UiO : Department of Informatics
University of Oslo

Exploratory Analysis of Caching Mechanisms in Siddhi Stream Processing Engine

Dan Rachou

Master's Thesis Spring 2023

Exploratory Analysis of Caching Mechanisms in Siddhi Stream Processing Engine

Dan Rachou

May 2023

Abstract

This thesis focuses on the modeling of performance non-determinism in Siddhi, a complex event processing system. The main objective is to assess the sufficiency of the modeling methodology proposed by Kristansen et al.[24] in capturing performance non-determinism. The research methodology employed involves an investigation into the performance non-determinism induced by cache memories.

The key findings of the study reveal that the time component of cache memories significantly contributes to performance non-determinism in event processing within Siddhi. This non-determinism can impact the overall efficiency and reliability of the system. However, the proposed modeling methodology does not adequately consider the time dimension, highlighting a limitation in its applicability to model performance non-determinism accurately.

Based on these findings, the conclusions drawn from this research suggest that the existing modeling methodology needs to be extended to incorporate the time dimension.

Acknowledgements

I would like to extend my sincere thanks to my supervisors, Thomas Peter Plagemann and Espen Volnes, for their support and valuable feedback on my writing. Their guidance has been invaluable throughout this process.

I am grateful to my employers at Dolphin Interconnect Solutions, Roy Nordstrøm and Hugo Kohman, for accommodating my study and writing commitments while working full-time.

I want to express my appreciation to my mother, Eva Synnøve Drægni Rachou, for her unwavering support in my studies.

I want to thank my father, Zaki Rachou, who sadly passed away during the writing of this thesis, for giving me tea and freshly cut fruit whenever I was staying up late studying. I miss you dearly and wish you were still here.

I want to thank my parents-in-law, Leticia Nũnez and Olivier Graziani, for proof-reading and providing valuable insights.

Last but not least, I want to thank the love of my life, Cristina Maria Nes Nũnez, for patiently letting me ramble on about bursts and caching, and for supporting me to the very end. Te amo.

Contents

1 Introduction **3**
1.1 Problem statement 5
1.2 Methodology and approach 6
1.3 Contributions 6
1.4 Outline 7

2 Background **9**
2.1 The modeling methodology 9
2.1.1 Core concepts and challenges 10
2.1.2 The iterative modeling process 12
2.1.3 Model evaluation 16
2.1.4 The methodology in practice 16
2.2 Complex event processing (CEP) 17
2.2.1 Events and event relationships 18
2.2.2 Event pattern, rules and constraints 21
2.2.3 Event hierarchies and personalized views 22
2.2.4 Summary 23

3 Methodology **25**
3.1 System under test (SUT) 29
3.1.1 Event stream 29
3.1.2 Pre-generated events 29
3.1.3 Injecting events 29
3.1.4 Stream call-back 30
3.2 Instrumentation 31
3.2.1 Burst processing times 31
3.3 Metrics 32
3.4 Tracing framework 33
3.4.1 Requirements 33
3.4.2 Tracing utility 33
3.4.3 Evaluation 34
3.5 Summary 35

4 Analysis **37**
4.1 Processing event bursts 37
4.1.1 Cache memory 37

4.1.2 Experimental design 38
4.1.3 Analysis 40
4.1.4 Head-tail comparison analysis 42
4.1.5 Distribution analysis 43
4.2 Spikes in burst processing time 46
4.2.1 Experimental design 48
4.2.2 Analysis 48
4.2.3 Garbage collection in the HotSpot VM 49
4.2.4 Experimental design 52
4.2.5 Analysis 52
4.2.6 Experimental design 53
4.2.7 Analysis 53
4.2.8 Experimental design 54
4.2.9 Analysis 54
4.2.10 Experimental design 55
4.2.11 Analysis 55
4.3 Bimodal distribution 56
4.3.1 Java Virtual Machine (JVM) and JIT compilation . . . 56
4.3.2 Experimental design 58
4.3.3 Analysis 59
4.3.4 Experimental design 59
4.3.5 Analysis 60
4.3.6 Virtual memory 62
4.3.7 Experimental design 64
4.3.8 Analysis 64
4.4 Elevated event processing 65
4.4.1 Dynamic voltage and frequency scaling (DVFS) . . . 65
4.4.2 Experimental design 66
4.4.3 Analysis 67
4.4.4 Experimental design 68
4.4.5 Analysis 69
4.4.6 Experimental design 69
4.4.7 Analysis 70
4.5 Asymmetric event processing 70
4.5.1 Cache state 70
4.5.2 Experimental design 71
4.5.3 Analysis 72
4.5.4 Experimental design 74
4.5.5 Analysis 74
4.6 Summary 79

5 Conclusion **81**
5.1 Contributions and critical reflections 81
5.1.1 Contributions 82
5.1.2 Critical reflections 83
5.2 Open problems and future work 83
5.2.1 Isolation of CPU core 84
5.2.2 Control task scheduling 84

5.2.3 Cache model 84

List of Figures

3.1 System stack 26
3.2 Flowchart of system analysis and modeling approach. 28
3.3 Evaluation and comparison of monitor overhead when using 7 and 2 tracepoints to capture event processing timing, in addition to a timestamp-enabled trace monitor. 36

4.1 Comparison of access latency for L1, L2 and L3 cache, as well as RAM. Each vertical line denote the end of a storage unit. The graph has been generated based on `lm_mem_rd` microbenchmark from `lmbench`[32]. 38
4.2 Burst processing of burst distances longer than 300 ms. . . . 41
4.3 Burst processing of burst distances between 100 and 300 ms. 41
4.4 Comparison of burst processing for a constant stream and a bursty stream with 1000 d_{ms}. 42
4.5 Distribution of burst processing times in S_{50}^{1000} with 1000 d_{ms}. 44
4.6 Distribution of event processing times in S_{50}^{1000} with 1000 d_{ms}. 45
4.7 Distribution of event processing times shorter than 300 µs in S_{50}^{1000} with 1000 d_{ms}. 46
4.8 Burst processing in S_{50}^{1000} with 1000 ms. 47
4.9 Comparison of burst processing behavior for three different burst sizes: 500, 1000 and 2000 events. 49
4.10 Distribution of event processing times in S_{200}^{500} with 1000 d_{ms}. 50
4.11 Distribution of event processing times in S_{200}^{1000} with 1000 d_{ms}. 50
4.12 Distribution of event processing times in S_{200}^{2000} with 1000 d_{ms}. 51
4.13 A comparison of the impact of heap settings on burst processing in S_{50}^{1000} with 1000 d_{ms}. 55
4.14 Distribution of event processing time in S_{1000}^{50} with 1000 d_{ms} and without garbage collection. 56
4.15 Burst processing times in S_{50}^{1000} with burst distances ranging from 400 ms to 1000 ms and without garbage collection. The dashed line in dark gray denotes the end of the JVM warm-up. 60
4.16 Comparison of burst processing in S_{50}^{1000} with burst distances ranging from 400 ms to 1000 ms. No garbage collection occurrences during runtime and JIT compiler is disabled. . . 61
4.17 Distribution of event processing time in S_{1000}^{50} with 1000 d_{ms}. No garbage collection occurrences during runtime and JIT compiler is disabled. 61

4.18 Distribution of event processing time for S^{50}_{1000} where the JIT compiler is disabled, no garbage collection has occurred during execution and the entire heap has been mapped physical memory. 64
4.19 Comparison of JVM factors impacting asymmetric event processing in short bursts of 1000 events. 74
4.20 Distribution of event processing time for the first, second, fifth, 10th, 20th, and 25th event in event bursts for S^{50}_{1000} with the JIT compiler disabled and no garbage collection or page fault occurrences during execution. 75
4.21 Distribution of event processing time for the the 25th and 1000th event in event bursts for S^{50}_{1000} with the JIT compiler disabled and no garbage collection or page fault occurrences during execution. 76
4.22 Distribution of event processing time of the first three events in event bursts for S^{50}_{1000} with the JIT compiler disabled, no occurrences of garbage collection during execution, and entire JVM heap is pre-touched. CPU frequency is locked to 2.8 GHz. 77
4.23 Distribution of event processing time of $S_{*,3}$, $S_{*,7}$ and $S_{*,1000}$ in S^{50}_{1000} with the JIT compiler disabled and no occurrences of garbage collection during execution and JVM heap pre-touched. CPU frequency is locked to 2.8 GHz. 78
4.24 Configurations inducing various degrees of amplifications of caching behavior in relation to performance non-determinism. 80

List of Tables

4.1 Statistics of burst processing times from processing S_{50}^{1000}, including JVM and default settings. Time measurements are in milliseconds. 40
4.2 Head-tail analysis of a bursty stream S_{50}^{1000} with 1000 $d_m s$. Time measurements are in microseconds. 43
4.3 Head-tail analysis of a constant stream S_{50}^{1000}. Time measurements are in microseconds. 43
4.4 The percentage of events in S_{50}^{1000} with 1000 d_{ms} that fall within a given time different boundaries. Time measurements are in microseconds. 45
4.5 Statistics of burst and event processing time per burst for the ten bursts with the highest processing time in S_{50}^{1000} with 1000 d_{ms}. Time measurements are in milliseconds. 47
4.6 Garbage collection activities during processing of S_{50}^{1000} with 1000 d_{ms}. Timestamp is in seconds and Duration is in milliseconds. 53
4.7 Garbage collection activities during processing of S_{50}^{1000} with 1000 d_{ms} using the timestamp-enabled trace monitor. Timestamp is in seconds and Duration is in milliseconds. . . 54
4.8 Garbage collection activities during processing of S_{50}^{1000} with 1000 d_{ms} running on the JVM with initial heap size of 4GB. Timestamp is in seconds and Duration is in milliseconds. . . 54
4.9 Event processing time exceeding 500 µs in S_{50}^{1000} when no garbage collection occurs during runtime. Time measurements are in microseconds. 57
4.10 The 10 events with the highest processing time in S_{50}^{1000} with 1000 d_{ms}. No garbage collection occurrences during runtime and the JIT compiler is disabled. Time measurements are in microseconds. 62
4.11 Burst processing times in S_{50}^{1000} with default JVM settings and `powersave` governor. Time measurements are in milliseconds. 68
4.12 Burst processing times in S_{50}^{1000} with default JVM settings and `performance` governor. Time measurements are in milliseconds. 68
4.13 Burst processing times in S_{50}^{1000} with JIT compilation disabled, no garbage collection or page fault occurrences, and `powersave` governor. Time measurements are in milliseconds. 69

4.14 Burst processing times in S_{50}^{1000} with default JVM settings and static CPU frequency of 2.8 GHz. Time measurements are in milliseconds. 70
4.15 Head-tail comparison analysis of S_{50}^{1000} with JIT enabled and no occurrences of garbage collection. JVM warm-up not included. Experiment executed under the `powersave` scaling governor. Time measurements are in microseconds. 72
4.16 Head-tail comparison analysis of S_{50}^{1000} with JIT disabled and no occurrences of garbage collection. Initialization phase not included. Experiment executed under the `powersave` scaling governor. Time measurements are in microseconds. 73
4.17 Head-tail comparison analysis of S_{50}^{1000} with JIT disabled, no occurrences of garbage collection, and JVM heap pre-touched. Initialization phase not included. Experiment executed under the `powersave` scaling governor. Time measurements are in microseconds. 73
4.18 Head-tail comparison analysis of S_{50}^{1000} with JIT disabled, no garbage collection occurrences, and pre-touch enabled. Initialization phase not included. Executed with the CPU locked at 2.8 GHz. Time measurements are in microseconds. 75

Chapter 1

Introduction

In the realm of distributed applications, a significant number of them demand real-time or near-real-time processing of high volume and high velocity data, as it flows from the periphery to the center of the system. The data stream attributes precludes the data from being stored on disk before analysis. Examples include: market feed processing and electronic trading on financial markets[1]; Intrusion detection systems which analyze network traffic in real-time to identify possible attacks[13]; ubiquitous computing[25], such as environmental monitoring applications which process raw data coming from wireless sensory networks (WSN)[6]; and Internet of Things (IoT)[12].

To address these requirements, complex event processing (CEP) systems have been developed. CEP systems enable the processing of data as it flows through the system, allowing users to define new queries for complex events while the system is operational. These queries can include filters, aggregates, and correlation of data, providing system administrators with the ability to notify interested parties about notable results, anomalies, or significant findings. The development of CEP systems has greatly enhanced the capabilities of real-time data analysis and decision-making in diverse domains.[11]

In the classical architecture of CEP systems, event processing is performed within a single node or cluster of tightly interconnected nodes. However, certain CEP use-cases, such as automated traffic control and smart cities need insights from large sets of data quickly to maintain continuous situational awareness. This with the aim of reacting to certain events as fast as possible. Event processing in a single CEP node struggles to meet the speed requirement of real-time analytics demanded by these types of use-cases[22].

In distributed CEP, complex event processing is performance across several nodes[37, 22]. It enables the system to scale performance and handle higher workloads in real-time. Additionally, it allows the system to perform in-network processing. This can reduce the amount of data that has to be

transferred through the network, which is especially important in networks with bandwidth limitations, such as mobile networks and wireless sensory networks. However, designing and implementing distributed systems can be exceptionally challenging. The performance characteristics of distributed applications are intricate, often plagued by "soft failures" where the application produces correct results but experiences lower throughput or higher latency than expected[17].

Identifying and troubleshooting performance problems can prove difficult due to the intricate interplay among various components in a distributed system. Performance issues in one area may manifest themselves elsewhere, making it challenging to track down the root causes. Bottlenecks can arise at different points along the data flow paths, including the applications themselves, the operating systems, the device drivers, the network adapters on both sending and receiving hosts, as well as network components like switches and routers[41].

Therefore, having a thorough understanding of the significant performance pitfalls and considerations associated with designing distributed systems is detrimental in one's success. Being proactive and addressing these challenges beforehand can greatly benefit the development and deployment of distributed systems. Simulation plays a vital role in this process by providing a platform to explore and evaluate various performance aspects of distributed systems. By simulating different scenarios and configurations, one can gain valuable insights into the potential performance issues and make informed decisions to optimize system design and performance[21]. Thus, simulation serves as a valuable tool for mitigating risks and ensuring the success of distributed system designs.

DCEP-Sim, an open distributed CEP framework built on ns-3[37], provides a simulation framework to run large-scale experiments of distributed CEP solutions. By utilizing DCEP-Sim, the overall cost and effort associated with evaluating distributed CEP approaches can be significantly reduced. This is particularly valuable since the proper evaluation of distributed CEP approaches often require networks with several hundred nodes, making real-world experiments unfeasible. However, DCEP-Sim is omitting one substantial element in its simulations – the *intra-node* processing time[45].

Capturing the intra-node processing time is crucial in simulation studies as it allows for a comprehensive assessment of event delivery latency. If only network latency is considered and the intra-node processing time is omitted, the simulation results would lack the processing delay component, leading to inaccuracies in the estimated event delivery latency. By including the intra-node processing time in simulations, researchers can obtain a more realistic representation of the overall latency experienced by events as they traverse the system. This enables more accurate evaluations and insights into system performance and behavior.

Kristiansen et al.[24] presents a methodology for modeling the execution of communication software in multi-threaded systems. The methodology

has been used to successfully extend ns-3 to run accurate simulations of wireless sensory networks by modeling the packet processing in motes[44]. Furthermore, in a demonstration by Volnes et. al[45], the methodology was used to model the software execution of T-Rex, a prototype pub/sub CEP server system, that was integrated into DCEP-Sim. It revealed that processing delay introduced by software execution in T-Rex is up to several times higher than transmission delay between nodes. This demonstration showcased that the methodology is able to model software execution of systems with higher complexity.

Nevertheless, T-Rex is more straightforward to model than modern stream-processing engines in terms of programming language, system architecture, and key properties such reliability, fault tolerance and quality of service[45]. For instance, consider the complexity of a CEP system like Siddhi[39]. The intra-node processing time encompasses multiple layers, including the execution of Siddhi itself, the Java Virtual Machine (JVM), and the underlying operating system. Each of these layers introduces potential performance non-determinism, making it challenging to precisely predict and control system behavior. In this thesis, however, the focus is on investigating a specific known source of performance non-determinism, namely *cache memories*[3].

1.1 Problem statement

The primary objective of this thesis is to improve our understanding of how to model the performance non-determinism associated with event processing in a modern CEP system called Siddhi. Specifically, the research aims to investigate the relationship between wall clock time and non-deterministic software execution, seeking to uncover any correlations or dependencies between the two factors. By studying this relationship, valuable insights can be gained into the impact of time on the variability of software execution in CEP systems.

Additionally, the thesis intends to evaluate the modeling methodology proposed by Kristiansen et al.[24] in the context of non-determinism. The goal is to determine whether this existing methodology is limited in its ability to effectively model non-determinism, and whether it should be extended to address the unique challenges posed by performance non-determinism in event processing. By identifying the limitations and potential gaps in the current modeling methodology, the research aims to contribute to the development of more comprehensive and accurate models for non-deterministic software execution in CEP systems.

Research questions:

1. How can we effectively model the performance non-determinism of event processing in Siddhi to enhance our understanding?

2. What is the correlation between wall clock time and non-deterministic

software execution, and how does it impact the performance of Siddhi?

3. To what extent is the modeling methodology proposed by Kristiansen et al.[24] limited in representing non-determinism, and what potential extensions can be made to overcome these limitations?

1.2 Methodology and approach

To address the research questions and account for the complexity of the application and its execution environment, we follow an empirical method of observation, experimentation, and data analysis. This approach allows us to assess the impact of caching on event processing time in Siddhi.

In multi-programming environments, where multiple processes running concurrently[40], caching is known to cause non-deterministic behavior due to the time component of processes competing for its resources. To expose caching behavior, we subject Siddhi to a stream of events consisting of multiple short bursts. Our results demonstrate that when Siddhi processes a burst of events, the initial events within the burst experience significantly longer processing times compared to the subsequent events. This behavior is much like the execution pattern commonly observed in cache *misses*, followed by a sequence of cache *hits*.

We have devised a methodology to isolate and analyze the impact of caching on event processing, due to the high complexity of Siddhi and the execution environment. This methodology entails an iterative process that involves instrumentation, human investigation of trace files, evaluation of process non-determinism, and system tuning. Through this iterative approach, we strive to achieve a satisfactory level of performance determinism, allowing us to design an accurate software execution model that captures the caching behavior effectively.

1.3 Contributions

This thesis contributes valuable insights into the realm of performance non-determinism in event processing within the Siddhi stream processing engine. These insights encompass comprehensive assessments of the impact of various system attributes, including garbage collection, just-in-time compilation, paging, dynamic voltage and frequency scaling, and caching, on event processing. These assessments consider both the relationship between these attributes and performance non-determinism.

The obtained insights provide descriptive characterizations of the behavior of event processing resulting from the interplay of these system attributes. By shedding light on the effects of these factors on performance, the thesis enhances our understanding of the complexities and challenges associated with performance non-determinism in event processing in modern CEP systems.

Additionally, the thesis presents an effective methodology for fine-tuning the execution environment, specifically addressing performance non-determinism. This methodology enables the creation of an optimal environment for modeling purposes, allowing for accurate and detailed investigation of performance non-determinism. By providing a systematic approach for fine-tuning the execution environment, the thesis equips researchers with a valuable tool for conducting further studies and analyzes related to performance non-determinism in event processing.

1.4 Outline

The thesis is organized into several chapters, each addressing specific aspects of the research. The structure of the thesis is as follows:

Chapter 1: This chapter serves as an introduction, presenting the problem statement, outlining the methodology employed in the research, and summarizing the contributions of the thesis.

Chapter 2: In this chapter, a comprehensive background is provided on the modeling methodology proposed by Kristiansen et al.[24] as well as complex event processing. This background sets the foundation for the subsequent chapters.

Chapter 3: This chapter delves into the methodology used to evaluate the impact of caching on event processing. It covers various aspects, including the application under test, the instrumentation techniques employed, and the tracing framework utilized.

Chapter 4: Rigorous experimentation is conducted in this chapter to provide a detailed analysis of performance non-determinism in event processing. The findings and insights gained from these experiments are presented and discussed.

Chapter 5: The final chapter presents the conclusive results of the research. It reflects on the conducted research, summarizes the key findings, and provides suggestions for future work, thereby concluding the thesis.

By following this structured approach, the thesis systematically progresses from the introduction to the background, methodology, analysis, and ultimately, the conclusion, ensuring a comprehensive exploration of the research topic.

Chapter 2

Background

In this chapter, we establish a strong foundation by exploring the modeling methodology proposed by Kristiansen et al.[24] and CEP technology. Section 2.1 introduces the fundamental concepts and abstractions of the modeling methodology. We delve into the iterative modeling process, which have drawn inspiration from to design our own approach. Additionally, we examine how the methodology evaluates its models and highlight success stories where it has been effectively utilized.

Moving forward, Section 2.2 focuses on CEP, which serves as the core technology in Siddhi. Gaining a deep understanding of its principles is crucial for comprehending the intricate nature of the system. We explore key principles, such as events and their relationships, as well as event patterns, rules, and constraints. Moreover, we explore event hierarchies and views, providing a comprehensive overview of this domain.

By thoroughly examining both the modeling methodology and complex event processing, we lay a solid groundwork for our research

Disclaimer: The word "event" is heavily used in the Section 2.1 and 2.2. The word in each of these sections refer to completely different concepts. In the modeling methodology, an "event" correspond to instructions that impact execution flow according to context[24], i.e. the state of shared resources and threads. Whereas in the section about CEP, an "event" refers to a computational object that is a record of an activity in a system[27].

2.1 The modeling methodology

The modeling methodology defined by Kristiansen et al.[24] describes how to use traces from real systems to create execution models. One of the strong suits of this methodology is that the models are reusable. They are only required to be created once, and can later be integrated with other simulators. The methodology has previously been used to model execution of communication software on mobile devices such as Galaxy Nexus, Google Nexus, and Nokia N900.

The basis of this methodology is a set of high-level abstractions and event definitions. These abstractions facilitate modeling of a wide range of device types and may even be extended to new types of software and hardware when desired. The methodology also provides design principles on how to extend discrete event network simulators. It separates the modeling of devices from the process of extending a network simulator with new device types. Following these two principles results in models that are independent of network simulators, which reduces the overall modeling effort. Instead of having to create a model of the same device multiple times, the device models can be distributed and reused in other discrete event network simulators, given that the simulator provides a well-defined extension.

The models created are defined by high-level statements, enabling reparameterization and allowing alternative compositions of models. These types of configurations all impact the simulation results to some degree, and this level of modularity of *Service Execution Model* (SEM) definitions encourages low-effort studies on the impacts of modifications. The overall goal of the modeling methodology is to create models that predict the impact of protocol processing on system performance.

2.1.1 Core concepts and challenges

In this section, we cover the core concepts and abstractions given in [24]: *events*, *execution units*, *services*, *execution context and behavior*, and *processing stages*. Then we examine how the methodology measures packet processing delays.

The methodology is designed to model the execution of communication software in multi-threaded systems. Operating system design and computer architecture make describing software execution in multi-threaded systems a challenging task. Multi-threaded systems may have separate physical execution units executing software in parallel, such as the CPU, DMA controller and NIC. Examples are the CPU running software on multiple cores, or the DMA controller handling multiple DMA transfers simultaneously, or the NIC receiving data from memory in parallel. The methodology has to take into account that threads and interrupt handlers share CPU time, and execute the software in a serialized manner. Furthermore, software execution in multi-threaded systems are impacted by the complex communication and synchronization behaviors of threads, and these behaviors are depended on the execution context.

The execution behavior of the communication software is described through (1) the events that impact the execution flow of the program, (2) the processing durations of the protocol services, (3) the interactions between services, and (4) the impact of execution context.

To create the models of communication software, it is crucial to determine which events describe the program execution, and then capture these events. Determining events is part of the stage of system analysis which we

expand on later. Three important abstractions of multi-threaded dynamics are used to determine events. They are the following: (1) execution units, (2) services, and (3) behavior and context.

Abstractions of multi-threaded dynamics

Execution units are a high level abstraction of software execution in a multi-threaded system. There are two types of execution units: Physical Execution Units (PEUs) and Logical Execution Units (LEUs). The PEUs represent the hardware executing the communication software. The LEUs are the threads and interrupts in the operating system. The operating system assigns programs to run inside LEUs. It also schedules the LEUs execution, and provides communication between individual LEUs. After the OS activates an LEU, it dispatches the LEU for execution at some later point in time. Interrupts are activated by PEUs, while threads are activated by LEUs.

Services are an abstraction of software running within LEUs. The services are separate modules assigned to perform particular tasks. They are implemented by a special function, which calls other functions either directly or indirectly from within itself. The only functions that cannot be called from a function implementing a service, are functions that implement other services, i.e. other special functions. Services are communicating through LEU-requests. The methodology defines two types of LEU requests: intra- and inter-LEU request. An *intra*-LEU request is a function call from a service running within the same LEU, and an *inter*-LEU request is a function call to a service running within another LEU. Inter-LEU requests are passed via a queue to the targeted service.

