Title
Accurate Estimation of Occurrences of Performance Events

Permalink
https://escholarship.org/uc/item/5rw08880

Author
Carrillo Cisneros, David

Publication Date
2015

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA, IRVINE

Accurate Estimation of Occurrences of Performance Events

DISSE TATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

David Carrillo Cisneros

Dissertation Committee:
Professor Isaac D. Scherson, Chair
Assistant Professor Guoqing Xu
Associate Professor Zhaoxia Yu

2015
DEDICATION

To my parents and sisters.
TABLE OF CONTENTS

LIST OF FIGURES vi
LIST OF TABLES vii
LIST OF ALGORITHMS viii
ACKNOWLEDGMENTS ix
CURRICULUM VITAE x
ABSTRACT OF THE DISSER TATION xi

1 Introduction 1
1.1 Challenges in the Utilization of PMUs 3
1.1.1 Perturbation Caused by Sampling of HPCs 4
1.1.2 Error and Non-determinism 4
1.2 Performance Diagnostics using Performance Events 6
1.3 Association with other Performance Events 8
1.4 Associating Performance Events to Instructions 9
1.5 Our Contribution 11

2 Preliminaries 13
2.1 Common Terms 13
2.2 Sampling of HPCs 14
2.2.1 Register Overflow 16
2.3 Formal Definitions 16
2.3.1 Notation 16
2.3.2 Basic Definitions 17

3 Problem Statement 20

4 Related Work 23
4.1 Traditional Methodology to Associate Performance Events and Instructions 23
4.1.1 Example 24
4.1.2 Limitations of traditional approach 26
# Linear Time Algorithms for Regression of Performance Events Counts

## 7 Estimation as a Isotonic Regression Problem

### 7.1 Isotonic Regression

### 7.1.2 Pool Adjacent Violators Algorithm

### 7.1.3 PAVA to Solve Estimation Problem

## 7.2 Estimation as a Generalization of the Isotonic Regression Problem

### 7.2.1 Motivation

### 7.2.2 Generalization of PAVA

## 7.3 Experiments

## 7.4 Conclusions

## Bibliography

## Appendix
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Example of time series for <strong>energy-cores</strong> (Energy consumed per core)</td>
<td>30</td>
</tr>
<tr>
<td>5.2</td>
<td>Example of interpolation techniques</td>
<td>31</td>
</tr>
<tr>
<td>5.3</td>
<td>An example of a sequence of sample points</td>
<td>32</td>
</tr>
<tr>
<td>5.4</td>
<td>Problems with no regularized interpolation</td>
<td>39</td>
</tr>
<tr>
<td>5.5</td>
<td>Reconstruction error (MASE)</td>
<td>45</td>
</tr>
<tr>
<td>6.1</td>
<td>Relative error of different estimation techniques versus known ratio of retired instructions per IP</td>
<td>79</td>
</tr>
<tr>
<td>6.2</td>
<td>Relative error of different estimation techniques versus estimated ratio of retired instructions per IP for L1-dcache misses</td>
<td>80</td>
</tr>
<tr>
<td>6.3</td>
<td>Relative error of different estimation techniques versus estimated ratio of retired instructions per IP for Branch misses</td>
<td>81</td>
</tr>
<tr>
<td>7.1</td>
<td>Running times versus number of samples of various algorithms</td>
<td>93</td>
</tr>
<tr>
<td>7.2</td>
<td>Running times versus number of samples for linear time algorithms</td>
<td>94</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

4.1 Example of executable program ........................................... 25
4.2 Example of sample points .................................................. 25
4.3 Example of estimation of rates of performance event rate using the observed ratio estimator ............................................. 25

5.1 Summary of reconstruction error. Reconstruction techniques are ordered in increasing error (MASE). ............................................. 46

6.1 Example of execution trace .................................................. 51
6.2 Relationship between $C_{\varepsilon}(t)$ and $C_{\varepsilon}(d)$ .................. 53
6.3 Example of estimation of performance events per estimation block .......... 68
6.4 Example of different methodology to estimate performance event per instruction. 69
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A linear time algorithm to align <em>sample points</em> to instruction trace.</td>
<td>77</td>
</tr>
<tr>
<td>The Pool of Adjacent Violators Algorithm (PAVA).</td>
<td>88</td>
</tr>
<tr>
<td>Modified PAV Algorithm.</td>
<td>92</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to thank my family and friends for their support in this journey towards a Ph.D.

I would like to thank my advisor, Isaac D. Scherson, for his patience and mentorship.

Also, I would like to acknowledge the funding sources that made my dissertation research possible. Fellowship support was provided by the University of California Institute for Mexico and the United States (UC-MEXUS) and the Consejo Nacional de Ciencias y Tecnología (CONACYT) for the UCMEXUS-CONACYT fellowship that covered tuitions, fees and stipend during five years of graduate studies. Additional fellowship support was provided by the UCI Graduate Division through the Miguel Velez Scholarship.

Formatting of this dissertation is based on Lars Otten’s LATEX[38] and Pawel Pszona’s dissertation [39].
CURRICULUM VITAE

David Carrillo Cisneros

EDUCATION

Doctor of Philosophy in Computer Science 2015
University of California, Irvine Irvine, California

Master of Science in Statistics 2015
University of California, Irvine Irvine, California

Master of Science in Computer Science 2012
University of California, Irvine Irvine, California

Bachelor of Engineering in Computer Systems 2007
Instituto Politécnico Nacional Mexico City, Mexico

RESEARCH EXPERIENCE

Graduate Research Assistant 2009–2014
University of California, Irvine Irvine, California

TEACHING EXPERIENCE

Teacher Assistant 2013–2015
University of California, Irvine Irvine, California
ABSTRACT OF THE DISSERTATION

Accurate Estimation of Occurrences of Performance Events

By

David Carrillo Cisneros

Doctor of Philosophy in Computer Science

University of California, Irvine, 2015

Professor Isaac D. Scherson, Chair

Improvements in performance and energy efficiency often require deep understanding of the complex interactions between hardware and software components. Hardware performance events provide low-level insights about the behavior of a program in real hardware while imposing negligible overhead. Such low overhead allows real-time profile in production environments and makes them useful tools for Feedback Directed Optimization, Software and Hardware Validation, Security, and Performance Analysis among many other applications.

When performance counters are sampled, the perturbation imposed on the monitored task increases with the sampling frequency; restricting the maximum number of sample points that can be acquired in a single run to a few hundreds of sample points per second of execution. This limited sampling frequency, together with the non-deterministic nature of many performance events and the systematic and random errors in the number of measured events caused by bugs and design limitations, had restricted the utilization of performance events in applications that require fine granularity and precision. The error and non-determinism of performance events makes imperative to develop solid methodologies to analyze the data collected from performance counters.

This dissertation presents what we believe is the first formal treatment of methodologies aimed to increase the precision of estimates of performance events in two scenarios: (1)
The first scenario is estimation of a trend performance trace; the problem of estimating the cumulative number of performance events that have occurred at each moment during the execution of a task. We propose the utilization of a regression model that combines sample points from different executions and enforces monotonicity constraints in order to increase precision and granularity of the estimator. Additionally, we compare multiple approaches to interpolation of unobserved points in the trace. We present experimental verification of the superiority of the proposed methodology. (2) The second scenario is the attribution of performance events to dynamic instructions. We propose a methodology to build a regression model that associates performance events with dynamic instructions, rather than static ones, allowing to separately estimate performance events for different instances of the same static instructions, ie. first iteration of a loop versus others. We demonstrate that the presented methodology provide more accurate estimates than the traditional approach, while requiring much less sample points.

Finally, we study the structure of the regression problems created in each scenario and show two particular cases that can be solved in linear time and space, either by mapping the problem into a total order Isotonic Regression problem, solvable in linear time with the well-known Pool of Adjacent Violators algorithm, or using a new dynamic algorithm, linear time algorithm proposed in this dissertation. In both cases, the linear time complexity is a significant improvement over the traditional Non-negative Least Squares, Linear or Quadratic Programming utilized to solve the general case.
Chapter 1

Introduction

Accurate profiling is an indispensable tool to understand the complex interactions between hardware and software in modern systems[53]. In the hardware side, heterogeneous systems and multi-core microprocessors with complex cache and memory hierarchies are the norm. Modern software is concurrent, and often executed in virtualized environments that execute mixtures of heterogeneous applications, making unfeasible to predict before-hand the performance impact of new designs and implementations.

Together with software and hardware, the methodology and tools necessary to accurately observe and measure the behavior of modern systems had increased their complexity. The quest towards energy efficiency has turned the traditional performance optimization problem in a trade-off between speed and energy consumption. The variability in execution time, behavior and sensitivity to execution context of systems has increased due to energy conservation techniques such as low power states and frequency scaling, ubiquitous in modern hardware. Furthermore, the performance and energy consumption is often affected by timing and synchronization issues, specially in modern multi-threaded applications, limiting the overhead that a profiler can impose in a task before rendering the profiling data unacceptable.
Many tools that in the past were commonly utilized to understand the interaction between software and hardware are being confined to niches. For example, the high complexity of modern hardware and the effect of system load, temperature, and interactions with external subsystems and tasks have increased the cost and running time of instruction level simulations to a level unpractical for most applications[3]. For low-level profiling, software based solutions such as instrumentation and software-based sampling techniques typically impose enough overhead to make them unsuitable for production environments. Even when adequate, without specific hardware support, it is impossible or impractical to observe many of the circumstances that caused stalls[57], over-utilization of memory subsystems[23], or speculation problems among other problems.

As a way to address the overhead of profiling and to provide additional information about the internals of the microprocessor, hardware supported methods for profiling have slowly but steadily gained ground in recent years[23]. Most modern microprocessors include a number of Performance Monitoring Units (PMUs)[14, 20] that monitor the occurrences of performance events. Performance events are instances of hardware related activities performed during the execution of a program. Some examples of performance events are instructions-fetched, cache-misses, branch miss-predictions, and write-memory accesses. The types of performance events available in each microprocessor model varies widely between manufacturers and architectures. Although originally designed for validation and testing, PMUs have been constantly refined to improve their suitability for profiling and to incorporate features that allow new applications[44].

A PMU contains a small number of special registers named Hardware Performance Counters (HPCs) [14][20] that are designed to count the occurrences of performance events. Depending on the microprocessor model, there may exist constraints in the performance events that can be programmed simultaneously in the set of HPCs in a microprocessor. The number of programmable HPCs, although variable among microprocessors models, rarely exceeds six
in commercially available microprocessors[14][20]. The limited number of counters available usually requires to either execute a program multiple times, each time with different events, or periodically change the events that are programmed in the HPCs counter during the execution of a program. The latter is known as multiplexing of performance events[41].

Most types of HPCs can be configured to trigger a hardware interruption once a predetermined number of events has occurred. Usually the operative system’s kernel handles this interruption and retrieves the value stored in the HPC that triggered the interruption event and possibly other attributes of interest such as Instruction Pointer (IP), hardware clock, state of the software stack, recent branches registers[11], or others, depending on the hardware and software support[13]. Such interruption and the subsequent collection of information is referred to as a sampling event.

1.1 Challenges in the Utilization of PMUs

Initially, HPCS were intended for development, testing and validation of microprocessor’s architecture. Manufacturers did not emphasize their correctness and the documentation was scarce[52]. Within the last decade, as new usage scenarios are discovered, software support has matured, and looser transistor budgets had become available for chip designers, manufacturers have increased the emphasis on the development and testing of performance monitoring features.

There is a trade-off in microprocessor design between speed, energy consumption, and cost versus completeness and accuracy of monitoring capabilities. Even in present hardware, multiple limitations in the implementation of PMU persist that increase the complexity of the analysis of information collected using HPC. This section explores some of them, namely, perturbation, counting errors and non-deterministic results.
1.1.1 Perturbation Caused by Sampling of HPCs

Although there are not significant performance penalties for keeping track of the counts of performance events, the execution of the interruption routine may take a considerable amount of computing time\[52, 54, 17\] and affect the timing of other instructions in the pipeline. In architectures that allow precise sampling of instructions, such as Intel’s PEBS or AMD IBS, selected instructions are executed through a modified pipeline that collects detail information about it at the cost of performance \[3\].

Furthermore, the execution of the interruption routine usually modifies the behavior of the profiled task significantly. Loading the software routine pollutes caches and MRB registers, and affects timing of asynchronous subsystems such as I/O, among many other subtle effects\[23\]. The perturbation caused by each sampling interruption is significant enough to discourage high sampling rates\[52, 17\]. Software tools such as *perf event*\[13\] limit the sampling frequency to a few thousands of sample points per second of execution to avoid excessive miss-count caused otherwise \[52\].

Some recent improvements have been rolled out by hardware manufacturers to decrease the cost of reading from performance counters. In x86 microprocessors the instruction **RDHPC** allows reading the value of an HPC from user space. Reducing perturbation imposed by the interruption routine is also a primary objective in many software libraries that wrap around HPC \[17, 50, 54\].

1.1.2 Error and Non-determinism

A difficulty when working with performance counters is that the information collected through sampling is not always accurate. Some of the sources of error are:
1. The microprocessor’s state that is captured by the software interruption is distinct to the microprocessor’s state at the moment that the performance event that triggered the sample interruption occurred, due to delay in the trigger mechanism of the PMU [14, 20].

2. Bugs and limitations in the design of microprocessor are known to cause both systematic and non-deterministic errors even in commonly used performance events such as instruction-retired[52, 53, 11].

3. Bugs and limitation in software interruption routines of software libraries[13, 6] may affect the count and other values collected during sampling[53].

4. Due to micro-operands fusion, out-of-order execution and other optimization techniques, there are some scenarios where the mapping of program instructions to samples of performances[10] is not direct. ie. micro-operands of two instructions may be fused together there may be no sample points reporting the IP of one of the static instructions even if such instruction would have generated a performance event had it been executed within a different sequence.

To reduce the error in the collected information, manufactures have devised precise sampling modes that guarantee that the collected instruction pointer (IP) accurately corresponds to the instruction that caused the occurrence of the performance event that triggered the sampling interruption. Unfortunately only few existing performance events are supported in precise sampling modes. The precise sampling mode is discussed in full detail in section 2.2.

Besides accuracy problems, the information gathered using HPCs tends to be non-deterministic due to numerous factors. In modern microprocessor, frequency scaling or activation of low power states has significant impact in performance events that depend on timing or that signal resource contention, such as cache-misses. Additionally, changes in the system load and
competition for scheduled time and other resources (I/O, network, etc.) affect the behavior of the task even in controlled environments.

1.2 Performance Diagnostics using Performance Events

The information collected through sampling of HPCs is commonly utilized by programmers and/or tools to improve performance, reduce power consumption, isolate bugs, detect security and reliability risks, and schedule tasks among many others. Not all applications require the same type of profiling information; some applications may require to precisely associate performance events with instruction while for others it suffices to know the general trend of occurrences of performance event with respect to time or another performance event or simply the total counts of performance events.

In this work we focus on two different signals of information that can be obtained from a trace of performance events:

1. Association of performance event of interest with timer and/or other performance events.

2. Association of performance event of interest with machine instructions.

Since a timer can also be considered a performance event, the previous distinction in point (1) is merely didactic.

Applications that utilize signal in point (1) usually utilize a trace of performance event as source of information. Some examples are:

- Characterization of task through traces of performance counters, including power consumption profiles[26, 47, 59].
• Content aware scheduling. Adaptive scheduling of threads (or processes) to minimize resource contention[47].

• Malware detection through signatures of performance events traces[19].

• Detection of race conditions[46].

• Dynamic optimization to improve performance or decrease energy consumption[32].

Applications that utilize signal in point (2) usually aim to identify parts of the binary that create bottlenecks or induce other performance problems. These application usually utilize a frequency table of IP registers (See 4.1) as a summary of the information collected from the sample. Some examples are:

• Identifying instructions or regions of code that generate excessive memory problems (data/instruction cache-misses, TLB misses, excessive bandwidth consumption.[23, 31, 51].

• Identify regions of code that generate a high number of branch mis-predictions in order to refactoring it[33].

• Estimation of frequency of execution of basic block to guide Feedback Directed Optimization (FDO)[11, 10].

• Identify branches or loop that should be refactored by a FDO compiler in order to improve effectiveness of hardware prediction and/or hardware pre-fetching.

• Identify data and instruction that cause false-sharing.

