
7	4	{18, 30, 166, 187}
8–9	3	{87, 90, 158}, {149, 150, 152}
10	1	{88}

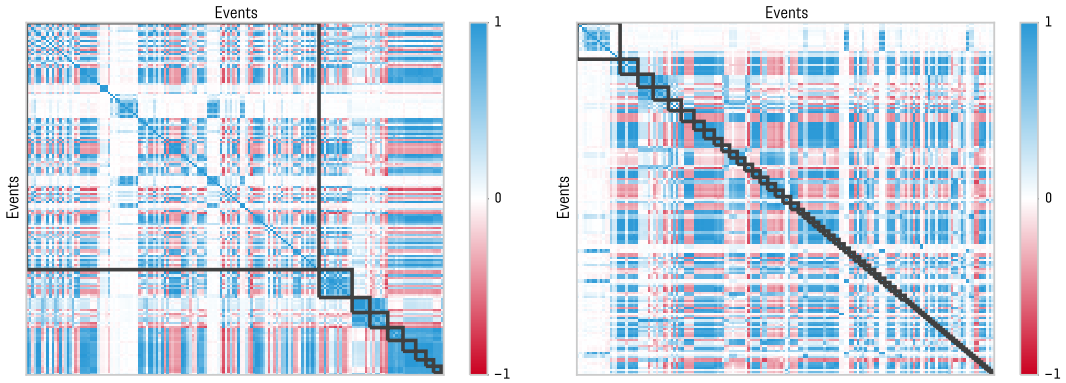
Table 4.6: Event groups generated by k-means ($k = 10$, dist = Euclid).

For $k = 75$, the sizes of groups are closer to methods derived from graph algorithms. This includes a long "tail" of many singleton groups although only approximately half are uncorrelated events. A large value of k does not appear to be effective either as group sizes are rather small.

Group No(s).	Size	Group(s)
1	17	{41, 42, 43, 44, 50, 51, 52, 56, 57, 58, 121, 141, 142, 173, 175, 181, 182}
2	7	{7, 8, 36, 84, 159, 174, 185}
3–4	6	{63, 68, 69, 70, 82, 156}, {71, 72, 73, 74, 78, 79}
5–6	5	{10, 15, 80, 127, 160}, {94, 97, 99, 103, 104}
7–8	4	{3, 11, 135, 157}, {153, 154, 155, 172}
9–19	3	{2, 21, 91}, {4, 9, 14}, {5, 12, 130}, {17, 27, 124}, {25, 125, 169}, {28, 29, 31}, {75, 102, 106}, {93, 105, 147}, {98, 145, 163}, {111, 164, 184}, {129, 177, 178}
20–40	2	{1, 22}, {6, 85}, {16, 86}, {18, 187}, {23, 83}, {24, 108}, {30, 166}, {33, 34}, {37, 38}, {49, 119}, {64, 131}, {96, 179}, {107, 161}, {109, 165}, {112, 170}, {114, 115}, {118, 120}, {144, 180}, {150, 152}, {151, 186}, {171, 188}
41–75	1	{13}, {19}, {20}, {26}, {32}, {35}, {39}, {40}, {65}, {67}, {76}, {77}, {81}, {87}, {88}, {89}, {90}, {92}, {95}, {100}, {101}, {110}, {113}, {116}, {117}, {126}, {128}, {132}, {146}, {148}, {149}, {158}, {162}, {176}, {190}

Table 4.7: Event groups generated by k-means ($k = 75$, dist = Euclid).

The heatmaps for both values of k are displayed in Figure 4.10. The largest group for $k = 10$ gives visual confirmation that the Euclidean distance is a very different metric compared to correlation. Interestingly, events in smaller groups are somewhat correlated. Common to all methods so far is the "tail" of small groups. This suggests there are a number of events which cannot be grouped.



(a) $k = 10$ (b) $k = 75$

Figure 4.10: Event heatmaps with k-means (dist = Euclid) groups.

4.5.5 k-means Clustering with Correlation

The disadvantages of the Euclidean distance were previously discussed, and it is evident such a notion of similarity is not always desirable. Therefore, we suggest an alternative application of k-means with the correlation coefficient as the distance function.

First, if the distance function returns a smaller value, then it indicates the objects are closer. On the other hand, variables which are more positively correlated have a greater correlation coefficient. Consequently, we define the distance between two events to be the additive inverse of the correlation coefficient. Second, events cannot be "averaged" to obtain a centroid, so it is necessary to select a representative event from each group. Technically, such events are *medoids* instead of *centroids*, but we will continue to use the term centroid. The centroid is the event whose total distance to other events in the same cluster is minimized.

We perform clustering with k = 10 and k = 75. The groupings are shown in Tables 4.8 and 4.8, respectively. For $k = 10$, the largest group is considerably smaller compared to k-means with the Euclidean distance while the remaining groups are also larger. This signifies the groups are more balanced. The uncorrelated events are more evenly distributed among groups.

Group No(s).	Size	Group(s)
1	73	{1, 2, 3, 4, 7, 8, 11, 13, 16, 18, 19, 20, 21, 22, 23, 28, 29, 30, 31, 32, 33, 34, 35, 65, 77, 80, 83, 84, 86, 87, 88, 89, 90, 91, 92, 94, 96, 107, 108, 110, 113, 114, 115, 135, 144, 148, 149, 150, 152, 153, 154, 155, 157, 158, 159, 160, 161, 162, 166, 169, 172, 173, 174, 175, 176, 180, 181, 182, 184, 185, 187, 188, 190}
2	34	{5, 12, 15, 17, 26, 27, 40, 64, 75, 93, 95, 97, 98, 99, 103, 104, 105, 111, 112, 124, 127, 129, 130, 131, 145, 147, 151, 163, 165, 170, 177, 178, 179, 186}
3	14	{9, 14, 25, 36, 63, 67, 68, 69, 70, 71, 73, 74, 81, 156}
4	11	{42, 49, 50, 52, 56, 58, 117, 120, 121, 141, 142}
5	10	{10, 24, 76, 100, 101, 102, 106, 109, 116, 164}
6	7	{72, 78, 79, 82, 125, 132, 146}
7	5	{6, 85, 126, 128, 171}
8	4	{41, 43, 44, 57}
9–10	3	{37, 38, 39}, {51, 118, 119}

Table 4.8: Event groups generated by k-means ($k = 10$, dist = corr).

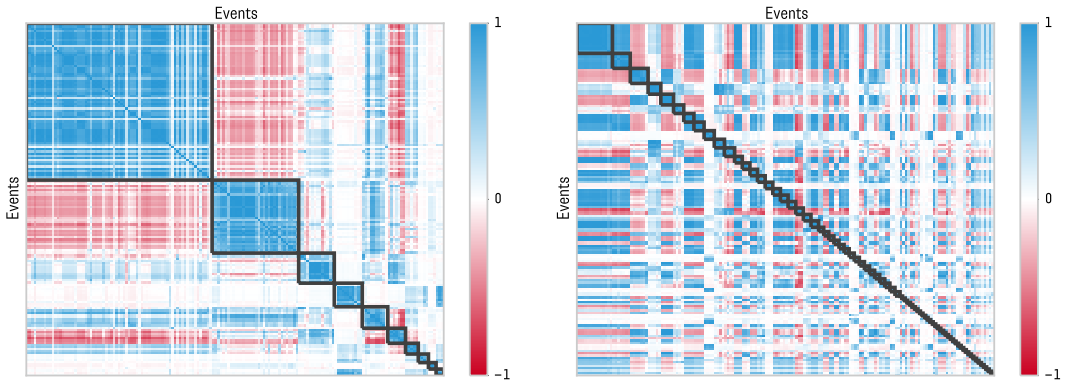
Group No(s).	Size	Group(s)
1	14	{7, 8, 84, 87, 90, 107, 153, 154, 155, 158, 159, 172, 180, 182}
2–3	7	{1, 2, 20, 22, 149, 150, 152}, {98, 129, 145, 165, 177, 178, 179}
4–5	5	{63, 69, 70, 73, 74}, {64, 93, 97, 99, 131}
6–11	4	{9, 10, 14, 25}, {28, 29, 30, 31}, {33, 34, 35, 148}, {49, 50, 56, 120}, {67, 68, 71, 156}, {72, 78, 79, 82}
12–22	3	{3, 11, 157}, {5, 130, 186}, {23, 32, 83}, {24, 76, 102}, {37, 38, 39}, {65, 96, 174}, {77, 108, 113}, {94, 187, 188}, {125, 132, 146}, {126, 128, 171}, {135, 166, 185}
23–38	2	{6, 85}, {15, 127}, {16, 86}, {17, 124}, {18, 88}, {21, 91}, {43, 57}, {51, 121}, {75, 163}, {80, 81}, {100, 101}, {105, 170}, {109, 116}, {114, 115}, {118, 119}, {162, 164}
39–75	1	{4}, {12}, {13}, {19}, {26}, {27}, {36}, {40}, {41}, {42}, {44}, {52}, {58}, {89}, {92}, {95}, {103}, {104}, {106}, {110}, {111}, {112}, {117}, {141}, {142}, {144}, {147}, {151}, {160}, {161}, {169}, {173}, {175}, {176}, {181}, {184}, {190}

Table 4.9: Event groups generated by k-means ($k = 75$, dist = corr).