Behavior and *context* are abstractions of the variations in service execution. The context is the part of the state that affects the service, while the behavior is the sequence of instructions executed by the service under a given context. A service can exhibit a number of behaviors less than or equal to the number of contexts it may execute under.

The program flow within individual services are determined by branching instructions and processing durations. The overall program flow of the multi-threaded system is determined by the synchronization and communication between services running in separate execution units.

Certain conditions in the system affect the service behavior in various ways. These sets of conditions are called *execution contexts*. To model the service effectively, we need to trace all relevant execution contexts of the targeted service. An execution context is set by the packet characteristics, the state of queues in the system, and the protocol state. The state of the threads is included in the execution context for multi-threaded systems.

Packets characteristics such as packet size and type can impact software execution, for instance large packets may take longer time to process than small ones. The queue states in the system determine whether an incoming

packet needs to wait before being processed. If the queue is empty, the packet can be processed immediately, but if the queue is semi-full, the packet must wait. In the worst case, the queue is full and drops the packet. Thread state, such as *active* or *ready*, also impact processing times, e.g., threads failing to acquire a lock and having to wait.

Measuring packet processing delays

Packet processing delays are measured by the number of cycles spent per packet and the clock frequency of the CPU. The number of cycles spent is dependent on the instructions executed. Packet processing delay is therefore determined by the complexity of the service implementation and also its execution. The number of cycles are spent per instruction may vary due to pipeline flushing, branch misprediction, and cache misses. CPU cycles is an accurate and efficient metric of packet processing delay. It is effective in estimating processing durations under varying CPU frequencies.

2.1.2 The iterative modeling process

In this section, we outline the iterative modeling process used in the methodology. We begin by providing a concise overview of each processing stage, followed by a comprehensive examination of each stage in detail.

The goal of the iterative modeling process is to create trace-based models with sufficient accuracy. Each model obtained is a model of the service behavior under each individual context. These models are represented as sequences of events and distribution functions of processing stage durations between consecutive events. Model accuracy is affected by the granularity of the model, the number of execution contexts, and the number of iterations in the modeling process.

The following list is an overview of the step-wise approach of the modeling process:

- **Instrumentation:** instrument the software with *tracepoints* to capture all relevant temporal behavior of a service.
- **Tracing:** collect the traces from the knowledgeable trace experiments, and assess how different parameters impact timing results.
- **Automatic analysis:** transform the traces into a set of *signatures* through filtering and clustering.
- **Human investigation:** inspect the quality of the traces, and place the signatures in a *device file*.
- **Model creation:** parse the device file and combine signatures into *Service Execution Models*.

The iterative modeling process consist of 5 stages. The modeler performs modeling stage 1 through 4 on every service of the communication software. For every service, the modeler performs stage 1 through 3 to achieve acceptable quality of instrumentation. The resulting models collectively model the execution of the target communication software in a real device. The models can later be used to extend existing network simulators.

Instrumentation

The first stage is *instrumentation*. The modeler places tracepoints into the code of the service that needs to be captured, and executes it with different workloads. A tracepoint is merely a function call that generates an event with a set of attributes, such as the event type, the value of the CPU cycle counter, and the memory address of from which the tracepoint was called. The remaining attributes are used to store state variables and identify shared resources. The memory addresses identify unique locations in the final model, and these locations can further be associated with functions in the objective network simulator.

To make the instrumentation effective, the modeler has to find the locations of the events that characterizes execution behavior of a service. They will likely perform multiple series of instrumentation, tracing and human investigation of the service to find all significant events. The methodology presents multiple classes of events, however we are not delving into every class in this discussion. Instead, we focus on selected classes and provide only minor details about them.

Traces of Class 1 and 3 events are used to distinguish between LEUs and capture interactions with the thread scheduler. Class 1 events are captured from functions performing context switching and in the moments right before and after execution of interrupt handlers. Class 3 events are captured from functions that manipulate the ready queue and functions that interact with synchronization primitives.

Class 1, 4, 5 and 6 events are used to instrument services. SRVEntry events are captured at the beginning of a service function, and SRVExit are captured right before the return statement of the function. Certain device-specific drivers and non-standard or experimental networking protocols are not included in the mainline kernel distributions, which means that they are not covered by the one-time OS instrumentation. If the networking simulator supports any of these services, they also need to be instrumented.

Class 6 events are determined by the state variables that have non-negligible impact on service behavior. The addition of Class 6 events increases the number of attributes in the trace events since it requires the events to also hold the values of state variables. Identifying these state variables are crucial to the modeling process. The Class 6 events must be captured at the point where the state variables start to impact the behavior of a service. For instance De-Queue events are placed at the beginning of

queuing functions to capture packet characteristics, and StateRead events are placed as closely as possible to conditional branching points to capture the value that determines the branching decision.

Tracing

The next stage is *tracing*. Here the instrumented software is executed `n` times with a static context. Multiple executions are necessary to collect sufficient amount of data for plotting the empirical processing distributions. The set of contexts are defined by the parameter values that cover the parameter spaces. In practice of the methodology, finer granularity of values increases the accuracy of the final models, but higher modeling effort also increases the complexity of the resulting models. It is the main responsibility of the modeler to find the appropriate balance between sufficient accuracy and simulation overhead.

Certain contextual parameters are difficult to control, especially the parameters that are part of the non-deterministic behavior of the operating system. These are parameters such as the state of queues of threads. A service must therefore be executed under a subset of c' of defined context c. Of all possible contexts, we can only stage deliberately a subset of them. An example of a context that can be controlled is the *packet size*. Here the modeler can inject a stream of packets of the same size into the system such that the instrumented service handles each packet. The resulting event traces are used at input to the automatic trace analysis.

Automatic trace analysis

In the stage of automatic trace analysis, a trace file is transformed into a set of *signatures*. Each signature describes one behavior of the traced service. A signature is represented as a sequence of events and processing duration distributions between these events. The signatures are created through *filtering* and *clustering*.

A trace file may contain events that are not specifically related to the target service S. To extract the relevant events that are part of S, it utilizes a filtering process. This filtering step aims to isolate and extract the events that are directly associated with the execution of the service. As a result, it produces one sequence of events per execution of the service. Each sequence of events is called a *case* and describes a *single execution* of S. The number of extracted cases n is equal to the number of times the service was executed to generate the trace file. Loops are also modeled as services, where each iteration yields a case.

Clustering sort the n cases in m groups. Each group describes one behavior of the traced service. All cases within a group have the same behavior, i.e. the same sequence of events and all events in the sequence share the same attribute values. The only difference between cases in a group is the execution times of processing stages. This is caused by the

non-deterministic behavior of the CPU, like memory caching, and minor variations in the number executed instructions.

Signatures are generated by calculating a probability function that fits the processing times of each case in a group. Signatures, cases and processing durations are all stored in human readable form, including metadata that describe which cases are used to create which signatures. This information is used during human investigation.

Human investigation

The fourth stage is *human investigation*. Here, the modeler analyzes the quality of a signature and investigates whether the collective signatures is a good model of the service or not. The quality of the instrumentation is assessed based on the resulting cases from the previous stage and the signatures themselves. This stage is important since insufficient instrumentation leads to inaccurate models. The instrumentation needs to be extended for another iteration of the modeling process if the quality of the signature does not meet the quality requirements. It is worth to note that the location attribute of the event traces provide valuable hints of where to place additional tracepoints. The methodology introduces two techniques to analyze the quality of a signature.

The first technique is to make sure that the automatic analysis generated one signature per context, i.e. the number of signatures m are equal to the number of contexts c'. All context parameters are stored in Class 6 events. Cases with identical event sequences but different attribute values will be clustered in separate groups, which results in multiple signatures supposedly describing the same behavior. This implies that the target service is not properly captured. A signature that is aggregated by a very low number of cases, describes most likely rare behavior, and can therefore be excluded from the model.

The second technique is to plot the empirical probability distribution graph of the processing stage durations. In the cases of one group, the amount of processing performed in equivalent processing stage should be the same. In other words, approximately the same number of instructions are executed in each case. *Unimodal* probability distributions indicate proper instrumentation, while *multimodal* distributions with distant modes, imply that multiple completely different tasks were executed between events. In simpler terms, important events were not captured.

Combining signatures

Lastly is the stage of model creation, where the modeler combines the signatures into an execution model. The iterative modeling process has to be done for every service of interest of the target communication software. The end result is collection of all the signatures for all services. This collection is stored in a *device file*. The device file is passed to a parser which

combines the signatures for each service and outputs one executable model per service. These models are called *Service Execution Models* (SEMs).

Multiple signatures may share common subsets of events from the beginning of the sequence until a Class 6 event. However, the sequence of events following the Class 6 event differ between these signatures. The signatures with a common subset can be combined into a tree, where they follow the same path of events from the root until they reach the Class 6 event. Here, a conditional statement creates a branching point based on the attribute values of the Class 6 event. Each value of the state variable creates another subtree. The execution of the service statements start at the root of the tree, and once it encounters a conditional statement, the modeled state variable is read to select the appropriate subtree to be executed next.

Extending network simulators

The methodology defines a highly flexible mechanism to map SEMs onto protocol models in existing discrete event simulators. Mapping SEMs onto protocol models enhances the models with the timing behavior of real software implementations.

2.1.3 Model evaluation

Each SEM models individual portions of software execution. When the modeler evaluates the SEMs, it is important that they compare the behavior of SEMs during execution to the behavior of running the real device. By making this comparison, the modeler can efficiently assess the accuracy of the final model.

2.1.4 The methodology in practice

In recent research by Espen Volnes et al.[44] the methodology was used to simulate protocols in wireless sensory networks. At the time of the research, the network simulators available ignored the processing delay of packets within nodes, since intermediate nodes in classical networking architectures uses specialized hardware that yield almost zero delay in packet processing. In the case of wireless sensory networks, each node in the network is a general purpose computing unit with small amount of processing power. These units are called *motes*. The intra-OS delay is substantial in these motes and therefore cannot be neglected. An alternative was to run emulated motes in the Cooja network simulator which would capture the packet processing delay. However, WSNs are usually built out of hundreds to thousands of nodes, and emulating each node in a large simulation is simply not scalable. The modeling methodology was successful in simulating the intra-OS delay in a TelosB mote running TinyOS. The results showed that models created with this modeling methodology can provide accurate software execution simulations in motes. These models can also effectively be integrated into general-purpose simulators such as ns-3.

2.2 Complex event processing (CEP)

In our every day language, we refer an "event" to something that happens[28], while in the context of CEP, an "event" refers to an object that can be processed by a computer. CEP consists of a set of techniques and tools to help us grasp and control event-driven information systems. Techniques such as tracking causal histories of events, using patterns of events and event relationships to recognize the presence of complex events, and older techniques found in rule-based systems. A *complex event* is an event could that only happen if a collection of other events happened.

CEP takes advantage of the fact that there exists some form of relationship between the events in a system. It focuses on three types of relationships: *time*, *cause*, and *aggregation*. The presence of relationships between events add another dimension to event processing. It enables the system to answer questions raised about events that are more than low-level network activities, such as questions about high-level activities related to business- and strategic-level events. For example, "Is our system providing the correct level of service to our customers?", "What caused the router in Frankfurt to be overloaded?", and "Is the system under a denial-of-service attack?". For the system to answer the third question regarding a DoS attack, it depends on real-time recognition of complex patterns of events that indicate such an attack.

Lets look at an example of the *completion of a financial transaction*. It involves a bundle of financial contracts. In the span of a couple of days, several merchant banks and brokerage houses participate in the transaction. The event itself, i.e. the transaction, is the result of hundreds or thousands of electronic messages and database entries around the world. These simple events of sending messages and entries to databases, are not necessarily happening in a linear fashion. They will most likely happen simultaneously and independently of others, even mixed with events of other transactions.

The communication flexibility of the Internet makes it challenging to track the cause of an event. An event can arrive from any place in the Internet, and when it does, there is no apparent cause behind it. Say we have a transaction that failed to complete. *Why did it happen? Which events caused it?* Without CEP, we would resort to processing many large low-level network logs in order to answer these questions. Not only is this tedious work, but it is inefficient, primitive and can easily fail. The fact is that it is very hard to understand network traffic if we can not, at any point in time, see who is communicating with whom, or get a view of the events that cause another event to happen in the network. If we only monitor message traffic in our systems, we fail to recognize a lot information. The messages would simply be passed silently between information systems as unrelated pieces of communication. But when events aggregated together, correlated, and their relationships understood, they offer great deal of information.

2.2.1 Events and event relationships

In this section, we cover the main concepts of CEP. First we explain what *events* are, where to find them, and how they are created. Then we discuss the different relationships that exist between events: *time*, *cause*, and *aggregation*. All definitions and quotes are adopted from the pioneering textbook on CEP systems by David Luckham[27].

"*An event is an object that is a record of an activity in a system.*"([27], page 88) The event *signifies* the activity, and may be related to other events in the system.

There are three aspects of an event:

- **Form:** An event is formed as an object. It may have attributes or data components. The form is often a tuple of data components, such as the time period of the activity, the location of occurrence, who did the activity, and other data.
- **Significance:** An event *signifies* an activity. The activity is the *significance* of that event. A description of the activity is included in the form of the event.
- **Relativity:** An event is related to other events by *time*, *cause*, and *aggregation*. The relationships between an event an other events are called its *relativity*. An event's form usually contains methods that can be invoked to reconstruct the relationships with other events.

Creating events

The process of creating events is divided into two parts: *observation* and *adaption*. The part of observation entails observing the activities happening at any level of the system without affecting behavior. In the part of adaption, observations are transformed into event object that can be processed by CEP. The tools performing the transformations are called *adapters*.

There are three major sources of events:

- **IT layer:** The communication between components at different levels of the system can be observed from the IT layer. This communication is in the form of messages or method calls.
- **Sensors:** Sensors detect events and changes in their environment. An event detected by a sensor can be a button press, a noise or temperature level hitting a certain threshold, etc.
- **Instrumentation:** Components of the target system are instrumented to create events signifying the activities happening in the component. Low-level events can, for instance, be generated from status probes in the operating systems, or heartbeats and alerts generated by network management systems. Higher-level events can be created from the instrumentation of applications.

- **CEP:** Events created by CEP.

Event relationships

Time: *"Time is a relationship that orders events – for example, event A happened before event B."*([27], page 94)

A time relationship between events depends on a clock. When there is an occurrence of an activity, that has an event that signifies it, the event is created and given a timestamp. The timestamp is generated from a read of one of the clocks in the system at the time of the activity. The order of timestamps defines the time relationship between events. Similar to systems having multiple clocks, events may have multiple time relationships, one for each clock. The timing relationship between two events may not be possible if the clocks are not comparable. This is dependent on the information available on how the clocks are related (e.g. synchronized or independent).

Cause: *"If the activity signified by event A had to happened in order for the activity signified by event B to happen, then A caused B."*([27], page 95)

The definition shows a dependency relationship between activities in the system. An event *depends* on other events only if it happened because the other events happened. Consequently, events can be *caused* by other events. Two events are independent if neither caused the other.

Aggregation: *"If event A signifies an activity that consists of the activities of a set of events, B_1, B_2, B_3, ..., then A is an aggregation of all events B_i. Conversely, the events, B_i, are members of A."*([27], page 95)

Aggregation is an abstraction relationship between a single event and a collection of events. From the definition A yields a high-level event that signifies a complex activity. The activity is *complex* in the sense that multiple event has to happen, possibly in different locations and points in time. Therefore we call A a *complex event*. One can argue that a "composite event" would be a more suitable name for an event such as A, since A is technically a composition of events that creates the high-level event. The members of a complex event most likely happened at different times, which means that a single timestamp would not be representative of the timing of a complex event. Instead, a time interval attribute is used, where the timing starts from where the earliest activities of the member events start, and ends where the latest activities of its member events ends.

Genetic parameters in events

An event's timing and causal relationships to other events are stored in its form. This special data structure is the genetic parameters of an event, as they give information about the event's timing and causal history. There are only two genetic parameters: *timestamps* and *causal vector*.

Timestamps refers to the readings of various clocks in the system at the

start and end time of the activity signified by the event. Some events only contain one time timestamp that correspond to a single point in time when the activity occurred. The events may also contain a timestamp of when the event was observed by some observer in the system. In that case, the timestamp refers to the *arrival time* at that particular observer. The timing relationships is especially useful when evaluating the system's performance, as well as debugging. For instance, if the system is not performing properly, we can use timing as a filter to identify which event happened before the event that is being investigated, consequently narrowing our search space to the earlier events.

Causal vector is a collection of the unique event identifiers of the events that caused the event. They define the causal attribute of the event. By storing the causal vector in the event makes it easier to trace causal relationships in complex systems. Instead of relying on separate components to fetch the information, such as a database, one can easily locate the data in the event itself. This has the benefit of facilitating the task of building tools for causal tracking.

Event causality

To really understand events, we need to know what *caused* them, also while they are happening. When the causing event and the caused events are happening on the same conceptual level in the overall system, it is referred to as *horizontal causality*. Complex events are build by lower-level events events, and these events can be considered as components or *members* of the complex event. This causal relationship is called *vertical causality*.

Vertical causality has two important use-cases:

- Knowing how are business event is broken down by a group of lower level-events, is helpful in understanding the properties of the higher-level event. With this knowledge we can identify bottlenecks in the business logic, and improve performance at the business level.
- We can use vertical causality to group lower-level events together based on the higher-level activities they signify. This makes it easier for us to understand the vast amount of low-level events observed in network monitoring tools.

Before we can connect a group of lower-level events as the cause of a single high-level event, the high-level event must already be identified. The identification of a high-level event are usually rooted in some perception, suspicion, or a list of complaints on the behavior of the system. However, whether we can properly identify a high-level event is dependent on our understanding of the lower level events. If we can not make sense of the great number of events happening at the lower layers, we are unable to connect them to high-level events. Similar to downward tracking of vertical causality, we can aggregate the lower-level events into a single higher-level event. The resulting event expresses the collective meaning of the

aggregated lower-level events.

2.2.2 Event pattern, rules and constraints

A fundamental part of viewing and controlling event-driven systems, is the ability to select a set of events that are of interest as they happen from large event executions. The selection of events is the result of events matching defined *event patterns*. Event patterns, event pattern rules, and event pattern constraints are the primary concepts that form the foundation for CEP applications.

An *event pattern* is a template that matches specific sets of events. The pattern describes the events, including their causal relationship, timing, data parameters, and context. Hence, an event pattern is also a template for *posets*.

An *event pattern rule* is a rule that specifies a set of actions to execute in response to occurrences of events that match a specific event pattern. It implies a causal relationship between the events that trigger it, and the events that are created when the rule performs its actions.

The event pattern rule is essentially reactive and consists of two parts:

- "*A* trigger*, which is the event pattern*"([27], page 119)
- "*An* action*, which is an event that is created whenever the trigger matches*"([27], page 119)

The *causal implication* is that the triggering events matching the event pattern *causes* the new event to be created. The triggering events are causal ancestors of the higher level event.

Event pattern rules can either be *sequential* or *parallel*. When a sequential rule is triggered, all of its actions are executed in a sequential manner. As a result, all the events created are causally ordered in a sequence. Conversely, when a parallel rule is triggered, the actions are executed independently from one another. As a consequence, the events are causally independent, except if the events created on one triggering, match a second triggering of the rule.

An *event constraint* expresses a condition that must be met by the events observed in a system. Constraints are used to specify how a system should behave, and how its users should use it. A constraint tests for a certain behavior of the system during runtime. Once the rule of the constraint is triggered, it produces a result; *violated* or *satisfied*. A violation event is created every time a violation occurs, while a satisfied result occurs when passing a certain time limit or when the target system ceases operation. Constraints neither enforce or guarantee behavior, they are simply *watchdogs* of the system.

2.2.3 Event hierarchies and personalized views

In this section, we start by covering *event abstraction hierarchies*, which is an important concepts in developing CEP applications. Next, we give and overview of *personalized and role-based view of hierarchical systems*.

Event hierarchy

An event abstraction hierarchy is how CEP organizes the events in a system. It enables CEP to infer higher-level events based on the lower-level events observed in the system. The inferred events assist us in understanding the activities that can happen. CEP is limited in what it can infer by what it can observe at the lowest level of the system.

Event abstraction hierarchies consist of two elements and they are defined as follows:

- "*A sequence of levels of activities. Each level consists of a set of descriptions of system activities and, for each activity, a specification of the types of events that signify instances of that activity. Level 1 is the lowest level.*"([27], page 131)
- "*A set of event aggregation rules for each level. For each level (except level 1), there must be a rule for creating each type of event at that level as an aggregation of events at levels below.*"([27], page 131)

Personalized view

Event abstraction hierarchies are used to specify and implement *personalized views* of the system.

There is an important distinction between *monitoring* and *viewing* the system. Monitoring refers to observing and analyzing level 1 events, while viewing refers to computing and analyzing higher-level complex events. The complex events are computed using aggregation rules, and analyzed using graphical and drill-down techniques. Drill-down refers to the process of backtracking from a complex event to its members.

CEP provides great flexibility by allowing configurations of existing hierarchy definitions, as well as creation of aggregation rules while the system is running. This flexibility greatly benefits the stakeholders of the system, as there is natural changes of interests in what the system is doing. Also, it is difficult to predict the most useful views in advance. Additionally, views and complex event types can be defined to fit specific roles and needs, and personalized views can be applied to any level of system activity, such as the network level and the application level.

Creating a personalized view is a two-step process:

- Specify the types of events of interests at each hierarchical level as CEP event types. This is the step of *personalization*. It is important

that the events are well organized, since grouping high-level and low-level events at the same level would interfere with the second step.

- Define an aggregation rule for each type of event at each level above level 1. The rule has to define the event pattern that describes the set of lower-level events and their relationships that yields the higher-level event. Next, it has to specify the event object that is created when the event pattern is matched.

2.2.4 Summary

CEP has a very broad applicability due to the fact that information system are all driven by events. It provides techniques for defining and applying relationships between different kinds of events. It even lets you define your own events as patterns of the events in your system, making your defined events appear when the patterns are matched, and giving you a better understanding of what is going on in your system. When CEP is applied to information systems, not only can we recognize when a complex event happens, but we can also predict whether it is going to happen, or if it is tracking off and why.

Chapter 3

Methodology

The goal of the performance evaluation in this thesis is to model the impact of caching on event processing performance in Siddhi. Caching is a widely used technique to improve system performance in various levels of the system stack. At the operating system level, the buffer cache and the page cache serve to minimize the number of disk accesses by storing recently used disk blocks and file data in low latency storage. This approach enables faster read operations, circumventing access to data located in slower disk storage. Given the historically high latency of disk I/O, numerous layers of the software stack attempt to avoid it by caching reads and buffering writes. However, the types of caches present within a system vary depending on the environment and intended use-cases. For example, web browser cache serves as a client cache, while the MYSQL database management system stores frequently used data in the MySQL buffer cache[16].

Considering that caches exist in multiple levels of the system stack, identifying their location can be crucial in developing accurate execution models. The system under test (SUT) is the Siddhi CEP-engine. This is a highly complex system as it is written in Java, which runs on the JVM and therefore is prone to all sorts of runtime optimization. Furthermore, the JVM runs as a software layer on top the operating system and is affected by optimizations on the hardware level. The system stack is depicted in Figure 3.1.

These four levels may each contain a caching mechanism that affects event processing performance:

- **Java program (running Siddhi):** Due to the overall complexity, we have written a simple Java program. Although our program itself does not contain any software caches, it is dependent on the Siddhi Core Libraries, which have numerous dependencies. For instance, `siddhi-core` module, has 21 dependencies alone. If we were to go through the complete dependency graph, it is hard to imagine that there are no software caches being utilized. Nevertheless, the concern really is: *Are there any software caches that affect the event processing time?*

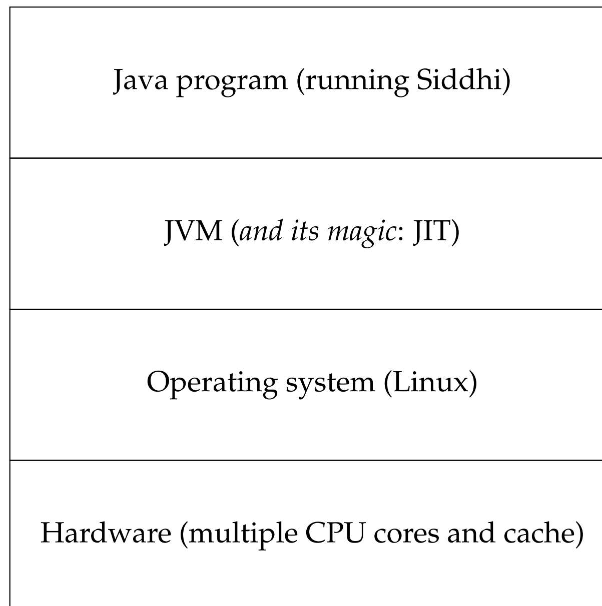

Figure 3.1: System stack

- **JVM (*and its magic*: JIT):** The JVM optimizes Java bytecode at runtime, which can cause non-deterministic execution behavior. This process of optimization is referred to as the *JVM warm-up*. Fortunately, this optimization can be completely turned off. Typically, people often care more about software performance than performance determinism in real-world applications. Thus, we are forced to model the optimized, and in affect non-deterministic execution of the program to create accurate simulations.