In the following sections we detail in the difficulties associated with collecting information for each one of the two scenarios described above:
1.3 Association with other Performance Events

In this scenario, the trace of performance events can be interpreted as a time series (the trace of performance events is formally defined in section 2.3.2). Due to the restricted sampling rate that is achievable without excessively perturbing the task (as discussed in section 1.1.1), it is common to combine traces from multiple executions to increase the resolution of the sample. The resulting set of sample points has certain attributes that usually complicate its interpretation:

1. There may exist more than one sample point indexed at the same time (multiple observations at the same time point).

2. There may exist no sample points indexed at some values in time. In fact, it is common that most of the possible values in the index event or timer have no associated sample point.

3. Due to variable behavior of the task between executions, there is no guarantee that the counts of performance events in successive sample points are monotonically increasing, as would be expected of a non-decreasing counter.

In literature it is common to combine sample points from multiple executions of the program in a single time series and interpolating between neighboring points, ignoring the execution that originated the sample, despite the problem described above. Finding a good estimator of attributes or parameters of the process that generates the performance events as a function of the reference index is an important problem in its own right, since better estimations can increase the performance of classification and predictions techniques that utilize the “improved” data. Nevertheless, no formal treatment of estimation of trends or parameters for trace of performance events has been found in literature. It is intuitive that better methods of estimating trends exists. An ideal estimator would combines sample points from
multiple executions without violating the monotonicity constraint among successive counts. It would also provide estimations to values of the reference index even if they are not part of the sampled performance trace. Chapter 5 explores these ideas to find a more efficient way to estimate trends and extract information from traces of performance events.

1.4 Associating Performance Events to Instructions

Existing approaches to map occurrences of performance events to machine instructions utilize the value of the Instruction Pointer (IP) reported in a sample point[45, 10], summarizing it in a frequency table of samples per IP. A caveat of this approach is that, unless precise IP sampling is utilized (see section 2.2), there is no guarantee that such value really corresponds to the IP value of the instruction that generated the performance event that triggered the sample.

Even with precise IP sampling, the frequency table approach suffers multiple shortcomings such as: (1) it discards all information about instructions that did not trigger a sampling event but contributed to the overall count of performance events, (2) different executions of same instructions are indistinguishable (i.e., it is not possible to estimate separately the first iteration of a loop from others), and (3) it requires very large number of sample points in order to estimate the number of performance events associated with instructions that are rarely executed or that generate small number of performance events.

The estimation of the number of performance events caused by specific instructions could be improved by incorporating additional information besides the trace of performance events. Some sources of additional information that can be included and that are derivable from the binary executable are:

- Machine instructions in binary executable.
- Instruction trace.
- Known relationship between machine instructions and performance events.
- Other events traces.

A possible approach to improve the accuracy of instruction level estimation of occurrences of performance events is to associate the sampled value of a performance counter with the dynamic instructions that were executed between the times the samples were taken. This approach incorporates information from the instruction trace and the task’s binary to estimate the number of performance events generated by any dynamic instruction and not only for those ones whose address is present in the IPs collected during a sample event.

Nevertheless, traditional samples from performance counters do not contain information about which dynamic instructions had been executed between sample points, except by the one associated with the performance event that triggered the sampling event. A natural approach is to collect a instruction trace for the execution of the task, but as there is no information about the position in the instruction trace of an instruction associated to a sample-point, there is no straightforward way to map IPs in sample points to instruction trace. The problem is complicated furthermore by the factor described in sections 1.1.1 and 1.1.2.

An additional complexity is introduced by the difficulty of interpreting the information collected. As discussed in section 1.1.2, the deep pipelines, super-escalar design, and speculative execution ubiquitous in modern microprocessors imply that tens of instructions can be executed concurrently, some of them speculatively; making it difficult to associate the state of a microprocessor with the effects caused by particular dynamic instructions. Furthermore, the translation to microcode and its optimization carried by the front-end in modern x86 microprocessors [14, 20] implies that impact of an instruction in the back-end is dependent on surrounding instructions.
1.5 Our Contribution

In the preceding sections we hinted the problems and solutions included in this dissertation. Now we put them all together and summarize our contribution:

1. In Chapter 5 we present a formal treatment to the problem of estimating a trend for the time series given by the set of performance event traces collected during sampling. We present a simple and formal framework that separates the information collected during the sample from assumptions about the underlying process that generates the performance events. We pose the intuitive methodology that is common in literature using in the proposed framework model and compare it with other two proposed model that enforce monotonicity in the estimation of the trend. We compare the predictive power of interpolation techniques and of enforcing monotonicity in each one of the three models.

2. In Chapter 6 we present a methodology to combine information from both performance events traces and instructions traces in order to estimate the number of performance events generated by individual dynamic instructions. The proposed methodology:

   - Associates instructions traces to performance event traces from multiple, distinct executions of the task.

   - Allows to specify groups of dynamic instructions to estimate separately, rather than only static instructions, as the state of the art allows (ie. first iteration in a loop versus the rest of iterations).

   - Map problems of estimation of performance events to a regression problem where instructions (or groups of them) are predictors and the occurrence of performance events are the predicted variables.

   - Specify conditions for the estimation problem to be estimable.
3. In Chapter 7 we study the regression problems proposed in chapters 5 and 6 and propose solutions in linear time to some common particular cases. One particular case occurs when assumptions consistent with monotonicity between successive samples are imposed. This approach induces regression problem that can be efficiently solved using the well-known Pool of Adjacent Violators[4]. The second particular case explored occurs when assumptions consistent with non-negativity increments between sample points are imposed. This approach induces a regression problem that can be efficiently solved using a novel, linear-time dynamic algorithm that is proposed in the same chapter.
Chapter 2

Preliminaries

2.1 Common Terms

- **Indexed Family**: a collection of values indexed by an index set.
- **Code-path**: a code-path is a the path in the Control Flow Graph of a program that is followed during an execution.
- **Static Instruction**: a machine instruction in a program.
- **Dynamic Instruction**: a instruction executed by a microprocessor.
- **Performance Events Trace**: the collection of samples of performance events collected during one or multiple executions, usually indexed by time. A detailed definition is given in Definition 2.5.
- **Instruction Trace**: a sequence of dynamic instruction executed by a microprocessor, ordered by the time that the IP pointed to them.
- **Trace of Performance Events**: a sequence of performance, indexed by time.
- **Basic block**: a sequence of static instructions with has a single entry point and single exit point.
- **IP skid**: The number of instructions in the instruction trace between the position in
the instruction trace of the dynamic instruction that is associated with a sample point
collect during the sampling interruption to the position in the instruction trace of the
dynamic instruction that really generated the performance event that triggered the
sampling event.

2.2 Sampling of HPCs

During asynchronous sampling of performer events. A HPC is programmed with an event
code and a sampling period, denoted by $p$. The event code denotes the performance event
to be counted and $p$ denotes the number of occurrences of the performance event to count. A
sampling event is triggered on the $p$-th occurrence of the performance event, although it may
not occur immediately[11], we refer to these delays as sampling delay. When a sampling
event occurs, a routine collects the values of registers in the microprocessor and/or software
variables.

In this work we concentrate in two specific values, out of the many that could ideally be
collected:

- The memory address of the instruction that generated the performance event that trig-
gerated the sampling event. A sampling methodology can collect such memory address
accurately it has the property of precise IP sampling.

- The count of performance events occurred since the last sampling event. A sampling
methodology that can collect such count has the property of precise count sampling.

Precise IP sampling is instrumental for the work in chapter 6. In most software libraries, a
software interruption routine will report the value of the program counter stored in the stack
at the moment that the hardware interruption was triggered; unfortunately, without explicit
hardware support there is no guarantee that the program counter in the stack is that of the instruction that triggered the sampling event. This is due to numerous factors such as:

1. In most architectures, there is a delay between the overflow of a HPC register and the trigger of the hardware execution [14, 20].

2. In multiscalar and pipelined architectures, any instruction in some stage of execution can generate a performance event, regardless of the current value of the program counter.

The problems above cannot be handled without hardware support. In x86 architectures, manufactures provide features that avoid the previous problems for some specially designed performance events. These are:

1. **Instruction Based Sampling (IBS)**: In AMD microprocessor, IBS allows the microprocessor to tag a specific instruction and monitor its execution, in order to specially recover registers associated to it in the case of that a performance event is generated[3].

2. **Precise Event Based Sampling (PEBS)**: In Intel microprocessors, PEBS allows the microprocessor directly stores sample points with a fixed format in a specially designated area of memory, with the consequent benefits due to decrease perturbation. The samples are guaranteed to contain an IP skid of at most one[3].

Another set of problems affect the accuracy of the count of performance events. Notably:

1. Let instruction $d$ be the dynamic instruction that generated the $c$-th performance such that $c = p$ where $p$ is the sampling period. Due to out-of-order execution, there may be another instruction, $d'$, that generated the $(c - k)$-th event, where $k \in \mathbb{N}$ and such that $d$ appears before $d'$ in the instruction trace.
2. The occurrences of performance events generated between the triggering of a sampling event and the occurrence of the sampling delay are not counted.

3. Hardware and software bugs introduced miscounts. One common source of over-count is the execution of instructions within the software interruption routine, after the HPC hardware has been re-enabled for counting but before the interruption routine returns.

There are not known hardware architectures and software libraries that provide precise count sampling.

2.2.1 Register Overflow

In practice, register overflows may occur, but this case is handled appropriately in all software libraries he have surveyed, including Linux’s perf event[13], Perfmon2[22], Limit[17] and PAPI[6]. Therefore, it is guaranteed that the collected count from performance counters is non-decreasing. This guarantee of non-decreasing counts is enforced as monotonicity constraint throughout the paper.

2.3 Formal Definitions

2.3.1 Notation

- $[a, b, a, c, \ldots]$ denotes a multiset.

- $\langle a, b \rangle$ denotes a tuple with elements $a$ and $b$.

- $\mathbb{1}_y(x)$ denotes the indicator function that is equal to one if $y = x$ and zero otherwise.
• $\mathcal{P}(A)$ denotes the power set of the set $A$. This is the set of all subsets of $A$, including $\emptyset$ and $A$ itself.

• $S(B)$ denotes the set of all non-empty contiguous subsequences of the sequence $B$.

• $\odot$ entry-wise product between two vectors or matrices of equal dimensions.

• $\mathbb{N}^0$ and $\mathbb{R}$ denotes the set of all natural numbers, including zero and the set of all real numbers, respectively.

2.3.2 Basic Definitions

Definition 2.1. An executable-program is represented by the 3-tuple $P = (I, \Pi, \theta)$ where:

• $I$ is a set of distinct static instructions (including instruction’s operands).

• $\Pi : \{\pi_i\}$ is a set of instructions’ addresses in a executable-program.

• $\theta : \Pi \rightarrow I$ is a surjective function that maps memory addresses in $\Pi$ to instructions in $I$.

Our goal is to associate performance events to instructions in $P$. To collect traces of performance events, the executable-program $P$ is executed $R$ times. Note that $R = 1$ is a valid particular case. For the $r$-th execution, $n_r$ sampling points are collected (as defined in Definition 2.4). Totaling $n = \sum_{r=1}^{R} n_r$ sampled points where $r = 1, \ldots, R$.

Note that definition 2.1 does not restrict $P$ to be a whole task or program. It is any set of executable instructions.

Definition 2.2. The execution time of the $r$ execution of $P$ is given by the value of a non-decreasing counter denoted as $T$. The values taken by $T$ during the $r$-th execution range
from $t_{r,\alpha}$, at the beginning of the $r$ execution of $P$, to $t_{r,\omega}$, at the end of the $r$-th execution of $P$.

For the sake of notation brevity, when the context allows it, $T$ also denotes the set of possible values taken by the counter $T$.

**Definition 2.3.** The cumulative number of occurrences of an event of type $E_\sigma$ counted during the $r$-th execution of $P$ up to execution time $t_{r,i}$ is given by the non-negative, non-decreasing stochastic function $C_{E_\sigma}(t_{r,i})$.

**Definition 2.4.** A **sample point**, denoted by $\sigma_{r,i} = \langle t_{r,i}, c_{r,i}, \pi_{r,i} \rangle$, is the $i$-th observation of the value of a performance counter of interest during the $r$-th execution of the task where:

- $t_{r,i}$ is the value of the execution timer at the moment the values $c_{r,i}$ and $\pi_{r,i}$ are collected.

- $c_{r,i}$ is the value of the performance counter attributed to the sample event, this is $C_{E_\sigma}(t_{r,i}) = c_{r,i}$. Note that $c_{r,i} \geq 0$ and $c_{r,i-1} \leq c_{r,i}$ since the value of a performance counter is a non-negative, non-decreasing count. See section 2.2.1 for overflow and miscounts considerations in real systems.

- $\pi_{r,i} \in \Pi$ is the address of the static instruction attributed to the last performance event counted in $c_{r,i}$. Depending on the sampling mode it can be the exact IP of the instruction that triggered the sample (plus the skid) or another the value of the program counter at the moment of the sample collection.

Additional information may be collected during each sample point that, although useful in many scenarios, it is not required for the material exposed in this work. In real systems, the sampled values may not be entirely accurate. It is common that the value of $t_{r,i}$ is not exactly the one at the time the value reported $c_{r,i}$ was set, or that $\pi_{r,i}$ presents a skid. Details regarding the accuracy of the $\pi_{r,i}$ value are discussed in section 2.2. In this work we assume
that the values are correct unless otherwise stated and let the estimation methodology to
handle the errors introduced by the measurement mechanisms.

**Definition 2.5.** A *performance events trace* for $R$ executions is given by $\Sigma$ where
$\Sigma = (\sigma_{1,1}, \ldots, \sigma_{1,n_1}, \sigma_{2,1}, \ldots, \sigma_{R,n_R})$ is the sequence of all sample points collected in the $R$
executions. The choice of order for the sequence $\Sigma$ is arbitrary.

In literature, it is common to refer to all the sample points of a execution as a *trace of
performance events.*
Chapter 3

Problem Statement

The estimation of the number of hardware performance events (i.e., branch misses, instructions retired, stalled cycles) generated by dynamic instructions provides information about the interaction of software, hardware, and execution context. Such information is utilized by programmers and/or tools for many uses including performance analysis, reduction of power consumption, isolation of bugs, detection of security and reliability risks, and tasks scheduling.

In modern CPUs, polling and/or sampling of hardware performance counters are common techniques utilized to obtain a sample of a trace of the performance events occurred during the execution of a task. However, the applications of such events trace is limited due to the imprecision of the observations and the relatively low sampling rate that is achievable without disturbing the profiled task. Despite these problems, trace of performance events has become a standard tool to understand the behavior of tasks and new usage scenarios for it are constantly discovered[9, 11, 10, 2, 23].

In some scenarios, when insight about the relationship between particular instructions and performance is required, it is necessary to associate performance events with specific ma-
chine instructions or with other events of interest. Existing approaches map occurrences of performance events to machine instructions by counting the number of sample points with the same value on the Instruction Pointer (IP) collected during the sampling event. Although simple, this approach suffers shortcomings such as: (1) imprecision and nonsensical results caused by miscounts and skids in the sampled values introduced by hardware and software limitations, (2) it discards all information about performance events generated by instructions that did not trigger a sampling event but contributed to the overall count of performance events, (3) do not distinguish between dynamic instructions (i.e., it is not possible to estimate separately the first iteration of a loop from others), and (4) it requires very large number of sample-points in order to estimate the number of performance events associated with instructions that are rarely executed or that generate small number of performance events.

Three techniques to alleviate the problems enlisted above without increasing the perturbation of the sampling process are:

1. Extending the event trace with an instruction trace. The dynamic instructions that were executed between successive sample points in the events trace provide information about the sequence of operations carried by the microprocessor. Combining the events trace collected through sampling with the instruction trace would allow to perform more accurately estimate the number of performance events generated by each instructions and allow to distinguish between different dynamic instructions of the same static instructions. Nevertheless, associating events trace with instructions trace is complicated by factors such as: (1) the variability of the instruction trace itself that is caused by dynamically shared libraries and variations in the execution path, and (2) the discrepancies between the executed micro-operations executed by the microprocessor and the instruction trace that is caused by out-of-order execution, fusion of microinstructions and branch prediction.
2. Increase the number of observations in the events trace by appropriately merging event traces from different executions. Since different executions are run into different contexts, even repeated executions of a task may generate different event trace. Combining sample points from different executions and reconstructing a single consistent event trace in a time efficient manner is a problem on its own.