For $k = 75$, we see a distribution of group sizes akin to that of k-means with

the Euclidean distance. However, it is evident the uncorrelated events are mainly in singleton groups or groups of size 2.

Figure 4.11 shows events within groups are more correlated with each other, and the intensity increases with k. It appears that the k-means algorithm has similar tendencies with respect to group sizes regardless of the chosen distance function. Therefore, an suitable value of k must be selected to obtain desired group sizes.



(a) $k = 10$ (b) $k = 75$

Figure 4.11: Event heatmaps with k-means (dist = corr) groups.

4.6 Comparison of Grouping Methods

Up to now, we have not discussed the events themselves in different groupings since a side by side comparison would present a better view than discussing them individually. Groupings serve two purposes: (1) they reduce the number of events which need to be measured, and (2) they allow users without domain knowledge to have a better understanding of events.

4.6.1 Validity and Effectiveness

For the first objective, we need to confirm the groupings are both *valid* and *effective*. By valid, we mean the events in each group must be related to each other according to the notion of similarity adopted by the grouping method. We examine the following set of four events in the groupings:

- `L2_LINES_IN:ALL` (event 22)
- `LLC_REFERENCES` (event 1)
- `LLC_MISSES` (event 2)
- `OFFCORE_RESPONSE_0:ANY_REQUEST:LLC_MISS_LOCAL` (event 91)

`L2_LINES_IN:ALL` counts the number of filled L2 lines, so it should be representative of L2 cache misses. Each L2 cache miss becomes an L3 cache reference, which is counted by `LLC_REFERENCES`.[1] Thus the two event counts should be

[1]For this particular processor, the last-level cache is the L3 cache.

similar. In all our applications, the L3 cache miss rate is relatively stable, so the number of references is proportional to the number of misses. By definition of the correlation coefficient, they are correlated with each other. However, the same cannot be said of the Euclidean distance since it is affected by the magnitude of counts. Finally, the last event counts the number of memory accesses, which should also be close to the number of L3 cache misses. In summary, we expect the following to hold for event counts:

- `L2_LINES_IN:ALL` $\approx$ `LLC_REFERENCES`
- `LLC_REFERENCES` $\propto$ `LLC_MISSES`
- `LLC_MISSES` $\approx$ `OFFCORE_RESPONSE_0:ANY_REQUEST:LLC_MISS_LOCAL`

With Greedy-Max-Clique, all four are placed in the same group along with several additional events (c.f. Table 4.5). In contrast, Min-Clique put `LLC_MISSES` in a group different from the other three (c.f. Table 4.4). CC is a generalization of clique-based methods, so naturally all were placed in the same group (c.f. Table 4.3).

From this particular example, we see the graph-based methods generate valid groupings, for correlated events are associated with each other. In particular, it is not affected by the magnitude of event counts as long as they are correlated. However, Min-Clique did "misplace" one event—while the grouping is valid, we expected all four to be put together. This can be attributed to the fact the implementation simply enumerated all possible groupings, and selected one with the lowest number of groups. Namely, it tries to find *fewer* groups and not *larger* groups. On the other hand, Greedy-Max-Clique tries to find the maximum clique, so there is a higher possibility of correlated events actually being placed together in larger groups.

For k-means with $k = 10$, the four events were grouped together for both Euclidean distance and correlation. For $k = 75$, it is unable to associate `LLC_MISSES` and `LLC_REFERENCES` using the Euclidean distance, which is affected by the magnitude of events. In addition, correlation places the last event in a separate cluster, but this appears to be the result of specifying a higher number of clusters and selection of centroids.

4.6.2 Inference of Similar Events

While our final goal is to move toward automation and remove the need for expert knowledge, groupings still help users form useful conclusions regarding events. There are often similar events with minor differences, and users may not be privy to all the details. Consider the following events:

- `UOPS_ISSUED:ANY` (event 63)
- `UOPS_RETIRED:ALL` (event 67)
- `UOPS_EXECUTED:THREAD` (event 69)
- `INSTRUCTIONS_RETIRED` (event 156)

Suppose we want to count the number of instructions executed by the CPU. Then which event do we measure? Are instructions or micro-operations (also known

as micro-ops or μops) better candidates? And what are the differences between issued, retired, and executed micro-operations? The answers are not apparent for anyone without a deep understanding of processor internals. This is not an isolated case as there are more examples such as the following:

- `ITLB_MISSES:MISS_CAUSES_A_WALK` (event 37)
- `ITLB_MISSES:WALK_COMPLETED` (event 38)
- `ITLB_MISSES:WALK_DURATION` (event 39)
- `ITLB_MISSES:STLB_HIT` (event 40)

If we simply want to count the number of instruction TLB (ITLB) misses, which event should be monitored? Are four events actually necessary?

All methods except Min-Clique and k-means with $k = 75$ (both distance functions) confirm all events in the first set are similar. For the second set, the first three events are similar since every method apart from k-means with $k = 75$ and correlation place them in the same group. We have already discussed the consequences of Min-Clique not attempting to find larger groups, and it is applicable here. Similarly, $k = 75$ may be too large.

4.6.3 Summary

We have seen two categories of grouping methods; ones which rely on graph algorithms and others which use k-means clustering. Different notions of similarity were also investigated.

The main different between the categories is the latter allows the resulting number of groups to be specified. The correlation cutoff, denoted by c, may be adjusted for the former, but it does not provide fine-grained control. The Euclidean distance is unable to associate events having counts where are not of the same magnitude. Thus, it is advisable to use the correlation coefficient.

For the graph-based algorithms, Greedy-Max-Clique appears to be superior due to its tendency to form larger groups. It is possible to have alternative implementations Min-Clique, but more computation is required to try to find larger groups. CC is useful for exploiting weaker correlations.

To overcome the shortcomings of Min-Clique, we present an approximation algorithm for the minimum clique cover problem. It is based on the greedy vertex colouring algorithm shown in Algorithm 3. Colours are represented by nonnegative integers. Vertex colouring is done on the complement of the correlation graph, and nodes with the same colour form a group.

The resulting grouping is in Table 4.10. Approx-Min-Clique is like Greedy-Max-Clique in the sense it attempts to generate larger groups since we colour vertices with larger degrees first. In fact, the groupings are quite similar (c.f. Table 4.10). We omit the heatmap for brevity as no further conclusions can be drawn from it.

Group No(s).	Size	Group(s)
1	33	{3, 7, 8, 11, 16, 18, 23, 28, 29, 30, 31, 65, 77, 83, 84, 86, 87, 90, 94, 107, 108, 135, 153, 154, 155, 157, 158, 159, 166, 172, 174, 182, 185}

Group No(s).	Size	Group(s)
2	15	{64, 75, 93, 95, 97, 98, 99, 103, 104, 105, 129, 131, 145, 147, 165}
3	11	{1, 2, 22, 33, 34, 35, 91, 148, 149, 150, 152}
4	8	{63, 67, 68, 69, 70, 73, 74, 156}
5	7	{49, 50, 52, 56, 58, 118, 120}
6–7	4	{6, 85, 128, 171}, {17, 26, 27, 124}
8–11	3	{5, 12, 130}, {37, 38, 39}, {72, 79, 82}, {100, 101, 179}
12–21	2	{4, 96}, {9, 10}, {15, 127}, {32, 187}, {76, 113}, {102, 106}, {114, 115}, {132, 146}, {141, 142}, {177, 178}
22–71	1	{13}, {14}, {19}, {20}, {21}, {24}, {25}, {36}, {40}, {41}, {42}, {43}, {44}, {51}, {57}, {71}, {78}, {80}, {81}, {88}, {89}, {92}, {109}, {110}, {111}, {112}, {116}, {117}, {119}, {121}, {125}, {126}, {144}, {151}, {160}, {161}, {162}, {163}, {164}, {169}, {170}, {173}, {175}, {176}, {180}, {181}, {184}, {186}, {188}, {190}

Table 4.10: Event groups generated by Approx-Min-Clique.

The k-means algorithm is difficult to use because of the challenges involved in determining k. One possibility is to exclude the uncorrelated events before performing clustering. We leave this as future work. Moreover, k-means is also affected by the random selection of initial centroids. Restarting is one option, but we see some variation in the results. Furthermore, the effectiveness is heavily contingent on selecting an appropriate value of k. In contrast, the graph-based algorithms exhibit more stability when c is changed.