- **Operating system (Linux):** The Linux operating system uses a disk cache, which is a software mechanism that allows the kernel to keep some data in memory that is normally stored on a disk, such that subsequent references to that data can be quickly satisfied without a slow access to the disk itself[8]. *Are there software mechanisms in the operating system, such as disk cache, that affect event processing performance?*

- **Hardware (multiple CPU cores and cache):** CPUs have at least three independent caches: (1) the instruction cache, (2) data cache, and (3) translation lookaside buffer (TLB). The cache controller is responsible for keeping track of what parts of memory are stored in the caches and where. The cache provides faster memory requests of instructions or data, given that they are present in the cache. Modern microarchitectures have three levels of cache: L1, L2 and L3. L1 has the fastest access time, as it is located closest to the CPU. L3 has the slowest access time and located the furthest from the CPU. The cache memory is small and is used by multiple processes in the system. Due to the size of the cache, the data stored is constantly replaced by

other recently used data. Exactly which data is replaced is dependent on the cache replacement policy of the CPU. CPUs can also change frequency depending on the workload, this is known as *dynamic voltage and frequency scaling* (DVFS) and can affect event processing time if the CPU suddenly changes frequency during execution.

The modeling methodology proposed by Kristiansen et al.[24] models software execution at a granular level, e.g., modeling branching statements. Our modeling target, however, is a highly complex system in each layer of the stack, and therefore we can not strictly follow the modeling methodology, as it would require too much modeling effort. Furthermore, we observe how *time* affects event processing behavior, a dimension that is not considered in the methodology. Therefore, we follow to some extent the iterate modeling approach by Kristiansen et al.[24] presented in Section 2.1.2, but rely more on human investigation.

Our modeling approach is as follows:

1. **Instrumentation:** Instrument the software with *tracepoints* to capture all relevant temporal behavior of the target service.
2. **Tracing:** Collect traces from knowledgeable trace experiments, and assess how different parameters impact timing results.
3. **Human investigation:** Inspect non-deterministic behavior in traces and investigate possible causes.
4. **System tuning:** Strip away external configurations in the SUT that may cause the non-deterministic timing behavior. Go back to step 2).
5. **Model creation:** Create execution model.

Figure 3.2 illustrates our modeling approach through a flow diagram. The process begins by instrumenting the SUT with tracepoints in the source code. Then we trace its execution, resulting in a trace file. As part of the human investigation, the file is parsed and analyzed. Based on the outcome of the analysis, we determine whether the system execution exhibits deterministic timing in relation to the service. If non-determinism in service execution is recognized, we investigate the service behavior further and assess the existing instrumentation.

Insufficient instrumentation of a service is often characterized as multi-modal distributions featuring distant modes within a probability distribution graph of the traced service[44]. This typically indicates a failure to instrument key features of the source code, such as branching statements or loops. If, despite proper instrumentation of the service, we continue to observe performance non-determinism, then we need to investigate the potential sources in the execution environment, and assess and mitigate their impact as far as possible. This represents an iterative process, continuing until a satisfactory level of determinism is attained.

The following sections of this chapter explores the SUT, its implementation, and the way it is instrumented. Furthermore, they define the metrics

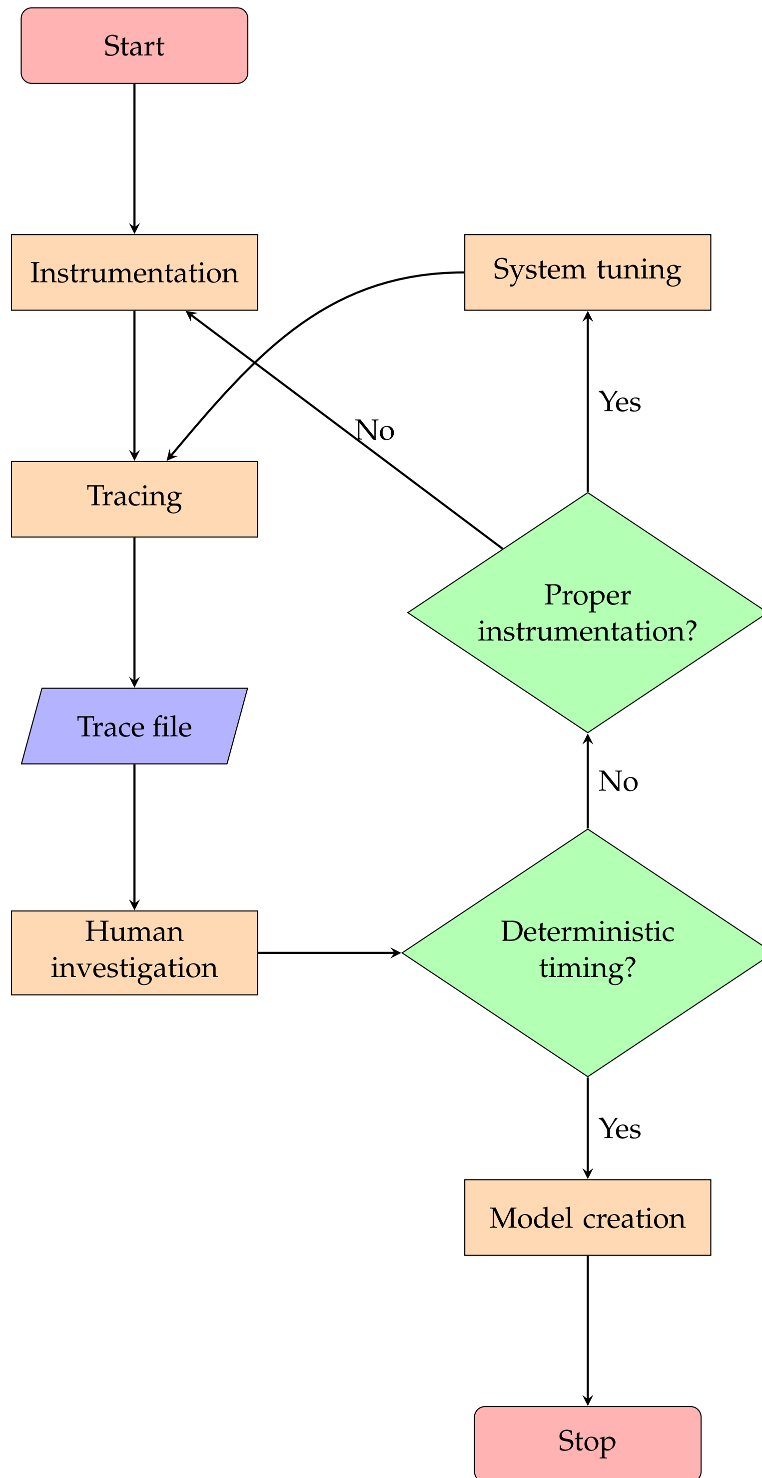

Figure 3.2: Flowchart of system analysis and modeling approach.

used throughout this thesis, and cover the tracing framework that was implemented to instrument the SUT.

3.1 System under test (SUT)

This section provides an overview of the SUT. We delve into its implementation and discuss how it is instrumented to capture event processing timings. The SUT in this thesis is the Siddhi stream processing engine. To showcase the SUT, we have created a simple application that generates and processes both bursty and non-bursty traffic of event tuples.

3.1.1 Event stream

The application defines an event stream, where each event tuple comprises a unique ID, a company symbol, a price, and a volume. See Listing 3.1 for further details.

```
StockStream(id long,symbol string,price float,volume long)
```

Listing 3.1: The defined stock stream.

3.1.2 Pre-generated events

The application parses a text file with 1000 pre-generated events in the format depicted in Listing 3.2. These pre-generated values can be reused in order to conduct experiments requiring a higher number of event to be processed. The parsed events are loaded into memory, and at the point of injection, each event is assigned a relative timestamp corresponding to the order they are injected into Siddhi. By storing the pre-generated events in memory prior to transmission, we aim to minimize the involvement of the file system in event processing as much as possible. Moreover, we strive to minimize the computational overhead associated with event injection. In other words, we aim to achieve a high event injection rate to accurately simulate event bursts.

```
<SYMBOL> <PRICE> <VOLUME>
```

Listing 3.2: Format of the text file of randomly generated stock events.

3.1.3 Injecting events

Events are injected into Siddhi for processing by calling the `send` method in the `InputHandler` class of Siddhi. Listing 3.3 shows how bursts of events are injected for processing in our application. In the inner loop, the application first calculates the next sequence number, extracts one of the pre-generated events, and injects it into Siddhi. The event ID is assigned as the event's timestamp. This process is repeated for the number of events specified in a burst. The outer loop puts the application thread to sleep for

a specified number of milliseconds (burstDelay) between each burst. The burst distance is provided by the user as a command-line argument.

```
private static void injectEventBursts(InputHandler inputHandler,
                                      int burstDelay)
        throws InterruptedException
{
        final int total = burstSize * count;
        final int id_idx = 0;

        for (int i = 0; i < total; i += burstSize) {
            for (int j = 0; j < burstSize; j++) {
                long event_id = (long)(i + j);
                Object[] obj = list.get((int)event_id % list.size());
                inputHandler.send(event_id, obj);
            }
            Thread.sleep(burstDelay);
        }
}
```

Listing 3.3: Code for injecting event burst from pre-generated events.

When the burst distance is zero, we call a different method that does not include the Thread.sleep() call. This is because calling Thread.sleep(0) will cause the thread to yield, and subsequently be rescheduled. Since we aim for the system to be as deterministic as possible, minimizing the rescheduling of processes is crucial.

3.1.4 Stream call-back

A StreamCallback instance can be mapped to a defined event stream in the Siddhi runtime and acts as the endpoint for received events satisfying the event query, as demonstrated in Listing 3.4. If the received event is marked with the relative timestamp -1, which denotes the end of the stream, it writes the collected traces from our tracing utility to a file and shuts down the Siddhi runtime, thereby terminating the application.

```
siddhiAppRuntime.addCallback("OutputStream", new StreamCallback() {
        @Override
        public void receive(Event[] events)
        {
            long id = (long) events[0].getTimestamp();
            if (id == -1) {
                writeTracesToFile();
                siddhiAppRuntime.shutdown();
            }
        }
    });
```

Listing 3.4: The Siddhi runtime callback invoked upon receiving events.

3.2 Instrumentation

The main goal of our instrumentation is to assess the performance of event processing by measuring the time taken to process each event in a specific stream. To achieve this, we instrument the event injection process by placing tracepoints directly before and after the `send` call, as shown in Listing 3.5. In order to capture the precise duration of single event processing, we measure the event processing time in nanoseconds. This level of granularity is necessary as measurements in milliseconds do not provide the required level of detail for accurate estimations of individual event processing.

```java
private static void injectEventBursts(InputHandler inputHandler,
                                      int burstDelay)
        throws InterruptedException
{
        final int total = burstSize * count;
        final int id_idx = 0;

        for (int i = 0; i < total; i += burstSize) {
            for (int j = 0; j < burstSize; j++) {
                long event_id = (long)(i + j);
                Object[] obj = list.get((int)event_id % list.size());
                TraceUtil.addTrace(event_id, System.nanoTime());
                inputHandler.send(event_id, obj);
                TraceUtil.addTrace(event_id, System.nanoTime());
            }
            Thread.sleep(burstDelay);
        }
}
```

Listing 3.5: Code for injecting event burst from pre-generated events.

3.2.1 Burst processing times

We measure the processing time of a burst from the moment the first event in the burst is injected into Siddhi to the moment when the last event in the burst is finished processing. Listing 3.6 shows a heavily truncated version of a trace file in which only the first and last event of the first three bursts are included. Each timestamp that is part of the calculation of burst processing time is highlighted with a unique color for each burst. The calculation of the processing time for these bursts is shown in Equation (3.1).

```
0,SiddhiApp.injectEventBursts(SiddhiApp.java:230), 25347858375282
0,SiddhiApp.injectEventBursts(SiddhiApp.java:232),25347865084941
. . .
999,SiddhiApp.injectEventBursts(SiddhiApp.java:230),25347910345938
999,SiddhiApp.injectEventBursts(SiddhiApp.java:232), 25347910375579
1000,SiddhiApp.injectEventBursts(SiddhiApp.java:230), 25348913577706
1000,SiddhiApp.injectEventBursts(SiddhiApp.java:232),25348913947198
. . .
1999,SiddhiApp.injectEventBursts(SiddhiApp.java:230),25348949802146
1999,SiddhiApp.injectEventBursts(SiddhiApp.java:232), 25348949828524
```

```
2000,SiddhiApp.injectEventBursts(SiddhiApp.java:230), 25349950154840
2000,SiddhiApp.injectEventBursts(SiddhiApp.java:232),25349950369623
. . .
2999,SiddhiApp.injectEventBursts(SiddhiApp.java:230),25349983716091
2999,SiddhiApp.injectEventBursts(SiddhiApp.java:232), 25349983741867
. . .
```

Listing 3.6: Excerpt from log file, cut for readability and brevity, with highlighted timestamps caption to show start and finish time of a burst. The colors are unique for each burst.

$$\begin{aligned} 25347910375579 - 25347858375282 &= 52000297 \\ 25348949828524 - 25348913577706 &= 36250818 \\ 25349983741867 - 25349950154840 &= 33587027 \end{aligned} \tag{3.1}$$

3.3 Metrics

In this section, we are defining the metrics and their notation that is used throughout this thesis:

- S^n_m: Event stream of m bursts with n events in each burst.
- $S_{i,*}$: i-th burst in stream S.
- $S_{*,j}$: j-th event in each burst in stream S.
- $S_{i,j}$: j-th event in the i-th burst in stream S.
- S_n: n-th event in stream S.
- $Time(S_{i,*})$: Processing time of the i-th burst in stream S.
- $Time(S_{i,j})$: Processing time of the j-th event in the i-th burst in stream S.
- $Time(S_n)$: Processing time of the n-th event in stream S.
- $Min(S_{i,*})$: Minimum burst processing time in stream S.
- $Min(S_{i,j})$: Minimum event processing time in the i-th burst in stream S.
- $Max(S_{i,*})$: Maximum burst processing time in stream S.
- $Max(S_{i,j})$: Maximum event processing time in the i-th burst in stream S.
- $Mean(S_{i,*})$: Average burst processing time in stream S.
- $Mean(S_{i,j})$: Average event processing time in the i-th burst in stream S.
- $std(S_{i,*})$: Standard deviation of burst processing time in stream S.
- $std(S_{i,j})$: Standard deviation of event processing time in the i-th burst in stream S.

- $Sum(S_{i,*})$: Sum of the processing time of all bursts in stream S.

In our tracing experiments we set a static distance in time between each burst injection. This is referred to as *burst distance* and is denoted by d_{ms}, where *ms* is specifies that the unit of time is milliseconds.

3.4 Tracing framework

This section covers the design of the tracing framework used in measuring the performance of event processing in Siddhi. First, we give a brief overview of our requirements for an effective monitor system. Then, we present the design and implementation of our tracing utility, and finally, we evaluate our design.

3.4.1 Requirements

The monitoring system employed to measure the performance of event processing should satisfy the following requirements: (1) flexibility; (2) ease of use; (3) minimal overhead; and (4) generation of easily parsable output. Flexibility and ease of use refers to the ability to place tracepoints anywhere within the source code of the SUT, without having to modify the build process or write complicated trace scripts.

Overhead pertains to the monitor's consumption of shared system resources, such as CPU and memory, which perturbs system operation[21]. Thus, minimizing monitor overhead is crucial in getting representative execution timings of the SUT. Parsability is an important attribute for fast and straightforward processing of trace files in preparation for subsequent analysis. Fulfilling these requirements will facilitate faster and accurate analysis of the targeted system.

3.4.2 Tracing utility

Using the monitor terminology from [21], we classify our tracing utility as a *software-event-driven-batch* monitor. As a software monitor, it oversees the SUT by inserting function calls to the tracing utility in the source code. It operates as an event-driven monitor by measuring the span from the moment an event tuple is injected into the process chain to when the event tuple's concludes. It functions as a batch monitor because the trace is saved on disk for later analysis using a separate analysis program.

The tracing utility is implemented as a global and static library in the `core` module of Siddhi Core Libraries[33]. Its CPU overhead is from the insertion of tracepoints, and its a memory overhead stems from storing the tracepoints in memory. The tracing utility can measure event processing with up to nanosecond precision. It writes the tracepoints to a file after all tracepoints from the stream have been collected.

Listing 3.7 presents the complete implementation of the tracing utility. A tuple data structure is used to represent a tracepoint, and a stack

is used to store the tracepoints in memory. We opted to use a stack because it allows for tracepoint insertion in constant time. The tuple comprises three elements: a sequence number, a string representation of the tracepoint location, and a timestamp indicating the time of the tracepoint. Tracepoints are placed in the source code by calling the `TraceUtil.addTrace()` function.

```
package io.siddhi.core.util;

import java.util.Stack;
import org.javatuples.Triplet;

public class TraceUtil
{
    public static Stack<Triplet<Long, String, Long>> traceStack =
        new Stack<Triplet<Long, String, Long>>();

    public static void addTrace(Long id, Long time)
    {
        traceStack.push
            (new Triplet<Long, String, Long>
             (id,
              Thread.currentThread().getStackTrace()[2].toString(),
              time));
    }
}
```

Listing 3.7: Tracing utility

Listing 3.8 showcases the format of the resulting trace file from the tracing utility. The trace file adopts the widely used comma-separated values (CSV) format, which facilitates easy parsing for subsequent processing and analysis. The first value is the event stream sequence number, i.e., the order in which the event tuple was injected into Siddhi. The second value denotes the precise location of the tracepoint in the source code (function, file, and file number). The third value is the time when the tracepoint was called.

```
<SequenceNumber>,<Function(<FileName>:<LineNumber>)>,<Timestamp>
```

Listing 3.8: The format of a trace file generated from instrumentation using the tracing framework.

3.4.3 Evaluation

Our tracing utility largely meets the requirements we have set for a monitoring system. Regarding flexibility and ease of use, the tracing utility allows us to place a tracepoint anywhere in the source code with a single function call. Moreover, because Siddhi is built with Maven and the tracing utility is implemented as a global static class in Siddhi, it is automatically included in the build process. The tracing utility also generates a trace file in a format that is easily parsable, making it swift and straightforward to process.

In order to evaluate the overhead of the tracing utility, we implemented a monitor that was solely focused on performance, as seen in Listing 3.9. This monitor measures event processing using a timestamp that is automatically added to the event tuple upon injection into Siddhi. When the event is received in the stream callback, the event processing time is measured and then written to a file. This approach significantly reduces memory consumption and halves the number of function calls needed for measuring event processing time.

```
siddhiAppRuntime.addCallback("OutputStream", new StreamCallback() {
        @Override
        public void receive(Event[] events)
        {
            long timestamp = System.nanoTime();
            long id = (long) events[0].getTimestamp();
            if (id == -1) {
                siddhiAppRuntime.shutdown();
            } else {
                fileLogger.trace(events[0].toString()+"␣"+timestamp);
            }
        }
    });
```

Listing 3.9: Timestamp-enabled trace monitor

For our experiment, we instrumented a simple query and processed 50 bursts containing 1000 events each, separated by sleep intervals of 1000 ms. This type of workload is the one we are using to measure the impact of caching on event processing. We conducted the workload measurement three times: (1) with the timestamp-enabled trace monitor, which measures from the injection of an event to it being received in the callback; (2) with the tracing utility using two tracepoints to measure the same path as (1); and (3) with the tracing utility measuring the same path as (1) and (2), but also other aspects of the event process path, seven tracepoints in total. As Figure 3.3 shows, the tracing utility introduces a noticeable overhead, and the overhead increases with the number of tracepoints.

Despite the significant overhead of our tracing utility, it provides the flexibility, ease of use, and parsable trace output that we need to investigate performance non-determinism of event processing in Siddhi. However, we need to be aware of the increasing timing that accompanies an increasing number of tracepoints.

3.5 Summary

This chapter focuses on the methodology employed in the study, which involves an iterative process of instrumentation, human investigation of traces, assessment of performance non-determinism, and system tuning. The systematic approach of the methodology is a valuable tool for managing the complexity of the SUT and the execution environment,

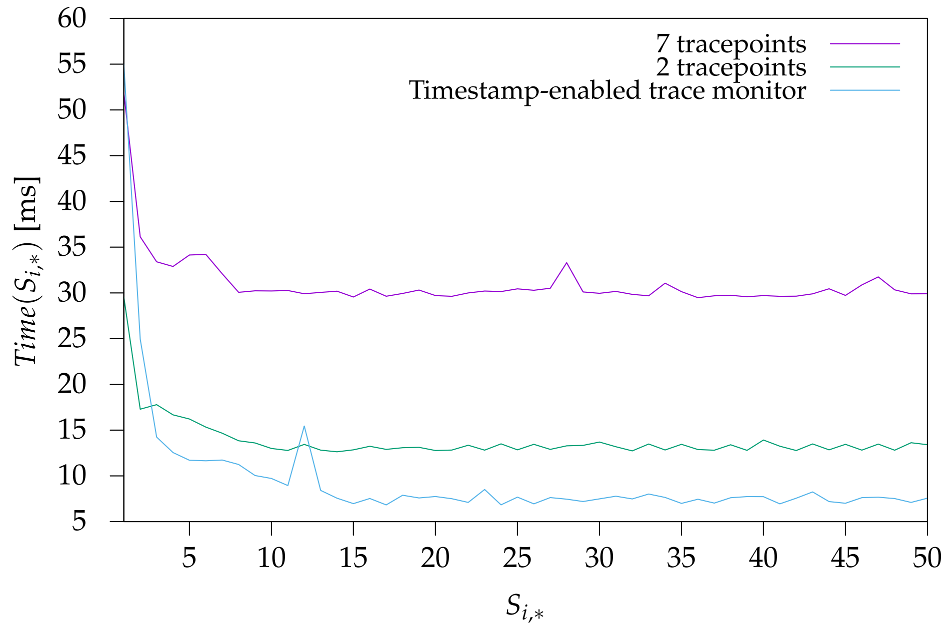

Figure 3.3: Evaluation and comparison of monitor overhead when using 7 and 2 tracepoints to capture event processing timing, in addition to a timestamp-enabled trace monitor.

providing clear steps and procedures to isolate the target behavior for modeling purposes.

The SUT in this study is a simple Java application utilizing the Siddhi CEP engine. The application is responsible for generating and processing a stream of events with different characteristics, such as bursty and non-bursty traffic. To measure the event processing time, the application has been instrumented with tracepoints before and after the injection of events into Siddhi.

A simple and flexible tracing framework has been designed, which is easy to use and deploy, while ensuring efficient utilization of CPU and memory resources. Additionally, metrics have been defined to enhance the reader's understanding and facilitate analysis of the experimental results.

Chapter 4

Analysis

This chapter presents the exploratory analysis of performance non-determinism in the Siddhi stream processing engine. Our approach involves an iterative process consisting of five steps. Firstly, we state our hypothesis. Secondly, we provide essential background information to support our hypothesis. Thirdly, we present an experimental design to test our hypothesis. Fourthly, we analyze the traces generated from our experiment to confirm or reject our hypothesis. Finally, we present a plan for future work based on the results obtained from our analysis.

4.1 Processing event bursts

H1 Performance non-determinism in Siddhi stream processing is caused by *caching*.

Caching is often believed to be a major source of performance non-determinism[19]. To identify the presence of caching in a black-box system, we subject the SUT to traffic consisting of short bursts of events with idle time between each burst. This idle time should cause the cache to replace the data associated with the previous burst with new data until the next incoming burst arrives. We expect as a result, the system exhibiting a pattern of *cache misses* followed by a sequence of *cache hits*. To better understand the relationship between bursty traffic and caching, we provide a brief summary of caching in Section 4.1.1.

4.1.1 Cache memory

Caching plays a vital role in modern computer systems as it enhances memory access time, which is often a performance bottleneck. To achieve better performance at lower costs, a hierarchy of memory layers is constructed with top layers having higher speed but smaller capacity. This hierarchy starts with CPU registers, followed by cache memory and main memory, which is divided into *cache lines*. The most heavily used cache lines are stored in a high-speed cache located inside or near the CPU.