3. Estimate the interactions between different performance events. Each performance event is a single facet of the underlying operations followed by a microprocessor and combining multiple performance events with the known relationship between often bring new insights into the behavior or the test of helps to reduce the error of the estimation.

Any of the three techniques mentioned (or combinations of them) implies additional processing that often represents significantly computation time (i.e. in modern commodity microprocessors, a typical instruction trace contains thousands of millions of instructions per second of running time) and attention must be taken to algorithm design and complexity.
Chapter 4

Related Work

It is a common strategy to complement instrumentation with samples of performance counters to diagnose performances or reliability problems [2] or optimization opportunities [9].

An empirical analysis of reconstruction methodologies for the problem of multiplexing of performance counters is covered in [36] and estimates the quality of imputation strategies to approximate total counts of a single run. In contrast, we aim to combine multiple runs into a single estimator, with much finer time resolution than what is required for the multiplexing problem.

4.1 Traditional Methodology to Associate Performance Events and Instructions

In this section we will analyze how performance events are associated with static instructions in existing literature and explore potential shortcomings in the existing methodology.

The ubiquitous approach to associate performance events to instructions in literature [10, 3,
and existing software \cite{13, 17, 6} consists in reporting the proportion of samples collected for each distinct value of \textit{IP}. This is effectively an estimator of the proportion of performance events triggered by a particular instruction. We refer to this estimator as the \textit{observed ratio estimator} and can be expressed as:

\textbf{Definition 4.1.} An \textit{observed ratio estimator} for the proportion of performance events triggered by instruction with address $\pi_a$ is given by:

$$\hat{\alpha}_r(\pi_a, \sigma) = \frac{\sum_{\sigma_{r,i} \in \Sigma} 1_{\pi_{r,i} = \pi_a}}{|\Sigma|}$$  \hspace{1cm} (4.1)

\subsection{4.1.1 Example}

Table 4.1 shows an example of an \textit{executable program} that is part of a larger program. The selected fragment contains a simple loop with three iterations that add one to register \texttt{EAX} and multiply by 3 each time. \texttt{EAX} takes the values 3, 12 and 39. Assume that such program is executed four times using random period-based sampling with sampling periods drawn from an discrete uniform distribution for the integers in the $[4, 6]$ interval. The example counts the occurrences of the event type \textit{retired microoperations}. Assume that the sample points shown in table 4.2 are collected. The periods in this examples are chosen to be extremely small for clarity of exposure, in real scenarios they are usually orders of magnitude larger. Note that after each sampling event, all instructions retired in the same cycle are not counted, as occurs in real performance counters\cite{10}. The \textit{observed ratio estimators} are given in table 4.3. Note that instructions at addresses 02 and 03 have attributed values of zero and that instruction 05 has an estimated value much larger that its true proportion.
Table 4.1: Example of executable program augmented with the number of micro-operations associated with each instruction.

<table>
<thead>
<tr>
<th>$\pi$ tag</th>
<th>Instruction</th>
<th># $\mu$ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>MOV EAX, 0x0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>MOV EBX, 0x3</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>MOV ECX, 0x0</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>ADD EAX, 0x1</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>MUL EBX</td>
<td>3</td>
</tr>
<tr>
<td>05</td>
<td>ADD ECX, 0x1</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>CMP EAX, 0x27</td>
<td>1</td>
</tr>
<tr>
<td>07</td>
<td>JNE loop</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.2: Example of sample points for four executions of the task ($r$ index) using random period-based sampling. Each period is chosen from an uniform distribution in the integers of the interval $[5, 9]$. The event type microoperations retired is counted. The value $c_{r,0}$ is the value of the performance counter before instruction in 00 is executed. When a sample occurs, the performance events generated by instructions in the pipeline that occur after the sampling interruption is triggered are not counted [10].

<table>
<thead>
<tr>
<th>$r$</th>
<th>1 1 1 1 1 2 2 2 2</th>
<th>3 3 3 3 4 4 4 4 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>0 1 2 3 4 0 1 2 3</td>
<td>0 1 2 3 0 1 2 3 4</td>
</tr>
<tr>
<td>$p_{r,i}$</td>
<td>6 7 8 6 9 1 7 8 6</td>
<td>0 5 6 8 5 7 6 9 6</td>
</tr>
<tr>
<td>$c_{r,i}$</td>
<td>6 7 15 21 30 1 7 15 21</td>
<td>0 5 11 19 5 7 13 22 28</td>
</tr>
<tr>
<td>$\pi_{r,i}$</td>
<td>- 00 06 05 07 - 04 05 04</td>
<td>- 04 04 05 - 01 05 07 06</td>
</tr>
</tbody>
</table>

Table 4.3: Estimated proportions of performance events per instruction using the observed ratio estimator. The value $o_r$ represents the true ratio and $o_{\pi a, \sigma}$ is the estimated ratio. The estimations for instructions at addresses 02 and 03 are zero since no sample points had such addresses associated to them. The estimation of 04 is significantly smaller than the true ratio.

<table>
<thead>
<tr>
<th>$\pi_a$</th>
<th>00 01 02 03 04 05 06 07</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_r$</td>
<td>1/24 1/24 1/24 3/24 9/24 3/24 3/24 3/24</td>
</tr>
<tr>
<td>$o_r(\pi_a, \sigma)$</td>
<td>1/15 1/15 0 0 4/15 4/15 2/15 2/15</td>
</tr>
</tbody>
</table>
4.1.2 Limitations of traditional approach

In this paper we discuss limitations on the traditional methodology to estimate the ratios, as presented that exists, regardless of the known problems in the real world implementations [55, 54, 10]:

**Low precision**

Only instructions that trigger a sample can be estimated. Instructions that are executed more often dominate making necessary to collect a large number of sample points to estimate performance events for instructions that do not dominate the generation of performance events. This makes harder to collect detailed information of instructions outside of hot paths. This problem is more serious when the benchmarks and test cases in which the profiling is analyzed is not representative of the real world. Furthermore, empirical evidence shows that, due to biases in the sampling mechanism, the distribution of sampling IPs does not always resemble the true IP distribution[34, 10].

**Estimation restricted to static instructions**

In many scenarios dynamic instructions are of interest. For example:

- Different iterations in a loop may have different behaviors. For example, branch prediction may be able to correctly predict branches in the body of a loop in some iterations, but not in other.

- The behavior of performance events in a block of code, or function can be different depending on the invocation context.
In the scenarios above can be useful to group together arbitrary sets of dynamic instructions, in order to obtain separate estimation of number of occurrences of performance events (or rates of occurrences) for each group. In the traditional approach, all dynamic instructions corresponding to the same static instruction are lumped together. This limitation of the traditional approach is one of the main motivations for the development of the methodology presented in chapter 6.
Chapter 5

Estimation of a Monotonic Trend in Trace of Performance Events

In this chapter we present explore different approaches to estimate trends in a trace of performance events. There are multiple scenarios where the goal in a time series constituted by sample points of values of a performance event as function of a timer or another performance event. Some of these scenarios are:

- Monitoring of uncore/northbridge and/or off-core events such as page-faults, energy-cores, energy-ram, and IO_TRANSACTIONS.

- Monitoring of in-core performance events that have no direct association with individual instructions such as CPU_CLK_UNHALTED (halted cycles) or ITLB_FLUSH (ITLB flushes) or prefetching events.

- Estimating trend for performance traces of software events such as page fault, cpu-migrations or context-switches.

- When the objective is the interaction between two or more performance events rather
than the instructions in the task.

- When the objective is to extract features of the performance trace for applications such as: task-load characterization, or some machine learning technique such as phase detection, clustering, among others.

- When the goal is to detect dynamic changes in behavior, such as online malware detection [19] or scheduling [47] or resource contention.

For the scenarios above, the objective is often to estimate a performance events trace indexed by a counter or performance event. A execution timer as defined in Definition 2.2 can be utilized as index for a performance events trace when certain conditions are met:

**Definition 5.1.** A execution timer, \( T \), is a reference timer for a events trace \( \Sigma \) if:

\[
c_{r,i} \neq c_{r,j} \Rightarrow t_{r,i} \neq t_{r,j} \quad \forall \sigma_{r,i}, \sigma_{r,j} \in \Sigma
\]  
(5.1)

Equation 5.1 implies that the granularity of the reference index is fine enough that sample points from a performance counter cannot take multiple values at the same sampled value of the reference timer in a single execution. For most performance events such condition is easily satisfied by commonly utilized timers such as of sufficiently high resolution, such as Linux’s high resolution timers[48], or by performance events such as instructions fetched or cycles count. This definition also allows other “slowly moving” performance events such as missed-branches, page-faults or energy counters such as RAPL[44], as long as the event \( \mathcal{E}_\sigma \) is sampled with periods large enough to guarantee that equation 5.1 holds.

Multiple events can share the same reference time as long as condition in equation 5.1 is satisfied for all of them. When profiling multiple executions it is preferred to utilize events that are not affected by frequency scaling, such as instructions-retired as index events;
Figure 5.1: Example of time series for energy-cores (Energy consumed per core). Each series of dots represents sample points from a different execution of the same task. Lines show linear interpolation between sample points in the same execution.

specially when the goal is to associate counts with program instructions[10]. Such case is the main topic of Chapter 5 in this dissertation.

This chapter present a study of the problem of reconstructing the trace of performance events for every value in the reference index. In these work we consider three aspects of these problem that affect the accuracy of a reconstruction:

- **Combining Multiple Event Traces**: As discussed in section 1.1.1, the sampling rate that is attainable without introducing significant perturbation is small (hundreds of samples per second)[18]. In order to collect enough sample points to obtain detailed, fine-grained information.

- **Execution Variability**: For most performance events, the behavior of the task differs from one execution to another due to the interactions of the profiled task with a chang-
Figure 5.2: Example of interpolation techniques. Sample points (in black) and lines showing different reconstruction techniques.

In execution context, such as other processes, CPU frequency rescaling, I/O delays, resource competition among processes, etc. Altogether with the inherent variability of the sampling process results in sample points that usually do not behave deterministically. This variability is illustrated in figure 5.1 for a number of sample points in a task repeatedly executed.

- **Interpolation**: The interest may be in values of the reference index that were not sampled, requiring to estimate the value of the performance counter at unobserved points in the execution trace. Interpolation between observed points is usually useful in this cases, but there is no existing research in this topic. Figure 5.2 illustrates several possible interpolation data applied to sample points from RAPL (power energy) counters from a running application.

According to definition 2.3, $C_{E}(t)$ is a non-decreasing non-negative stochastic function that represents the number of occurrences of performance events at time $t$. The goal in this section is to create an estimate of the values taken by $C_{E}(t)$ for all possible values of $t$. We
call this estimate a **monotone trend** of a trace of performance events.

## 5.1 Estimation of a Trace of Performance Events as a Regression Problem

**Definition 5.2.** Let $N_{E \sigma}(t)$ be a non-negative stochastic function such that $N_{E \sigma}(t) = \sum_{\tau \leq t} \tau$. $N_{E \sigma}(t)$ is the increment in the count of performance events $E \sigma$ that occurs during the clock cycles where counter $T$ had value equal to $t$.

**Definition 5.3.** For a sample $\Sigma$, let $U = \{0\} \cup \{ t_{r,i} : \forall \sigma_{r,i} \in \Sigma \}$ be an ordered set of unique values of the reference index.

The elements in $U$ denote the values of the *reference index*. The $i$-th element of $u$ is $u_i$. The first element in $U$ is $u_0 = 0$.

**Definition 5.4.** A **region**, $R_i$ is the the increasing sequence of values of $t \in T$ such that $u_{i-1} < t \leq u_i$.

**Definition 5.5.** The ordered set of regions is given by $R = \{R_i\}$ for $i = 1, \ldots, |U|$. The set is ordered in non-decreasing order of $\max(\{t : t \in R_i\})$. 

---

Figure 5.3: An example of a sequence of sample points. The sampling times $t_1 \ldots t_4$ and counts $c_{t_0} = 0, \ldots c_{t_4}$ are known.
**Definition 5.6.** Let $G \subseteq \mathcal{P}(R)$. The total number of performance events generated during a set of regions is given by the stochastic function $C_{E_a}(G) : T \to \mathbb{N}^0$ such that:

$$C_{E_a}(G) = \sum_{R_i \in G} \sum_{t_j \in R_i} N_{E_a}(t_j)$$ (5.2)

**Definition 5.7.** An imputed region is given by $s_i = \langle \delta_i, G_i \rangle$ where:

- $G_i \subseteq \mathcal{P}(R)$, a set of regions.
- $\delta_i \geq 0$ is the increment in the value of the performance counter of interest assumed to occur while the reference index took values in any $R_i \in G$.

The value $\delta_i$ in the $s_i$ is a realization of the function $C_{E}(G_i)$, as given in Definition 5.6.

**Definition 5.8.** Let $S = [s_i]$ be the multiset of all imputed sets.

**Definition 5.9.** A region imputation function is a function $I : \Sigma \to S$ that maps the set of sample period into the multiset of imputed sets.

For notation simplicity, let:

- $\Sigma(T)$ be the ordered set of $\sigma_{r,i} \in \Sigma$ ordered in non-decreasing order of $t_{r,i}$.
- $\sigma_i$ be the $i$-th element in $\Sigma(T)$.
- $t_i$ be the reference timer element in $\sigma_i$.
- $c_i$ be the count element in $\sigma_i$.

### 5.1.1 Types of Imputation

Similarly to the methodology presented on Chapter 6, the approach presented in this chapter transforms a set of sample points $\Sigma$ into imputation intervals. The choice of the region
imputation function, \( I \), captures the assumptions about the underlying, unobserved process that generates the performance events and the execution-trace. In this paper we consider two types of imputations: (1) **Preceding Regions Imputation** (PRI), that is implicitly used in the vast majority of surveyed literature and software. It assumes a increment in counts is due to all preceding regions to a *sample point*, and (2) **Observed Regions Imputation** (ORI), that assumes a increment in counts is due to regions occurred since the last sample point of the same repetition.

**Preceding Regions Imputation (PRI)**

This imputation ignores information about the execution that generated the samples and assumes that performance events in \( c_{r,i} \) were caused by all regions.

\[
I_p(\Sigma) = \{(c_i, \{R_k : \forall R_k \in R \text{ such that } \max(t \in R_k) \leq t_i \}) : \sigma_i \in \Sigma^{(T)} \}
\]  
(5.3)

**Observed Regions Imputation (ORI)**

This imputation considers only the occurrences of performance events occurred between successive samples in the same execution.

\[
I_o(\Sigma) = \{(c_{r,i} - c_{r,i-1}, \{R_k : \forall R_k \in R \text{ such that } t_{r,i-1} < \max(t \in R_k) \leq t_{r,i} \})\}
\]  
(5.4)

### 5.1.2 Regression Model

For a set of regions, \( R \), and a imputed set \( I \), let \( X \) be a \(|S| \times |R|\) binary matrix such where:

\[
x_{i,j} = 1_{R_j \in G_i}
\]  
(5.5)
where $G_i$ is the second element in the tuple $s_i$ where $s_i \in S$, and $R_j \in R$.

Let $Y$ be a vector of $|S|$ elements such that:

$$y_i = \delta_i$$  \hspace{1cm} (5.6)

where the $i$-th element corresponds to the $\delta_i$ of the $i$-th imputation in $S$. Then, the regression model is given by:

$$\min_{\beta} L (Y - X \beta)$$

subject to $g_i(\beta) \leq k_i$, $i = 1, \ldots, m$  \hspace{1cm} (5.7)

where $m \in \mathbb{N}^0$, $g_i(\beta) \leq k_i$ represent additional constraints that may be imposed to the optimization problem and $L : \mathbb{R}^{|B^K|} \to \mathbb{R}$ is the error function to be minimized.

The coefficients $\beta_i$ represent the estimated number of occurrences of performance events for the time interval represented by the $i$-th region. The set of coefficients $\beta$ can be interpreted as a new set of data points that have been regularized.

Note that matrix $X$ is guaranteed to be full-rank for any choice of $B$ (or $Q$) due to the choice of using regions rather than individual values of reference index.

**Monotonic Constraints**

A direct consequence that the fact that $C_{\mathcal{C}_{\sigma}}(t)$ is a non-decreasing function is that $N_{\mathcal{C}_{\sigma}}(t) \geq 0$.

It is a natural constraint to enforce such condition; when done, the estimates for $\beta$ can be solved using constrained estimation algorithms such as Non-negative Least Square Error or Linear Programming. Chapter 7 shows efficient algorithms to solve the two variations of constrained optimization that are generated by imputations type PRI and ORI. The effect
of enforcing monotonic constraints is verified experimentally in section 5.3.