Algorithm 3 Greedy Vertex Colouring

```
1: function Greedy-Vertex-Colouring(G)
2:     Sort V by vertex degree in descending order
3:     for all v ∈ V
4:         S = Numbers assigned to neighbours of v
5:         Assign smallest nonnegative integer not in S to v
```

4.7 Discussion

The application of PCA showed the intrinsic dimensionality of the event space is less than 10 thus suggesting only a subset of events need to be monitored. However, the dimensionality is not a reflection of the actual number of events to be measured, nor does it provide recommendations. This necessitated the exploration of various grouping methods. We conclude Max-Greedy-Clique or Approx-Min-Clique are most suitable for associating similar events. If fine-grained control of the number of groups is desired, then k-means with correlation is a viable alternative. Finally, CC is good if weaker correlations can be tolerated.

So far no discussion of the appropriate number of groups has taken place. This same applies to an actual validation of the effectiveness of generated event groups in detecting interference. In the next chapter, we will investigate strategies for detecting interference and evaluate the usefulness of groups in such a scenario.

5 Interference Detection

Previously, we proposed various methods for grouping performance events and compared their characteristics. In this chapter, objective evaluations are made concerning the suitability of methods for detecting interference in computer systems. Our analysis relies on a set of machine learning techniques named **support vector machines** (**SVMs**) [15]. All the SVM functions in this chapter are provided by the LIBSVM machine learning library [9, 10].

5.1 SVM

Select SVM models are capable of performing **binary classification** with **supervised learning**. Given a **sample**, i.e., a set of performance counter measurements, we want to determine whether there is interference. There are many possible extensions, e.g., determining the severeness or cause of interference, but a *binary decision*, i.e., yes/no, will be sufficient for now. We use the integers 1 and -1 to represent the two classes—a sample is labeled 1 if there is interference and -1 otherwise. *Supervised learning* involves providing the classifier with a **training set** of labeled samples it can "learn" from. The accuracy of the classifier is evaluated with a distinct **test set**. There are different variations of SVMs and numerous tunable parameters. A **kernel function** which evaluates the similarity of samples must also be defined.

LIBSVM has two SVM implementations which perform support vector classification (SVC): *C*-SVC and ν-SVC. In addition, four different kernel functions are made available to users:

- Linear kernel
- Polynomial kernel
- Radial basis function (RBF) kernel
- Sigmoid kernel

We evaluated all eight combinations with the default parameters. Only the linear kernel produced usable results. In most cases, ν-SVC outperforms *C*-SVC, so only the results of ν-SVC with the linear kernel are shown in this chapter.

5.1.1 Training and Classification

Both C-SVC and ν-SVC have a **training function** and a **classifier function**. The input for the training function is the training set while the classifier takes the test set. Let $E = \{e_1, \ldots, e_n\}$ be a set of n events (bear in mind E does not necessarily span the entire event space). Then the training set is a 2-tuple $(\boldsymbol{x}, \boldsymbol{y})$. Vector $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t)$ contains t samples, where each sample $\boldsymbol{x}_i = (d_1, \ldots, d_n) \in \mathbb{R}^n$ is a vector of deltas. Delta d_j is the delta for event e_j in the time slice corresponding to the sample. Alternatively, a sample can be thought of as a column of the previously defined matrix X, albeit only deltas corresponding to events in E are included. Vector $\boldsymbol{y} = (y_1, \ldots, y_t)$ contains labels, where each label $y_i \in \{-1, 1\}$ indicates the class of sample $\boldsymbol{x}_i$. The test set $(\boldsymbol{x}', \boldsymbol{y}')$ is analogous, but with t' samples. The classifier outputs a vector of labels $\tilde{\boldsymbol{y}}'$ consisting of the predicted labels for samples in $\boldsymbol{x}'$ and compares $\tilde{\boldsymbol{y}}'$ with $\boldsymbol{y}'$ to determine its accuracy.

5.1.2 Classification of Interference

The list of benchmarks was given in Table 4.1. Every application execution must be classified depending on whether the execution experienced interference. Naturally, all base cases, i.e., benchmarks of applications executing alone, are labeled -1 (technically, we label samples of executions and not the executions). For any benchmark involving two applications, we compute each application's interference factor by comparing its execution time with its base case execution time. To illustrate, for the benchmark SORT + PR (*shared cores*), the base case of SORT is simply SORT running alone. Recall PR has two baselines since the *shared sockets* scheme requires a special base case. We label the execution 1 if the interference factor is greater than or equal to 1.05 and -1 otherwise. In other words, we say an application experiences interference if its execution time increased by at least 5% compared to its baseline.

Note that we use the average execution time when calculating the interference factor (recall multiple iterations of each benchmark were required to measure all events). Table 5.1 shows the results of the labeling. Execution times are also provided for reference. If a benchmark contains two applications, then the labels are expressed as a 2-tuple with the first application corresponding to the first label. The format for execution times is analogous.

Application(s)	Scheme	Label(s)	Execution time(s) [s]
MM1		-1	28.69
MM2		-1	33.53
MM3		-1	27.28
SORT		-1	35.53
PR		-1	49.67
PR	*shared sockets*	-1	54.50
MM1 + SORT	*shared cores*	(1, 1)	(57.68, 71.12)
MM1 + SORT	*shared sockets*	(-1, -1)	(28.78, 35.53)
MM1 + SORT	*whole sockets*	(-1, -1)	(28.69, 35.06)
MM1 + PR	*shared cores*	(1, 1)	(60.89, 52.78)
MM1 + PR	*shared sockets*	(1, -1)	(31.10, 50.05)

Application(s)	Scheme	Label(s)	Execution time(s) [s]
MM1 + PR	*whole sockets*	(-1, -1)	(28.70, 49.25)
MM2 + SORT	*shared cores*	(1, 1)	(67.42, 71.17)
MM2 + SORT	*shared sockets*	(-1, -1)	(33.66, 35.61)
MM2 + SORT	*whole sockets*	(-1, -1)	(33.49, 35.08)
MM2 + PR	*shared cores*	(1, 1)	(70.10, 52.76)
MM2 + PR	*shared sockets*	(1, -1)	(35.66, 50.16)
MM2 + PR	*whole sockets*	(-1, -1)	(33.48, 49.23)
MM3 + SORT	*shared cores*	(1, 1)	(54.06, 71.14)
MM3 + SORT	*shared sockets*	(-1, -1)	(26.99, 35.62)
MM3 + SORT	*whole sockets*	(-1, -1)	(26.97, 35.06)
MM3 + PR	*shared cores*	(1, 1)	(55.87, 52.77)
MM3 + PR	*shared sockets*	(-1, -1)	(28.38, 50.10)
MM3 + PR	*whole sockets*	(-1, -1)	(26.98, 49.26)
SORT + PR	*shared cores*	(1, 1)	(76.34, 53.10)
SORT + PR	*shared sockets*	(1, -1)	(38.04, 51.17)
SORT + PR	*whole sockets*	(-1, -1)	(35.09, 49.22)
PR + PR	*shared cores*	(1, 1)	(104.34, 104.38)
PR + PR	*shared sockets*	(1, 1)	(68.52, 68.48)
PR + PR	*whole sockets*	(-1, -1)	(49.48, 49.50)

Table 5.1: Labels and execution times of executions.

Observe the executions of applications with the *shared cores* scheme typically result in severe performance degradations while interference is absent with *whole sockets*. The *shared sockets* scheme is somewhere in between as executions occasionally experience mild interference. This reflects the choice of thread placements, and will be useful in analyzing the classifier later on.

We acknowledge 5% is an arbitrarily chosen number. However, it is robust against variations in executions. The maximum relative standard deviation of the execution time across all executions is 0.96%. Assuming the distribution of execution times is normal, approximately 99.7% of data points are expected to lie within three standard deviations. Three standard deviations is only 2.88% of the mean, so it is statistically unlikely that executions are erroneously classified due to outliers skewing the average execution time.

5.2 Classification with Events

In this section, we evaluate the effectiveness of detecting interference with samples containing the complete set of events. Then we will investigate how subsets of correlated and uncorrelated events fare under the same conditions.

5.2.1 Training and Test Sets

Suppose we have t samples $x_1, \ldots, x_t$ from the execution of an application in a benchmark. Then all the odd-numbered samples $\{x_1, x_3, x_5, \ldots\}$ belong to the training set, and the even-numbered samples $\{x_2, x_4, x_6, \ldots\}$ belong to the test set. Hence the training and test sets both have samples from every execution, and they contain approximately the same number of samples. To be precise, the training

and test sets have 1283 and 1255 samples, respectively. However, the intersection of the two sets is always an empty set, so the classifier is never tested on samples it has "seen" before.

5.2.2 Complete Event Set

We first perform classification using all usable events. In other words, the event set E contains 164 events, so the samples are simply columns of the matrix X.