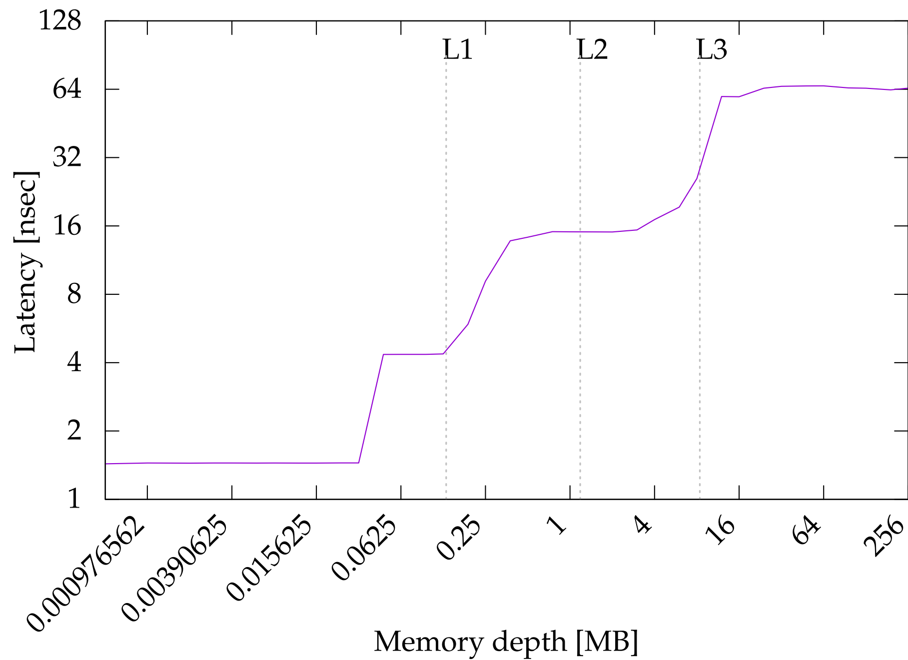

Figure 4.1: Comparison of access latency for L1, L2 and L3 cache, as well as RAM. Each vertical line denote the end of a storage unit. The graph has been generated based on `lm_mem_rd` microbenchmark from `lmbench`[32].

Accessing the main memory takes substantially longer time than accessing the cache, but not all caches are equal. Modern CPUs computers have three levels of cache, where the caches located closer to the CPU are smaller and feature faster accessing times. As a result, timing of cache hits depends on the cache level in which the memory word is stored, see Figure 4.1. L1 is the fastest but smallest cache and L3 is the slowest but largest cache. The L1 cache is always inside the CPU and can hold a few hundred kilobytes, while the L3 cache is located near the CPU and can store several megabytes of recently used words.

When a memory word needs to be accessed, the cache hardware verifies whether the requested line is present in the cache. If it is, the request is fulfilled from the cache, and no memory request is sent over the bus to the main memory, this is known as a *cache hit*. Cache hits typically require two to three clock cycles. In contrast, *cache misses* have to go to memory, resulting in a considerable time penalty, often at least one order of magnitude more expensive than accessing even the last level of the cache. The cache line containing the word is temporarily inserted into the cache for faster accesses in the future[40, 14, 19].

4.1.2 Experimental design

The SUT is executed on a Dell Precision 5520 workstation powered by a 2.9 GHz Intel Core i7-7820HQ Quad-Core processor (with 2 threads per core),

which can be overclocked up to 3.9 GHz using Intel Turbo Boost 2.0. The CPU has 128 KiB of data cache (L2d) and instruction cache (L1i), 1 MiB of L2 cache, and 8 MiB of L3 cache. The machine has 32 GB of memory and 1 TB of SSD memory, and runs Debian Linux 11 64-bit with kernel version 5.10.0. The SUT employs Siddhi Core Libraries version 5.1.20, given that it was the most recent release at the beginning of this thesis. The HotSpot JVM from OpenJDK has been selected due to its status as one of the most widely used JVM implementations to date[36].

To address Hypothesis H1, we instrument the processing behavior of a simple stream query and exposing it to both bursty and non-bursty traffic. The query, as shown in Listing 4.1, contains no filters or aggregates. It processes event tuples from a defined stream `StockStream`, which contains a set of four typed attributes: id, symbol, price, and volume. The `select` statement selects all four input attributes as query output attributes for `OutputStream`. The simple `select` query is to understand the basic functionality and behavior of the system before diving into more complex queries. It also allows us to establish a strong foundation and follow an incremental approach to model the targeted system.

```
from StockStream
select id, symbol, price, volume
insert into OutputStream
```

Listing 4.1: The instrumented event query.

The selected workload is an event stream consisting of 50 bursts of 1000 events each, referred to as S_{50}^{1000}. We define a burst as the injection of a given number of events at the maximum rate, and bursty traffic refers to scenarios where the burst distance is greater than zero. In contrast, non-bursty traffic occurs when the burst distance is zero. For S_{50}^{1000}, non-bursty traffic simply refers to a long burst of 50,000 events. We opt to use a workload of 50 bursts to ensure that we have an adequate number of data points for bursts, even when excluding the JVM warm-up. We select a burst size of 1000 events, as it is a reasonable number of events that can be considered a short burst and is large enough to capture cache misses and hits in a confined manner.

We instrument the query by placing a tracepoint immediately before and after the injection of an event into Siddhi, as illustrated in Listing 3.5. We also place tracepoints at several other locations within the source code of the Siddhi core engine to measure the time between different locations in the processing path. In total, we add seven tracepoints to the code, which capture every event and, as explained in Section 3.4.3, results in a significant overhead.

We execute the application default values and conduct each experiment only once[1] for each burst distance. Our primary variable of interest is

[1]It is not necessary to run the experiment several times, because the number of bursts in the sequence provides enough observations of the targeted behavior to verify the validity and reliability of the results.

the burst distance, which we varied from 0 to 1000 ms in steps of 100 ms with each experiment to evaluate the impact of burst distance on event processing. Here, 0 d_{ms} denotes non-bursty traffic. We expect to observe the first events in a burst to have substantially higher processing time than the subsequent events in the burst, which will demonstrate the presence of cache-like behavior in the SUT.

4.1.3 Analysis

Table 4.1 presents statistics of burst processing time for the different burst distances. The analysis reveals that processing the event stream as a constant stream, instead of dividing it into smaller bursts, leads to higher event processing throughput. When burst distances are between 100 and 200 ms, the highest processing times are observed, which are approximately 150% higher than those for burst distances longer than 300 ms. Additionally, the burst distance of 300 ms leads to high processing times and a greater variation in burst processing times. The observations of elevated processing time is discussed in detail in Section 4.4.

Table 4.1: Statistics of burst processing times from processing S_{50}^{1000}, including JVM and default settings. Time measurements are in milliseconds.

d_{ms}	$Mean(S_{i,*})$	$Min(S_{i,*})$	$Max(S_{i,*})$	$Sum(S_{i,*})$	$std(S_{i,*})$
0	29.73	24.54	66.19	1486.54	7.95
100	62.27	27.84	104.96	3113.38	10.98
200	64.35	28.18	103.17	3217.45	11.06
300	48.66	27.32	101.81	2432.84	18.98
400	33.35	29.17	66.25	1667.29	8.38
500	32.10	28.62	66.38	1604.99	7.22
600	32.22	28.55	62.11	1611.04	7.35
700	32.59	28.44	69.35	1629.27	8.87
800	33.44	28.47	69.08	1671.95	9.71
900	32.24	28.58	69.06	1611.95	8.36
1000	32.31	28.74	67.93	1615.39	7.50

In examining the effect burst distance length, we observe that each distance yield distinct processing behaviors. Burst distances greater than 300 ms demonstrate similar processing behaviors, as indicated by the line plot in Figure 4.2, whereas burst distances between 100 and 200 ms show similarities in their burst processing behaviors, as depicted in Figure 4.3. The high standard deviation observed when processing a stream with 300 ms burst distances is also shown in Figure 4.3. As demonstrated in Figure 4.4, the single long burst reaches a higher throughput compared to the event streams with a burst distance of 1000 ms. In summary, these findings demonstrate the emergence of different processing behaviors as a function of burst distances.

The processing times for the initial bursts in the sequence are significantly longer than those of the subsequent bursts. We postulate that this

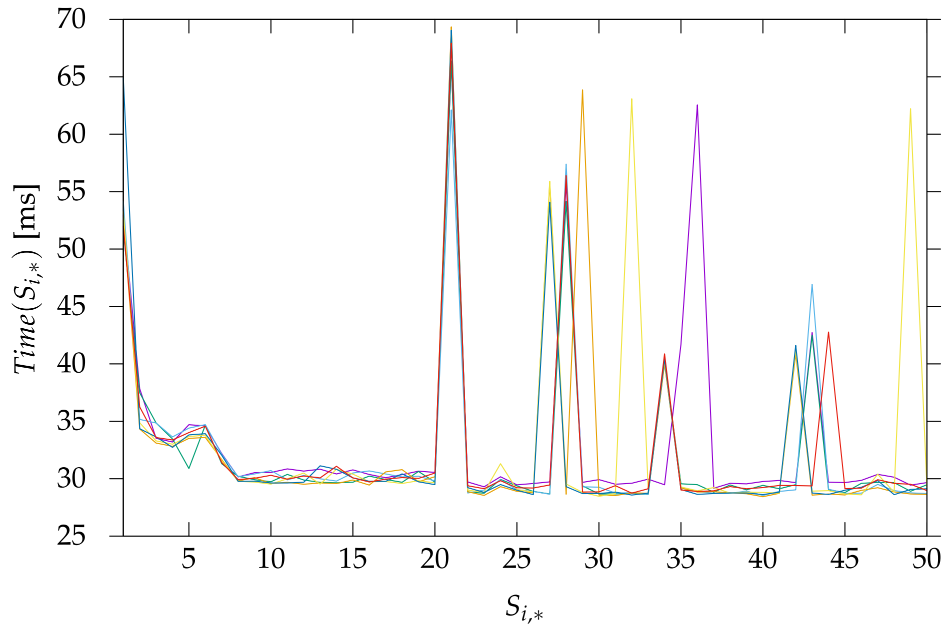

Figure 4.2: Burst processing of burst distances longer than 300 ms.

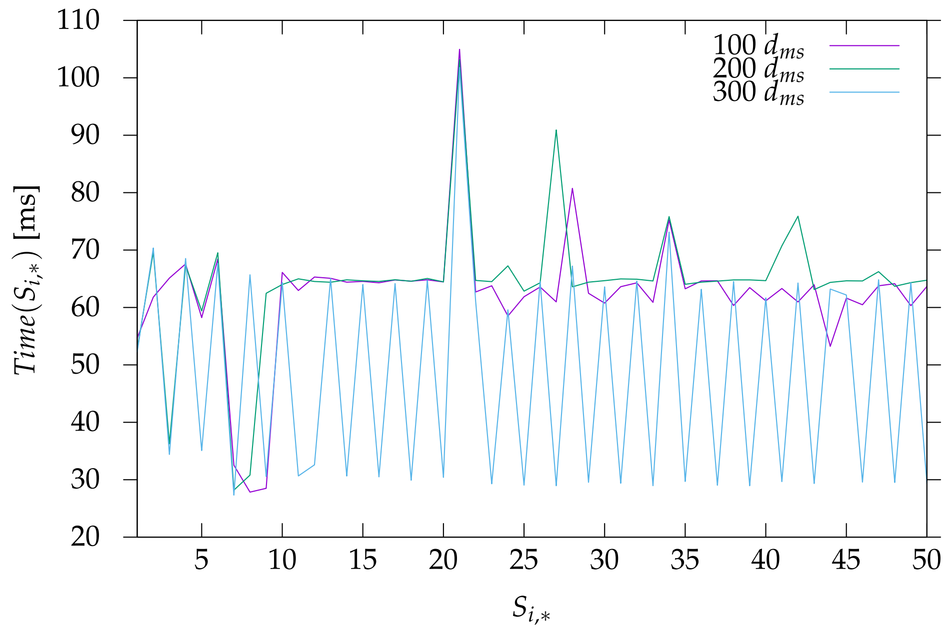

Figure 4.3: Burst processing of burst distances between 100 and 300 ms.

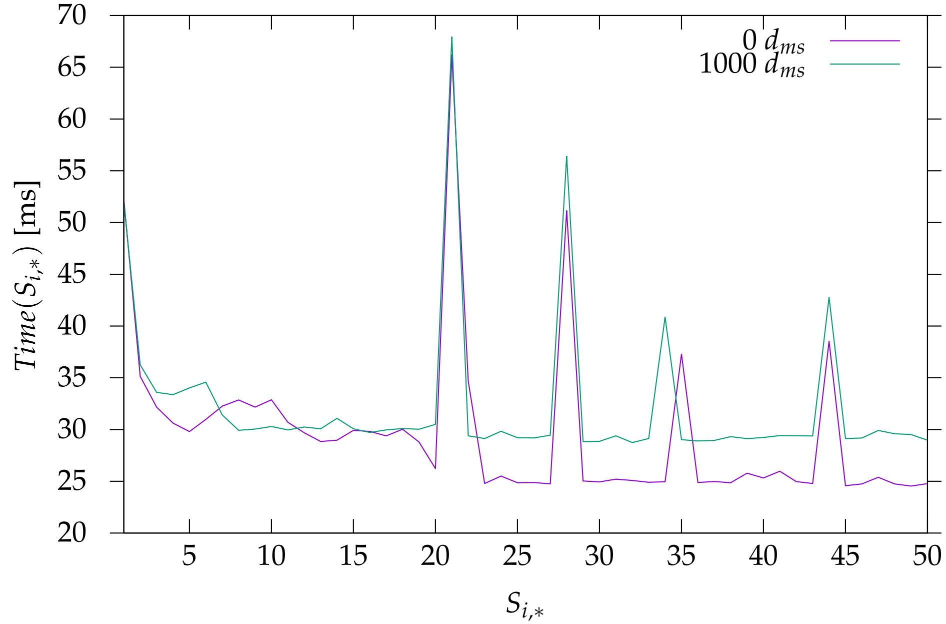

Figure 4.4: Comparison of burst processing for a constant stream and a bursty stream with 1000 d_{ms}.

phenomenon is due to the fact that the JVM has not yet been warmed up[29]. A detailed analysis of this observation is presented in Section 4.3. In addition, we note the existence of common processing time spikes around $S_{20,*}$, $S_{30,*}$, and $S_{40,*}$ for each burst distance. We hypothesize that this behavior is caused by the program reaching the limit of the allocated heap memory and the JVM having to perform clean-up of unused memory before the program can continue, or the JVM deciding to compile sections of the program to native code[5]. We provide further discussion on this topic in Section 4.2.

4.1.4 Head-tail comparison analysis

Upon closer examination of the event processing times, we have identified a processing pattern in which the initial events in a burst take significantly longer to process than the subsequent events. This behavior is believed to be attributed to caching within the system stack as stated in Hypothesis H1. To investigate this behavior, we conduct a *head-tail comparison analysis*. Head-tail comparison analysis entails comparing the average sum of the processing time for the first N events in a burst with that of the last N events in the burst for a given stream S. The average sum of the processing time of the first and last N events is denoted as A and B, respectively.

The results from the head-tail comparison analysis on the bursty stream S_{50}^{1000}, as presented in Table 4.2, suggest that the head of the burst takes

significantly more time to process than the tail, and there exists a negative correlation between the ratio in processing time of the head and tail and the number of events N. In contrast, the head-tail comparison analysis of the constant stream S_{50}^{1000} does not exhibit this behavior, as shown in Table 4.3.

The observation of the first few events in a burst taking longer to process than the remaining events, strongly indicates the presence of caching in our black-box system. However, it is not sufficient to confirm Hypothesis H1. We must isolate the asymmetric event processing in bursts by identifying and removing factors that cause undefined behavior in bursts, such as spikes and elevated processing times, which can obscure a potential caching model. The result is a well-tuned system for cache modeling purposes.

Table 4.2: Head-tail analysis of a bursty stream S_{50}^{1000} with 1000 $d_m s$. Time measurements are in microseconds.

N	A	B	$A-B$	A/B
1	226	2	224	113.00
5	269	10	259	26.90
10	313	21	292	14.90
20	401	46	355	8.72
30	558	69	489	8.09
40	646	90	556	7.18
50	734	110	624	6.67
100	1246	216	1030	5.77
500	4225	1115	3110	3.79

Table 4.3: Head-tail analysis of a constant stream S_{50}^{1000}. Time measurements are in microseconds.

N	A	B	$A-B$	A/B
1	56	159	-103	0.35
5	70	171	-101	0.41
10	83	184	-101	0.45
20	111	208	-97	0.53
30	142	234	-92	0.61
40	178	263	-85	0.68
50	213	288	-75	0.74
100	363	733	-370	0.50
500	3374	3301	73	1.02

4.1.5 Distribution analysis

We look at the distributions of processing times of single events and bursts to determine whether different tasks were executed during the processing of an event or in between events. A unimodal distribution indicates a high level of performance determinism, while a multimodal distribution

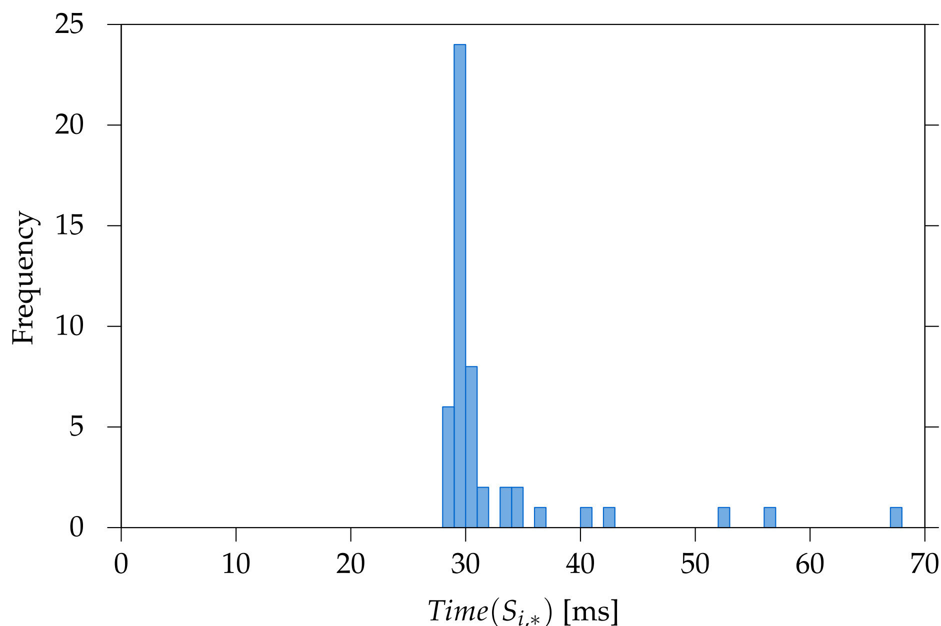

Figure 4.5: Distribution of burst processing times in S_{50}^{1000} with 1000 d_{ms}.

with distant modes, infers that performance non-determinism is present during the execution. The distribution will also reveal potential outliers. This technique is similar to one of the technique used in [24] to validate instrumentation of communication software.

According to Figure 4.5, out of 50 bursts consisting of 1000 events each, roughly 80 percent of the bursts take approximately 30 ms to process. Furthermore, we observe a few outliers between 40 and 70 ms, in which some of these outliers occur early in the event stream. Table 4.4 presents the distribution of event processing times. Approximately 92% of the event processing times fall within 30 µs. Similarly to the distribution of burst processing times, the event processing time has a few outliers, see Figure 4.6 for more details. We investigate these outliers further in Section 4.2.

Figure 4.7 provides a more detailed distribution of the first two bins presented in Figure 4.6. It is evident from this figure that the processing time for an event in a burst falls within the range of approximately 25 to 50 µs. This observation is also supported by the results presented in Table 4.4. The unimodal distribution of the processing times suggests that simple but accurate models of event processing times in bursts can be developed by confining the analysis within this range.

Table 4.4: The percentage of events in S_{50}^{1000} with 1000 d_{ms} that fall within a given time different boundaries. Time measurements are in microseconds.

$Time(S_n)$	%
< 10	0.000
< 20	0.000
< 30	91.676
< 40	93.566
< 50	94.166
< 100	98.938
< 150	99.614
< 200	99.948
< 250	99.968
< 300	99.974
< 350	99.978
< 400	99.980
< 500	99.982
< 700	99.986
< 750	99.992
< 6750	99.994
< 11750	99.996
< 27050	99.998
< 31350	100.000

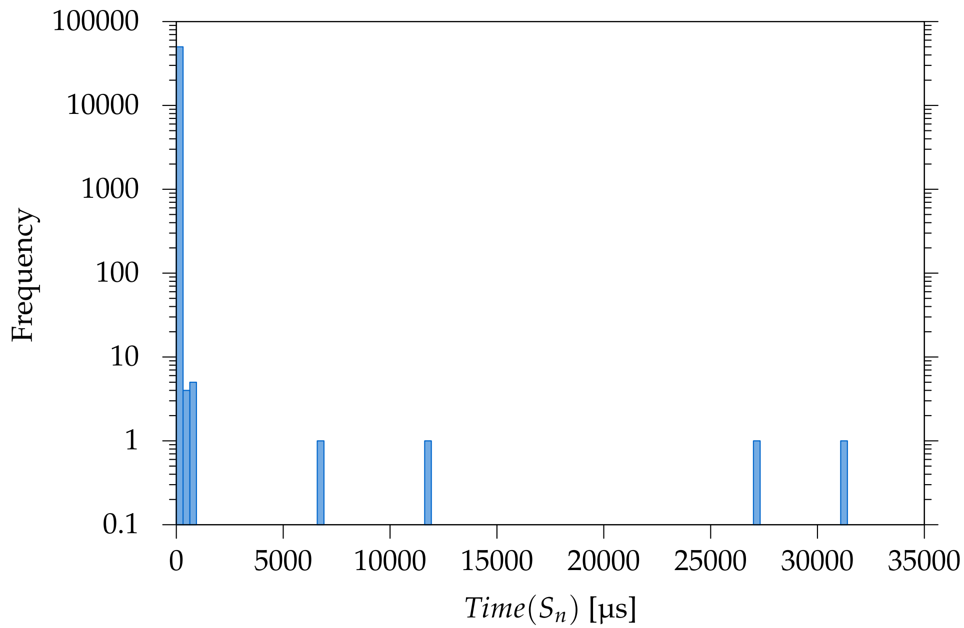

Figure 4.6: Distribution of event processing times in S_{50}^{1000} with 1000 d_{ms}.

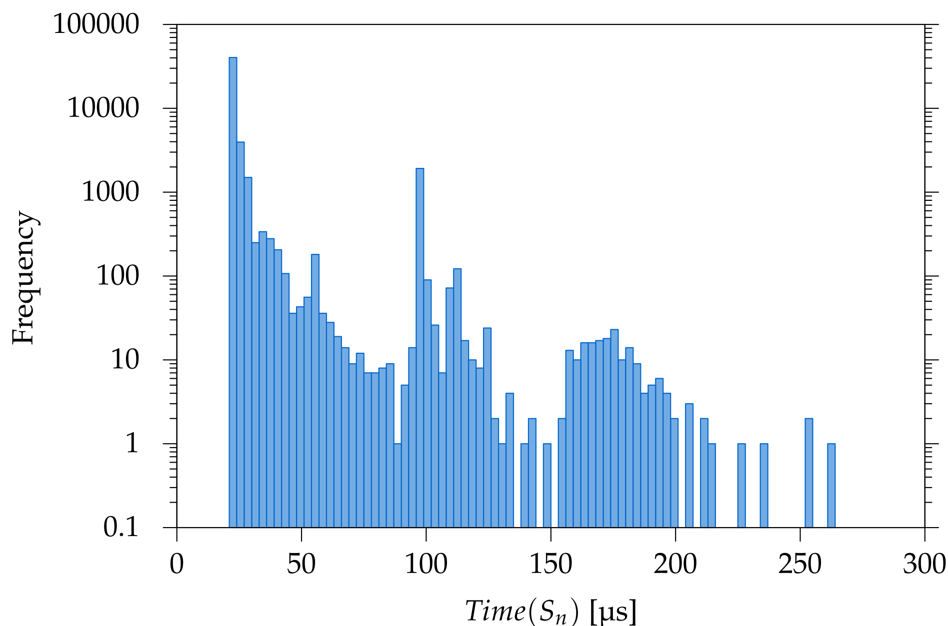

Figure 4.7: Distribution of event processing times shorter than 300 µs in S_{50}^{1000} with 1000 d_{ms}.