5.1.3 Imputation Types and Design Matrices

The matrix $X$ and vector $Y$ will take values depending of the imputation utilized. The two types studied in this chapter generate a lower triangular matrix.

PRI Model

**Lemma 5.1.** The design matrix $X$ created by PRI imputation is a full-rank binary matrix which $(i,j)$ is given by:

$$x_{i,j} = 1_{\max(t \in R_j) \leq t_i} \quad (5.8)$$

and the rows and columns in $X$ can be arranged to form a lower trapezoid matrix such that:

$$x_{i,j} = 0 \implies x_{i,k} = 0 \quad \forall \ k = j, \ldots, |R| \quad (5.9)$$

**Proof.** The lemma follows immediately from substituting values in equations 5.3 and 5.5. It yields that $x_{i,j} = 1$ if $R_j \in \{R_k : \max(t \in R_k) \leq t_i \ \forall \ R_k \in R\}$ which is equivalent to $x_{i,j} = 1_{\max(t \in R_j) \leq t_i}$. There at most one region per sample point, and therefore at most one region per imputation. Then $X$ is full-rank. Since $G_{i-1} \subseteq G_i$, then $X$ is lower triangular.

It is immediate to verify from definition 5.3 that, for PRI imputation model, the $i$-th element of $Y$ is given by: $y_i = c_i$. The resulting regression equation can be transformed into a Isotonic
Regression Problem[43]. This is shown in chapter 7.

**ORI Model**

**Lemma 5.2.** The design matrix $X$ created by ORI imputation is a full-rank binary matrix which $(i,j)$ is given by:

$$x_{i,j} = \mathbb{1}_{\max(t \in R_j) = t_i}$$  \hspace{1cm} (5.10)

and the rows and columns in $X$ can be arranged to form a lower trapezoid matrix such that:

$$x_{i,j} = 0 \implies x_{i,k} = 0 \quad \forall \ k = j, \ldots, |R|$$  \hspace{1cm} (5.11)

$$x_{i,j} = 0 \implies x_{i,j-k} = 0 \quad \forall \ k = j, \ldots, 0$$  \hspace{1cm} (5.12)

*Proof.* The lemma follows immediately from substituting values in equations 5.3 and 5.5. It yields that $x_{i,j} = 1$ if $R_j \in \{R_k : \max(t \in R_k) \leq t_i \ \forall \ R_k \in R\}$ which is equivalent to $x_{i,j} = \mathbb{1}_{\max(t \in R_j) \leq t_i}$.

There at most one region per sample point, and therefore at most one region per imputation. Then $X$ is full-rank. Since $G_{i-1} \subseteq G_i$, then $X$ is lower triangular. \qed

It is immediate to verify from definition 5.3 that, for PRI imputation model, the $i$-th element of $Y$ is given by: $y_i = c_i$.

### 5.2 Interpolation

The regression coefficients $\beta$ in the regression model only estimate number of counts for regions as a whole. If the interest is to estimate individual time points or to a finer granularity
than the available regions, then interpolation between estimated regions is required.

In this work, we present an experimental comparison of the effect of interpolation methodologies in the prediction accuracy of unobserved instructions (instructions for which no sample was collected). Each interpolation technique can be combined with a regularization technique, creating multiple combinations of multiple approaches. In this work we consider four different regularization techniques, created by the combination of the two imputation models introduced in the previous section: PRI and ORI, and whether non-negativity of increments is enforced or not. Sometimes we refer to the PRI model as Isotonic, due to its relationship with isotonic regression (discussed in section 7.1). Figure 5.2 illustrates examples of different interpolation approaches.

The following subsection introduces

5.2.1 Non-regularized Interpolation

Non-regularized interpolation does not perform any regularization in the original set of data points. In this paper we compare the performance of three commonly used interpolation techniques that are applied to the original sample points. These are described below:

Linear Interpolation

Reconstruction based in linear interpolation finds the piece-wise linear function, $\mu$, that connects all $s_k = (t_k, c_k)$ points in the training set. The predicted value for a new observation at time $t^*$ is given by $\mu(t^*)$.

Despite its widespread usage, linear interpolation is very sensitive to the intrinsic variability of $C_{E_{\sigma}}(t)$ an often yields estimates that violates the monotonicity property of $C_{E_{\sigma}}(t)$. Figure
5.4 illustrates a simple linear interpolation between successive points that ignores the execution that originated each sample point and produces estimator that are not monotonic, as represented by the red line in this figure.

**Smoothing Splines**

A smoothing spline\cite{49} is the estimator $\hat{\mu}$ of the unknown function $\mu$ that is twice differentiable and minimizes:

$$\sum_{k=1}^{n} (c_k - \hat{\mu}(t_k))^2 + \lambda \int_{t_1}^{t_n} \hat{\mu}''(t)^2 \, dx.$$  \hfill (5.13)

Note that the previous equation is the equivalent to the objective function in the regression model (equation 5.7) with a loss function of a penalized sum of squares. An illustration of the interpolation created by the smoothing spline is provided in figure 5.2.
k-NN Regression

This algorithm finds the $k$ nearest points in the training set to the time-point that must predict and calculates its count value by a weighted average of the nearest neighbors[1]. We utilize Euclidean distance. The parameter $k$ is a tuning parameter and is found via cross-validation. Figure 5.2 illustrates this problem.

5.2.2 Non-negative Interpolation

The three methods in this section shown require monotonic inputs to provide monotonic interpolation (hence the name monotonic preserving). This makes the estimation of the region increments ($\beta$) in the regression model a preprocessing step that regularizes the original points in preparation to the interpolation step.

Loess

Locally weighted Regression (or LOESS) is a smoothing procedure that fits a function of the independent variables locally and in a moving fashion analogous to how a moving average is computed for a time series[12]. The smoothing parameter required by LOESS was calculated for each application using cross-validation.

Hyman Cubic Spline

It finds a monotonicity preserving cubic Hermite interpolant, specially designed for the cases when rapid variation is presented [30].
**Spline Fritsch-Butland**

This is another cubic spline interpolant that preserves monotonicity and aims to provide maximum visual smoothness [25].

**Regularized + Linear Interpolator**

A linear interpolant also preserves monotonicity, making a suitable interpolant for the points calculated with isotonic regression.

### 5.3 Methodology

The experiments include benchmarks in the SPEC Suite for `L1-dcache-load-misses`, `branch-misses` and power consumption utilizing the RAPL on-chip power counters available in Intel’s Haswell[44].

#### 5.3.1 Objectives and Metrics

Given sample points from $R$ executions, our goal is to quantify the performance of different regression techniques to predict future observations of $C_{E_a}(t)$. For each regression technique we fit a model $m(t)$.

We measure the accuracy of $m(t)$ using the Mean Absolute Scaled Error (MASE) measure, as defined in [42]. Mean Absolute Scaled Error is a forecasting accuracy measure that performs well in time series data and that is robust to small sample and outliers..
For a fitted model $m(t)$ and a sample-point $s_{r,t_i}$, the *scaled error* is defined as:

$$ q_m(r, t_i) = \frac{(n_r - 1)(c_{r,i} - m(t_i))}{\sum_{i=2}^{n_r} |c_{r,i} - c_{r,i-1}|} $$(5.14)

The Mean Absolute Scaled Error (MASE) of the fitted model $m(t)$ when tested against the $r$-th repetition is denoted as $e_m(r)$ and given by:

$$ e_m(r) = \text{mean}(q_m(r, t_i)) = \sum_{i=1}^{n_r} |q_m(r, t_i)| / n_r $$ (5.15)

### 5.3.2 Experimental Setup

We choose a representative subset of benchmarks from the SPEC CPU2006 benchmark suite[27] to estimate the prediction performance of each reconstruction technique. The experiments focused on *branch-misses* and *L1-dcache-misses* events. These events are specially challenging due to their high variability and non-determinism [58]. For each benchmark and event combination we estimated the appropriate period, $p^*$, required to obtain an average sampling frequency of 1 sample per millisecond. Then we selected 100 random sampling periods within 10% of the value of $p^*$. We executed each benchmark once for each one of the randomly selected periods. The number of sample points for each run depends on the total count of events, which depends on random factors such as system load, operative system, and scheduling frequency scaling among others.

The experiments were performed in system with an Intel i7-970, 6 cores and Hyper-threading support and 12 GB DDR3 of system memory. Running Linux with 3.16.0 – 12 kernel version. The sampling of performance counters was done with custom written software that accessed
the _perf_event_ API for performance counters. The kernel version was modified to remove the upper cap to sampling rate (accessible in `/proc/sys/kernel/perf_event_max_sample_rate`).

5.3.3 Procedure

As defined in section 5.3.1, we utilize the metric MASE to evaluate the goodness of prediction between a reconstructed count and future observations. For each pair of benchmark and event, the set of 100 repetitions is split into 70% training set, D, and a 30% testing set T.

Cross-Validation

Validation data is obtained separately for k-NN regression, and optimal parameter k was calculated using cross-validation. We found the optimal parameter to be strongly correlated with the size of the training size (usually between 4 and 10) and to changes among most benchmarks. For each experiment, we utilized the optimal value of k found.

Sampling of training set

For a given benchmark and a value of $s_i$, $K$ samples of with $s_i$ elements are created by sampling with replacement from $\mathbb{D}$, this constitutes bootstrap sampling[24].

For a reconstruction technique $r$, a model $m_r^k$ is trained on the $k$-th bootstrap sample of the training set, subsequently, $m_r^k$ is test against $t \in T$ and its error recorded. The error for the model $m_r^k$ fit against the $t$-th sample case is denoted as $e_{m_r^k}(t)$ and is calculated according to equation 5.15. This process is repeated for each benchmark, event type and training size value. The above design was chosen since it requires only $K$ training for each $K \cdot |T|$ predictions and allows to reutilize the training set for each experiments.
Variance estimation

A consequence, verified experimentally, of reutilization test cases is that error from obtained against the same test case exhibit less variability that errors from different bootstrap training sets of the same size. This is, for $s_i$:

\[
\text{Cov} [e_{m^k_t}(t), e_{m^k_t}(t)] < \text{Cov} [e_{m^{k_t}_t}(t), e_{m^{k_t}_t}(tt)]
\] (5.16)

To model this, we follow the random effect model for bootstrap clustering presented in [24] which decomposes the error as follows:

\[
e_{m^k_t}(t) = \bar{e}_{m^k} + \hat{\beta}_t + \hat{\epsilon}_{t,k}
\] (5.17)

where $\bar{e}_{m^k}$ is the overall mean of all the errors among all test cases and bootstrap training sets. $\hat{\beta}_t$ is the random effect of the test case in the error and $\hat{\epsilon}_{t,k}$ is the error of that observation. Additional details about this model can be found in literature[24].

5.4 Results and Discussion

We observed a consistent relative performance about the benchmarks, for almost all of the benchmarks the best performing techniques were the ones with non-negativity constraints enforced with ORI imputation model, followed by the spline. There is a high variability in the performance of the reconstruction estimate within the same benchmark, i.e. for leslie3d
Figure 5.5: Reconstruction error (MASE) with error bars (standard deviation) for each evaluated method.
the reconstruction error for \texttt{branch-misses} is 3000\% higher than for \texttt{l1-dcache-misses}.

The reconstruction error is directly related with the variability of the sampled count and it can be utilized as a measurement of variability between and within the same task. Not surprisingly, the reconstruction error of one event does not seem to be directly associated with the other, at least in the two events studied.

Interestingly, the performance of \textit{Hyman} and \textit{Fritsch-Butland} splines seems to be slightly poorer than the much simpler Isotonic Regression + Linear Interpolation that runs in linear time.

### 5.5 Conclusions

We found that enforcing non-negativity to count increments between successive sample-points (Observed increments) yields the best prediction accuracy as depicted in figure 5.5. We found no significant difference in prediction accuracy between linear interpolation and
more complex (and slow) interpolation techniques.

The number of required executions to achieve an acceptable level of resolution can be reduced with statistical techniques that efficiently combines samples. By enforcing non-negativity in the estimates, we were able to generate more accurate reconstruction methodologies. Taking into account the tremendous difference in computation time and numerical stability problem of a cubic spline fitting versus linear interpolation [30, 25], the ORI + Monotonic + Linear Interpolation methodology is, hands down, the best choice for reconstruction out all the methods tried to estimate a trend in the performance trace.

The proposed methodology (ORI + Monotonic + Linear Interpolation) to aggregate samples from different executions allows to consistently obtain finer-granularity information about the occurrence of architectural events with greater accuracy and more efficiency than existing techniques. Allowing applications to obtain fine-grained estimates of occurrences of performance with less sample points than existing methodologies. This improved estimates can allow new application scenarios for performance analysis applications and for optimization tool such as Feedback Directed Optimized compilers and dynamic task schedulers.
Chapter 6

A Methodology to Unify Performance Events and Instruction Traces

In this chapter we present a model and a methodology to pose the problem of estimation the number of performance events generated by dynamic instructions as a regression problem. The new model links a trace of performance events with a trace of instructions while clearly encapsulating the assumptions required by it.

The methodology in this chapter is general enough to encompass the traditional approach shown in section 4.1. It solves the limitations of the traditional estimation methodology that are discussed in section 4.1.2. The new model allows to answer a new class of questions regarding the behavior of a profiled task, in particular it allows to estimate performance events generated by dynamic instructions (or groups of them) rather than only static instructions, as with the traditional approach.
6.1 A Model to Associate Performance Events and Dynamic Instructions

In this section we present a model that express the number of performance events observed during sampling events as a sum of events generated for each dynamic instruction executed during successive samples. This model will be utilized in the rest of this chapter.

6.1.1 The Instructions Trace

**Definition 6.1.** An *execution trace* is given by the surjective (but not necessarily injective) function \( \tau = D \rightarrow \Pi \) where \( D \) is an ordered index set for \( \Pi \). Therefore \( \tau \) maps instructions in the execution trace to their corresponding memory address in the executable program.

Therefore \( d_i \in D \) indexes a dynamic instruction that are successfully retired by the microprocessor during the execution of an executable program and \( d_1, \ldots, d_{|D|} \) is the sequence of indexes of all such dynamic instructions. Note that instructions and micro-operations executed speculatively are ignored.

**Definition 6.2.** The sequence of instructions given by \( \theta(\tau(d_1)), \ldots, \theta(\tau(d_{|D|})) \) \( \forall d_i \in D \) is the *instruction trace* as commonly referred in literature.

For the methodology presented in this thesis it is rarely necessary to know the whole execution trace; depending of the profiling problem to solve, some subsets of it may or may not be required.

There exist many tools and methodologies to collect an instruction trace. Binary instrumentation tools like DynamoRIO[7], and PIN[35] can be utilized to instrument a binary to collect
traces of basic blocks or individual instructions, although with relatively high overload. Intel has incorporated the so called “Processor Trace”[15] that provides hardware support to collect hardware support with significantly less overload than other implementations.

Nevertheless, the overload of collecting the trace is not a limitation of the approach in this work, since the collection of trace is disassociated from the collection of performance events. Therefore, an instruction trace needs to be collected only once, even with significant high overload and perturbation of the task, while the traces of performance events (sample points) can be collected in different executions, or even different systems.

In order to associate sample points to trace it is necessary to map from the $\pi_{r,i}$ value that is part of the sample point to the space of indexes $D$, the execution trace. This is done by augmenting the definition 2.4 with $d_{r,i}$ such that:

**Definition 6.3.** Let $d_{r,i} \in D$ be the index in the execution trace of the instruction that generated the occurrence $c_{r,i}$ of the performance event of interest.

Note that $d_{r,i}$ cannot be obtained from the sample points only, since sample point contain the memory address of the static instruction associated with the last performance event, but no information about the specific dynamic instruction that generated the event. The details of estimation of $d_{r,i}$ are covered in section 6.5. Until that section we assume that $d_{r,i}$ is known for every sample point.

An example of an execution trace is shown in table 6.1. It utilizes the executable program in example 4.1 and the sample points in table 4.2. The table shows the execution trace, the cumulative number of performance events ($\#$ of $\mu$ops) and its corresponding sample points.
Table 6.1: Example of execution trace. The index of the dynamic instruction in the `execution trace` is given by $d$. The Address of Instruction, $(\pi)$, refers to the memory address of the static instruction in the executable program. Sample point shows the `sample points` associated with the corresponding index in the trace $(d_{r,i})$.