The results are shown in Table 5.2. Executions are categorized according to application and partitioning scheme. Applications running alone are assigned the scheme *base*, and the applications MM1, MM2, and MM3 are shown collectively as MM. For instance, the cell (SORT, *whole sockets*) contains samples of SORT from the benchmarks MM + SORT (*whole sockets*) and SORT + PR (*whole sockets*). The fraction 67/68 denotes 67 samples out of a total of 68 were labeled correctly by the classifier. The accuracy is calculated using the fraction, i.e., $67/68 \approx 0.9853$.

Samples are also aggregated across applications and partitioning schemes. For example, the classifier performs relatively poorly on PR samples since it only has an accuracy of 82.59%. The accuracy rates for MM and SORT are 15% higher. Similarly, the classifier is least accurate for *whole sockets*, and we see the accuracy is skewed by PR.

In summary, the overall accuracy rate is 91.79%, which is considerably good. The majority of errors occurred when labeling PR samples with the partitioning schemes *whole sockets* and *shared cores*. In contrast, MM and SORT samples posed no problem.

	base	***shared cores***	***shared sockets***	***whole sockets***	***all***
MM	100.00% (43/43)	100.00% (178/178)	95.51% (85/89)	100.00% (86/86)	98.99% (392/396)
SORT	100.00% (17/17)	100.00% (142/142)	100.00% (69/69)	98.53% (67/68)	99.66% (295/296)
PR	98.00% (49/50)	83.01% (171/206)	93.87% (153/163)	63.89% (92/144)	82.59% (465/563)
All	99.09% (109/110)	93.35% (491/526)	95.64% (307/321)	82.21% (245/298)	91.79% (1152/1255)

Table 5.2: Classification with all events.

5.2.3 Correlated Event Set

We have established the full set of events provides sufficiently many data points to detect interference. Now we want to find out if there is a simple method of determining relevant events. Recall analysis of the correlation graph revealed there are 40 uncorrelated events (c.f. Table 4.2). A naive method is to partition the event space into correlated events, i.e., events with nonzero degrees, and uncorrelated events, i.e., events with degree zero.

Classification with the set of correlated events is shown in Table 5.3. The overall accuracy is slightly lower. We see significant drops in accuracy for MM and

SORT with *shared sockets*. As this particular scheme often only causes mild interference, a reduction in the number of data points may make classification more difficult, or some excluded uncorrelated events may be important for these categories. The same holds for PR with *shared cores*. On the other hand, the classifier showed improvements in labeling PR samples with *shared sockets* and *whole sockets*, especially for the latter. One possibility is the uncorrelated events generated noise for those two cases.

	base	*shared cores*	*shared sockets*	*whole sockets*	*all*
MM	100.00% (43/43)	100.00% (178/178)	64.04% (57/89)	100.00% (86/86)	91.92% (364/396)
SORT	100.00% (17/17)	100.00% (142/142)	73.91% (51/69)	100.00% (68/68)	93.92% (278/296)
PR	96.00% (48/50)	57.28% (118/206)	85.89% (140/163)	97.92% (141/144)	79.40% (447/563)
All	98.18% (108/110)	83.27% (438/526)	77.26% (248/321)	98.99% (295/298)	86.77% (1089/1255)

Table 5.3: Classification with correlated events.

5.2.4 Uncorrelated Event Set

In Table 5.4, we see a similar phenomenon when MM and SORT are executed with *shared sockets*. It is plausible both correlated and uncorrelated events are important in boundary cases. Classification of PR samples is very erratic—those with *shared cores* were labeled correctly 98.54% of the time, but results for the other schemes are unusable. This creates an interesting scenario where different types of events are required for different partitioning schemes, even for the same application.

	base	*shared cores*	*shared sockets*	*whole sockets*	*all*
MM	100.00% (43/43)	100.00% (178/178)	73.03% (65/89)	100.00% (86/86)	93.94% (372/396)
SORT	100.00% (17/17)	95.77% (136/142)	75.36% (52/69)	100.00% (68/68)	92.23% (273/296)
PR	28.00% (14/50)	98.54% (203/206)	58.90% (96/163)	38.89% (56/144)	65.54% (369/563)
All	67.27% (74/110)	98.29% (517/526)	66.36% (213/321)	70.47% (210/298)	80.80% (1014/1255)

Table 5.4: Classification with uncorrelated events.

5.3 Classification with Groupings

The main focus of Chapter 4 was to address the limited number of performance counters by grouping similar events. However, no studies on the validity of the grouping results were carried out. In this section, we classify samples using the groupings produced by the various proposed methods.

5.3.1 Training and Test Sets

The training and test sets in this section are identical to the ones described in Section 5.2.1. This allows us to identify grouping methods which are not suitable for reducing the event space.

5.3.2 Group Leaders

We now define a process for reducing the number of events which need to be measured by taking advantage of groups. The basic idea is to select a *group leader* from each group which is representative of remaining events. Since every group is reduced to a single event, the number of events we need to measure is equal to the number of groups.

The group leaders for each grouping are presented in Table 5.5. The group leaders are listed in the same order as the groups they were derived from. The first four groupings were obtained by applying graph algorithms to the graph representation of event correlations. Therefore, we define the group leader to be event with the largest degree. If there is a tie, then the event with the smallest event number is selected.

For k-means, group leader selection is dependent on the distance function. For the Euclidean distance, the centroid of each cluster is the mean of all points in the cluster. Therefore, we choose the event closest to the centroid to be the group leader. Use of the correlation coefficient as the distance dictates that centroids are actual events, so we define the centroids to be groups leaders.

Grouping	Group Leaders
CC	166, 64, 156, 49, 128, 9, 37, 15, 132, 141, 19, 24, 25, 36, 40, 41, 42, 43, 44, 51, 57, 80, 81, 88, 89, 92, 109, 110, 111, 112, 116, 119, 121, 125, 144, 151, 161, 162, 163, 164, 169, 170, 173, 175, 176, 180, 181, 184, 188, 190
Min-Clique	3, 17, 16, 28, 33, 50, 98, 63, 1, 2, 67, 100, 6, 20, 37, 72, 49, 76, 95, 9, 4, 78, 15, 126, 12, 132, 141, 77, 177, 5, 14, 19, 24, 25, 36, 40, 41, 42, 43, 44, 51, 57, 80, 81, 88, 89, 92, 109, 110, 111, 112, 116, 119, 121, 125, 144, 151, 161, 162, 163, 164, 169, 170, 173, 175, 176, 180, 181, 184, 188, 190
Approx-Min-Clique	3, 64, 1, 63, 49, 6, 17, 5, 37, 72, 100, 4, 9, 15, 32, 76, 102, 114, 132, 141, 177, 13, 14, 19, 20, 21, 24, 25, 36, 40, 41, 42, 43, 44, 51, 57, 71, 78, 80, 81, 88, 89, 92, 109, 110, 111, 112, 116, 117, 119, 121, 125, 126, 144, 151, 160, 161, 162, 163, 164, 169, 170, 173, 175, 176, 180, 181, 184, 186, 188, 190
Greedy-Max-Clique	3, 64, 1, 63, 49, 5, 6, 100, 37, 72, 4, 9, 15, 17, 76, 114, 132, 141, 177, 12, 13, 14, 19, 20, 21, 24, 25, 32, 36, 40, 41, 42, 43, 44, 51, 57, 71, 78, 80, 81, 88, 89, 92, 108, 109, 110, 111, 112, 116, 117, 119, 121, 125, 126, 144, 151, 160, 161, 162, 163, 164, 169, 170, 173, 175, 176, 179, 180, 181, 184, 186, 188, 190
k-means ($k = 10$, dist = Euclid)	15, 102, 68, 72, 91, 154, 30, 87, 150, 88
k-means ($k = 75$, dist = Euclid)	58, 8, 68, 79, 127, 99, 11, 155, 91, 9, 130, 124, 169, 29, 106, 105, 98, 184, 177, 1, 6, 86, 18, 23, 24, 30, 34, 37, 49, 64, 96, 161, 109, 112, 114, 118, 144, 150, 151, 188, 13, 19, 20, 26, 32, 35, 39, 40, 65, 67, 76, 77, 81, 87, 88, 89, 90, 92, 95, 100, 101, 110, 113, 116, 117, 126, 128, 132, 146, 148, 149, 158, 162, 176, 190

Grouping	Group Leaders
k-means ($k = 10$, dist = corr)	166, 75, 68, 120, 102, 79, 128, 43, 37, 118
k-means ($k = 75$, dist = corr)	155, 152, 98, 63, 97, 9, 28, 33, 49, 68, 82, 157, 5, 23, 102, 37, 65, 108, 187, 132, 128, 166, 6, 15, 16, 17, 18, 21, 43, 51, 75, 80, 100, 105, 109, 114, 118, 162, 4, 12, 13, 19, 26, 27, 36, 40, 41, 42, 44, 52, 58, 89, 92, 95, 103, 104, 106, 110, 111, 112, 117, 141, 142, 144, 147, 151, 160, 161, 169, 173, 175, 176, 181, 184, 190

Table 5.5: Group leaders of groupings.