4.2 Spikes in burst processing time

We observe four spikes in burst processing time when processing S_{50}^{1000} with 1000 d_{ms}. The spikes are presented in Figure 4.8 and Table 4.5. Table 4.5 displays in descending order the bursts that exhibit the longest processing time. Each of these bursts coincidentally contains an event with one of the highest processing times throughout the stream, with the exception of $S_{44,*}$. The average burst processing time for this workload is 32.31 ms, as demonstrated in Table 4.1. This indicates that these abnormal spikes in event processing time are the sole cause of the elevated burst processing time for the associated burst. However, the initial bursts in the sequence ($S_{1-7,*}$) are not exclusively impacted by a single event but rather by the high processing time of events in general. We suspect that the higher processing time of the first bursts is caused by the JVM warm-up process[29]. The very first event in the stream will always have a high processing time due to *class initialization*. Class initialization is the process by which a class object is loaded, linked, and initialized when used for the first time in a Java program[20].

There are several possible factors that can cause spikes in event processing time. One such factor is the activation of the garbage collector, which halts event processing until garbage collection is completed. Another factor is the Just-In-Time (JIT) compiler in the JVM, which may decide to compile sections of the program that have run "hot" to native code.

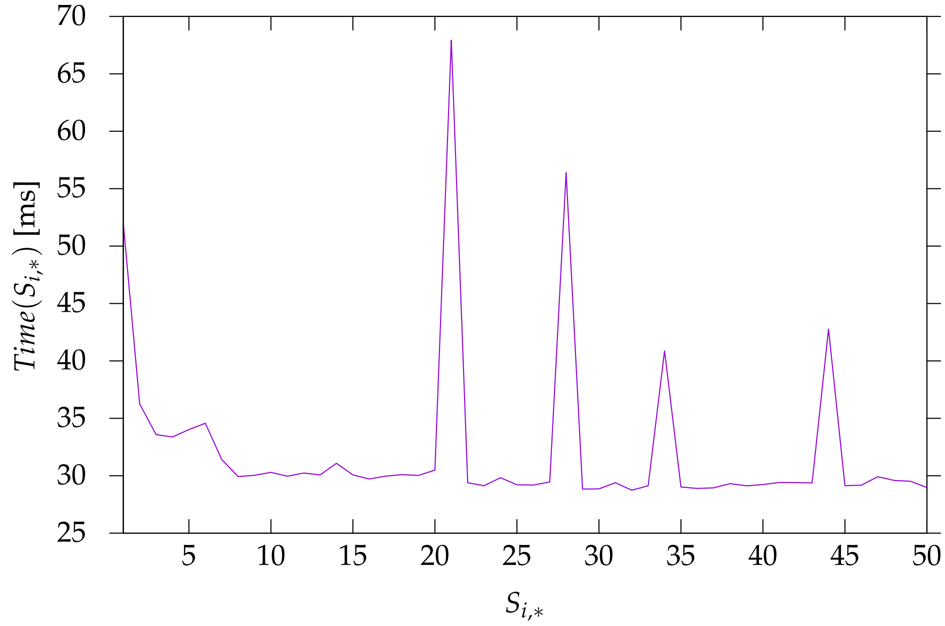

Figure 4.8: Burst processing in S_{50}^{1000} with 1000 ms.

Table 4.5: Statistics of burst and event processing time per burst for the ten bursts with the highest processing time in S_{50}^{1000} with 1000 d_{ms}. Time measurements are in milliseconds.

$S_{i,*}$	$Time(S_{i,*})$	$Max(S_{i,j})$	$Mean(S_{i,j})$	$std(S_{i,j})$
21	67.93	31.30	0.065	0.989
28	56.39	27.03	0.054	0.854
1	52.00	6.71	0.049	0.212
44	42.78	0.17	0.027	0.017
34	40.87	11.71	0.038	0.370
2	36.25	0.37	0.033	0.024
6	34.58	0.21	0.032	0.021
5	34.02	0.21	0.031	0.023
3	33.59	0.21	0.031	0.023
4	33.37	0.20	0.031	0.021
7	31.41	0.74	0.029	0.031

The compilation process, although running in the background, requires shared resources, such as main memory and cache, and thereby affecting performance, resulting in high processing time of events[29]. These factors can lead to unexpected variations in event processing time, which can impact the overall performance of the system. Therefore, understanding and mitigating the effects of these factors may be important in ensuring consistent and predictable event processing performance which we can build our model on.

However, if these spikes in event processing time are periodical or predictable, we can potentially save time and research effort into identifying their origin, and instead focus on modeling event processing.

We start the investigation of the spikes in event processing with Hypothesis H2:

H2 Event processing spikes are periodical or predictable.

4.2.1 Experimental design

To test Hypothesis H2, we extended the experiment by increasing the number of bursts for a fixed burst distance of 1000 ms. In a sequence of 50 bursts, we observed four significant spikes apart from the spike in the beginning of the stream. By extending the sequence to 200 bursts (4x), we expect that the spikes are going to be observed 16 times for S_{200}^{1000}. Furthermore, we can test Hypothesis H2 by halving the burst size to 500 events or doubling it to 2,000 events. For these burst sizes, we anticipate that the spikes are going to be observed eight and 32 spikes respectively in a sequence of 200 bursts. This will demonstrate that the event spikes are periodical or predictable, confirming Hypothesis H2.

4.2.2 Analysis

Figure 4.9 presents the extended stream of 200 bursts with 1000 ms burst distance for different burst sizes. The line plot illustrates the relationship between burst processing time and burst sizes, revealing a more frequent occurrence of spikes in burst processing time with larger burst sizes. This observation suggests that the occurrence of spikes is contingent on the quantity of processed events rather than the duration of the system's uptime.

The event distribution from the different burst sizes, as illustraded in Figure 4.10, 4.11 and 4.12, mark the boundary between *typical* event processing times and processing *spikes*. We can classify a processing spike when the processing time of an event exceeds 4 ms. Note that the boundary may be different on different hardware or if we use fewer tracepoints. With this defined boundary and not including the first event, we observe 8 event spikes for burst size of 500 events, 19 event spikes for burst size of 1000 events, and 93 event spikes for 2000 events. These results show that a correlation between spikes in event processing time, the number of

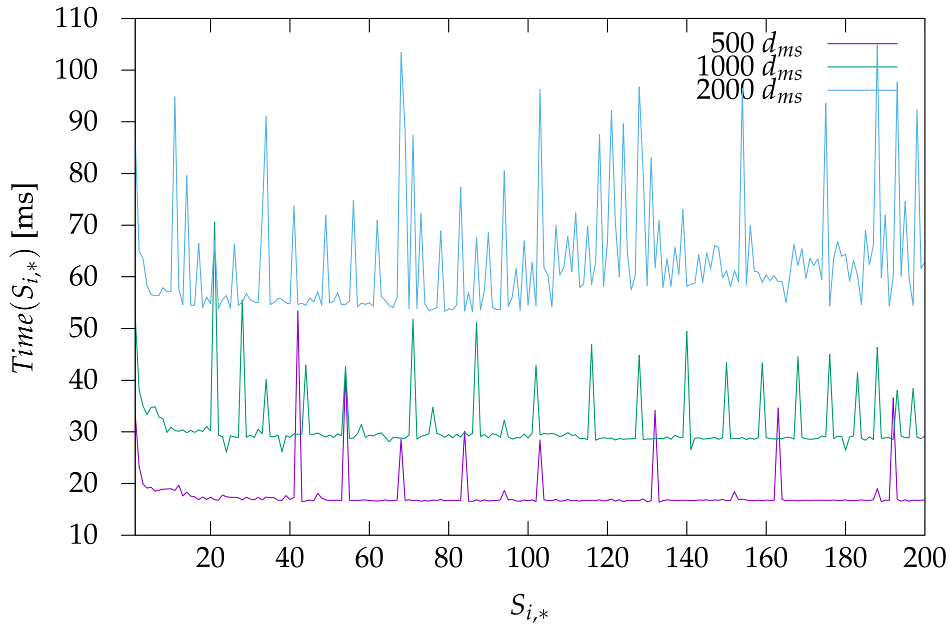

Figure 4.9: Comparison of burst processing behavior for three different burst sizes: 500, 1000 and 2000 events.

processed event and wall clock time is not to be found, thereby rejecting Hypothesis H2 forcing us to investigate event spikes further and identify their source.

Our tracing utility, which stores all tracepoints in memory prior writing them to file, or possible software caches utilized within the Siddhi engine, may activate the garbage collector. This would cause significant delays in processing time due to the *stop-the-world* pauses that occur during the garbage collection process[15]. These delays may be characterized by spikes in the event processing time.

H2 Event processing spikes are caused by garbage collection.

4.2.3 Garbage collection in the HotSpot VM

Garbage collection is a crucial process in managing memory in software applications. It involves identifying and freeing up memory that is no longer in use by the application. As such, it plays a critical role in ensuring the smooth running of the application and preventing memory leaks.

Tuning the garbage collector is essential for optimizing the performance of a Java application, as the different collectors available offer varying performance characteristics. The four main garbage collectors are the serial collector, throughput (parallel) collector, concurrent (CMS) collector, and the Garbage-First (G1) collector[29]. The G1 collector is a generational

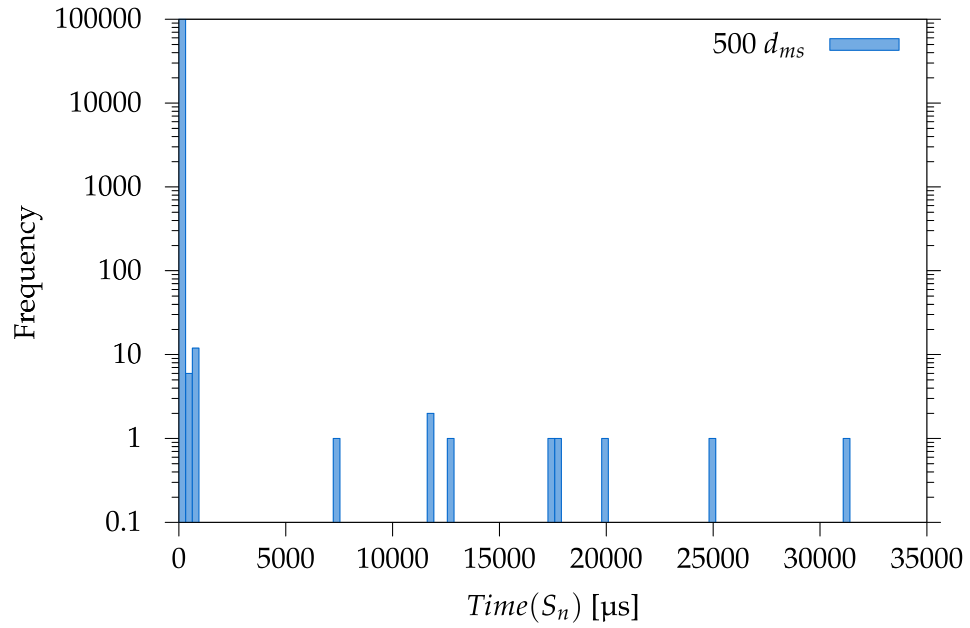

Figure 4.10: Distribution of event processing times in S_{200}^{500} with 1000 d_{ms}.

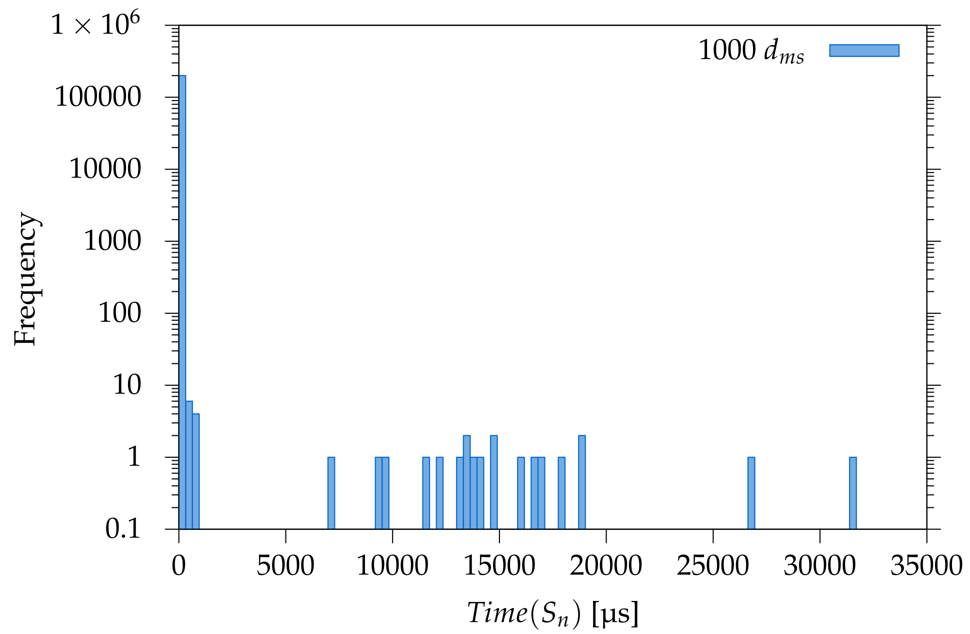

Figure 4.11: Distribution of event processing times in S_{200}^{1000} with 1000 d_{ms}.

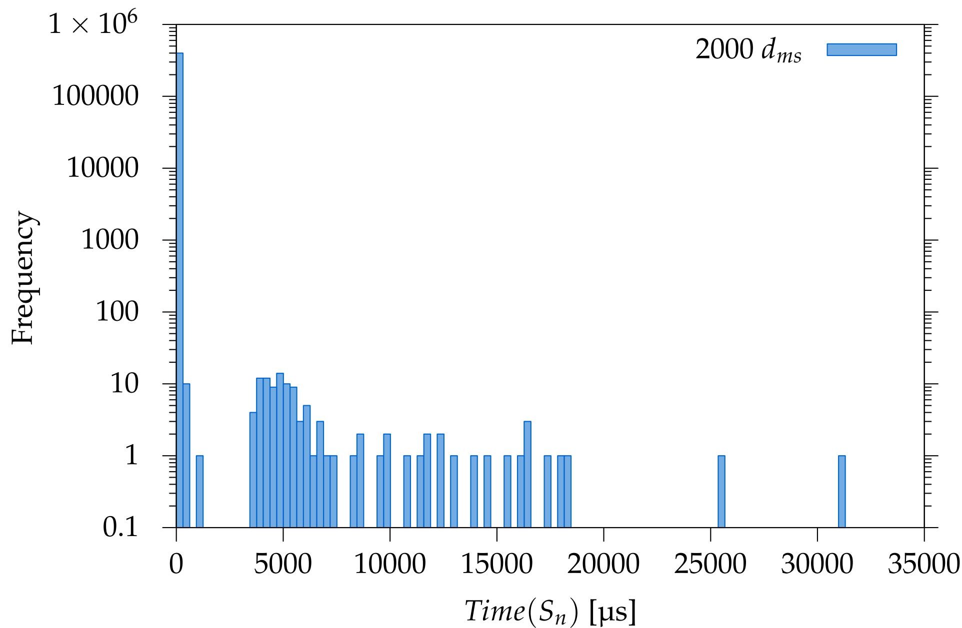

Figure 4.12: Distribution of event processing times in S_{200}^{2000} with 1000 d_{ms}.

garbage collector that is renowned for handling large heaps and reducing the frequency of *stop-the-world* pauses. This results in efficient and effective management of memory.

Stop-the-world pauses refer to situations where the garbage collector stops the application in order to recover space that is no longer in use. During these pauses, the entire application is temporarily suspended. Some operations, such as space-reclamation in the young generation, are performed in these pauses to improve overall throughput.

Generational garbage collectors divide the heap into multiple generations based on object age. The rationale behind this is that most objects die young, and so by segregating young objects and avoiding scanning older objects unnecessarily, the garbage collector can reduce the frequency and duration of garbage collection pauses. The young generation holds new objects that have just been created, while the old generation holds objects that have survived one or more garbage collection cycles.

The G1 collector's goal when performing space-reclamation to maximize the amount of space that is reclaimed in the old generation during a single garbage collection pause. It does so incrementally in steps and in parallel to keep stop-the-world pauses short for space-reclamation. A maximum pause-time goal is used by the garbage collector to limit the longest of these pauses to achieve predictability in performance.

While garbage collection is critical in managing memory, it can also be a

source of performance non-determinism. Since the garbage collector can run at any time, it can introduce unpredictable delays that can affect the overall performance of the application. Understanding this relationship is essential in tuning the garbage collection process for higher application performance or increase predictability.[30, 7]

4.2.4 Experimental design

To assess the activity of the garbage collector, we enable logging of garbage collection in the JVM by using the command-line option `-Xlog:gc:filename`. This sets the file, given by `filename`, to which information about the heap and garbage collection at each collection should be redirected for logging[31]. We process S_{50}^{1000} with 1000 d_{ms}, which should trigger garbage collections, considering the sheer amount tracepoints stored in memory. We expect to see logging information of each garbage collection occurring during the experiment, which can corroborate or confirm Hypothesis H2.

4.2.5 Analysis

According to the garbage collection log, our instance of the HotSpot JVM defaults to using the Garbage First (G1) collector and maintains a constant initial heap size of 504 megabytes throughout the experiment. Table 4.6 presents the parsed log file, demonstrating that the garbage collector is active multiple times during our experiment.

The garbage collection log provides detailed information about garbage collection events. Each event is identified by a timestamp, ID number, type, trigger, and other relevant parameters. The *timestamp* denotes the number of seconds the garbage collection occurs from the start time of the application, while the *ID number* indicates the order in which collections occur. The *type* specifies the type of garbage collection, such as a Pause Young (Normal) (PY(N)) collection, which collects data objects in the young generation. The *trigger* denotes the cause of the garbage collection, such as a G1 Evacuation Pause (G1EP), which is a garbage collection technique that reclaims space by collecting live objects in a selected memory area and copying them into new memory areas while compacting them in the process. The term "pause" in both the type and trigger refers to the fact that the garbage collector performs a stop-the-world event, in which it stops the application to recover memory that is no longer in use. The parsed log file, presented in Table 4.6, contains information about the memory usage in megabytes before and after each collection, denoted by the "Before" and "After" columns respectively, as well as the duration of each collection in milliseconds.[30]

As the garbage collection log does not provide information on whether the garbage collections occur in the background while the main application thread yields between bursts, it is insufficient to determine if these activities have an impact on the burst processing times. We suspect that the

Table 4.6: Garbage collection activities during processing of S_{50}^{1000} with 1000 d_{ms}. Timestamp is in seconds and Duration is in milliseconds.

Timestamp	ID	Type	Trigger	Before	After	Duration
0.355	0	PY(N)	G1EP	24	3	10.948
26.323	1	PY(N)	G1EP	299	41	31.087
32.530	2	PY(N)	G1EP	121	54	25.020
39.739	3	PY(N)	G1EP	142	68	10.563
47.989	4	PY(N)	G1EP	174	83	12.169
57.383	5	PY(N)	G1EP	209	102	13.103
58.003	6	PY(N)	G1EP	294	103	9.228
58.583	7	PY(N)	G1EP	321	102	1.107

tracepoints stored in memory during the experiment are the primary cause of the observed garbage collections. This is formulated as Hypothesis H3 and further investigated:

H3 Observed garbage collections are caused by the tracing utility memory overhead.

4.2.6 Experimental design

To test Hypothesis H3, we switch to the monitor with the smallest memory footprint. The monitor does not use the tracing utility to store tracepoints in memory, but instead stores tracing information in the event itself, see Section 3.4.3 for more details. We process the same workload of S_{50}^{1000} with 1000 d_{ms} as previously to see if a monitor with less memory trigger the same amount of garbage collections. Our expectation is that we are going to see fewer garbage collections in the resulting log, which confirms our hypothesis that the memory overhead of our tracing utility is invoking garbage collection.

4.2.7 Analysis

Analysis of the resulting garbage collection log reveals that only a single garbage collection occurs during S_{50}^{1000}, as demonstrated in Table 4.7. This finding confirms Hypothesis H3 and supports the conclusion that the number of garbage collections is proportional to the memory usage of the application. We hypothesize that allocating sufficient memory for the application would prevent tracepoints stored in memory from reaching a boundary that would trigger a garbage collection. Hypothesis H4 is introduced and further investigated:

H4 Garbage collection occurs less frequently when the heap size of the JVM is large.

Table 4.7: Garbage collection activities during processing of S_{50}^{1000} with 1000 d_{ms} using the timestamp-enabled trace monitor. Timestamp is in seconds and Duration is in milliseconds.

Timestamp	ID	Type	Trigger	Before	After	Duration
0.331	0	PY(N)	G1EP	24	3	9.840

4.2.8 Experimental design

To test Hypothesis H4, we switch back to monitor that uses the tracing utility, and increase the initial heap size of the JVM to maximum size of 4GB with the command-line option `-Xms4g`[31]. Then we process the workload of S_{50}^{1000} with 1000 d_{ms}. We expect to see fewer garbage collections during the experiment compared to the ergonomically selected heap size of the JVM. This demonstrates that we can mitigate the tracing utility's impact on application performance. If we can avoid garbage collections from occurring entirely during our measurement while using the tracing utility, we can shy from changing to a potentially more complex instrumentation method, saving time and effort to isolate caching behavior.

4.2.9 Analysis

Our analysis of the resulting garbage collection log, presented in Table 4.8, revealed that garbage collection still occurred five times during the experiment, which was only three times less than with the default initial heap size. This finding suggests that our hypothesis was incorrect, and that the frequency of garbage collections is not solely determined by the memory allocation of the application. However, the analysis of the garbage collection log revealed that the type of garbage collections during the experiment was predominantly "Pause Young (Normal)", referring to the young generation of the heap. To reduce the frequency of garbage collection, we hypothesize that dedicating a greater portion of the heap to the young generation would be beneficial. We continue our investigation with Hypothesis H5 formulated as follows:

H5 Garbage collection occurs less when the young generation of the heap is large.

Table 4.8: Garbage collection activities during processing of S_{50}^{1000} with 1000 d_{ms} running on the JVM with initial heap size of 4GB. Timestamp is in seconds and Duration is in milliseconds.

Timestamp	ID	Type	Trigger	Before	After	Duration
17.032	0	PY(N)	G1EP	204	27	23.357
31.461	1	PY(N)	G1EP	207	53	20.403
45.913	2	PY(N)	G1EP	231	78	21.222
57.538	3	PY(N)	G1EP	260	102	26.995
58.071	4	PY(N)	G1EP	284	102	9.182

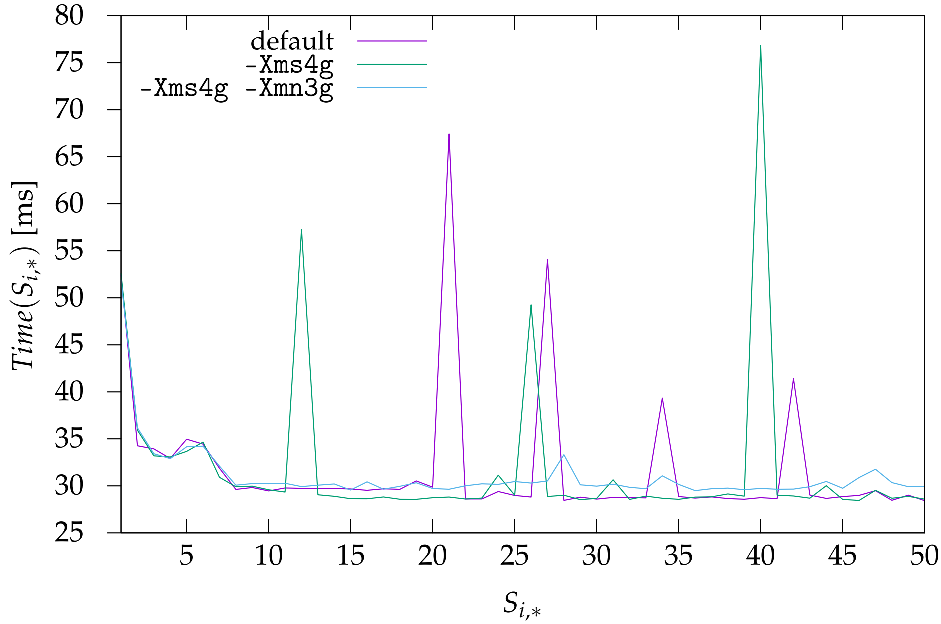

Figure 4.13: A comparison of the impact of heap settings on burst processing in S_{50}^{1000} with 1000 d_{ms}.

4.2.10 Experimental design

We dedicate 3GB of the heap to the young generation with the command-line option `-Xmn3g`. `-Xmn<size>` sets the initial and maximum `size` of the heap for the young generation[31]. We keep the initial heap size of 4GB and end up with a 3:1 ratio between the young and the old generation of the heap. To verify that the garbage collections occurring during our experiment are caused by the young generation filling up, we process again the workload of S_{50}^{1000} with 1000 d_{ms} once, but with the new heap settings. We expect zero garbage collections during the experiment because the sum of memory usage before each garbage collection for the given workload never exceeds 2GB, as illustrated in Table 4.6 and 4.8. This would provide us measurements of event processing without the involvement of garbage collection and how garbage collection affects event processing.