<table>
<thead>
<tr>
<th>$d$</th>
<th>$\pi$</th>
<th>$c$ (# µops)</th>
<th>Sample Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td>$\sigma_{1,1}$</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
<td>2</td>
<td>$\sigma_{4,1}$</td>
</tr>
<tr>
<td>3</td>
<td>02</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>03</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>04</td>
<td>7</td>
<td>$\sigma_{2,1}, \sigma_{3,1}$</td>
</tr>
<tr>
<td>6</td>
<td>05</td>
<td>8</td>
<td>$\sigma_{4,2}$</td>
</tr>
<tr>
<td>7</td>
<td>06</td>
<td>9</td>
<td>$\sigma_{1,2}$</td>
</tr>
<tr>
<td>8</td>
<td>07</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>03</td>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>04</td>
<td>14</td>
<td>$\sigma_{3,2}$</td>
</tr>
<tr>
<td>11</td>
<td>05</td>
<td>15</td>
<td>$\sigma_{1,3}, \sigma_{2,2}$</td>
</tr>
<tr>
<td>12</td>
<td>06</td>
<td>16</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>07</td>
<td>17</td>
<td>$\sigma_{4,3}$</td>
</tr>
<tr>
<td>14</td>
<td>03</td>
<td>18</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>04</td>
<td>21</td>
<td>$\sigma_{2,3}$</td>
</tr>
<tr>
<td>16</td>
<td>05</td>
<td>22</td>
<td>$\sigma_{3,3}$</td>
</tr>
<tr>
<td>17</td>
<td>06</td>
<td>23</td>
<td>$\sigma_{4,4}$</td>
</tr>
<tr>
<td>18</td>
<td>07</td>
<td>24</td>
<td>$\sigma_{1,4}$</td>
</tr>
</tbody>
</table>
6.1.2 Association between Performance Events and Dynamic Instruction

**Definition 6.4.** Let $N_{E_\sigma}(d,t) : D \times T \rightarrow \mathbb{N}^0$ a stochastic function that returns the number of performance events of type $E_\sigma$ generated by the dynamic instruction $d \in D$ at time $t$ such that:

$$C_{E_\sigma}(t) = \sum_{d \in D} N_{E_\sigma}(d,t) \tag{6.1}$$

**Definition 6.5.** Let $N_{E_\sigma}(d) : D \rightarrow \mathbb{N}^0$ such that:

$$N_{E_\sigma}(d) = \sum_{t \in T} N_{E_\sigma}(d,t) \tag{6.2}$$

denote the number of performance events of interest generated by the dynamic instruction $d \in D$.

The distinction between $N_{E_\sigma}(d,t)$ and $N_{E_\sigma}(d)$ is necessary to capture the fact that a dynamic instruction can generate performance events multiple times, depending on its stage of execution in the microprocessor’s pipeline. $N_{E_\sigma}(d)$ is effectively counting the number of events generated for a retired instruction. While $N_{E_\sigma}(d,t)$ counts the events occurred until the execution stage that the dynamic instruction $d$ has reached after time $t$.

**Definition 6.6.** The cumulative number of performance events up to the retirement of the dynamic instruction $d \in D$ is given by the stochastic function $C_{E_\sigma}(t) : D \rightarrow \mathbb{N}^0$ such that:

$$C_{E_\sigma}(d) = \sum_{d_i \leq d} N_{E_\sigma}(d_i) \quad \forall \ d_i \in D \tag{6.3}$$

Note that the functions $C_{E_\sigma}(t)$ and $C_{E_\sigma}(d)$ are related since they both describe the cumulative
Table 6.2: Relationship between $C_{\sigma}(t)$ and $C_{\sigma}(d)$. The columns represent different dynamic instructions and rows different times. A cell is the number of events generated by the $d$ instruction in the $t$ time point.

<table>
<thead>
<tr>
<th>$t \ \backslash \ d$</th>
<th>N(d=1)</th>
<th>N(d=2)</th>
<th>N(d=3)</th>
<th>N(d=4)</th>
<th>N(d=5)</th>
<th>$C_{\sigma}(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$C_{\sigma}(d)$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

number of performance events as a program executes. $C_{\sigma}(t)$ counts performance events as time passes while $C_{\sigma}(d)$ counts performance events as dynamic instructions are retired. The processes $C_{\sigma}(t)$ and $C_{\sigma}(d)$ are margins of an underlying bivariate process that is unknown an unobservable.

Our goal is to utilize the execution trace to improve the estimation of performance events. A key step in this procedure, that we call imputation, consists on associating subsets of the instructions trace with specific samples. The subset of the trace that is selected depends on the assumptions that we are willing to make about the architecture that generates performance events. Different types of imputation models are discussed in section 6.2.

Definition 6.7. An imputed set is given by $s_i = \langle \delta_i, D_i \rangle$ where:

- $D_i \subseteq D$, a subset of the execution trace.
- $\delta_i \geq 0$ is the increment in the value of the performance counter of interest attributed to the execution of the instructions in $D_i$.

Definition 6.8. Let $S = \{s_i\}$ be the multiset of all imputed sets.

The choice of $\mathcal{I}$ captures the assumptions about the underlying, unobserved process that
generates the performance events and the execution-trace. Complexities in the execution of instructions such as out-of-order execution and pipelining can be approximated by an adequate choice of \( I \). The different types of imputations (and their associated assumptions) are discussed in section 6.2.

For an imputed set \( s_i \) the observed increment in the count of performance events, \( \delta_i \) must be equal to the true, unknown increment. This is:

\[
\delta_i = \sum_{d \in D_i} N_{E_o}(d)
\]  

(6.4)

this equality is utilized in the rest of the section. The misspecification of the imputed set may introduce errors, as discussed in 2.2.

6.2 Types of Imputations of Performance Events and Underlying Assumptions

**Definition 6.9.** A *trace imputation function* is a function \( I : \Sigma \rightarrow S \) that maps the set of sample period into the multiset of imputed sets.

The choice of \( I \) captures the assumptions about the underlying, unobserved process that generates the performance events and the execution-trace, complexities in the execution of instructions such as out-of-order execution and pipelining can be approximated by an adequate choice of \( I \). In this paper we focus in three types of imputations: (1) **Point Imputation**, that is implicitly used in the vast majority of surveyed literature and software, (2) **Interval imputations**, that incorporates information contained in the magnitude of the sampling period and, (3) **Point-Interval imputations**, a combination of the two previous approaches and the basis for the reconstruction technique presented in this paper.
6.2.1 Point Imputation (PI)

**Definition 6.10.** Let $\alpha_{\theta(\pi_{r,i})} \in \mathbb{N}^0$ and $1 \leq \alpha_{\theta(\pi_{r,i})} \leq c_{r,i}$ be the number of performance events of the type of interest that the instruction $\theta(\pi_{r,i})$ generates when it triggers a sampling event.

The exact value of $\alpha_{\theta(\pi_{r,i})}$ for a specific machine instruction $\pi_{r,i}$ is usually available in the manufacturers’ manual. The traditional sampling process discussed earlier assumes that $\alpha_{\theta(\pi_{r,i})} = 1$ for all instructions, as shown in section 6.4.6.

A imputation model that imputes $\alpha_{\theta(\pi_{r,i})}$ events to the execution of the instruction $\pi_{r,i}$ associated with each *sample point* (recall definition 2.4) is called **point imputation** and it is represented by the function:

$$I_p(\Sigma) = [\langle \alpha_{\theta(\pi_{r,i})}, d_{r,i} \rangle : \sigma_{r,i} \in \Sigma] \quad (6.5)$$

Note that this imputation model discards information regarding the value $c_{r,i}$.

6.2.2 Interval imputation

The value $c_{r,i}$ conveys information regarding the occurrence of performance events prior to the last occurrence that is attributed to the *sample point*. This section introduces **interval imputations**, that attempt to utilize information about the count by assuming that the value of $c_{r,i}$ (or a function of it) is the summation of performance events generated by multiple instructions in the trace. In this paper we study four possible choices of *interval imputation*:
1) **Observed-blocks Imputation (OBI):**

\[ I_o(\Sigma) = \{\langle c_{r,i} - c_{r,i-1}, \{d \in D : d_{r,i-1} < d \leq d_{r,i}\} \rangle : \sigma_{r,i} \in \Sigma \} \]  \hspace{1cm} (6.6)

2) **Preceding-blocks Imputation (PBI):**

\[ I_p(\Sigma) = \{\langle c_{r,i}, \{d \in D : t \leq d_{r,i}\} \rangle : \sigma_{r,i} \in \Sigma \} \]  \hspace{1cm} (6.7)

The previous interval imputation models may be combined with the previous point imputation into new imputation models. Such models assumes that the instruction that triggered the sampling event generated exactly \( \alpha_{\theta(\pi_{r,i})} \) performance events and that the rest of the performance events in the count were generated by other instructions in the interval.

3) **Observed-blocks and Point Imputation (OBPI):**

\[ I_{op}(\Sigma) = \{\langle c_{r,i} - c_{r,i-1} - \alpha_{\theta(\pi_{r,i})}, \{d \in D : d_{r,i-1} < v < d_{r,i}\} \rangle, \langle \alpha_{\theta(\pi_{r,i})}, d_{r,i} \rangle : \sigma_{r,i} \in \Sigma \} \]  \hspace{1cm} (6.8)

4) **Preceding-blocks and Point Imputation (PBPI):**

\[ I_{fp}(\Sigma) = \{\langle c_{r,i} - \alpha_{\theta(\pi_{r,i})}, \{d \in D : d < d_{r,i}\} \rangle, \langle \alpha_{\theta(\pi_{r,i})}, d_{r,i} \rangle : \sigma_{r,i} \in \Sigma \} \]  \hspace{1cm} (6.9)

Note that **preceding-blocks imputations** associate the sampled count to all the instructions in the trace that preceded the instruction attributed to the sample, while **observed-blocks imputations** associate the count only to the instructions that, in the trace, occurred between two successive samples in the current execution of the task.

In general, **point imputation** makes the least assumptions out of the five imputation tech-
niques studied, but it also discards all information regarding the value of the sampled count. Other trace imputation functions could be designed to accommodate additional attributes of the architecture such as out-of-order execution, deep pipelines, speculative executions (for events types that count events generated by speculative instructions), among others.

6.2.3 Examples

We exemplify different imputations techniques using the executable program shown in table 4.1 and sample points in table 4.2.

Define: \( \alpha_i = \begin{cases} 3 & \text{if } i \text{ is MUL instruction type} \\ 1 & \text{else} \end{cases} \) for \( i \in I \). The sample point \( \sigma_{r,0} \) for all \( r = 1, \ldots, R \) is not considered for imputations since it is not a sample that was collected but added only for clarity of exposition.

Point Imputation

\[
\mathcal{I}_p(\Sigma) = \langle 1, \{1\} \rangle : 1, \langle 1, \{2\} \rangle : 1, \langle 3, \{5\} \rangle : 2, \langle 1, \{6\} \rangle : 1, \\
\langle 1, \{7\} \rangle : 1, \langle 3, \{10\} \rangle : 1, \langle 1, \{11\} \rangle : 2, \langle 1, \{13\} \rangle : 1, \\
\langle 3, \{15\} \rangle : 1, \langle 1, \{16\} \rangle : 1, \langle 1, \{17\} \rangle : 1, \langle 1, \{18\} \rangle : 1
\]

Full-blocks Imputation

\[
\mathcal{I}_o(\Sigma) = \langle 15, \{0, \ldots, 7\} \rangle, \langle 21, \{0, \ldots, 11\} \rangle, \langle 30, \{0, \ldots, 18\} \rangle, \\
\langle 15, \{0, \ldots, 11\} \rangle, \langle 21, \{0, \ldots, 15\} \rangle, \\
\langle 11, \{0, \ldots, 10\} \rangle, \langle 19, \{0, \ldots, 16\} \rangle, \\
\langle 13, \{0, \ldots, 6\} \rangle, \langle 22, \{0, \ldots, 13\} \rangle, \langle 28, \{0, \ldots, 17\} \rangle
\]
6.3 Construction of an Estimation Query

To understand the interaction of software, hardware and execution context it is often necessary to differentiate between dynamic instructions even when they are associated to the same static instruction. i.e. first iteration in a loop may present significantly different behavior than other iteration, or a basic block may have different behavior dynamic behavior depending of the trace of instructions that preceded it.

**Definition 6.11.** An estimation block, \( B_b \subseteq D \), is a set of dynamic instructions that are deemed to be indistinguishable from each other for the purpose of the profiling goals that guide the analysis. The ordered collection \( B = (B_b) \), indexed by \( b \), contains all estimation
blocks for a estimation problem.

Our objective is to specify a mapping between dynamic instructions and estimation-blocks. Such a mapping can defined as a query and it is defined as follows:

**Definition 6.12.** A query is given by \( Q = \{ (\pi_1, \phi_1, b_1), \ldots, (\pi_Q, \phi_Q, b_Q) \} \) where:

- \( \pi_i \in \Pi \).
- \( \phi_i : \mathcal{P}(D) \times D \to \{0, 1\} \) is as context filter. A function that takes a subset of the execution-trace and a dynamic instruction and returns 1 if such dynamic instruction must be part of the \( b_i \)-th block of interest, 0 otherwise.
- \( b_i \) is the index of the estimation-block to which the resulting traces will be attributed.

**Definition 6.13.** For \( Q_i \in Q \), the queried subtrace of the \( i \)-th query is given by:

\[
D_{Q_i} = \{ d \in D : \tau(d) = \pi_i \text{ and } \phi(D, d) = 1 \}
\]  

(6.10)

\( D_{Q_i} \) is the set of dynamic instructions that correspond to the static instructions \( \pi_i \) and that are not filtered out by \( \phi_i \).

**Definition 6.14.** A valid query is a query such that:

\[
D_{Q_i} \cap D_{Q_j} \neq \emptyset \Rightarrow b_i = b_j \quad \forall \ i, j = 1\ldots|Q|
\]  

(6.11)

**Definition 6.15.** The \( b \)-th estimation block, \( B_b \), is given by:

\[
B_b = \bigcup_{\forall i \mid b=b_i} D_{Q_i} \quad \text{for } i = 1\ldots|Q|
\]  

(6.12)
Equation 6.11 requires that the no dynamic instruction is associated to more than one estimation-block. It follows immediately that, for a valid query, all estimation blocks are disjoint. When defining queries, it is often useful to aggregate dynamic instructions that are not of interest into a single estimation block rather than keep them as independent estimation-blocks. Such aggregation helps to reduce the dimensionality of the problem, as discussed in section 6.4.3.

6.3.1 Examples

Some queries that are often useful to understand the behaviour of an executable program are:

**Example 6.1** (Query for an arbitrary function or code section). Let $\phi_a(D,d)$ be a constant function equal to 1. Using the example program in table 4.1, the query:

$$Q = \{(00, \phi_A, 1), (01, \phi_A, 1), (02, \phi_A, 1),$$

$$ (03, \phi_A, 2), (04, \phi_A, 3), (05, \phi_A, 4), (06, \phi_A, 5), (07, \phi_A, 6)\}$$

specifies a estimation problem where the number of performance events generated by each instruction in the loop are estimated individually, but instructions outside the loop are aggregated together.

$$D_{Q_1} = \{1\} \quad D_{Q_2} = \{2\}$$

$$D_{Q_3} = \{3\} \quad D_{Q_4} = \{4, 9, 14\}$$

$$D_{Q_5} = \{5, 10, 15\} \quad D_{Q_6} = \{6, 11, 16\}$$

$$D_{Q_7} = \{7, 12, 17\} \quad D_{Q_8} = \{8, 13, 18\}$$
$B_1 = \{1, 2, 3\} \quad B_2 = \{4, 9, 14\} \quad B_3 = \{5, 10, 15\} \\
B_4 = \{6, 11, 16\} \quad B_5 = \{7, 12, 17\} \quad B_6 = \{8, 13, 18\}$

**Example 6.2** (Query for a function in an arbitrary Call-stack). Let $I_C = (\iota_1, \ldots, \iota_a)$ be a sequence of CALL static instructions that describe the sequence of calls in the desired call-stack. Let $\iota_R$ a return instruction.

Let $D_p$ be the subset of $D$ that precedes $d$. Let $I_C$ the set of all CALL static instructions in $\rho(d) : d \in D_p$ that do not have a corresponding RET instruction.