5.3.3 Connected Components

The accuracy rates for CC in Table 5.6 are very close to those of the complete event space. MM with *shared sockets* is the only exception as the accuracy plummeted by a little over 30%. Nevertheless, this reduction is considered effective since samples with the *shared sockets* scheme are difficult to label.

	base	*shared cores*	*shared sockets*	*whole sockets*	*all*
MM	100.00% (43/43)	100.00% (178/178)	64.04% (57/89)	100.00% (86/86)	91.92% (364/396)
SORT	100.00% (17/17)	100.00% (142/142)	100.00% (69/69)	98.53% (67/68)	99.66% (295/296)
PR	96.00% (48/50)	83.01% (171/206)	90.18% (147/163)	63.19% (91/144)	81.17% (457/563)
All	98.18% (108/110)	93.35% (491/526)	85.05% (273/321)	81.88% (244/298)	88.92% (1116/1255)

Table 5.6: Classification with CC.

5.3.4 Minimum Clique Cover

The results for classification with Min-Clique-Cover are shown in Table 5.7. Despite the stronger condition on the correlation of events and more data points, the accuracy rate for SORT with *shared sockets* dropped by nearly 30%. This alone would not be a problem, but half of the samples for PR with *shared cores* were incorrectly labeled.

	base	*shared cores*	*shared sockets*	*whole sockets*	*all*
MM	100.00% (43/43)	100.00% (178/178)	64.04% (57/89)	100.00% (86/86)	91.92% (364/396)
SORT	100.00% (17/17)	100.00% (142/142)	72.46% (50/69)	100.00% (68/68)	93.58% (277/296)
PR	100.00% (50/50)	49.51% (102/206)	90.80% (148/163)	66.67% (96/144)	70.34% (396/563)
All	100.00% (110/110)	80.23% (422/526)	79.44% (255/321)	83.89% (250/298)	82.63% (1037/1255)

Table 5.7: Classification with Min-Clique.

5.3.5 Minimum Clique Cover Approximation

We presented Approx-Min-Clique as an alternative for Min-Clique which attempts to generate larger groups. However, there is little change in Table 5.8.

	base	*shared cores*	*shared sockets*	*whole sockets*	*all*
MM	100.00% (43/43)	100.00% (178/178)	64.04% (57/89)	100.00% (86/86)	91.92% (364/396)
SORT	100.00% (17/17)	100.00% (142/142)	72.46% (50/69)	100.00% (68/68)	93.58% (277/296)
PR	100.00% (50/50)	49.51% (102/206)	96.93% (158/163)	66.67% (96/144)	72.11% (406/563)
All	100.00% (110/110)	80.23% (422/526)	82.55% (265/321)	83.89% (250/298)	83.43% (1047/1255)

Table 5.8: Classification with Approx-Min-Clique.

5.3.6 Greedy Maximum Clique Cover

Greedy-Max-Clique finds the largest clique possible in every iteration, but we also see no improvement in Table 5.9. Therefore, we must conclude grouping methods which return clique covers generally are not better than CC even though CC produces significantly fewer data points.

	base	*shared cores*	*shared sockets*	*whole sockets*	*all*
MM	100.00% (43/43)	100.00% (178/178)	64.04% (57/89)	100.00% (86/86)	91.92% (364/396)
SORT	100.00% (17/17)	100.00% (142/142)	72.46% (50/69)	100.00% (68/68)	93.58% (277/296)
PR	100.00% (50/50)	49.51% (102/206)	94.48% (154/163)	66.67% (96/144)	71.40% (402/563)
All	100.00% (110/110)	80.23% (422/526)	81.31% (261/321)	83.89% (250/298)	83.11% (1043/1255)

Table 5.9: Classification with Greedy-Max-Clique.

5.3.7 *k*-means Clustering with Euclidean Distance

Tables 5.10 and 5.10 show the results of *k*-means with the Euclidean distance for k = 10 and 75, respectively. In all cases but one, increasing the number of clusters improves the likelihood of correctly labeling samples. In particular, the accuracy rate for PR with *shared cores* improved from 50% to 83.98%. However, many samples for PR with *whole sockets* were incorrectly classified as having interference. The numbers for $k = 75$ are comparable to those of the complete event set.

5.3.8 *k*-means Clustering with Correlation

As shown in Table 5.12, using the correlation coefficient as the distance function with $k = 10$ resulted in a noticeable regression for MM and SORT with *shared sockets*, implying it is not as capable of establishing the boundary between weak

	base	*shared cores*	*shared sockets*	*whole sockets*	*all*
MM	100.00% (43/43)	100.00% (178/178)	79.78% (71/89)	100.00% (86/86)	95.45% (378/396)
SORT	100.00% (17/17)	100.00% (142/142)	100.00% (69/69)	92.65% (63/68)	98.31% (291/296)
PR	100.00% (50/50)	50.00% (103/206)	84.05% (137/163)	98.61% (142/144)	76.73% (432/563)
All	100.00% (110/110)	80.42% (423/526)	86.29% (277/321)	97.65% (291/298)	87.73% (1101/1255)

Table 5.10: Classification with k-means ($k = 10$, dist = Euclid).

	base	*shared cores*	*shared sockets*	*whole sockets*	*all*
MM	97.67% (42/43)	100.00% (178/178)	91.01% (81/89)	97.67% (84/86)	97.22% (385/396)
SORT	100.00% (17/17)	100.00% (142/142)	100.00% (69/69)	95.59% (65/68)	98.99% (293/296)
PR	96.00% (48/50)	83.98% (173/206)	92.02% (150/163)	62.50% (90/144)	81.88% (461/563)
All	97.27% (107/110)	93.73% (493/526)	93.46% (300/321)	80.20% (239/298)	90.76% (1139/1255)

Table 5.11: Classification with k-means ($k = 75$, dist = Euclid).

interference and no interference. Additionally, it also erroneously labeled a considerable number of samples of PR with *whole sockets* as having interference.

The results of for $k = 75$ are presented in Table 5.12. Aside from being able to somewhat reliably detect when PR suffers from interference with shared cores, there are no significant changes. For both values of k, the correlation coefficient showed no advantages over the Euclidean distance.

	base	*shared cores*	*shared sockets*	*whole sockets*	*all*
MM	100.00% (43/43)	100.00% (178/178)	64.04% (57/89)	100.00% (86/86)	91.92% (364/396)
SORT	100.00% (17/17)	100.00% (142/142)	71.01% (49/69)	100.00% (68/68)	93.24% (276/296)
PR	100.00% (50/50)	49.51% (102/206)	90.80% (148/163)	66.67% (96/144)	70.34% (396/563)
All	100.00% (110/110)	80.23% (422/526)	79.13% (254/321)	83.89% (250/298)	82.55% (1036/1255)

Table 5.12: Classification with k-means ($k = 10$, dist = corr).

5.4 Classification of Unknown Applications

The previous training and test sets were designed so that the classifier labels samples from applications it has encountered before. Unfortunately, this will not be true for the majority of samples, for there are too many different applications out in the wild. In this section, we devise a scheme which emulates this scenario to test the robustness of the classifier. We use the complete event set since it has been

	base	*shared cores*	*shared sockets*	*whole sockets*	*all*
MM	100.00% (43/43)	100.00% (178/178)	64.04% (57/89)	100.00% (86/86)	91.92% (364/396)
SORT	100.00% (17/17)	100.00% (142/142)	72.46% (50/69)	100.00% (68/68)	93.58% (277/296)
PR	84.00% (42/50)	81.07% (167/206)	92.64% (151/163)	63.19% (91/144)	80.11% (451/563)
All	92.73% (102/110)	92.59% (487/526)	80.37% (258/321)	82.21% (245/298)	87.01% (1092/1255)

Table 5.13: Classification with k-means ($k = 75$, dist = corr).

shown accuracy rates are still acceptable after applying grouping methods.

5.4.1 Training and Test Sets

First we selected an application, e.g., SORT. Next, a list of benchmarks was compiled with benchmarks involving SORT removed. This applies to both benchmarks only containing SORT and those where SORT is paired with another application, e.g., SORT + PR (*shared cores*). The training set is the aggregation of all samples from the compiled list, and the test set contains all samples from executions of SORT.

This process was repeated for every application, i.e., MM1, MM2, MM3, SORT, and PR. In addition, we also did it for for all MM variants collectively. That is, MM1, MM2, and MM3 are all considered the same application under the name MM. Note that every application has different training and test sets. The training set sizes for MM1, MM2, MM3, MM, SORT, and PR are 1991, 1947, 2008, 870, 1439, and 891, respectively. The test set sizes can be found below.

5.4.2 Classification of Unknown Applications

Table 5.14 shows the classification results for each application. Each row is dedicated to a single application. For instance, the last row presents the statistics for classifying PR samples, i.e., only executions of PR were included in the test set whereas all benchmarks involving PR were excluded from the training set.