4.2.11 Analysis

We observed zero garbage collections during the experiment, confirming Hypothesis H5. The analysis of the burst processing time also reveals that the severe spikes in burst processing time were the result of garbage collections, as shown in Figure 4.13, concluding Hypothesis H2.

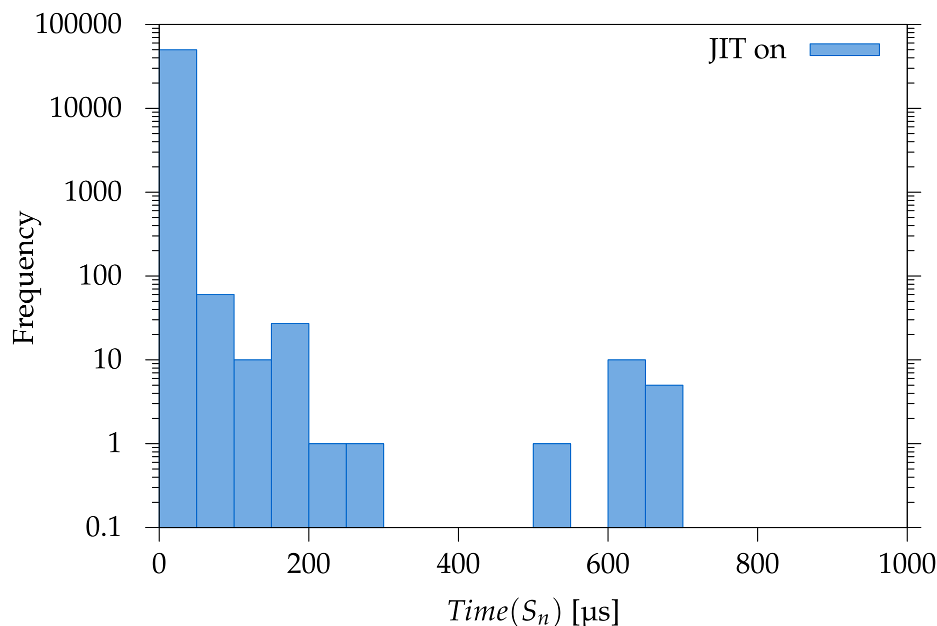

Figure 4.14: Distribution of event processing time in S^{50}_{1000} with 1000 d_{ms} and without garbage collection.

4.3 Bimodal distribution

Figure 4.14 presents the distribution of event processing times from the experiment in Section 4.2.10 where no garbage collection occurs during runtime. The results reveal a bimodal distribution for a logarithmic Y-scale. The first modal, between 0 and 400 µs, contains 99.7% of the events, while the second modal, between 500 and 700 µs, contains only 0.3% of the events. We have excluded the first event in the stream in the distribution plot, as it will always have abnormal processing time due to class initialization. Table 4.9 displays the events with processing times exceeding 500 µs and demonstrates that the majority of these events occur in bursts past $S_{15,*}$ in S^{1000}_{50}. This indicates that the majority of observed high event processing times are not caused by JIT compilation, as the JIT compilation process is at the start of the application lifetime. However, this process is known to take an unexpectedly long time[4].

We start the investigation of the bimodal distribution of event processing time with Hypothesis H6:

H6 Majority of high event processing time are caused by JIT compilation.

4.3.1 Java Virtual Machine (JVM) and JIT compilation

The Java Virtual Machine (JVM) is an abstract computing mechanism that forms the foundation of the Java platform[10]. Analogous to a tangible computing machine, the JVM operates by executing a set of instructions

Table 4.9: Event processing time exceeding 500 μs in S_{50}^{1000} when no garbage collection occurs during runtime. Time measurements are in microseconds.

i	j	$Time(S_{i,j})$
1	1	6734.27
26	118	678.04
11	241	675.43
35	259	661.99
24	159	656.72
28	77	650.67
13	161	640.96
44	196	636.19
48	114	629.21
39	177	623.34
46	155	622.49
37	218	615.58
19	38	614.27
17	79	610.56
6	83	607.76
15	120	606.80
20	344	501.34

and managing several memory regions during runtime. It serves as the integral component of the Java platform that enables it to operate independently of hardware and operating system dependencies. Thus, Java applications are capable of running seamlessly on any computing system equipped with a Java runtime environment[26].

The JVM runtime executes bytecode instructions generated through compilation of a Java program[10]. When bytecode instructions are interpreted for execution, the byte compiled Java program run innately slower compared to its C/C++ counterpart that is compiled to native machine code. The performance evaluation by Wentworth and Langan[47] shows that for a set of commonly known sorting algorithms, interpreted Java code has a performance degradation of 20.7 compared to C++.

To improve the performance of the JVM, just-in-time (JIT) compilation has been incorporated and is now a critical part of the JVM's runtime performance of Java applications. JIT compilation in the HotSpot VM works by compiling only the sections of code that are executed frequently, also known as "hot spots". The performance of an application depends primarily on how fast those hot spots are executed[29].

In the HotSpot VM, the JIT compiler comes in two flavors: client (C1) and server (C2). The C1 compiler is designed for use with GUI-based programs that prioritize fast startup times and generates lightly optimized code while having a small memory footprint. In contrast, the C2 compiler is a highly optimizing compiler and generates high-quality code at the cost of longer compilation. It is most often used with long running server-side

applications and perform better than the client compiler once the JVM has "warmed up".[18, 29]

The JVM warm-up refers to the time it takes for the JIT compiler to optimize code for peak performance[29]. The warm-up process begins in the initial phase of execution and continues until it reaches a *steady state of peak performance*. However, empirical research by Barret et al.[4] show that even widely studied, small and deterministic microbenchmarks, fail to consistently reach a steady state of peak performance on several common VMs, including the HotSpot VM, despite being executed in heavily controlled environments. Furthermore, the number of function calls and lines of code have a considerable influence on whether or not the microbenchmarks reach a steady state[23]. The warm-up period is characterized by slower execution times and higher memory usage, as the JVM perform expensive code verification, class loading, bytecode interpretation, profiling, and dynamic compilation[48].

Since the release of Java 8, the HotSpot VM uses a combination of the C1 and C2 compiler by default to compile Java bytecode to native machine code, a technique known as *tiered compilation*[18]. Tiered compilation systems compile methods multiple times using different compilers to optimize code at different levels of execution to provide a good trade-off between speed of compilation and generated code quality. There are five distinct levels of compilation, starting with interpreted code (level 0) and progressing through four levels of compiled code. The initial compilation typically performs only lightweight optimizations and instruments generated code to gather profiling information. The profiling information is used in a later compilation to generate high quality code[18]. The best case for performance is when methods are compiled consecutively from level 0 to level 4[29]. The generated machine code is cached to be used repetitively without recompilation[10]. Ensuring that the code cache is sized appropriately is essential for optimal performance[29].

Overall, the JIT compiler is a crucial factor in the performance of Java applications, and understanding how it works can help developers optimize their code for better performance.

4.3.2 Experimental design

Achieving accurate performance measurements of software systems through microbenchmarks requires careful consideration of the warm-up period for each microbenchmark. The warm-up period provides the compiler with an opportunity to finish producing optimal code, ensuring that the measured performance is representative of the steady-state of peak performance of the system. Therefore, identifying the end of the warm-up period is essential to create precise and effective models of long running systems.

We test Hypothesis H6 by processing S_{50}^{1000} with burst distances ranging from 400 ms and 1000 ms, as burst distances below 400 ms exhibit different

behavior. In each experiment, we vary the burst distance in steps of 100 ms. We set the heap size of the JVM to be 4GB with 3GB dedicated to the young generation to guarantee that no garbage collections occur during the experiment and enable logging of garbage collection for validation. We keep the default JIT compilation settings of tiered compilation running in the background.

To mitigate the overhead of instrumentation and focus on the event processing as a whole, we reduce the number of tracepoints in the source code to two per event. These tracepoints are placed before and after event injection, thus prioritizing the overall processing of events rather than the internal behavior of the Siddhi event processing chain. This approach mitigates the overhead of instrumentation and allows for a more efficient and accurate evaluation of the event processing performance.

In our experiments, we anticipate that the burst processing in S_{50}^{1000} for each burst distance will reach a stable state of peak performance. We define the point in the burst sequence in which all streams have reached steady-state as the end of the warm-up period.

4.3.3 Analysis

Figure 4.15 displays the burst processing time for S_{50}^{1000} with different burst distances, and demonstrate that for our complex event query, we can expect to reach a steady-state for peak performance after ten bursts, i.e. 10000 events, when burst distances are greater than 400 ms. This might not be the case for shorter burst distances, as the compilation threads run in the background. Table 4.9 shows that the events with the highest processing times when there are no garbage collections, occurs after the JVM warm-up, which indicate that the higher event processing are not caused by our JIT compilation. However, microbenchmarks often fail to consistently reach a steady-state of peak performance[4], which means that the JIT compiler might still compile in the background later than we anticipate. Due to high memory and CPU usage of compilation[48], it can cause delays in event processing.

4.3.4 Experimental design

We note that there is still a possibility that higher event processing times are caused by compilation overhead, as the JVM may run JIT compilation after the warm-up period. Considering that the HotSpot JVM uses tiered compilation, the probability is even higher since tiered compilation compiles the same section of code multiple times.

To evaluate the impact of compilation overhead on event processing times, we conduct experiments using the same workloads and JVM configuration as in our previous experiments, in addition to disabling the JIT compiler. The JIT compiler is disabled by the command-line option `-Xint`[31], causing the JVM to interpret the bytecode entirely. This approach allows

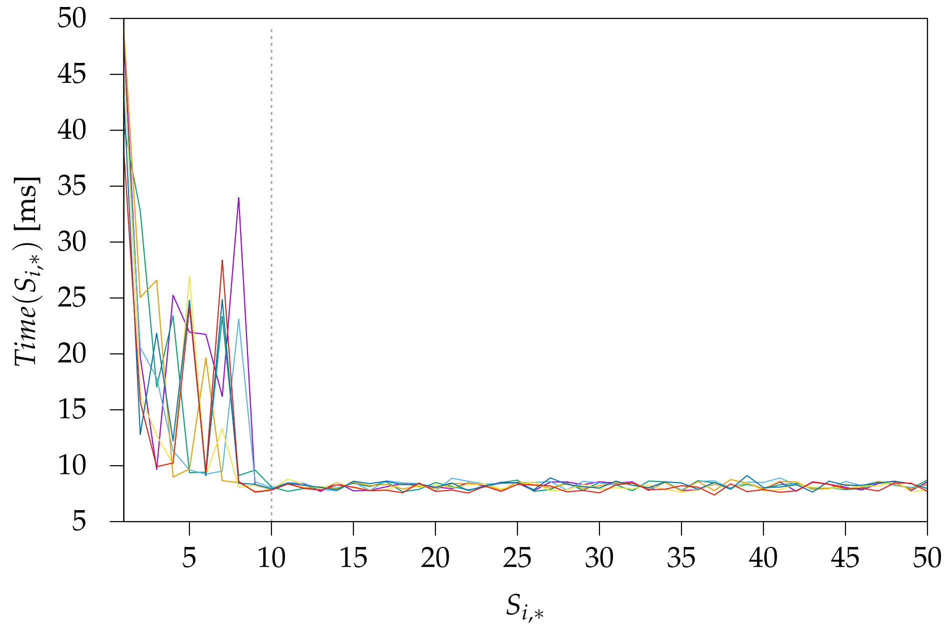

Figure 4.15: Burst processing times in S_{50}^{1000} with burst distances ranging from 400 ms to 1000 ms and without garbage collection. The dashed line in dark gray denotes the end of the JVM warm-up.

us to measure the performance of the system without the optimization benefits provided by the JIT compiler, which is useful for understanding the impact of compilation overhead on event processing times.

First and foremost, we expect to see behavior characterized as JVM warm-up to disappear which is a phenomenon caused by JIT compilation. This will confirm that the higher processing time of the first 10 bursts are caused by JIT compilation. Next, we predict a reduction of higher event processing time, which will be demonstrated by a unimodal distribution of event processing times. This will verify that the higher event processing times are attributed to JIT compilation.

4.3.5 Analysis

Figure 4.16 confirms that the higher processing times of $S_{1-10,*}$ are part of the JVM warm-up and caused by JIT compilation. The distribution of event processing time for S_{50}^{1000} with 1000 d_{ms}, depicted in Figure 4.17 reveals that many of the high event processing times were also caused by JIT compilation, as we observe only two spikes around 800 µs, in contrast to 16 spikes in the range of 500 to 700 µs. These findings validate Hypothesis H6 that JIT compilation can cause high event processing times.

Nevertheless, the data in Figure 4.17 reveals that there are still a couple

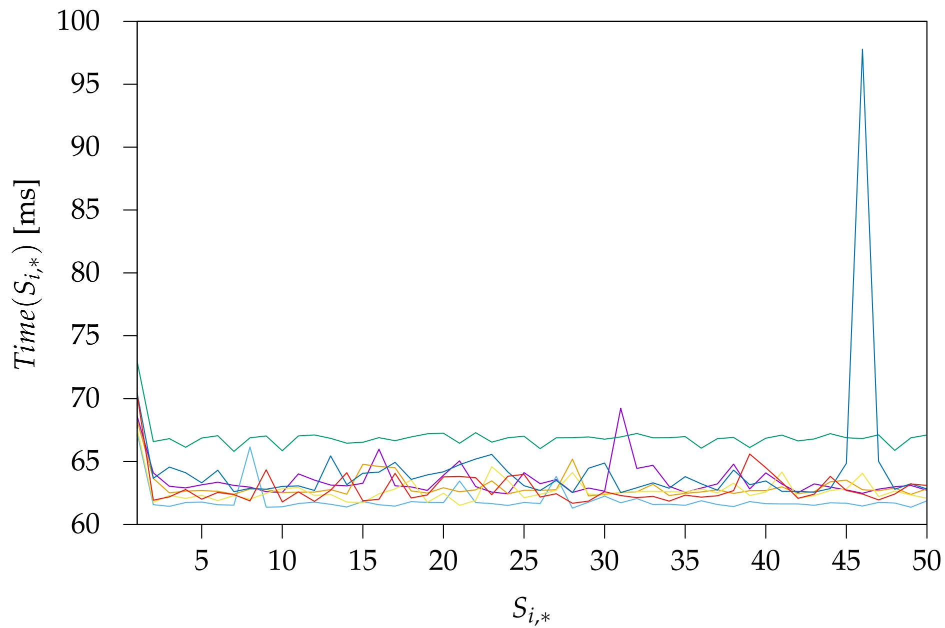

Figure 4.16: Comparison of burst processing in S^{1000}_{50} with burst distances ranging from 400 ms to 1000 ms. No garbage collection occurrences during runtime and JIT compiler is disabled.

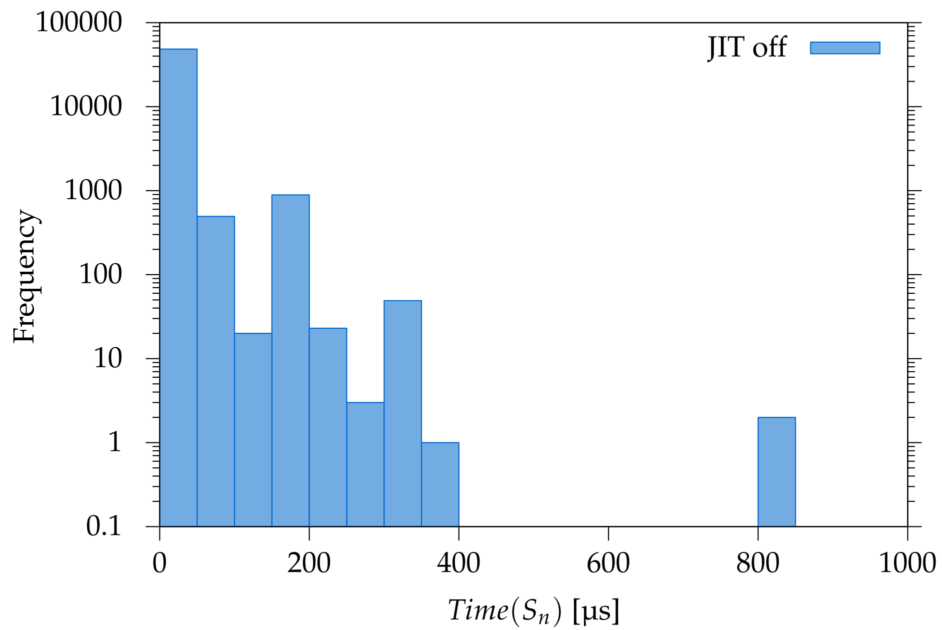

Figure 4.17: Distribution of event processing time in S^{50}_{1000} with 1000 d_{ms}. No garbage collection occurrences during runtime and JIT compiler is disabled.

of minor spikes event processing times, occurring halfway and at the end of the stream, as illustrated in Table 4.10. The amount of data stored in memory throughout the experiment is drastically larger than in our initial experiment, since we have increased the allocated space for the young generation of the heap. We postulate that these spikes are caused by *page faults*. During initialization, the JVM does not by default get all the pages of its heap into memory; instead pages are committed as the JVM heap space fills up[31], which introduces processing delays[40]. This is expressed as Hypothesis H7 and further investigated:

H7 High event processing times are caused by page faults.

Table 4.10: The 10 events with the highest processing time in S_{50}^{1000} with 1000 d_{ms}. No garbage collection occurrences during runtime and the JIT compiler is disabled. Time measurements are in microseconds.

i	j	$Time(S_{(i,j)})$
1	1	9722.73
50	12	815.07
25	8	814.27
38	1	378.93
46	1	340.16
20	1	337.40
12	1	331.82
49	1	331.52
17	1	331.18
23	1	331.16

4.3.6 Virtual memory

Virtual memory is a crucial component of modern operating systems which helps in managing memory resources while giving processes the illusion of owning the entire or more physical memory than what is available.

To manage virtual memory, the operating system breaks up a process's address space into fixed-sized blocks of virtual memory called *pages*, and maintains a map between virtual pages and physical *page frames* in a data structure known as a *page table*. When a process references an unmapped address, i.e., a page that is not in physical memory, a *page fault* occurs, which triggers the operating system to complete the page retrieval and placement process. This process is carried out by the *page fault handler*.

The page fault handler first saves the process' "state", a routine to push general registers and volatile information, such as local variables, hardware registers, and program counter on the stack. Then it acquires a page frame from the available free memory page pool and updates the page table to reflect the new mapping. For *major page faults*, the acquired page frame is filled with data content from a storage device corresponding to the referenced virtual address, while for *minor page faults*, the page frame is

zeroed out. Once the page fault handling is complete, the process' state is restored and the instruction that caused the page fault is re-executed.

When a program allocates memory, it is very rare for it to access (touch) all pages it requests immediately after the allocation request. Typically, program working sets are much smaller than their whole memory footprints. Based on this property of programs, the memory subsystem is not required to allocate memory at the moment of the initial allocation request. Memory is only allocated when it is required, upon its first usage. As a result, the operating system can run multiple processes concurrently without creating a scarcity of memory, and thereby improve the overall system performance.

This strategy is known as *lazy allocation* and is a key source of minor page faults. When the memory is allocated, only a region of the virtual address space is created by the operating system for the calling program. The page frames are not allocated to pair with those virtual pages until they are touched. A memory access within this newly created region triggers a minor page fault and the page fault handler will check and confirm this access as legal, and in turn allocate a page frame for it.

Processes produce page faults whenever they access virtual memory regions which are not mapped to a physical page. Page fault handling involves significant overhead, including disk I/O, memory allocation, pushing onto and popping from stack, and context switches results in pollution of architectural resources like caches, TLB and branch predictors. Hence, implementing efficient page replacement algorithms can significantly improve system performance. Furthermore, if the available memory is too small to hold the working set of a process, it will cause frequent page faults. Consequently, a high numbers of page swaps to disk are required, which initiates a vicious cycle of more page faults and excessive CPU utilization (thrashing), drastically reducing system performance.

To alleviate thrashing, the operating system may swap some processes out to disk temporarily, freeing up all the pages they are holding, and allocate more memory to the processes that are experiencing thrashing. By reducing the number of processes that are competing for memory, the system can potentially run more efficiently.

As the memory footprints of a process grow, they gradually induce minor page faults due to lazy allocation. Handling of each minor page fault can take a few thousands of CPU-cycles and blocks the application until the operating system finds a free physical frame. Thus, a high frequency of minor page faults spread across a process' lifetime can be detrimental to its performance. The study by Tirumalasetty et al.[42] shows that minor page faults alone can cause an overhead of 29% of execution time[40, 42].

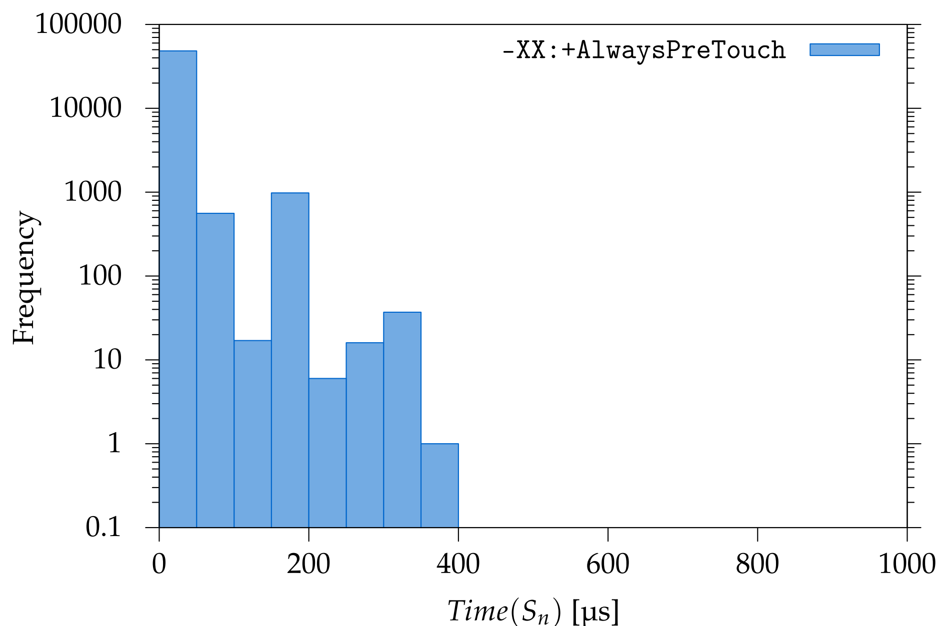

Figure 4.18: Distribution of event processing time for S_{1000}^{50} where the JIT compiler is disabled, no garbage collection has occurred during execution and the entire heap has been mapped physical memory.

4.3.7 Experimental design

To evaluate the impact of page fault handling on event processing time, we conduct an experiment on the workload of S_{50}^{1000} with 1000 d_{ms}. The selected workload is used to induce caching behavior. We set the heap size of the JVM to 4GB and dedicate 3GB to the young generation, to ensure that no garbage collection events occurred during our experiment. We disable the JIT compiler to eliminate any effects the dynamic compilation process have on event processing time. Furthermore, to minimize the number of page faults during our experiment, we instructed the JVM to touch every page in the allocated heap before entering the main method. This is enabled with the command-line option `-XX:+AlwaysPreTouch`[31]. We expect from the experiment a reduction in higher event processing times, demonstrated by a unimodal distribution, which would confirm Hypothesis H7 of higher event processing time being caused by page fault handling.

4.3.8 Analysis

The resulting event processing times are presented in Figure 4.18, demonstrating that we only observe processing times between 0 and 400 µs, which affirms Hypothesis H7.

4.4 Elevated event processing

Table 4.1 in Section 4.1 displays the burst processing time statistics for S_{50}^{1000} across varying burst distances. It is evident that the total processing time for bursts distances at 100, 200, and 300 ms are significantly higher compared to the total processing time for longer burst distances. The mean processing time for these shorter burst distances is between 48 and 65 ms, and the maximum never surpasses 105 ms. Surprisingly, burst processing at shorter burst distances leads to considerably increased processing times, even though our Siddhi program should have on average completed processing the entire event burst in a timely manner before the subsequent burst is injected for processing. Initially, we speculated that an ill-timed garbage collector scheduling during burst processing could be the underlying cause; however, as demonstrated in Section 4.2, garbage collection impacts burst processing time across all tested burst distances. Following our previous hypothesis, we suggest that dynamic voltage and frequency scaling (DVFS) causes the elevated event processing times, as DVFS adjusts the CPU frequency and voltage supply based on workload to minimize energy usage, thus substantially changing one's perception of performance.