Let $\phi_c(D, d) = 1$ if the sequence of the last $a$ elements in $I_C$ are equal to $I_C$, zero otherwise. The function $\phi_c(D, d)$ is a context filter that selects only the instructions in the desired call-stack described by $I_C$.

**Example 6.3** (Query Arbitrary iterations in a Loop). Using the example program in table 4.1. Let $\phi_l(D, d)$ be equal to 1 if subtrace $D$ that precedes $d$ does not contain the $i$-th full iteration of the loop and zero otherwise. Then, a query that estimates separately the dynamic instructions of the $i$-th iteration versus other iterations is given by:

$$Q = \{(00, \phi_l, 1), (01, \phi_l, 1), (02, \phi_l, 1), (03, \phi_l, 2), (04, \phi_l, 3), (05, \phi_l, 4), (06, \phi_l, 5), (07, \phi_l, 6), (03, 1 - \phi_l, 7), (04, 1 - \phi_l, 8), (05, 1 - \phi_l, 9), (06, 1 - \phi_l, 10), (07, 1 - \phi_l, 11)\}$$
will estimate separately between the instructions in the first iteration and other iterations. This distinction between iterations in a loop is impossible with the traditional approach. It is straightforward to generalize the context filter to any iteration in the loop.

**Example 6.4** (Query for Distinct Traces). Let $A_1, A_2, A_3, A_4$ be four basic blocks of static instructions and $D_1 = A_1 A_2 A_3$ and $D_2 = A_1 A_3 A_4$ two possible subsequences of the instruction trace. Let the functions $\phi_{D_1}(D,d)$ that return 1 for all $d \in D_q$ such that $d \in S(D_1)$, zero otherwise and $\phi_{D_2}(D,d)$ that return true if $d \in D_q$ is in $D_2$, zero otherwise. $\phi_{D_1}(D,d)$ and $\phi_{D_2}(D,d)$ can be utilized to estimate separately dynamic instructions in $D_1$ and $D_2$. Note that this distinction is impossible with the traditional approach.

### 6.4 Estimation as a Regression Problem

A valid query defines a mapping between dynamic instructions and estimation blocks. In this section we combine the concepts introduced before. We utilize a estimation blocks and a set of imputed sets to pose the problem of estimation of counts performance events per
estimation block as a regression problem with the sample points from the events trace as observations.

6.4.1 Building a Design Matrix

In this subsection we build a design matrix for the regression problem using:

1. A set of imputed sets \( S \) (Definition 6.7)

2. A estimation block, \( B \) (Definition 6.15).

**Definition 6.16.** A estimation subtrace, \( D_B \), is given by \( D_B = \bigcup_{B_b \in B} B_b \) and denotes the subset of the trace that is required by a set of estimation blocks.

**Definition 6.17.** For \( S_a \subseteq S \), an observed subtrace, \( O(S_a) \) is given by:

\[
O(S_a) = \bigcup_{s_i \in S_a} D_i
\]  

(6.13)

and denotes the subset of the execution trace that is also present within the dynamic instructions of any imputed set in \( S_a \).

For arbitrary \( S_a \) and \( B \), any dynamic instruction in \( D_B \setminus O(S_a) \) is not estimable since there is no imputed set that contains such element in its set of observations. For these cases the query \( Q \) (and hence the estimation block) should be modified. Some techniques to modify \( Q \) aimed to eliminate no estimable blocks are discussed in section 6.4.2. For the rest of this section, assume that \( D_B \subseteq O(S_a) \). We name such subsets of \( S \) as follows:

**Definition 6.18.** For a estimation subtrace \( D_B \), a subset of imputed sets, \( S_K \subseteq S \), and \( K_B = O(S_K) \). \( S_K \) is a imputations cover for \( D_B \) if and only if \( D_B \subseteq K_B \).
The set of dynamic instructions in \( K_B \setminus D_B \) are not mapped to any estimation block and therefore have limited utility to gain insight about the performance problem. Nevertheless, they may be useful as additional predictors in the regression problem to increase the precision of the estimation. We expand the set of blocks of interest to include such dynamic instructions in the regression problem (possibly aggregating them as seem fit).

**Lemma 6.1.** For a set of blocks \( B \), and a imputations cover \( K_B \). Let \( A^{B,K} \) be a partition of the set \( K_B \setminus D_B \) and let \( B^K = B \cup A^{B,K} \). Then \( D_B^K = K_B \).

**Proof.** From definition 6.16, \( D_{A^{B,K}} = \bigcup_{B_b \in A^{B,K}} B_b = K_B \setminus D_B \). Therefore, \( B^K = (K_B \setminus D_B) \cup D_B = K_B \). □

We utilize \( B^K \) in the rest of this section and consider the additional blocks in \( A^{B,K} \) as part of the estimation problem. The choice of the partition \( A^{B,K} \) embeds the desired role of the elements in \( D_B \setminus K_B \). Different choices of \( A^{B,K} \) impact features of the regression matrices such as multicollinearity as variance of the estimates. Particular values \( A^{B,K} \) and may be chosen to utilize elements in \( D_B \) as precision variables. These usage scenarios are discussed in section 6.4.3.

**Definition 6.19.** Let \( \nu(s_j, B_i) : \sum_{b \in B_i} \sum_{d \in D_j} 1_b(d) \) is a function \( S_K \times B^K \to \mathbb{N}^0 \) that denotes the number of times that a dynamic instruction in the estimation block \( B_i \) appears in \( D_j \), the subtrace of the imputed set \( s_j \).

**Definition 6.20.** For an imputation cover, \( S_K \), and a set of estimation blocks, \( B^K \), a design matrix, denoted by \( X \), is a \( |S_K| \times |B^K| \) matrix where the entry at the \( i \)-th row and \( j \)-th column is given by:

\[
x_{i,j} = \nu(s_i, B_j)
\]

where \( s_i \in S_K \) and \( B_j \in B^K \).
Let $Y = \langle \delta_i \rangle$ be a column vector of $|S|$ elements where $\delta_i$ is the first element of the $i$-th element in the $s_i$ tuple, then an estimation problem can be expressed as a regression problem where the goal is to find the values of $\beta = \langle \beta_i \rangle$, a column vector of size $|B^K|$ such that:

$$\begin{align*}
\text{minimize}_{\beta} & \quad L(Y - X\beta) \\
\text{subject to} & \quad g_i(\beta) \leq k_i, \ i = 1, \ldots, m
\end{align*}$$

for some $m \in \mathbb{N}^0$, where $g_i(\beta) \leq k_i$ represent $m$ optional constraints that may be imposed to the optimization problem, and $L : \mathbb{R}^{|B^K|} \to \mathbb{R}$ is the error function to be minimized.

Note that matrix $X$ is not guaranteed to be full rank for any choice of $B$ (or $Q$), as opposed to the similar regression problem in Chapter 5, equation 5.7. We discuss ways to specify the problem in order to avoid rank deficiency in section 6.4.3.

Additionally, note that the subset of the execution trace made of the dynamic instructions in $B^K$ contains all the elements of the trace that are required for the estimation. This can be summarized in the following remark:

**Remark 6.1.** The set $D_{BK}$ is a superset of the subset of $D$ necessary to estimate $B^K$ with $S_K$.

A direct consequence is that for some queries and choices of trace imputation function it is not required to collect a trace at all since the estimation of $\beta$ is trivially reduced. The observed ratio estimator presented in section 4.1 is one of those cases, as shown in section 6.4.6.
6.4.2 Incomplete Imputations Cover

In order for a set of imputation blocks to be estimable, $S_K$ must be an imputation cover of $D_B$ otherwise the columns of the matrix $X$ corresponding to the blocks with no imputation will contain rows of all zeros, creating a degenerated design matrix.

6.4.3 Handling Rank Deficiency and Collinearity

Even if $S_K$ is a imputation cover, misspecification of $B$ and $A^{B,K}$ can cause of rank deficiency or collinearity in $X$. This sections discusses how to avoid common problems in the specification of $B$ and $A^{B,K}$. Some of the common problems are:

- **Insufficient imputation sets**: The size of $A^{B,K}$ determines the number of variables that are estimated. If not enough imputation sets are available, elements in $B$ or $B^K$ can merged to reduce the number of variables to estimate.

- **Replications and collinearity**: Blocks that contain dynamic instructions that are associated with static instructions in the same basic block will tend to create very similar columns in the design matrix. Merging these blocks together or with another blocks will decrease the rank deficiency and collinearity in the design matrix.

Although collection more samples (and therefore more imputed sets) could allow to build a new matrix $X$ that is full-rank, reducing the number of estimation blocks may also avoid the problem. The following is a common procedure to select a subset of blocks in $A^{B,K}$ such that the resulting design matrix is full-rank:

1. Obtain the $QR$ decomposition of $X$. Recall that each column in $R$ corresponds to one column in $X$.
2. Obtain the first non-zero elements for each column in $R$ (left to right). The position of this element corresponds to the first column of $Q$ that such column depends on.

3. Build a new matrix $X^N$ with only columns of $X$ that have distinct first non-zero elements in $R$.

### 6.4.4 Monotonic Constraints

Although dependent on the definition of the error function $E$, the interpretation of the coefficients $\beta_i$ is commonly associated with a likelihood that the dynamic instructions in $\beta_i$ generate a performance event. Since the performance counters are non-decreasing counters (except by overflow that is handled in software, see section 2.2) it is natural to impose a non-negativity constraint to the estimation of $\beta_i$. When this is done, the regression model can be expressed as:

\[
\begin{align*}
\text{minimize} \quad & E(Y - X\beta) \\
\text{subject to} \quad & \beta_i \geq 0 \quad \text{for} \quad i = 1, \ldots, |\beta| \\
\end{align*}
\]  

(6.16)

and solved using restricted estimation methods such as non-negative least squares. A detailed evaluation of different estimation techniques is given in Chapter 5.

### 6.4.5 Examples

Using the query shown in example 6.1 and the *Observed-Blocks Point Imputation* model (as shown in example 6.2.3) yields the design matrix shown in appendix A. The estimation for such regression problem using unconstrained least squares are shown in table 6.3. For this
Table 6.3: Estimation of performance events per estimation block for query in example 6.1

<table>
<thead>
<tr>
<th>Method</th>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( b_3 )</th>
<th>( b_4 )</th>
<th>( b_5 )</th>
<th>( b_6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least Squares</td>
<td>1.024</td>
<td>0.844</td>
<td>3.141</td>
<td>0.939</td>
<td>1.015</td>
<td>0.952</td>
</tr>
<tr>
<td>True</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

example using non-negative least squares yields the same results, a complete evaluation of different estimation techniques is given in Chapter 5. Due to the definition of the \( \nu(s_j, B_i) \) function (definition 6.19), the estimators of blocks that contain multiple instruction are average values for all instructions in such estimation block. There is no information about the number of instructions generated by individual instruction in such estimation blocks.

6.4.6 Comparison with Existing Methodology

Observed Ratio Estimator

The methodology presented in this work is a generalization of the traditional methodology shown in section 4.1. The traditional approach represents the trivial case of of the framework presented in this paper and can be expressed in the following manner:

- Let \( \Sigma \) be the set of sampled values, as defined in definition 2.5.
- Let \( U \) be the set of unique \( \pi_{r,i} \) for every \( \sigma_{r,i} \in \Sigma \). Where \( U_i \) is the \( i \)-th element in \( U \).
- Let \( S = \mathcal{I}_p(\Sigma) \) be the set of imputed sets using point-imputation with \( \alpha_{\theta(\pi_{r,i})} = 1 \)
- Let \( \phi_A(D, d) \) be a context filter that equal 1 if \( d = d_{r,i} \) for some \( r = 1, \ldots, R \), \( i = 1, \ldots, n_r \) and zero otherwise.
- Let \( Q = \{ (U_i, \phi_A, i) : i = 1, \ldots, |U| \} \). This is a query that maps all dynamic instructions associated with a static instruction in \( U \) into one single block.
Table 6.4: Estimation of rate of performance events per instruction for query in example 6.1 versus traditional approach.

<table>
<thead>
<tr>
<th>Instruction at:</th>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least Squares</td>
<td>1.024</td>
<td>1.024</td>
<td>1.024</td>
<td>0.844</td>
<td>3.141</td>
<td>0.939</td>
<td>1.015</td>
<td>0.952</td>
<td>9.963</td>
</tr>
<tr>
<td>True Ratio</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>True Ratio</td>
<td>0.103</td>
<td>0.103</td>
<td>0.103</td>
<td>0.085</td>
<td>0.315</td>
<td>0.094</td>
<td>0.102</td>
<td>0.096</td>
<td></td>
</tr>
<tr>
<td>True Ratio</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.3</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>True Ratio</td>
<td>0.067</td>
<td>0.067</td>
<td>0</td>
<td>0</td>
<td>0.267</td>
<td>0.267</td>
<td>0.133</td>
<td>0.133</td>
<td></td>
</tr>
</tbody>
</table>

Therefore, \( D_{Q_i} = \{d_i : \forall u \in U \text{ s.t. } \tau(d_i) = u\} \). Since there is one element in \( Q \) per instruction in \( U \), note that \(|Q| = |B|\) and \( B_j = D_{Q_j} \) for all \( j = 1, \ldots, |B|\).

Therefore the design matrix, \( X \), is given by: \( x_{i,j} = 1_{\sigma_i \in B_j} \) where \( \sigma_i \) is the \( i \)-th element in \( \Sigma \).

From definition of point imputation, \( Y_i = \delta_i = 1 \), the solution to the minimization problem is trivial and \( \beta_i = 1 \) for \( i = 1, \ldots, |\beta| \).

This example illustrates how the traditional approach is a degenerated case of the methodology presented in this section. No information about the instruction trace is required and the solution is trivially estimable. Table 6.4 shows the solution of applying the observed ratio estimator of the traditional approach (as defined in 4.1 using both the count estimators calculated in example 6.4.5 (regression with Point-Interval Imputation) and the using simple frequency counts as in table 4.3.

### 6.5 Mapping Sampling Points to Execution Trace

The methodology presented in previous sections of this chapter assumes that there is a known mapping between sample points and a constant, unique execution trace. Such mapping is necessary to make interval attribution possible since imputation associates individual sample points with instructions in the execution trace. Unfortunately, the sample points containing
no information about their position in the execution trace and additional steps are required to build such a mapping. This section present methodology to address such problem.

### 6.5.1 Difficulties

The exposition in section 6.4 is based into two specific assumptions:

- **Assumption 1**: The *execution trace* given by the mapping $\tau$ is the same for all repetitions of a task. This assumption is embedded in the definition of an executable trace (Definition 6.1).

- **Assumption 2**: The existence of $d_{r,i}$; an index in the *execution trace* that represents the dynamic instruction that generated the last performance event in the count corresponding to the sample point $\sigma_{r,i}$. This assumption is utilized by definition 6.3.

Assumption 1 does not hold in general since modern software usually contain call to system call, synchronization primitives, I/O operations and other software routines for which the code-path followed depends on the current state of the system and/or a myriad of parameters external to the *executable task* itself. Section 6.5.3 demonstrate a methodology to guarantee that the sections of the *execution trace* utilized for estimation are the same for all executions.

To understand Assumption 2, recall that a *sample point* is given by $\sigma_{r,i} = \langle t_{r,i}, e_{r,i}, \pi_{r,i} \rangle$ where $\pi_{r,i}$ is the value of the program counter of the instruction associated with the program event of interest up to a *skid* (or no skid when exact instruction sampling is available). In the sample itself there is no information about the position in the *instruction trace* of the instruction associated to the sample. Therefore, the value of $d_{r,i}$ cannot be estimated from the set of sample points alone and additional information is required. Section 6.5.4 presents methodology to obtain additional information required to map sample points to execution trace.
Section 6.5.5 presents an algorithm that is guaranteed to find an optimal mapping between sample points and execution trace using the proposed methodologies devised to guarantee that assumptions 1 and 2 hold.

### 6.5.2 Collecting Information about Execution Trace from Performance Events

The association between the event of interest, $\mathcal{E}_\sigma$, and instructions may be very complex or unknown. However, there exists some performance events that present a clear association with the progress of the trace, such as $\text{instructions-retired}$ or $\text{branch-instructions-retired}$. The key idea in this section is to simultaneously sample the event of interest with another event with a strong, known association with the execution trace and utilize this known relationship to approximate the position of the performance events of interest within the execution trace.