Observe that all the classifier has accuracy rates of above 95% for every case except PR. In particular, it is able to classify executions with the *shared sockets* scheme. In contrast, PR samples are incorrectly labeled over 50% of the time. This is consistent with previous classifications in the sense that results for PR are often below average.

5.5 Discussion

In summary, the classifier is able to label samples according to our definition of interference with good accuracy rates. The schemes *shared cores*, *shared sockets*, and *whole sockets* were designed so that they represent cases with severe, mild, and no interference. Since *shared sockets* is a grey area, the accuracy rates of this scheme

	base	*shared cores*	*shared sockets*	*whole sockets*	*all*
MM1	100.00% (28/28)	100.00% (117/117)	86.21% (50/58)	100.00% (56/56)	96.91% (251/259)
MM2	100.00% (33/33)	100.00% (136/136)	92.65% (63/68)	100.00% (66/66)	98.35% (298/303)
MM3	100.00% (27/27)	100.00% (108/108)	88.89% (48/54)	100.00% (52/52)	97.51% (235/241)
MM	98.86% (87/88)	100.00% (361/361)	88.89% (160/180)	98.28% (171/174)	97.01% (779/803)
SORT	100.00% (35/35)	100.00% (285/285)	100.00% (142/142)	96.32% (131/136)	99.16% (593/598)
PR	100.00% (102/102)	0.00% (0/414)	59.70% (197/330)	66.32% (193/291)	43.27% (492/1137)

Table 5.14: Classification of unknown applications.

for MM and SORT are often lower. The classifier is also reasonably robust as it was able to classify MM and SORT samples without having seen them before. However, it struggles with PR samples, and this is applicable to other evaluations we performed. One plausible explanation is that PR is a multithreaded application, and aggregation of counter values across threads causes problems. As discussed before, an alternative is to analyze each thread separately. Finally, the grouping methods are able to reduce the number of events which need to be measured. Of all the graph-based methods, CC produced the best results while the Euclidean distance is better than the correlation coefficient for k-means. In many cases, requiring all events in the same group to be correlated did not improve accuracy. It remains to be seen if there are specific groups are more suitable for classification than others.

6 Future Work

In this chapter, we describe possible directions for future work. Most of the items listed are improvements or extensions of results presented in this thesis.

Expansion of the application set: Given the diverse range of applications today, the application set presents a rather narrow view of the world. The application set can be easily expanded beyond the current applications with the inclusion of more algorithms. Furthermore, multithreaded programs should be well-represented as they are becoming increasingly important.

Classification with event groups: We focused on reducing the event space by nominating group leaders, so only one event per group is measured. However, certain groups may be better than others at classifying samples, so it is desirable to verify whether individual groups are capable of labeling samples accurately.

Tuning of SVM parameters: SVM has adjustable parameters such as the kernel function which evaluates similarity and other implementation-dependent options (e.g., the value of ν for ν-SVC). Tuning these parameters may improve accuracy as there were specific cases where classification did not work well. Replacing the kernel function is also a possibility. Furthermore, additional SVM implementations should be explored.

Multiclass interference classification: Samples were labeled with yes/no, but in reality binary classification cannot sufficiently describe the severity or types of interference. There are machine learning techniques designed to classify samples into multiple classes, and we may be able to apply these to obtain more descriptive classifications.

Alternative classification techniques: There are many other machine learning techniques commonly used for classification such as Bayesian networks and neural networks. It is plausible alternative techniques may be superior to SVMs. One-class classification techniques may also complement binary classification.

Recommendation of thread placements: The detection of interference alone is useful, but does not directly result in tangible benefits. A scheme for recommending improved thread placements must be devised to mitigate interference. Analysis of the benchmarks has shown that placement strategies similar to *shared cores*, *shared sockets*, and *whole sockets* are good candidates. While *whole sockets* typically causes the least interference, it also decreases resource utilization, so the other two should be preferred if interference is within an acceptable range.

Phase detection: Applications in the application do not have distinct phases unlike other programs. However, phase detection is relatively straightforward as we only need to periodically reevaluate thread placements. We only need to determine the frequency at which it is carried out.

7 Conclusion

The first contribution of this thesis is a survey of PMUs. In contrast to previous work which focused on applications of PMUs, we investigate properties of PMUs themselves with a view of addressing inherent shortcomings of performance counters. Of all the limitations, the limited availability of performance counters, the intricacies of performance events, and the portability of events are the primary concerns.

We proposed various grouping methods to reduce the event space by associating similar events. Two different definitions of similarity were considered: the Euclidean distance and correlation coefficients. We also derived a graph representation of the similarity of events using the latter. Graph algorithms and k-means clustering were applied to produce events groupings.

Since end users rarely possess domain knowledge about hardware and performance events, this presents a barrier to an otherwise useful tool. We applied an SVM model to detect interference by classifying event samples. This approach has the benefit of not requiring knowledge of performance events, and classification is accurate even for a reduced set of events such as the event groupings.

Furthermore, all the techniques in this thesis are not constrained to a particular machine. This is a departure from previous approaches which targeted specific processors. Consequently, our approach has the potential of solving the issue of portability.

As progress is made towards the removal of these obstacles, PMUs will be more easily accessible to users. This would be greatly beneficial as numerous applications of PMUs have been demonstrated.

A Benchmark Configurations

We document all the machines and applications used in this thesis here. In addition, the partitioning schemes employed to allocate processor cores for applications and the method for instrumenting PMUs are also discussed.

A.1 Machines

Appenzeller

Appenzeller has 4 AMD Opteron 6174 processors. Each processor comprises 2 six-core NUMA nodes, and each NUMA node has its own memory controller and L3 cache of 6 MB. Every core has 512 KB of L2 cache. *Appenzeller* runs Linux 4.1.

Babybel

Babybel has 2 Intel Xeon E5-2670 v2 processors. Each processor has 10 physical cores and 25 MB of L3 cache. Every core has 256 KB of L2 cache. SMT is disabled, and the machine has 256 GB of RAM. *Babybel* runs Linux 4.4.

There are actually 4 machines with this configuration, but since they are identical, we refer to them as if they were a single machine.

Gottardo

Gottardo has 4 Intel Xeon L7555 processors. Each processor has 8 physical cores and 24 MB of L3 cache. Every core has 256 KB of L2 cache. SMT is enabled, and the machine has 128 GB of RAM. *Gottardo* runs Linux 4.2.

A.2 Applications

Green-Marl

Green-Marl provides implementations of several graph algorithms [21]. We use the following three:

- PageRank (PR)
- Hop distance (HD)

- Single-source shortest path (SSSP)

They are all parallel implementations capable of spawning any given number of threads. Since the programs were compiled with libgomp, an implementation of OpenMP, threads may be pinned to specific cores using the environment variable `GOMP_CPU_AFFINITY`. The input for these applications is a graph based on the Twitter social network [27].

Matrix Multiplication

Matrix multiplication (MM) was written using the ATLAS library, which provides an implementation of BLAS for C [42]. It is single-threaded. We use various matrices from the University of Florida Sparse Matrix Collection (UFSMC) as input [16]:

- MM1: Multiply MathWorks/tomography by itself for 1000 iterations.
- MM2: Multiply Bai/tub1000 by itself for 150 iterations.
- MM3: Multiply Hamm/add32 by itself for 1 iteration.

MM1 and MM2 perform matrix multiplication multiple times to prolong the execution time.

Sort

Sort (SORT) was implemented using `qsort` from glibc [33]. It is single-threaded. The input for SORT is an array consisting of the elements of the matrix Schenk/nlpkkt200 from UFSMC [16].

A.3 Partitioning Schemes

We run at most two applications simultaneously. When two applications are run together, there are four possible partitioning schemes for allocating processor cores:

- *shared cores:* Applications share processor cores. If SMT is available and enabled, then each application is allocated 1 SMT thread in each physical core, assuming every physical core has 2 SMT threads. If SMT is disabled or not available, then applications share every physical core.
- *shared sockets:* Applications share processors, i.e., cores in the same CPU are allocated to different applications, but each core can only be used by one application.
- *whole sockets:* Each application receives whole processors, i.e., each CPU is assigned to only one application.
- *default:* Thread placement is delegated to the Linux scheduler.

A.4 PMU Instrumentation

Performance counters are read with `perf_event_open` in sampling mode. The target frequency is 100 samples per second, and counters are configured to monitor children, i.e., the `inherit` flag is set. For any measured event, each process opens a separate file descriptor for every core in the system. For example, if there are x measured events and y cores, then each application has xy file descriptors. Samples are only read at the end of execution, and the `mmap` ring buffers are large enough to hold all samples. A wrapper was written to allow event names to be passed in through an environment variable. Each application was modified at the beginning to monitor the specified events and at the end to read all samples.