We start the investigation of elevated event processing time with Hypothesis H8:

H8 Elevated event processing is caused by dynamic voltage and frequency scaling.

4.4.1 Dynamic voltage and frequency scaling (DVFS)

Dynamic voltage and frequency scaling (DVFS), also known as CPU frequency scaling, is a widely-used power management technique in modern processors[49]. By adjusting clock frequencies and voltage supply based on CPU workload, DVFS can reduce the energy cost of computation. Specifically, DVFS can decrease processor clock frequency together with the supply voltage, resulting in energy savings. This technique has been demonstrated to be particularly effective for memory-bound workloads, where the CPU wastes a significant portion of CPU cycles waiting for the memory subsystem to provide operands for instructions[35, 46]. By scaling down the frequency and voltage when the processor is idle, DVFS can reduce power consumption and improve energy efficiency.

DVFS helps to strike a balance between the CPU's capacity and power consumption, resulting in more efficient and sustainable operation. However, Le Sueur and Heiser[38] show that the energy saving benefits of using DVFS are diminishing. Firstly, lower memory access latency reduces pipeline stalls, which reduces the opportunities to save energy using DVFS. This means that the benefits of DVFS are reduced when memory access latency is lower. Secondly, the benefits of using DVFS are less when the power used in an idle mode is lower. This is because overall, less energy is

used by running at a high frequency and then spending longer in an idle state. Thirdly, DVFS becomes more complicated to implement on multi-core processors, which can limit its effectiveness. This is because each core on a package must operate at the same frequency and voltage, which can constrain the ability to scale frequency for workloads running on multiple cores. Finally, transistor sizes get smaller, more current is lost into the transistor substrate, which reduces the dynamic range of power consumption that DVFS can use. This means that DVFS becomes less effective at reducing power consumption and can actually increase static power consumption.

Linux supports DVFS by the means of CPUFreq subsystem[49]. The subsystem consists of three layers of code: the core, scaling governors, and scaling drivers. The core provides the common code infrastructure and user space interfaces for all platforms that support CPU performance scaling. Scaling governors implement algorithms to estimate the required CPU capacity, and as a rule, each governor implements one scaling algorithm. Scaling drivers talk to the hardware, providing scaling governors with information on the available P-states and accessing platform-specific hardware interfaces to change CPU P-states as requested by scaling governors.

A P-state refers to a processor's voltage and frequency configuration that is used to achieve a certain performance level. Scaling drivers talk to the hardware and provide scaling governors with information on the available P-states and access platform-specific hardware interfaces to change CPU P-states as requested by scaling governors. Typically, they are used along with algorithms to estimate the required CPU capacity, so as to decide which P-states to put the CPUs into.

In principle, all available scaling governors can be used with every scaling driver based on the observation that the information used by performance scaling algorithms for P-state selection can be represented in a platform-independent form in the majority of cases. However, performance scaling algorithms based on information provided by the hardware, e.g., through feedback registers, is typically specific to the hardware interface it comes from, and may not be easily represented in an abstract and platform-independent way. As a result, CPUFreq allows scaling drivers to bypass the governor layer and implement their own performance scaling algorithms. The `intel_pstate` scaling driver is an example of such a driver.[49]

4.4.2 Experimental design

Our computer system uses the `intel_pstate` scaling driver and operates in *active mode* with hardware-managed P-states (HWP) enabled. P-states refers to the clock frequency and voltage configurations for modern CPUs. The computer system has two scaling governors available, `powersave` and `performance`, and selects the `powersave` governor by default. The `powersave` governor supposedly sets the CPU frequency statically to the lowest frequency within the current frequency limits, while the

`performance` governor sets it to the highest frequency[49, 50, 9]. If we change the scaling governor it may exhibit significant differences in performance, despite the Intel P-state driver overriding our ability to set a static CPU frequency[4],

We process S_{50}^{1000} with burst distances ranging from 0 ms to 1000 ms. We vary the burst distance in steps of 100 ms for each experiment. These workloads are processed twice; once with the `powersave` scaling governor and once with the `performance` scaling governor. We use the JVM default settings as in our initial experiment and instrument the query with two tracepoints per event. By varying the burst distances, we aim to identify any discrepancies in processing time between each workload for both scaling governors, in order to assess the impact of DVFS on event processing.

We expect the `powersave` governor to yield similar results to those in Table 4.1 since it is the default scaling governor in our system. Specifically, we expect to observe the fastest average burst processing time for a constant stream, elevated processing time for burst distances between 100 ms and 300 ms, and typical burst processing times for burst distances greater than 300 ms. For the `performance` scaling governor, we expect the scaling governor to prioritize performance, and thus try to run at the highest possible CPU frequency, resulting in similar average burst processing times for each workload. This will indicate that the elevated processing times are caused by DVFS and that we should further investigate DVFS by processing the workloads at a fixed CPU frequency.

4.4.3 Analysis

Table 4.11 presents statistics of burst processing time in S_{50}^{1000} for various burst distances when using the `powersave` scaling driver. The results show some similarities to those of our initial experiment, as shown in Table 4.1, albeit with lower values. The highest average burst processing time is still observed for burst distances between 100 and 300 ms. However, the lowest average burst processing times are observed for 1000 d_{ms}, while 0 d_{ms} exhibit the second-lowest average burst processing time. Moreover, we observe a descending average processing time from 100 to 1000 d_{ms}, indicating that burst processing improves with longer burst distances. This finding is different from our initial experiment, and we suspect that the contrasting behavior is caused by the higher instrumentation overhead, since the experiment collected seven tracepoints per event.

Although our hardware is compromised by Intel's scaling driver to set a static CPU frequency, changing the governor exhibit significant differences in CPU utilization, as shown in Table 4.12. The findings show that burst distances ranging from 100 ms to 1000 ms yield similar performance. In contrast, a constant stream exhibits the least efficient event processing time, with a factor of two compared to the other burst distances. The contrasting event processing performance between the `powersave` and

Table 4.11: Burst processing times in S_{50}^{1000} with default JVM settings and `powersave` governor. Time measurements are in milliseconds.

d_{ms}	$Mean(S_{i,*})$	$Min(S_{i,*})$	$Max(S_{i,*})$	$Sum(S_{i,*})$	$std(S_{i,*})$
0	10.13	3.82	47.57	506.33	10.41
100	19.69	4.10	62.03	984.52	11.60
200	17.89	7.85	63.08	894.68	12.16
300	16.40	7.70	51.97	819.95	11.21
400	12.31	7.50	61.90	615.54	11.31
500	10.90	7.85	46.43	545.09	7.72
600	10.67	7.63	47.84	533.61	7.67
700	10.63	7.55	49.11	531.45	8.18
800	10.62	7.58	48.83	530.84	7.85
900	10.50	7.52	43.60	525.19	7.60
1000	9.76	7.62	38.87	487.94	5.97

`performance` scaling governor supports Hypothesis H8 of DVFS affecting event processing time. This might not be the case for the fine-tuned JVM runtime accomplished at the end of Section 4.3. Since the garbage collector and JIT compiler are not involved during execution, the event processing impacts the system differently, which might change the scaling driver's reaction to the same workload. This is formulated as Hypothesis H9:

H9 Elevated event processing are caused by DVFS even for a more deterministic JVM runtime.

Table 4.12: Burst processing times in S_{50}^{1000} with default JVM settings and `performance` governor. Time measurements are in milliseconds.

d_{ms}	$Mean(S_{i,*})$	$Min(S_{i,*})$	$Max(S_{i,*})$	$Sum(S_{i,*})$	$std(S_{i,*})$
0	11.19	3.69	64.74	559.64	12.84
100	6.22	3.89	31.89	311.10	6.64
200	5.80	3.74	41.68	290.07	6.75
300	5.90	3.71	35.07	295.14	6.51
400	5.97	3.74	41.55	298.62	7.04
500	6.20	3.73	31.42	310.02	6.88
600	5.70	3.62	41.62	284.97	6.76
700	5.90	3.71	30.87	295.21	6.07
800	5.96	3.72	36.24	298.13	6.45
900	5.82	3.65	41.85	291.13	7.07
1000	6.28	3.67	46.09	314.17	7.90

4.4.4 Experimental design

In order to investigate whether DVFS causes lower event processing performance on certain workloads in a more deterministic JVM runtime, we conduct experiments on S_{50}^{1000} using varying burst distances ranging from 0 to 1000 ms. The burst distance is increased in steps of 100 ms for each

experiment. We use the `powersave` scaling governor and adopt the JVM configurations from Section 4.3.7. We anticipate to observe similar event processing results as presented in Table 4.11, which will provide further evidence to support Hypothesis H8 and H9.

4.4.5 Analysis

Table 4.13 demonstrates that DVFS reduces event processing performance for certain workloads even when JIT compilation is disabled and garbage collection and page faults are avoided. This supports Hypothesis H8 and H9. To assess the impact of DVFS on event processing, we replicate the experiments from Section 4.4.2 with a fixed CPU frequency.

Table 4.13: Burst processing times in S_{50}^{1000} with JIT compilation disabled, no garbage collection or page fault occurrences, and `powersave` governor. Time measurements are in milliseconds.

d_{ms}	$Mean(S_{i,*})$	$Min(S_{i,*})$	$Max(S_{i,*})$	$Sum(S_{i,*})$	$std(S_{i,*})$
0	61.73	60.56	68.55	3086.26	1.46
100	95.02	68.32	98.85	4751.03	4.02
200	96.09	68.67	99.85	4804.65	4.38
300	82.74	63.18	102.78	4136.83	17.63
400	63.50	62.44	69.25	3175.09	1.34
500	66.91	65.81	72.96	3345.31	0.95
600	61.94	61.30	67.25	3096.87	1.08
700	62.99	62.28	68.24	3149.73	0.99
800	62.68	61.53	67.60	3133.78	0.96
900	64.43	62.54	97.78	3221.68	4.98
1000	62.88	61.69	70.28	3143.99	1.38

4.4.6 Experimental design

The experiments delineated in Section 4.4.2 are replicated under a fixed CPU frequency. This system employs the `acpi_cpufreq` frequency scaling driver, which enable us to set a static CPU frequency. We lock the CPU at the highest possible frequency of 2.8 GHz. By locking the CPU frequency, potential fallacies regarding event processing performance under DVFS are exposed, as the operating system abstains from adjusting CPU frequency based on workload.

In the initial investigation of DVFS, a correlation between event processing performance and burst distances ranging from 100 to 1000 ms is observed under the `intel_pstate` driver using the `performance` scaling governor. This system configuration results in higher event processing performance for workloads with longer burst distances. When processing identical workloads on a CPU with a static frequency, we anticipate that the correlation will be absent. Additionally, we expect that event processing performance for the constant stream workload will yield the worst processing time, analogous to operating under the `performance` scaling

driver. These findings ultimately demonstrate that DVFS indeed distort one's perception of performance.

4.4.7 Analysis

Table 4.14 demonstrates that a static CPU frequency nearly double burst processing time for a constant processing stream compared to streams with a burst distance of 100 d_{ms} or higher. Similar behavior was observed when using the `performance` governor, as illustrated in 4.12. However, the average burst processing time does not exhibit a decrease with longer burst distances, which illustrates that setting a static CPU frequency is necessary to avoid any misconceptions of event processing performance.

Table 4.14: Burst processing times in S_{50}^{1000} with default JVM settings and static CPU frequency of 2.8 GHz. Time measurements are in milliseconds.

d_{ms}	$Mean(S_{i,*})$	$Min(S_{i,*})$	$Max(S_{i,*})$	$Sum(S_{i,*})$	$std(S_{i,*})$
0	13.71	4.89	62.25	685.27	12.01
100	7.53	5.05	40.35	376.59	7.61
200	7.64	5.16	59.72	381.80	9.05
300	7.50	5.01	40.44	375.02	7.27
400	7.60	5.16	48.21	380.09	7.80
500	7.69	5.08	40.63	384.60	7.79
600	7.63	5.18	40.94	381.42	7.26
700	7.65	5.03	43.56	382.58	7.91
800	7.91	4.98	44.76	395.29	7.93
900	7.48	5.17	54.00	373.86	8.36
1000	7.57	5.07	40.53	378.26	7.51

4.5 Asymmetric event processing

In one of our initial experiments, we processed S_{50}^{1000} with 1000 d_{ms} under default JVM settings and scaling governor. A head-tail comparison analysis of the results indicates that the initial events in an event burst take significantly longer time to process than the subsequent events, as detailed in Section 4.1.3. It is hypothesized that this behavior is attributable to cache memories, which we explore further in this section as Hypothesis H10:

H10 Asymmetric event processing in bursts are caused by cache memory.

4.5.1 Cache state

In [16], the following terminology used to describe cache state:

- **Cold:** A cold cache refers to an empty cache, or one that is populated with unwanted data. The hit ratio for a cold cache is zero or near zero as it warms up.

- **Warm:** A warm cache refers to a cache that is populated with useful data, but does not sufficiently yet have a high hit ratio to be deemed *hot*.
- **Hot:** A hot cache refers to a cache that is populated with frequently requested data and has a high hit ratio, such as over 99%.
- **Warmth:** Cache warmth describes the extent to which a cache is hot or cold. An activity that improves cache warmth is on that strives to improve the cache hit ratio.

Upon initialization, caches start in a cold state and gradually warm up over time. When the cache has a large capacity or the next-level of storage has high access times, the cache may take a long time to become populated and warm.

4.5.2 Experimental design

Three experiments are conducted to process S_{50}^{1000} with 1000 d_{ms} under the `powersave` scaling driver, each employing a distinct tuning configuration of the JVM:

1. Enabling JIT compilation and preventing garbage collection.
2. Disabling JIT compilation and preventing garbage collection.
3. Disabling JIT compilation, preventing garbage collection, and pre-touching the entire JVM heap.

In each experiment, the selected JVM configuration aims to highlight the impact of a particular aspect of the JVM on asymmetric event processing. This is accomplished by consciously eliminating the target aspect in each experiment and in the subsequent experiments. The first and second configurations concentrate on the potential distortion of caching behavior by garbage collection and JIT compilation, respectively. Both the garbage collector and the JIT compiler are distinct processes likely to occupy sections of the shared cache memory, potentially replacing data relevant to the main application thread processing the event stream, and thereby impacting the cache miss ratio of the main application thread. The third configuration underscores the influence of pre-touching the entire allocated JVM heap on caching behavior. Pre-touching the JVM heap simulates a long-running system with all virtual pages mapped to physical page frames, thereby replicating the environment intended for modeling caching behavior.

The Siddhi application, responsible for generating the event stream, invokes the standard library function `Thread.sleep(long millis)` upon completion of injecting an event burst. This call suspends the active thread for `millis` number of seconds, prompting the operating system to schedule it in the task/process queue and prevent its execution until the predetermined number of milliseconds have elapsed. Concurrently, the cache lines associated with the running Siddhi application may be replaced

by cache lines of other processes. As the number of processes competing for the same resources is reduced with each experiment, it is anticipated that asymmetric event processing will progressively diminish as relevant data at the start of each burst is closer to the CPU. These findings will underscore the significance of finely tuning the JVM in order to accurately assess the impact of caching on event processing in Siddhi on a given system.

4.5.3 Analysis

Table 4.15 presents the results of a head-tail comparison analysis of S_{50}^{1000} with 1000 d_{ms} experiencing no occurrences of garbage collection. The JVM warm-up is not taken into account, since we are only interested in how caching impacts event processing in long-running systems. As with the head-tail analysis in Section 4.1.3, the result demonstrates that the head of the burst takes significantly longer to process than the tail, showing that asymmetric event processing in bursts is still present even when garbage collection does not occur during execution.

Turning off the JIT compiler results in a significant decrease in the head-tail ratio, particularly between the first and last event in a burst, as demonstrated in Table 4.16. The avoidance of paging, in addition to disabling the JIT compiler, only yields a minor decrease in the head-tail ratio, as shown in Table 4.17. The impact of JVM factors on asymmetric processing in bursts, depicted in Figure 4.19, suggest that memory plays a critical role in the observed behavior, given that the JIT compiler competes for memory resources and pre-touching the JVM heap populates the cache with relevant data prior to execution.

Table 4.15: Head-tail comparison analysis of S_{50}^{1000} with JIT enabled and no occurrences of garbage collection. JVM warm-up not included. Experiment executed under the `powersave` scaling governor. Time measurements are in microseconds.

N	A	B	$A-B$	A/B
1	60	1	59	60.00
5	99	13	86	7.62
10	143	23	120	6.22
20	229	43	186	5.33
30	316	64	252	4.94
40	418	85	333	4.92
50	505	105	400	4.81
100	978	208	770	4.70
500	3417	1067	2350	3.20

Furthermore, Figure 4.20 and 4.21 demonstrate that event processing time stabilizes after 25 events when JIT compilation is disabled, no occurrences of garbage collection, and the entire JVM heap is pre-touched. We hypothesize that DVFS is amplifying the observed processing behavior by swiftly adjusting the CPU frequency after processing a small number of

Table 4.16: Head-tail comparison analysis of S_{50}^{1000} with JIT disabled and no occurrences of garbage collection. Initialization phase not included. Experiment executed under the `powersave` scaling governor. Time measurements are in microseconds.

N	A	B	$A-B$	A/B
1	322	36	286	8.94
5	1043	183	860	5.70
10	1925	366	1559	5.26
20	3530	739	2791	4.78
30	3920	1108	2812	3.54
40	4285	1485	2800	2.89
50	4649	1856	2793	2.50
100	6474	3708	2766	1.75
500	21171	18487	2684	1.15

Table 4.17: Head-tail comparison analysis of S_{50}^{1000} with JIT disabled, no occurrences of garbage collection, and JVM heap pre-touched. Initialization phase not included. Experiment executed under the `powersave` scaling governor. Time measurements are in microseconds.

N	A	B	$A-B$	A/B
1	305	37	268	8.24
5	961	185	776	5.19
10	1768	372	1396	4.75
20	3341	746	2595	4.48
30	3900	1119	2781	3.49
40	4269	1493	2776	2.86
50	4636	1866	2770	2.48
100	6483	3731	2752	1.74
500	21369	18672	2697	1.14

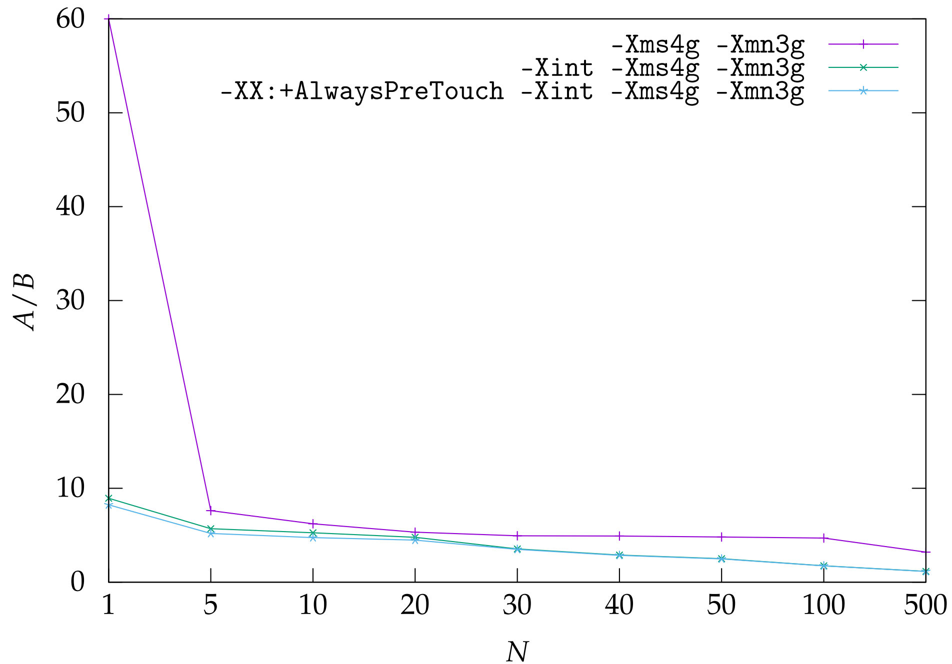

Figure 4.19: Comparison of JVM factors impacting asymmetric event processing in short bursts of 1000 events.

events in a burst to adhere to the scaling driver's configured goal. This is formulated as Hypothesis H11 and explored further:

H11 DVFS amplifies asymmetric event processing in event bursts.

4.5.4 Experimental design

We process S_{50}^{1000} with 1000 d_{ms} under a fixed CPU frequency of 2.8 GHz and replicate the third JVM configuration from the previous experimental design, with the aim of further isolating caching. This eliminates the potential impact of DVFS on asymmetric event processing in bursts. DVFS represents the final factor observed to cause performance non-determinism. We anticipate that the extent of asymmetric event processing in bursts will decrease, once again underscoring how DVFS can distort one's perception of performance and highlighting its importance for consideration when modeling software.

4.5.5 Analysis

When locking the CPU frequency, we observe an even smaller head-tail ratio, as demonstrated in Table 4.18. In addition, as indicated in Figure 4.22 and 4.23, event processing time stabilizes even faster, around $S_{*,3}$, thereby confirming Hypothesis H11 that DVFS amplifies asymmetric event processing in bursts.

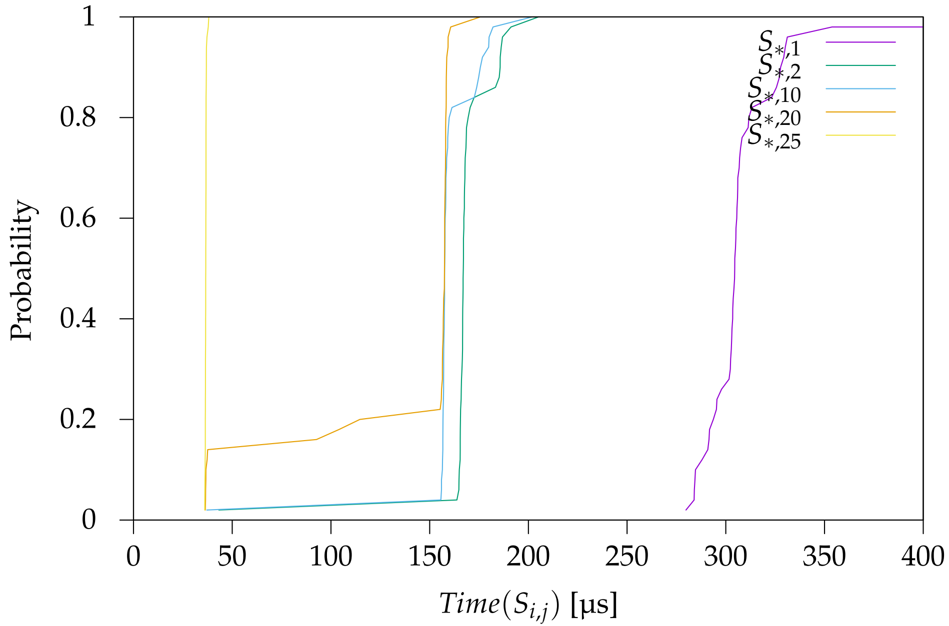

Figure 4.20: Distribution of event processing time for the first, second, fifth, 10th, 20th, and 25th event in event bursts for S_{1000}^{50} with the JIT compiler disabled and no garbage collection or page fault occurrences during execution.

Table 4.18: Head-tail comparison analysis of S_{50}^{1000} with JIT disabled, no garbage collection occurrences, and pre-touch enabled. Initialization phase not included. Executed with the CPU locked at 2.8 GHz. Time measurements are in microseconds.

N	A	B	$A-B$	A/B
1	129	53	76	2.43
5	346	266	80	1.30
10	612	533	79	1.15
20	1146	1065	81	1.08
30	1678	1599	79	1.05
40	2211	2132	79	1.04
50	2741	2664	77	1.03
100	5402	5325	77	1.01
500	26536	26509	27	1.00

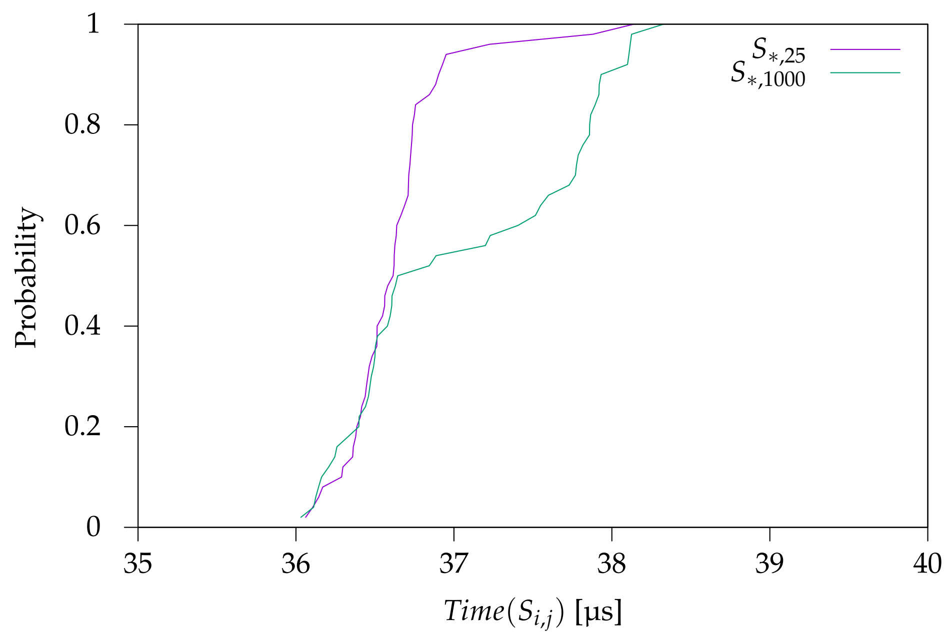

Figure 4.21: Distribution of event processing time for the the 25th and 1000th event in event bursts for S_{1000}^{50} with the JIT compiler disabled and no garbage collection or page fault occurrences during execution.