**Definition 6.21.** Let $\mathcal{E}_\eta$ a performance event type such that for a sampled count at execution time $k_{r,i}$, $C_{\mathcal{E}_\eta}(t_{r,i}) = \eta_{r,i}$, there exist functions $B_L(\eta_{r,i}, D)$ and $B_U(\eta_{r,i}, D)$ such that:

$$B_L(\eta_{r,i}, D) \leq d_{r,i} \leq B_U(\eta_{r,i}, D)$$

(6.17)

for $d_{r,i} \in D$ defined as in Definition 6.3.

All surveyed PMUs provide multiple hardware performance counters that allow simultaneous profiling of multiple performance events [14, 20]. Therefore, the events $\mathcal{E}_\sigma$ and $\mathcal{E}_\eta$ can be programmed to be counted simultaneously and the values of their performance counters can be collected together during each sampling interruption of $\mathcal{E}_\sigma$. In Linux’s `perf event` built-in support for this is available and is known as group-sampling [13].
The functions $B_L(\eta_{r,i}, D)$ and $B_U(\eta_{r,i}, D)$ provide a bounded space for the indexes of instructions that generated the count of interest, providing a restricted search space for the minimization problem described in 6.5.4.

### 6.5.3 Finding Common Instruction Traces in Multiple Executions

Due to the complexity of modern operative systems and programming languages, developers usually rely on abstractions that encapsulate and/or hide many interactions with the environment of a task into dynamically linked libraries, system calls or routines that are often called transparently and executed in user space without being part of an executable binary. Such routines may execute different code-paths even during repeated executions of the task, depending on the state of the underlying system. The execution of different code-paths makes execution traces of different executions to be different from each other.

When there is no guarantee that all samples were taken using the same code-path, then the imputation of performance events for the sample points of each execution should utilize its corresponding execution trace. Nevertheless, collecting such a trace is memory intensive (and CPU intensive when not assisted by hardware). Executions of the task that for which the execution trace is collected are significantly perturbed and rarely exhibit the behavior of the normally executed execution task.

An alternative is to restrict the estimation problem (and the required sample points) to subsets of the instruction traces in each execution of the executable program $P$ that are equal across all executions, the set of such subsets is called a repeatable trace. For most scenarios, restricting estimation to a repeatable trace have no significant impact since a repeatable trace usually contains the region of the instruction trace that is desired to be profiled and optimized since they are generated by the static instruction that the developer
is currently measuring and tweaking.

Some software libraries for performance events expose the ability to restrict the counting of events to certain execution contexts or levels, in particular Linux's `perf_event` allows to separate events occurred when a process runs in user space from events generated while the process runs in kernel space or in a hypervisor[13]. Unfortunately, the levels provided by a library do not coincide with the levels required for the analysis in this work.

Another way to exclude counts of performance events during routines that are not part of the repeatable trace is to guard regions in the executable trace that are guaranteed to be part of the repeatable trace with instrumentation or another mechanism that allows to register the values of the performance counter at the entrance to each region. This way, the events generated during parts of the trace that are not repeatable can be subtracted.

**Finding Common Subtraces using Hardware Breakpoints**

Modern commercial x86 microprocessors support hardware breakpoints [14, 20] that can trigger an interruption when a region of the code is being profiled is executed, allowing to collect values of the performance events of interest right at the entrance of such regions without other overhead than the aforementioned interruption. This method is utilized in this section since it has the additional advantage that it does not require modifications to the binary, unlike an instrumentation alternative.

**Definition 6.22.** Some basic definitions required for this section are:

- Let $D^{(r)}$ be the execution trace of the $r$-th execution of the task of interest, not necessarily equal to other execution traces of different executions.
- Let $S^r \subseteq S$ be the subset of imputed sets that only contains imputed sets generated from sample points of the $r$-th execution.
• $\tau^r$ is the mapping between $D^R$ and $\Pi$ as defined in Definition 6.1.

**Definition 6.23.** Let $M \in S(\Pi)$ a sequence of instruction addresses and $\{M_i\}$ a partition of $M$ such that:

- $M_i \in \mathcal{S}(M)$.
- $\tau^r(O(S^r)) \in \mathcal{S}(M)$ $\forall r = 1, \ldots, R$

This is, $M_i$ is a contiguous subsequence of $M$ such that the instructions traces of the imputations covers of $S^r$ are contain in $M$ for all $r = 1, \ldots, R$.

If the positions where the common sections of the instruction trace appear in each instruction is known, then the following theorem provides a straightforward mapping between each the execution trace in each execution and "common" trace, $D$.

**Theorem 6.1.** Let $D \subseteq \mathbb{N}^0$, $\tau(D) = M$. Let $\omega_i$ be the first element of $M_i$ in $D$. Let $D^r \subseteq \mathbb{N}^0$ $\forall r = 1, \ldots, R$. Let $G^r = \{\langle m^r_g, \omega^r_g \rangle\}$ such that $\langle m^r_g, \omega^r_g \rangle \in G^r$ implies that:

1. $M^r_{m^r_g} \in \{M_i\}$ was executed on the $r$-th execution. This is, $\exists H^r_g \in \mathcal{S}(D^r)$ s.t. $\tau^r(H^r_g) = M^r_{m^r_g}$.

2. $\omega^r_g \in D^r$ is the index in $D^r$ of the first element of $H^r_g$.

Then the function:

$$m(d^r) = d^r - \omega^r_j + \omega^r_{m_j} \quad \text{for } j = \arg \min_{g=1,\ldots,|G^r|} d^r - \omega^r_g \geq 0$$  \hspace{1cm} (6.18)

maps $d^r \in H^r_j$ to $d \in D$ such that:

$$\{\tau(m(d^r)) : d^r \in H^r_j\} = \{\tau^r(d^r) : d^r \in H^r_j\} \text{ for all } j = 1, \ldots, |G^r|$$  \hspace{1cm} (6.19)
Proof. For \( d^r \in H^r_i \), \( \omega^r_i \) is the index of its first element, therefore \( d^r - \omega^r_i \) is the offset of \( d^r \) in \( H^r_i \). Since \( \tau^r(H^r_i) = M^r_{m^r_i} \), then \( d^r - \omega^r_i + \omega^r_g \) is the position of an instruction such that \( \tau(m(d^r)) = \tau^r(d^r) \).

If \( d^r \notin H^r_i \), but \( d^r \in H^r_j \), then either \( d^r - \omega^r_i < 0 \), if \( H^r_j \) is after than \( H^r_i \) in the trace \( D^r \) or there exists \( \omega^r_j \) such that \( d^r - \omega^r_j < d^r - \omega^r_i \), if \( H^r_j \) is before than \( H^r_i \) in the trace \( D^r \). Therefore, the \( g \) with \( \omega_g \) with the minimum positive \( d^r - \omega_g \) is such that \( d^r \in H^r_g \). \( \square \)

Theorem defines a function \( m(d) \) that allows to map any instruction in a trace \( D^r \) into a “unified” trace \( D \), given a set \( G^r \) with information about the starting indexes of the common areas between \( D \) and \( D^r \).

### 6.5.4 Mapping Sample Points to Repeatable Traces

The skid of a sample point is the difference between the \( \pi_{r,i} \) in \( \sigma_{r,i} \) and the address of the instruction that really generated the sample point. This skid is architecture and event dependent and can be fixed, systematic or random [53]. Let \( skid_l \in \mathbb{Z} \) and \( skid_u \in \mathbb{Z} \) be the minimum and maximum skid that the architecture and sampling mode allows for a \( E_{\sigma} \) event.

**Definition 6.24.** Let \( \mathcal{M} : D \rightarrow \Pi \) be a mapping such that:

\[
\mathcal{M}(d) = \{ \tau(d_c) : d_c \in D \text{ for } d + skid_l \leq d_c \leq d + skid_m \} \quad (6.20)
\]

Thus \( \mathcal{M}(d) \) represents the set of memory addresses of static instructions that correspond to the elements of the trace \( D \) that are within skid distance from \( d \).

The problem of mapping sampling points to the execution trace can be posed as an opti-
mization problem where the goal is to find $\eta^* = \{\eta_i^*\}$ such that.

$$\arg \min_{\eta^*} \sum_{i=1}^{\eta} L(\eta_i^* - \eta_i)$$

subject to: $\pi_{r,i} \in S(\sum_{k=1}^{i} \eta_k^*)$

$$\mathcal{V}(\phi(\tau(\sum_{k=1}^{i} \eta_k^*)), E_\sigma) > 0$$

(6.21)

The constraints of this problem only allow a few possible trace indexes as candidates for each sample point. Let $b$ be the maximum distance between $\eta_i^*$ and $\eta_i$.

6.5.5 Algorithm to Map Sample points to Repeatable Subtraces

The following algorithm allows to find the optimal $\eta$ for the samples of a given execution of the task.: 

---

76
Algorithm 1 A linear time algorithm to align sample points to instruction trace.

**Input:** $r, \pi_{r,1}, \ldots, \pi_{r,n_r}, \eta_{r,1}, \ldots, \eta_{r,n_r}$

**Output:** $\eta^*$

1: $i = 1, t = 0, \eta^* = []$
2: for all $i \leq n_r$ do
3: \hspace{1em} $\eta_{r,i}^* = \eta_{r,i}$
4: \hspace{1em} while $\pi_{r,i} \notin \tau(t + \eta_{r,i}^*)$ and $\mathcal{V}(\phi(\tau(\sum_{k=1}^{i} \eta_{k}^*)), \mathcal{E}_o) \neq 1$ do
5: \hspace{2em} $d = \eta_{r,i}^* - \eta_{r,i}$
6: \hspace{2em} $\eta_{r,i}^* = \eta_{r,i} - d$
7: \hspace{1em} if $0 \leq d$ then
8: \hspace{2em} $\eta_{r,i}^* = \eta_{r,i}^* + 1$
9: \hspace{1em} end if
10: \hspace{1em} end while
11: \hspace{1em} add $\eta_{r,i}^*$ to $\eta^*$
12: \hspace{1em} $i = i + 1$
13: end for

**Proof.** A simple inspection of the inner loop shows that the values taken by $\eta_{r,i}^*$ are $\eta_{r,i}, \eta_{r,i} + 1, \eta_{r,i} - 1, \eta_{r,i} + 2, \eta_{r,i} - 2, \ldots$. Those values increase the loss function as the iteration progresses. Since the maximum distance between $\eta_{i}^*$ and $\eta_{i}$ is $b$, then the inner loop will iterate up to $bn$ times and the algorithm is linear in the number of sample points.

6.6 Experimental Verification

In order to verify the quality of the estimation, a ground truth must be known or assumed. Some performance events are estimable directly from the executable binary and exhibit a deterministic behavior (ie. branch-instructions or retired-instructions at least
up to relative small error. Nevertheless, most events are not directly estimable from the executable binary or the source code (unsurprisingly since the reason of being for performance events is that they provide information that is usually unavailable otherwise). The next subsections refer to the experimental verification for each one of these events. For both types we utilized the SPEC CPU2006 benchmark suite[27] and a system with an Intel i7-970, 6 cores and Hyper-threading support and 12 GB DDR3 of system memory. Running Linux with 3.16.0 – 12 kernel version. The sampling of performance counters was done with custom written software that accessed the perf_event[13] API for performance counters. The kernel version was modified to remove the upper cap to sampling rate (accessible in /proc/sys/kernel/perf_event_max_sample_rate).

6.6.1 Events that can be verified independently

For this events we calculate the true value using binary instrumentation using DynamoRIO[7] and measure the relative error between the true value and the estimated values. One of the few events that are almost deterministic and estimable from binary is retired-instructions. Although the total count is affected by overcount and other issues, the error is relatively small[53] and considered negligible for this experiment. The total counts of events for each IP in the sample points is collected using binary instrumentation and utilized to obtain the ratio of performance events generated by each instruction. Figure 6.1 illustrates the relative error of each methodology versus the known ratio of estimated instructions per IP.

6.6.2 Non-verifiable events

For those events whose true value is not estimable or non-deterministic, we estimated a ground-truth by collecting an extremely large amount of sample points and utilizing the traditional approach to estimate ratios. Then we compare the relative error of each estima-
Figure 6.1: Relative error of different estimation techniques versus known ratio of retired instructions per IP.
Figure 6.2: Relative error of different estimation techniques versus estimated ratio of retired instructions per IP for L1-dcache misses.
Figure 6.3: Relative error of different estimation techniques versus estimated ratio of retired instructions per IP for Branch misses
tion methodology as the number of sample points increases. Figures 6.2 and 6.3 illustrate the cases of l1-dcache-misses and branch-misses events. For all experiments, the OBPI (Observed Point Imputation) Model presented significantly smaller relative error.

6.7 Conclusion

Existing methodology to associate performance events to instructions only utilize the information provided by a frequency table of IPs in the performance events trace. We proposed a novel technique that augments the trace of performance events with the instruction trace, increasing the precision of the estimates and allowing to separately estimate dynamic instructions within the instruction trace.

The distinction between dynamic instruction allows a new dimension to performance analysis applications, that allows scenarios such as: (1) separately estimate the performance events for the same static instruction in each iteration of a loop, (2) distinguish between traces of basic blocks, (3) distinguish between function calls based on calling context. These new dimension facilitates obtaining a deeper understanding of the behavior of the profiled task.
Chapter 7

Linear Time Algorithms for Regression of Performance Events Counts

Chapters 5 present methodologies to pose the problem of estimating the trend in a performance trace as a linear regression problem with non-negativity constraints. The material presented in Chapter 6 enables to build a regression problem to estimate the number of performance events generated by groups of dynamic instructions where the regression problem may be optionally constrained to non-negativity constraints.

The estimation for the regression problem of both methodologies can be performed using Non-negative Least Squares, and Integer, Linear or Quadratic Programming, depending on the form of the loss function and the properties of the design matrix.

Due to the great speed of modern hardware, non-invasive estimation of performance events tends to require large amount of sample-points. As an example, in a modern x86 commercial microprocessor, that commonly executes over 1 billion of instructions per second, trend estimation of performance trace of one second of execution, with a average sampling resolution of one sample point per thousand instructions will require 1000 executions. In many
instances of the regression problem the estimation matrix can be very large, i.e. for the case just described, the estimation of the trend would generate a design matrix with dimensions up to $10^6 \times 10^6$.

Existing algorithms such as Least Squares, Integer, Linear and Quadratic Programming have have quadratic or higher asymptotic complexity and are not suitable for matrices of the size generated by performance trace estimation. In this chapter we study the constrained estimation of special cases of triangular design matrices that we found to be common when solving trace reconstruction problems and estimation of performance events per instruction. The first part of this chapter shows how Isotonic Regression[4, 8] arises when Previous Regions Imputation is utilized (equations 5.3 and 6.7). In this scenarios the design matrix is a lower triangular matrix that can be transformed into an Isotonic Regression problem and solved in linear space and time using the Pool of Adjacent Violators Algorithm (PAVA)[4]. The second part presents a new algorithm, a generalization of the PAVA that allows to solve, in $bn$ time a non-negative regression problem with a lower triangular matrix which certain constrains, where $n$ is the number of imputations and $b$ is usually a small constant.

In this chapter we focus on p-norms as loss function for the regression models defined in equations 5.7 and 6.15. Assuming non-negativity constraint for each coefficient $\beta_i$, then the regression equations become:

\[
\begin{align*}
\text{minimize} & \quad \|Y - X\beta\|^p \\
\text{subject to} & \quad \beta_i \geq 0 \quad \forall \beta_i \in \beta
\end{align*}
\]  
(7.1)
7.1 Estimation as a Isotonic Regression Problem

In this section we introduce Isotonic Regression (IR) and the Pool Adjacent Violators Algorithm (PAVA), a linear time and space algorithm to solve particular cases of IR.

7.1.1 Isotonic Regression

Isotonic Regression is a mature technique to estimate a regression model with isotonic constraints. In this work we limit to the one-dimensional version where the goal is:

**Definition 7.1.** Given $\vec{c} \in \mathbb{R}^m$, a set of non-negative weights $0 \leq \vec{w}, \vec{w} \in \mathbb{R}^m$, and a not contradictory set of order constraints $\mathcal{K}$. An isotonic regression problem is:

$$
\begin{aligned}
\text{minimize} & \quad \| \vec{c} \odot \vec{w} - \vec{\gamma} \|_p \\
\text{subject to} & \quad \gamma_i \leq \gamma_j \quad \forall (i,j) \in \mathcal{K}
\end{aligned}
$$

(7.2)

The order imposed by the set $\mathcal{K}$ can be a partial or a total order. The appropriate solution algorithm depends on the dimensionality of the problem, the type of norm of the objective function, and the structure of the order relationship imposed by $\mathcal{K}$.