Note that the execution time for each application varies. Therefore, when applications are paired together, short-running ones are run for multiple iterations to guarantee the total execution time of an application is no less than the paired application's first iteration execution time. For instance, if the execution times of PR and MM are t_1 and t_2, respectively, and $t_1 > t_2$, then MM is run n times so that $t_1 \leq n \times t_2$. While we collect data for all iterations, we typically only consider samples from the first iteration of each application. Analogously, the execution time of an application is understood to be the execution time of the first iteration unless specified otherwise.

Applications obtained from Green-Marl required further changes:

- Thread initialization was deferred until after the configuration of performance counters to allow children to be monitored as well.
- A command line parameter was added to allow applications to be run for multiple iterations.
- SSSP originally randomly assigned edge weights sequentially. The loop was parallelized and made deterministic using edge IDs.

B Event Space

Table B.1 presents the list of events which were selected in Chapter 4. The events are shown with libpfm event names instead of Intel event names. Apart from a few exceptions, the names are identical or similar.

The following events were excluded from the matrix X since they only had zero deltas: 45, 46, 47, 48, 53, 54, 55, 59, 60, 61, 62, 66, 122, 123, 133, 134, 136, 137, 138, 139, 140, 143, 167, 168, 183, 189.

Event No.	Event Name
1	LLC_REFERENCES
2	LLC_MISSES
3	L2_RQSTS:ALL_DEMAND_DATA_RD
4	L2_RQSTS:DEMAND_DATA_RD_HIT
5	L2_RQSTS:ALL_RFO
6	L2_RQSTS:RFO_MISS
7	L2_RQSTS:ALL_CODE_RD
8	L2_RQSTS:CODE_RD_MISS
9	L2_RQSTS:ALL_PF
10	L2_RQSTS:PF_MISS
11	L2_TRANS:DMND_DATA_RD
12	L2_TRANS:RFO
13	L2_TRANS:CODE_RD
14	L2_TRANS:ALL_PF
15	L2_TRANS:L1D_WB
16	L2_TRANS:L2_FILL
17	L2_TRANS:L2_WB
18	L2_TRANS:ALL
19	L2_LINES_IN:I
20	L2_LINES_IN:S
21	L2_LINES_IN:E
22	L2_LINES_IN:ALL
23	L2_LINES_OUT:DEMAND_CLEAN
24	L2_LINES_OUT:DEMAND_DIRTY
25	L2_LINES_OUT:PF_CLEAN
26	L2_LINES_OUT:PF_DIRTY
27	L2_LINES_OUT:DIRTY_ALL
28	DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK
29	DTLB_LOAD_MISSES:WALK_COMPLETED
30	DTLB_LOAD_MISSES:WALK_DURATION
31	DTLB_LOAD_MISSES:LARGE_WALK_COMPLETED

Event No.	Event Name
32	DTLB_LOAD_MISSES:STLB_HIT
33	DTLB_STORE_MISSES:MISS_CAUSES_A_WALK
34	DTLB_STORE_MISSES:WALK_COMPLETED
35	DTLB_STORE_MISSES:WALK_DURATION
36	DTLB_STORE_MISSES:STLB_HIT
37	ITLB_MISSES:MISS_CAUSES_A_WALK
38	ITLB_MISSES:WALK_COMPLETED
39	ITLB_MISSES:WALK_DURATION
40	ITLB_MISSES:STLB_HIT
41	MEM_LOAD_UOPS_LLC_HIT_RETIRED:XSNP_MISS
42	MEM_LOAD_UOPS_LLC_HIT_RETIRED:XSNP_HIT
43	MEM_LOAD_UOPS_LLC_HIT_RETIRED:XSNP_HITM
44	MEM_LOAD_UOPS_LLC_HIT_RETIRED:XSNP_NONE
45	MEM_LOAD_UOPS_LLC_MISS_RETIRED:LOCAL_DRAM
46	MEM_LOAD_UOPS_LLC_MISS_RETIRED:REMOTE_DRAM
47	MEM_LOAD_UOPS_LLC_MISS_RETIRED:REMOTE_HITM
48	MEM_LOAD_UOPS_LLC_MISS_RETIRED:REMOTE_FWD
49	MEM_LOAD_UOPS_RETIRED:L1_HIT
50	MEM_LOAD_UOPS_RETIRED:L2_HIT
51	MEM_LOAD_UOPS_RETIRED:L3_HIT
52	MEM_LOAD_UOPS_RETIRED:L1_MISS
53	MEM_LOAD_UOPS_RETIRED:L2_MISS
54	MEM_LOAD_UOPS_RETIRED:L3_MISS
55	MEM_LOAD_UOPS_RETIRED:HIT_LFB
56	MEM_UOPS_RETIRED:STLB_MISS_LOADS
57	MEM_UOPS_RETIRED:STLB_MISS_STORES
58	MEM_UOPS_RETIRED:LOCK_LOADS
59	MEM_UOPS_RETIRED:SPLIT_LOADS
60	MEM_UOPS_RETIRED:SPLIT_STORES
61	MEM_UOPS_RETIRED:ALL_LOADS
62	MEM_UOPS_RETIRED:ALL_STORES
63	UOPS_ISSUED:ANY
64	UOPS_ISSUED:FLAGS_MERGE
65	UOPS_ISSUED:SLOW_LEA
66	UOPS_ISSUED:SINGLE_MUL
67	UOPS_RETIRED:ALL
68	UOPS_RETIRED:RETIRE_SLOTS
69	UOPS_EXECUTED:THREAD
70	UOPS_EXECUTED:CORE
71	UOPS_DISPATCHED_PORT:PORT_0
72	UOPS_DISPATCHED_PORT:PORT_1
73	UOPS_DISPATCHED_PORT:PORT_2
74	UOPS_DISPATCHED_PORT:PORT_3
75	UOPS_DISPATCHED_PORT:PORT_4
76	UOPS_DISPATCHED_PORT:PORT_5
77	IDQ:EMPTY
78	IDQ:ALL_DSB_CYCLES
79	IDQ:ALL_DSB_CYCLES_4_UOPS
80	IDQ:ALL_MITE_CYCLES
81	IDQ:ALL_MITE_CYCLES_4_UOPS
82	IDQ:ANY_UOPS
83	OFFCORE_REQUESTS:DEMAND_DATA_RD

Event No.	Event Name
84	OFFCORE_REQUESTS:DEMAND_CODE_RD
85	OFFCORE_REQUESTS:DEMAND_RFO
86	OFFCORE_REQUESTS:ALL_DATA_RD
87	OFFCORE_REQUESTS_OUTSTANDING:DEMAND_DATA_RD
88	OFFCORE_REQUESTS_OUTSTANDING:DEMAND_CODE_RD
89	OFFCORE_REQUESTS_OUTSTANDING:DEMAND_RFO
90	OFFCORE_REQUESTS_OUTSTANDING:ALL_DATA_RD
91	OFFCORE_RESPONSE_0:ANY_REQUEST:LLC_MISS_LOCAL
92	OFFCORE_RESPONSE_1:ANY_REQUEST:LLC_MISS_REMOTE
93	BR_INST_EXEC:NONTAKEN_COND
94	BR_INST_EXEC:TAKEN_COND
95	BR_INST_EXEC:TAKEN_DIRECT_JUMP
96	BR_INST_EXEC:TAKEN_INDIRECT_JUMP_NON_CALL_RET
97	BR_INST_EXEC:TAKEN_NEAR_RETURN
98	BR_INST_EXEC:TAKEN_DIRECT_NEAR_CALL
99	BR_INST_EXEC:TAKEN_INDIRECT_NEAR_CALL
100	BR_INST_EXEC:ALL_BRANCHES
101	BR_INST_RETIRED:ALL_BRANCHES
102	BR_INST_RETIRED:COND
103	BR_INST_RETIRED:NEAR_CALL
104	BR_INST_RETIRED:NEAR_RETURN
105	BR_INST_RETIRED:NOT_TAKEN
106	BR_INST_RETIRED:NEAR_TAKEN
107	BR_INST_RETIRED:FAR_BRANCH
108	BR_MISP_EXEC:NONTAKEN_COND
109	BR_MISP_EXEC:TAKEN_COND
110	BR_MISP_EXEC:TAKEN_INDIRECT_JUMP_NON_CALL_RET
111	BR_MISP_EXEC:TAKEN_NEAR_RETURN
112	BR_MISP_EXEC:TAKEN_INDIRECT_NEAR_CALL
113	BR_MISP_EXEC:ALL_BRANCHES
114	BR_MISP_RETIRED:ALL_BRANCHES
115	BR_MISP_RETIRED:CONDITIONAL
116	BR_MISP_RETIRED:NEAR_TAKEN
117	CYCLE_ACTIVITY:CYCLES_L2_PENDING
118	CYCLE_ACTIVITY:CYCLES_LDM_PENDING
119	CYCLE_ACTIVITY:CYCLES_L1D_PENDING
120	CYCLE_ACTIVITY:CYCLES_NO_EXECUTE
121	CYCLE_ACTIVITY:STALLS_L2_PENDING
122	CYCLE_ACTIVITY:STALLS_L1D_PENDING
123	CYCLE_ACTIVITY:STALLS_LDM_PENDING
124	L2_L1D_WB_RQSTS:HIT_E
125	L2_L1D_WB_RQSTS:HIT_M
126	L2_L1D_WB_RQSTS:MISS
127	L2_L1D_WB_RQSTS:ALL
128	L2_STORE_LOCK_RQSTS:MISS
129	L2_STORE_LOCK_RQSTS:HIT_M
130	L2_STORE_LOCK_RQSTS:ALL
131	FP_COMP_OPS_EXE:X87
132	FP_COMP_OPS_EXE:SSE_FP_PACKED_DOUBLE
133	FP_COMP_OPS_EXE:SSE_FP_SCALAR_SINGLE
134	FP_COMP_OPS_EXE:SSE_PACKED_SINGLE
135	FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE

Event No.	Event Name
136	FP_ASSIST:ANY
137	FP_ASSIST:SIMD_INPUT
138	FP_ASSIST:SIMD_OUTPUT
139	FP_ASSIST:X87_INPUT
140	FP_ASSIST:X87_OUTPUT
141	OTHER_ASSISTS:AVX_TO_SSE
142	OTHER_ASSISTS:SSE_TO_AVX
143	OTHER_ASSISTS:AVX_STORE
144	OTHER_ASSISTS:WB
145	MOVE_ELIMINATION:INT_NOT_ELIMINATED
146	MOVE_ELIMINATION:SIMD_NOT_ELIMINATED
147	MOVE_ELIMINATION:INT_ELIMINATED
148	MOVE_ELIMINATION:SIMD_ELIMINATED
149	RESOURCE_STALLS:ANY
150	RESOURCE_STALLS:RS
151	RESOURCE_STALLS:SB
152	RESOURCE_STALLS:ROB
153	UNHALTED_CORE_CYCLES
154	UNHALTED_REFERENCE_CYCLES
155	CPU_CLK_UNHALTED:THREAD_P
156	INSTRUCTIONS_RETIRED
157	L1D:REPLACEMENT
158	L1D_PEND_MISS:PENDING
159	ICACHE:MISSES
160	ICACHE:IFETCH_STALL
161	LD_BLOCKS:STORE_FORWARD
162	LD_BLOCKS:NO_SR
163	LD_BLOCKS_PARTIAL:ADDRESS_ALIAS
164	MISALIGN_MEM_REF:LOADS
165	MISALIGN_MEM_REF:STORES
166	ARITH:FPU_DIV_ACTIVE
167	SIMD_FP_256:PACKED_SINGLE
168	SIMD_FP_256:PACKED_DOUBLE
169	LOAD_HIT_PRE:HW_PF
170	LOAD_HIT_PRE:SW_PF
171	CPL_CYCLES:RING0
172	CPL_CYCLES:RING123
173	LOCK_CYCLES:SPLIT_LOCK_UC_LOCK_DURATION
174	LOCK_CYCLES:CACHE_LOCK_DURATION
175	ILD_STALL:LCP
176	ILD_STALL:IQ_FULL
177	DSB2MITE_SWITCHES:COUNT
178	DSB2MITE_SWITCHES:PENALTY_CYCLES
179	DSB_FILL:EXCEED_DSB_LINES
180	TLB_FLUSH:STLB_ANY
181	TLB_FLUSH:DTLB_THREAD
182	ITLB:ITLB_FLUSH
183	MACHINE_CLEARS:MASKMOV
184	MACHINE_CLEARS:MEMORY_ORDERING
185	MACHINE_CLEARS:SMC
186	RS_EVENTS:EMPTY_CYCLES
187	LSD:UOPS

Event No.	Event Name
188	OFFCORE_REQUESTS_BUFFER:SQ_FULL
189	ROB_MISC_EVENTS:LBR_INSERTS
190	BACLEARS:ANY

Table B.1: List of selected events.

Bibliography

[1] Reza Azimi, Michael Stumm, and Robert W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 101–110, 2005.

[2] Reza Azimi, David K. Tam, Livio Soares, and Michael Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *SIGOPS Oper. Syst. Rev.*, 43(2):56–65, April 2009.

[3] Keith A. Bare, Soila Kavulya, and Priya Narasimhan. Hardware performance counter-based problem diagnosis for e-commerce systems. In *2010 IEEE Network Operations and Management Symposium*, NOMS '10, pages 551–558, 2010.

[4] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, pages 217–235, 1999.

[5] Sapan Bhatia, Abhishek Kumar, Marc E. Fiuczynski, and Larry Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pages 103–116, 2008.

[6] Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 180–186, 2010.

[7] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, 2000.

[8] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 185–197, 2007.

[9] Chih-chung Chang and Chih-jen Lin. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, May 2011.

[10] Chih chung Chang and Chih jen Lin. LIBSVM: A Library for Support Vector Machines. https://www.csie.ntu.edu.tw/~cjlin/libsvm/. Accessed 12 September 2016.

[11] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer Manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. Accessed 26 August 2016.

[12] Intel Corporation. Intel® Performance Counter Monitor - A better way to measure CPU utilization. https://software.intel.com/en-us/articles/intel-performance-counter-monitor. Accessed 26 August 2016.

[13] Intel Corporation. Intel® VTune™ Amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe. Accessed 26 August 2016.

[14] Intel Corporation. perfmon. https://download.01.org/perfmon/. Accessed 26 August 2016.

[15] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995.

[16] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[17] Advanced Micro Devices. BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors. http://support.amd.com/TechDocs/31116.pdf. Accessed 26 August 2016.

[18] Advanced Micro Devices. Developer Guides, Manuals & ISA Documents. http://developer.amd.com/resources/developer-guides-manuals/. Accessed 26 August 2016.

[19] Paul J. Drongowski. Basic Performance Measurements for AMD Athlon™ 64, AMD Opteron™ and AMD Phenom™ Processors. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Basic_Performance_Measurements.pdf. Accessed 1 September 2016.

[20] Stephane Eranian. perfmon2. http://perfmon2.sourceforge.net/. Accessed 26 August 2016.

[21] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A DSL for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 349–362, 2012.

[22] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24(6):417–441, September 1933.

[23] ICL group and contributors. PAPI. http://icl.cs.utk.edu/papi/. Accessed 26 August 2016.

[24] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, May 2008.

[25] Joseph Koshy. FreeBSD Library Functions Manual: hwpmc(3). https://www.freebsd.org/cgi/man.cgi?query=hwpmc&manpath=FreeBSD+11.0-RELEASE. FreeBSD 11.0-RELEASE. Accessed 1 September 2016.

[26] Joseph Koshy. FreeBSD Library Functions Manual: pmc(3). https://www.freebsd.org/cgi/man.cgi?query=pmc&manpath=FreeBSD+11.0-RELEASE. FreeBSD 11.0-RELEASE. Accessed 1 September 2016.

[27] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, 2010.

[28] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, 2015.

[29] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 1:1–1:16, 2016.

[30] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, 1967.

[31] John M. May. MPX: Software for multiplexing hardware performance counters in multithreaded programs. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, IPDPS '01, 2001.

[32] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 153–166, 2010.

[33] David Metcalfe. Linux Programmer's Manual: qsort(3). http://man7.org/linux/man-pages/man3/qsort.3.html. Version 4.07. Accessed 1 September 2016.

[34] Mozilla Developer Network and contributors. Building with Profile-Guided Optimization. https://developer.mozilla.org/en/docs/Building_with_Profile-Guided_Optimization. Accessed 26 August 2016.

[35] Numpy developers. NumPy. http://www.numpy.org/. Version 1.11. Accessed 5 September 2016.

[36] Numpy developers. NumPy Manual: numpy.cov. http://docs.scipy.org/doc/numpy-1.11.0/reference/generated/numpy.cov.html. Version 1.11. Accessed 5 September 2016.

[37] Numpy developers. NumPy Manual: numpy.linalg.eigh. http://docs.scipy.org/doc/numpy-1.11.0/reference/generated/numpy.linalg.eigh.html. Accessed 5 September 2016.

[38] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros Cabezas, Daniele G. Spampinato, and Markus Püschel. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '14, pages 76–85, 2014.

[39] Subhash Saini, Haoqiang Jin, Robert Hood, David Barker, Piyush Mehrotra, and Rupak Biswas. The impact of hyper-threading on processor resource utilization in production applications. In *2011 18th International Conference on High Performance Computing*, HiPC '11, pages 1–10, 2011.

[40] Karan Singh, Major Bhadauria, and Sally A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News*, 37(2):46–55, July 2009.

[41] Vince Weaver. Linux Programmer's Manual: perf_event_open(2). http://man7.org/linux/man-pages/man2/perf_event_open.2.html. Version 4.07. Accessed 26 August 2016.

[42] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, January 2001.

[43] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[44] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, 2013.

[45] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 129–142, 2010.

[46] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared

resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, December 2012.