The resolution for the problem greatly simplifies when the constraints define a total order. In this work we focus on the unweighted, totally ordered case. This is the case where:

- For a given $\vec{t} \in \mathbb{R}^m$, there exist pairwise order constraints given by:

$$
(i,j) \in \mathcal{K} \iff t_i < t_j \quad \text{for } t_i, t_j \in \vec{t}
$$

- The weights are $w_i = 1 \quad \forall w_i \in \vec{w}$. 

85
Resulting in the following problem:

**Definition 7.2.** *Given m elements such that \( (t_i, c_i) \in \mathbb{R} \times \mathbb{R} \). An unweighted, totally-ordered isotonic regression problem is given by:*

\[
\begin{align*}
\text{minimize} & \quad \| \bar{c} - \bar{\gamma} \|_p \\
\text{subject to} & \quad t_i \leq t_j \implies \gamma_i \leq \gamma_j
\end{align*}
\]  

(7.3)

**7.1.2 Pool Adjacent Violators Algorithm**

The **Pool Adjacent Violators Algorithm** (PAVA) finds the value of the optimal \( \bar{\gamma} \) in linear time and memory.

Before studying PAVA, we show an equivalent representation of problem shown in Definition 7.2 that is specially amenable to understand PAVA.

**Definition 7.3.** Let \( U = \{t_i\} \) and ordered set of unique elements for of \( \bar{t} \), clearly \( i = 1, \ldots, m \). Let \( u_j \) denote the \( j \)-th element of \( U \).

**Definition 7.4.** Let \( \bar{\alpha} \in \mathbb{N}^m \) such that \( \alpha_i \) is the index of \( u \in U \) such that \( u = t_i \) for each \( i = 1, \ldots, m \).

**Definition 7.5.** Let \( P_u = \{c_i\} \) be a multiset where \( c_i \in \mathbb{R} \). A **summary function** is a real function \( \mu(P_u) \) that minimizes:

\[
\sum_{c_i \in P_u} \| \mu(P_u) - c_i \|_p
\]  

(7.4)

By definition, \( \mu(\emptyset) = 0 \).

The summary function for \( l^1 \), \( l^2 \), \( l^\infty \), and \( l^\infty^- \) norms are the median, the mean, the maximum, and the minimum, respectively[56].

86
**Theorem 7.1.** Let $\vec{\gamma}^U$ such that:

$$\minimize_{\vec{\gamma}^U \in \mathbb{R}^{\lvert U \rvert}} \sum_{u=1}^{\lvert U \rvert} \|\mu(\{c_i : \alpha_i = u\}) - \gamma^U_u\|^p$$

subject to $u_i \leq u_j \implies \gamma^U_i \leq \gamma^U_j$ for $u_i, u_j \in U$ \hspace{1cm} (7.5)

Then $\gamma_i = \gamma^U_j$ for all $i$ such that $\alpha_i = u_j$, with $\vec{\gamma}$ as given in definition 7.2.

**Proof.** By substituting the definition of $\mu(P_u)$ into the objective function and applying norm properties. Then replacing constraints in definition 7.2 by their equivalents in $\alpha$. \hfill $\square$

Theorem 7.1 allows to express the IR problem in terms of the order relationship between the $t_i$ values rather than the $t_i$ values themselves. Therefore disassociating the space of indexes from the solution of the problem. This is summarized in the following remark.

**Remark 7.1.** The solution for the IR minimization problem does not depend on the values of $t_i$ or $U$. It only depends on the indexes $\alpha$.

The fundamental idea in PAVA is to group elements, components $c_i \in \vec{c}$ into pools of adjacent elements, denoted by $P_u$, based on the value of their corresponding $\alpha_i \in \vec{\alpha}$. Initially, one pool is created per value in $U$.

Such initial choice of $P$ pools together all $c_i \in \vec{c}$ with the same index. For each pool, a summary value is calculated using a summary function $m(P_u)$. If the summary value of a pool does not satisfy the isotonic constraint (non-decreasing blocks), its elements are transferred to the previous pool. Since . The above is formalized in Algorithm 2.
Algorithm 2 The Pool of Adjacent Violators Algorithm (PAVA).

Input: \( \tilde{\alpha}, \tilde{c} \)

Output: \( \tilde{\gamma} \)

Functions:

move_elements(source, destination): Move elements from source to destination.

next_nonempty(P_i): Get next non-empty set in \( P \), \( \emptyset \) if no next.

prev_nonempty(P_i): Get previous non-empty set in \( P \), \( \emptyset \) if no previous.

1: procedure PAVA(\( \tilde{\alpha}, \tilde{c} \))
2: Initialize \( P \) such that \( P_u = \{ c_k : \alpha[k] = u \} \)
3: if \( |P| = 1 \) then
4: return \( \tilde{\gamma} = \mu(P_1) \)
5: end if
7: while \( P_c \neq \emptyset \) do
8: if \( \mu(P_p) \leq \mu(P_c) \) then
9: \( P_p = P_c \)
10: \( P_c = \text{next\_nonempty}(P_c) \)
11: else
12: while \( P_p \neq \emptyset \) and \( \mu(P_p) > \mu(P_c) \) do
13: \( \text{move\_elements}(P_c, P_p) \)
14: \( P_c = P_p \)
15: \( P_p = \text{prev\_nonempty}(P_c) \)
16: end while
17: end if
18: end while
19: return \( \tilde{\gamma} = \langle \mu(P_{\alpha_i}) : i = 1, \ldots, m \rangle \)
20: end procedure

PAVA is a well-studied algorithm and many proofs of its correctness and asymptotic linearity are available in literature[4, 8].
7.1.3 PAVA to Solve Estimation Problem

The models chapters 5 and 6 have the following form when a $l^p$-norm and non-negative constraints for the $\beta_i$ coefficients:

\[
\begin{align*}
\min_{\beta} & \quad \|\vec{y} - X\vec{\beta}\|^p \\
\text{subject to} & \quad 0 \leq \beta_i \quad i = 1, \ldots, m
\end{align*}
\] (7.6)

with a matrix $X$ of dimensions $|S| \times m$.

**Theorem 7.2.** Let $\vec{x}_i$ the $i$-th row of matrix $X$. If $\vec{x}_i \preceq \vec{x}_j \forall i, j = 1, \ldots, |S|$, where $\preceq$ is the partial order of vector in $\mathbb{R}^m$. Then equation 7.6 can be solved using PAVA with:

- $\vec{c} := \vec{y}$.
- $U$ is total ordered set of unique row vectors of $X$.
- $\vec{\gamma} := X\vec{\beta}$.

**Proof.** Let $\alpha_i$, as defined in 7.4, be the index of $x_i$ in $U$. Since there is partial order among the $x_i$'s, there is a total order among the $u_i$'s in $U$. Therefore the elements $y_i$ can be pooled in pools of distinct indexes $\alpha_i$ and a total order between the $\mu(\alpha_i)$'s must exist, as required by PAVA. \hfill \Box

**Examples**

As stated in lemma 5.1, when using *Preceding Regions Imputation* (PRI), the design matrix forms a full-rank, binary matrix lower trapezoidal matrix. It is trivial to verify that a total order exists between the row vectors in $X$ and therefore PAVA, from theorem 7.2 is applicable.
7.2 Estimation as a Generalization of the Isotonic Regression Problem

In Chapter 5 we found that the combination of observed increments and enforcing monotonicity by non-negative increments, what we call Observed Intervals Imputation (ORI) in section 5.1.1, has the most predictive power to predict counts of performance events as a function of reference index. As shown in lemma 5.2, there is no total order between the rows of $X$ (a key difference between ORI and PRI) and therefore the PAV algorithm cannot be utilized.

This section shows a novel dynamic programming algorithm, inspired in PAVA, that allows to solve the regression problem generated by ORI in linear time. The algorithm generalizes to any regression problem with non-negativity constraints and with conditions as expressed in equations 5.11 and 5.12.

7.2.1 Motivation

The imputation model used in Chapter 5 and formalized in Definition 5.7 expresses the count in an imputation $s_i$ as a function of the regions associated to it. In the case of PRI, this is equivalent to state:

$$\delta_i = c_i = \sum_{t_j \leq t_i} N_{\xi_n}(t_j) = C_{\xi_n}(t_i)$$ \hspace{1cm} (7.7)

since $C_{\xi_n}(t)$ is a non-decreasing stochastic function, it is straightforward to enforce isotonicity in the estimation of the coefficients, as shown in the previous section. Nevertheless, other
imputations do not impute the full count $c_i$ to all preceding regions (and all counts). For this cases there may not be an isotonic constraint to enforce. For example, the ORI imputation model:

$$\delta_i = \sum_{t_{r,i-1}<t_j\leq t_i} N_{\xi_\sigma}(t_j) \quad (7.8)$$

assign only a subset and there is no guarantee that $\delta_i < \delta_j$ even for $i < j$.

Nevertheless, the monotonicity of $C_{\xi_\sigma}(t)$ is also captured by the non-negativity of $N_{\xi_\sigma}(t)$. The key idea in this section is to create an efficient algorithm that enforces non-negativity in the intervals themselves rather than monotonicity in the cumulative sums of intervals.

### 7.2.2 Generalization of PAVA

The generalized version of PAVA enforces non-negative in the increments rather than monotonicity in the counts. If the function $\mu(\cdot)$, as defined in 7.5, is modified to be the summary of the increment of the current block, then the algorithm 3 will find an optimal set of points $\mu$ that solve the regression problem subject to non-negativity constraints for the coefficients.
Algorithm 3 Modified PAVA: A dynamic programming algorithm to solve estimation problems with non-negative constraints and special structure design matrix.

Input: $\vec{\alpha}, \vec{\delta}$

Output: $\vec{\gamma}$

Functions:

$move\_elements(source, destination)$: Move elements from source to destination.

$next\_nonempty(P_i)$: Get next non-empty set in $P$, $\emptyset$ if no next.

$prev\_nonempty(P_i)$: Get previous non-empty set in $P$, $\emptyset$ if no previous.

1: procedure $MPAVA(\vec{\alpha}, \vec{c})$
2: Initialize $P$ such that $P_u = \{c_k : \alpha[k] = u\}$
3: if $|P| = 1$ then
4:     return $\vec{\gamma} = \mu(P_1)$
5: end if
7: while $P_c \neq \emptyset$ do
8:     if $\mu(P_c) < 0$ then
9:         $P_p = P_c$
10:        $P_c = next\_nonempty(P_c)$
11:     else
12:         while $P_p \neq \emptyset$ and $0 > \mu(P_c)$ do
13:             $move\_elements(P_c, P_p)$
14:             $P_c = P_p$
15:             $P_p = prev\_nonempty(P_c)$
16:         end while
17:     end if
18: end while
19: return $\vec{\gamma} = \langle \mu(P_{\alpha_i}) : i = 1, \ldots, m \rangle$
20: end procedure
7.3 Experiments

We are interested in comparing the execution speed of an implementation of algorithm 3 with PAV and other algorithms commonly utilized to solve estimation problems. We implemented algorithm 3 using \( C++ \) and build a simple R wrapper for its mPAV procedure. We refer to this implementation as mPAVA. We compared two variations of the mPAVA algorithm: (1) mpava.median: that solves the problem with a \( l^1 \) norm objective function and (2) mpava.mean: that solves for a \( l^2 \) norm objective function. We compare the execution speed of the R wrapped mPAVA. Both versions where compiled gcc 4.9. with O2 optimization level, with the following implementations of solvers available in R packages:

- **isoreg**[40, 4, 8]: A popular implementation of the PAV algorithm. Empirically found to be the faster implementation of PAV available in the data analysis community[21]
Figure 7.2: Running times versus number of samples for PAV algorithm (isoreg R package), and the proposed modified PAV Algorithm with Absolute Error (median) and Squared Error (mean) loss functions.

- **isotone**: Solves the isotonic regression problem using an active set algorithm.

- **linprog**: A linear programming solver using the Simplex algorithm.

- **nnls**: An implementation of the Fast NNLS algorithm.

The running times for each algorithm are shown in figure 7.1. As expected, algorithms with super linear time complexity becomes increasingly slow on large sample sizes. Figure 7.2 compares only implementations of algorithms known to have linear or log-linear time, these are: PAVA, mPAVA.mean and mPAVA.median. The implementation of isoreg is 42% faster than mPAVA.median and 24% faster than mPAVA.mean when 15,000 sample points are utilized. Although isoreg is significantly faster than both versions of mPAVA, the grow rate of running time as function of sample size is consistent with the expected theoretical results.
7.4 Conclusions

This chapter have presented how two classes of regression problems that commonly arise in estimation of a trend in a trace of performance counters can be solved using linear time and space algorithms. The proposed techniques allow to estimate trend using millions of sample points within minutes, using commercially available hardware. The improved computation time allow a new set of usage scenarios for the performance trace. One of them is to allow accurate trend estimation of thousands of sample points for interactive performance analysis tools. Another application scenario is to allow more efficient task and thread scheduler methodologies to be build that can exploit data from more data points, using more accurate estimations for their scheduling policies.
Bibliography


Appendix A

Appendix

Design matrix for first two sample points of first two repetitions of example in section 6.4.5.

\[
X = \begin{pmatrix}
 s_{i,r} & I_{op} & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & \delta \\
 s_{1,1} & \langle 1, \{1\} \rangle & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
 s_{1,2} & \langle 7, \{2, \ldots, 6\} \rangle & 2 & 1 & 1 & 1 & 0 & 0 & 7 \\
 s_{1,2} & \langle 1, \{7\} \rangle & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
 s_{1,3} & \langle 5, \{8, \ldots, 10\} \rangle & 0 & 1 & 1 & 0 & 0 & 1 & 5 \\
 s_{1,3} & \langle 1, \{11\} \rangle & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
 s_{1,4} & \langle 8, \{12, \ldots, 17\} \rangle & 0 & 1 & 1 & 2 & 1 & 8 \\
 s_{1,4} & \langle 1, \{18\} \rangle & 0 & 0 & 0 & 0 & 1 & 1 \\
 s_{2,1} & \langle 3, \{5\} \rangle & 0 & 0 & 1 & 0 & 0 & 0 & 3 \\
 s_{2,2} & \langle 7, \{6, \ldots, 10\} \rangle & 0 & 1 & 1 & 1 & 1 & 1 & 7 \\
 s_{2,2} & \langle 1, \{11\} \rangle & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
 s_{2,3} & \langle 3, \{12, \ldots, 14\} \rangle & 0 & 1 & 0 & 0 & 1 & 1 & 3 \\
 s_{2,3} & \langle 3, \{15\} \rangle & 0 & 0 & 1 & 0 & 0 & 0 & 3 \\
\end{pmatrix}
\]
Design matrix for first two sample points of last two repetitions of example in section 6.4.5.

\[
X = \begin{array}{c|c|cccccc}
  s_{i,r} & \mathcal{I}_{op} & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & \delta \\
  \hline 
  s_{3,1} & \langle 3, \{5\} \rangle & 0 & 0 & 1 & 0 & 0 & 0 & 3 \\
  s_{3,2} & \langle 3, \{6, \ldots, 9\} \rangle & 0 & 1 & 0 & 1 & 1 & 1 & 3 \\
  s_{3,2} & \langle 3, \{10\} \rangle & 0 & 0 & 1 & 0 & 0 & 0 & 3 \\
  s_{3,3} & \langle 7, \{11, \ldots, 15\} \rangle & 0 & 1 & 1 & 1 & 1 & 1 & 7 \\
  s_{3,3} & \langle 1, \{16\} \rangle & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
  s_{4,1} & \langle 1, \{2\} \rangle & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
  s_{4,2} & \langle 5, \{3, \ldots, 5\} \rangle & 1 & 1 & 1 & 0 & 0 & 0 & 5 \\
  s_{4,2} & \langle 1, \{6\} \rangle & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
  s_{4,3} & \langle 8, \{7, \ldots, 12\} \rangle & 0 & 1 & 1 & 1 & 2 & 1 & 8 \\
  s_{4,3} & \langle 1, \{13\} \rangle & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
  s_{4,4} & \langle 5, \{14, \ldots, 16\} \rangle & 0 & 1 & 1 & 1 & 0 & 0 & 5 \\
  s_{4,4} & \langle 1, \{17\} \rangle & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\end{array}
\]  

(A.2)