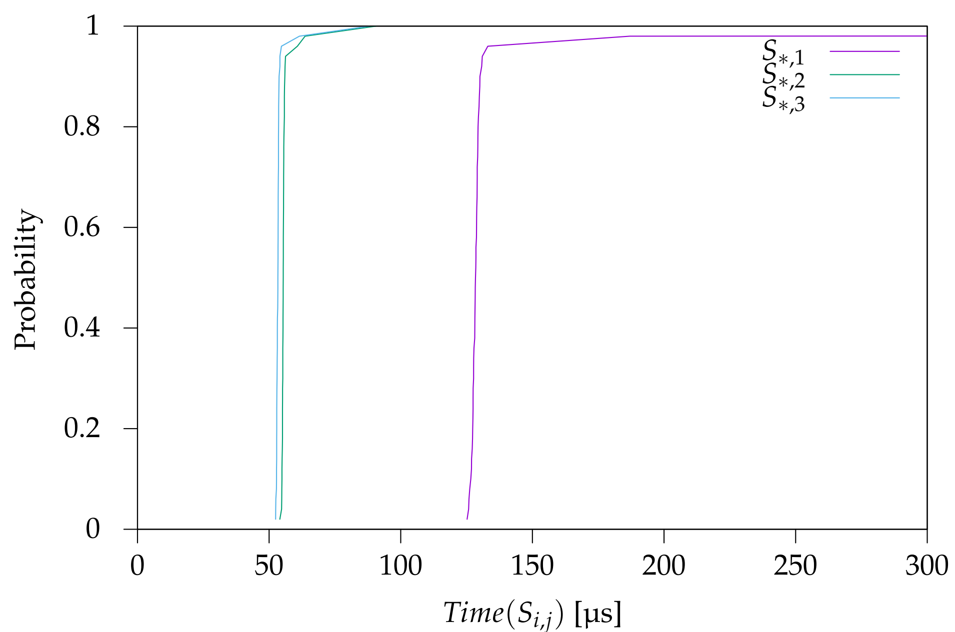

Figure 4.22: Distribution of event processing time of the first three events in event bursts for S_{1000}^{50} with the JIT compiler disabled, no occurrences of garbage collection during execution, and entire JVM heap is pre-touched. CPU frequency is locked to 2.8 GHz.

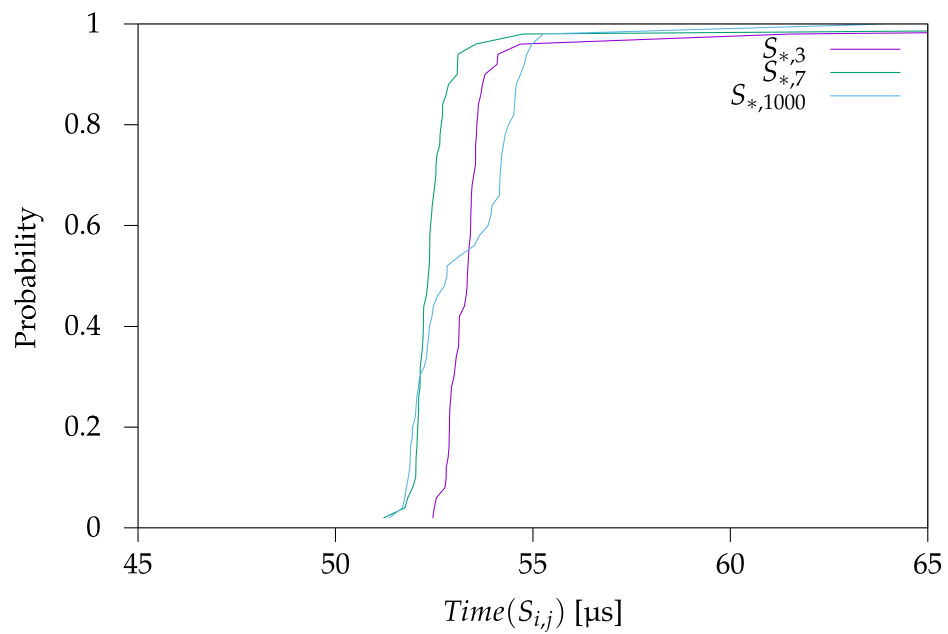

Figure 4.23: Distribution of event processing time of $S_{*,3}$, $S_{*,7}$ and $S_{*,1000}$ in S^{50}_{1000} with the JIT compiler disabled and no occurrences of garbage collection during execution and JVM heap pre-touched. CPU frequency is locked to 2.8 GHz.

4.6 Summary

This chapter delves into the analysis of event processing in Siddhi for a simple query when subjected to a sequence of short bursts and a constant stream. Through meticulously designed and conducted experiments, the analysis succeeds in pinpointing the performance non-determinism related to caching, resulting in a finely-tuned computer environment optimal for modeling the impact of caching on event processing. The analysis further recognize several attributes of the JVM, including garbage collection and JIT compilation, as significant contributors to performance non-determinism. Likewise, certain features of the computer system, such as DVFS, are recognized as notable factors that can distort one's perception of cache performance.

Figure 4.24 provides an illustration of how each mentioned attribute amplifies the impact of caching on performance non-determinism. The default configuration includes all attributes, namely garbage collection, JIT compilation, JVM heap not being pre-touched, and DVFS. The incremental configurations are as follows: Config 1 prevents garbage collection; Config 2 disabled JIT compilation; Config 3 pre-touches the entire JVM heap; and Config 4 locks the CPU frequency to 2.8 GHz.

Based on the results, JIT compilation emerges as the greatest amplifier of cache-induced performance non-determinism, followed by garbage collection and DVFS. This observation can be attributed to the memory-intensive nature of JIT compilation, which involves compiling code to native format and storing it in a dedicated code cache[18]. Garbage collection, on the other hand, operates as a background process, competing for cache resources and contributing to performance variability.

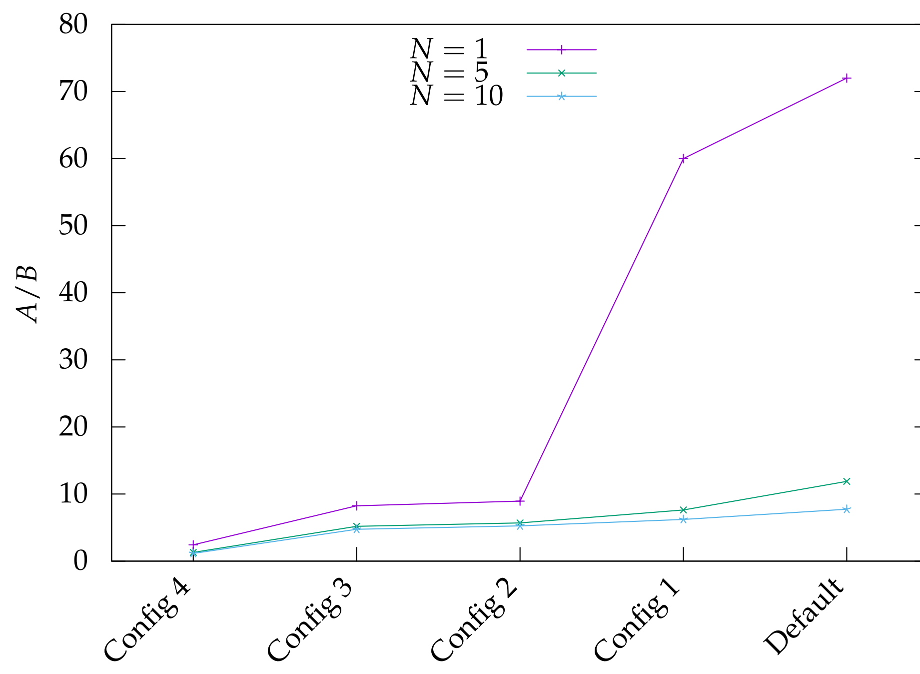

Figure 4.24: Configurations inducing various degrees of amplifications of caching behavior in relation to performance non-determinism.

Chapter 5

Conclusion

In this thesis, we set out to improve our understanding of performance non-determinism in Siddhi, specifically focusing on the modeling of caching behavior. Furthermore, we aimed to explore the limitations of the modeling methodology proposed by Kristiansen et al.[24] when it comes to capturing performance non-determinism accurately.

To guide our research, we formulated the following three research questions:

1. How can we effectively model the performance non-determinism of event processing in Siddhi to enhance our understanding?
2. What is the correlation between wall clock time and non-deterministic software execution, and how does it impact the performance of Siddhi?
3. To what extent is the modeling methodology proposed by Kristiansen et al.[24] limited in representing non-determinism, and what potential extensions can be made to overcome these limitations?

In Section 5.1, we provide a concise summary of the key contributions made in this thesis. These contributions cover the development of a tuning methodology, an in-depth analysis of event processing time with a focus on caching, and critical reflections on the conducted work and its results. Additionally, section 5.2 discusses open problems and identifies potential avenues for future research in this field.

5.1 Contributions and critical reflections

The analysis of performance non-determinism in event processing was conducted using a systematic methodology that involved several iterative steps. This methodology consisted of instrumentation, human investigation of traces, assessment of non-deterministic behavior, and system tuning. The iterations continued until a satisfactory level of determinism was achieved.

In analyzing the traces obtained through instrumentation, basic statistical techniques were employed. This involved computing various statistical metrics such as the average, minimum, maximum, and standard deviation of burst processing time in the event streams. These metrics were used to gain insights into the overall performance of the system. Additionally, empirical distribution graphs of event processing time were plotted to identify instances of performance non-determinism. A head-tail comparison analysis was designed to explore cache behavior and highlight differences in event processing time between the edges of a burst. The cumulative distribution function (CDF) was utilized to determine the warm-up time of caching.

From the analysis, it was observed that event processing in Siddhi is indeed affected by cache memories, and wall clock time plays a crucial role in this behavior[34]. Furthermore, JVM features like JIT compilation and garbage collection, as well as system attributes like DVFS, amplify the impact of caching behavior. Garbage collection introduces unpredictable spikes in event processing, while JIT compilation leads to variability in event processing time throughout the application's lifetime, as it struggles to reach a steady state of peak performance[4]. Minor page faults also contribute to higher event processing time due to the overhead of page replacement algorithms. Hence, when modeling event processing in Siddhi within a production environment, these attributes must be considered to ensure accurate simulation of software execution.

5.1.1 Contributions

This research contributes to the understanding of performance non-determinism in Siddhi and emphasizes the importance of considering temporal factors in modeling approaches. The findings highlight the need for further research and development in modeling methodologies to address performance non-determinism effectively in order to conduct accurate simulations.

How can we effectively model the performance non-determinism of event processing in Siddhi to enhance our understanding?

The analysis reveals that cache memories, a known source of performance non-determinism, significantly impacts event processing in Siddhi. The time dimension, represented by wall clock time, plays a critical role in this behavior. Thus, to effectively model performance non-determinism in Siddhi, it is important to incorporate the impact of cache memories and the associated time dimension into the modeling approach.

What is the correlation between wall clock time and non-deterministic software execution, and how does it impact the performance of Siddhi?

The analysis reveals a strong correlation between wall clock time and non-deterministic software execution in the context of Siddhi. Specifically, when processing short bursts of event tuples, the duration of idle time

between each burst significantly affects the measured processing time of the initial events in each burst. Furthermore, the behavior of event processing, driven by caching, is subject to the influence of various factors including JIT compilation, garbage collection, and DVFS. These factors amplify the impact of wall clock time on event processing, further accentuating the observed event processing behavior related to caching.

To what extent is the modeling methodology proposed by Kristiansen et al. limited in representing non-determinism, and what potential extensions can be made to overcome these limitations?

The analysis indicates that the modeling methodology proposed by Kristiansen et al.[24] has limitations in representing non-determinism, particularly in relation to wall clock time. The methodology does not account for the time component of cache memories, garbage collection spikes, and JIT compilation on software execution. To overcome these limitations, potential extensions to the methodology should incorporate the time dimension and its effects on cache memories to provide a more comprehensive representation of non-deterministic behavior in Siddhi.

5.1.2 Critical reflections

The process of methodically isolating cache behavior turned out to be more time-consuming than initially anticipated, which prevented the design of a model capturing the impact of caching on event processing. In hindsight, it is apparent that employing microbenchmarks would have been a more effective approach for assessing the impact of caching on software execution in relation to performance non-determinism, especially when considering the JVM as the runtime system. Microbenchmarks are specifically designed to isolate and measure the performance of specific components or functionalities within a system. By using microbenchmarks, it would have been possible to focus on the intricacies of caching behavior and evaluate its impact on software execution in a controlled and precise manner. Moreover, it would facilitate the design of experiments that specifically target the complex attributes of the JVM, such as JIT compilation and garbage collection. By modeling each of these attributes individually and gradually combining them, a more comprehensive understanding of their impact on performance could be achieved. Such an approach aligns well with the modeling abstractions of service execution models in [24].

5.2 Open problems and future work

This thesis has laid a solid foundation for further exploration and development in modeling performance non-determinism in Siddhi, specifically related to caching. In this section, we discuss some open problems and future work that could further build knowledge on the topic of modeling software execution of modern CEP systems.

5.2.1 Isolation of CPU core

Although we have made progress in isolating caching behavior, additional empirical evidence would be valuable. One possible approach is to run the application on a dedicated CPU core using techniques like `isolcpus`[2] or `cset`[43]. This would minimize the number of competing processes, preventing cache data relevant to the application from being displaced by other processes. It is expected that the cache would maintain a higher degree of warmth between each burst. However, it should be noted that the L3 cache is a shared resource, so there is still a possibility of variability in event processing times induced by caching.

5.2.2 Control task scheduling

To further reduce variability in event processing time, setting the highest possible priority value for the running application using `chrt --rr 99` can be considered. This would prevent the operating system scheduler from interrupting the process mid-burst or at least minimize the duration of interruptions. Another approach could be using a hard real-time operating system like RTLinux[51], which would provide a deterministic execution environment, facilitating the modeling process.

5.2.3 Cache model

The highest priority in the research is to model caching behavior within the system configuration. This priority is driven by the need to challenge the modeling methodology introduced by Kristiansen et al.[24] and potentially propose an extension to enhance its applicability. Specifically, the immediate focus is on modeling the impact of caching on the `select` query under various workloads. This includes exploring constant streams of different rates, workloads triggering alternating cache states of different warmth levels (hot, cold, and warm), and workloads with varying tuple sizes. Subsequently, the modeling expands to different queries or operators and their performance implications in relation to cache states.

Bibliography

[1] Ajay Acharya and Nandini S. Sidnal. "High Frequency Trading with Complex Event Processing." In: *2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)*. 2016, pp. 39–42. DOI: 10.1109/HiPCW.2016.014.

[2] *An overview and comparison of the isolcpus kernel parameter - Red Hat Customer Portal*. (Accessed on 05/14/2023). May 2022. URL: https://access.redhat.com/solutions/480473.

[3] *ARM Cortex-R Series Programmer's Guide*. (Accessed on 05/01/2023). 2014. URL: https://developer.arm.com/documentation/den0042/a/Caches/Cache-drawbacks.

[4] Edd Barrett et al. "Virtual machine warmup blows hot and cold." In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), pp. 1–27. ISSN: 2475-1421. DOI: 10.1145/3133876. URL: http://dx.doi.org/10.1145/3133876.

[5] Edd Barrett et al. *What Exactly do we Mean by JIT Warmup?* (Accessed on 04/24/2023). Apr. 2016. URL: https://soft-dev.org/events/bench16/slides/Edd_Barrett.pdf.

[6] R. Bhargavi et al. "Complex Event Processing for object tracking and intrusion detection in Wireless Sensor Networks." In: *2010 11th International Conference on Control Automation Robotics Vision*. 2010, pp. 848–853. DOI: 10.1109/ICARCV.2010.5707288.

[7] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. "Myths and Realities: The Performance Impact of Garbage Collection." In: *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '04/Performance '04. New York, NY, USA: Association for Computing Machinery, 2004, pp. 25–36. ISBN: 1581138733. DOI: 10.1145/1005686.1005693. URL: https://doi-org.ezproxy.uio.no/10.1145/1005686.1005693.

[8] Daniel Pierre Bovet. *Understanding the Linux kernel*. eng. Sebastopol, Calif.

[9] Dominik Brodowski. *Linux CPUFreq — CPUFreq Governors*. (Accessed on 04/11/2023). n.d. URL: https://kernel.org/doc/Documentation/cpu-freq/governors.txt.

[10] Yang Byung-Sun. "Java Virtual Machine Optimizations for Java and Dynamic Languages." PhD thesis. Seoul National University,

Department of Electrical Engineering and Computer Science, College of Engineering, Feb. 2017.

[11] Otávio M de Carvalho, Eduardo Roloff, and Philippe OA Navaux. "A Survey of the State-of-the-art in Event Processing." In: *Proceedings of the 11th Workshop on Parallel and Distributed Processing (WSPPD)*. 2013, p. 35.

[12] Ching Yu Chen et al. "Complex event processing for the Internet of Things and its applications." In: *2014 IEEE International Conference on Automation Science and Engineering (CASE)*. 2014, pp. 1144–1149. DOI: 10.1109/CoASE.2014.6899470.

[13] Gianpaolo Cugola and Alessandro Margara. "Processing Flows of Information: From Data Stream to Complex Event Processing." In: *ACM Comput. Surv.* 44.3 (June 2012). ISSN: 0360-0300. DOI: 10.1145/2187671.2187677. URL: https://doi.org/10.1145/2187671.2187677.

[14] Ulrich Drepper. *What Every Programmer Should Know About Memory*. 2007. URL: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

[15] Hua Fan et al. "Understanding the Causes of Consistency Anomalies in Apache Cassandra." In: *Proc. VLDB Endow.* 8.7 (Feb. 2015), pp. 810–813. ISSN: 2150-8097. DOI: 10.14778/2752939.2752949. URL: https://doi.org/10.14778/2752939.2752949.

[16] Brendan Gregg. *Systems Performance, 2nd Edition*. eng. 2020.

[17] D. Gunter et al. "NetLogger: a toolkit for distributed system performance analysis." In: *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.PR00728)*. 2000, pp. 267–273. DOI: 10.1109/MASCOT.2000.876548.

[18] Tobias Hartmann, Albert Noll, and Thomas Gross. "Efficient Code Management for Dynamic Multi-Tiered Compilation Systems." In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '14. Cracow, Poland: Association for Computing Machinery, 2014, pp. 51–62. ISBN: 9781450329262. DOI: 10.1145/2647508.2647513. URL: https://doi.org/10.1145/2647508.2647513.

[19] Michal Hocko and Tomas Kalibera. "Reducing Performance Non-Determinism via Cache-Aware Page Allocation Strategies." In: *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*. WOSP/SIPEW '10. San Jose, California, USA: Association for Computing Machinery, 2010, pp. 223–234. ISBN: 9781605585635. DOI: 10.1145/1712605.1712640. URL: https://doi-org.ezproxy.uio.no/10.1145/1712605.1712640.

[20] *How the JVM Locates, Loads, and Runs Libraries*. (Accessed on 04/24/2023). URL: https://blogs.oracle.com/javamagazine/post/how-the-jvm-locates-loads-and-runs-libraries.

[21] Raj Jain. *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling*. eng. New York, 1991.

[22] Sachini Jayasekara et al. "Wihidum: Distributed complex event processing." eng. In: *Journal of parallel and distributed computing* 79-80 (2015), pp. 42–51. ISSN: 0743-7315.

[23] Amirmahdi Khosravi Tabrizi and Naser Ezzati-Jivan. "Software Mining–Investigating Correlation between Source Code Features and Michrobenchmark's Steady State." In: *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*. 2023, pp. 107–111.

[24] Stein Kristiansen, Thomas Plagemann, and Vera Goebel. "A Methodology to Model the Execution of Communication Software for Accurate Network Simulation." eng. In: *ACM transactions on modeling and computer simulation* 26.1 (2015), pp. 1–31. ISSN: 1049-3301.

[25] Santosh Kumar. "Challenges for Ubiquitous Computing." In: *2009 Fifth International Conference on Networking and Services*. 2009, pp. 526–535. DOI: `10.1109/ICNS.2009.79`.

[26] Tim Lindholm et al. *The Java® Virtual Machine Specification*. Java SE 11. Oracle Corporation, Aug. 2018.

[27] David C Luckham. *The power of events : an introduction to complex event processing in distributed enterprise systems*. eng. Boston, 2002.

[28] Merriam-Webster. *Event*. In: *Merriam-Webster.com dictionary*. URL: `https://www.merriam-webster.com/dictionary/event` (visited on 10/31/2021).

[29] Scott Oaks. *Java performance : the definitive guide*. eng. Sebastopol, California, 2014.

[30] Oracle. "Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide." eng. In: Release 11. Oracle, 2022. Chap. 3-5,9. URL: `https://docs.oracle.com/en/java/javase/11/gctuning/hotspot-virtual-machine-garbage-collection-tuning-guide.pdf` (visited on 04/06/2023).

[31] Oracle. "Java Platform, Standard Edition Tools Reference." eng. In: Release 11. Oracle, 2021. Chap. 2. URL: `https://docs.oracle.com/en/java/javase/11/tools/tools-reference.pdf` (visited on 04/06/2023).

[32] Joshua Ruggiero. "Measuring Cache and Memory Latency and CPU to Memory Bandwidth." In: (Dec. 2008).

[33] *Siddhi Core Libraries*. Version 5.1.20. 2021. URL: `https://github.com/siddhi-io/siddhi`.

[34] Alan Jay Smith. "Cache Memories." In: *ACM Comput. Surv.* 14.3 (Sept. 1982), pp. 473–530. ISSN: 0360-0300. DOI: `10.1145/356887.356892`. URL: `https://doi.org/10.1145/356887.356892`.

[35] David C. Snowdon et al. "Koala: A Platform for OS-Level Power Management." In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: Association for Computing Machinery, 2009, pp. 289–302. ISBN: 9781605584829.

DOI: 10.1145/1519065.1519097. URL: https://doi.org/10.1145/1519065.1519097.

[36] Snyk. *JVM Ecosystem Report 2021*. (Accessed on 05/11/2023). 2021. URL: https://res.cloudinary.com/snyk/image/upload/v1623860216/reports/jvm-ecosystem-report-2021.pdf.

[37] Fabrice Starks, Stein Kristiansen, and Thomas Peter Plagemann. "DCEP-Sim: An Open Simulation Framework for Distributed CEP." In: ACM Digital Library, 2017.

[38] Etienne Le Sueur and Gernot Heiser. "Dynamic voltage and frequency scaling: the laws of diminishing returns." In: 2010.

[39] Sriskandarajah Suhothayan et al. "Siddhi: A Second Look at Complex Event Processing Architectures." In: *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*. GCE '11. Seattle, Washington, USA: Association for Computing Machinery, 2011, pp. 43–50. ISBN: 9781450311236. DOI: 10.1145/2110486.2110493. URL: https://doi-org.ezproxy.uio.no/10.1145/2110486.2110493.

[40] Andrew S Tanenbaum. "Modern operating systems : Global edition." eng. In: 4th ed. Upper Saddle River, N.J: Prentice Hall, 2014, p. 25. ISBN: 9781292061429.

[41] B. Tierney et al. "The NetLogger methodology for high performance distributed systems performance analysis." In: *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No.98TB100244)*. 1998, pp. 260–267. DOI: 10.1109/HPDC.1998.709980.

[42] Chandrahas Tirumalasetty et al. "Reducing Minor Page Fault Overheads through Enhanced Page Walker." In: *CoRR* abs/2112.14013 (2021). arXiv: 2112.14013. URL: https://arxiv.org/abs/2112.14013.

[43] Alex Tsariounov. *Ubuntu Manpage: cset-shield - cpuset supercommand which implements cpu shielding*. (Accessed on 05/14/2023). Sept. 2011. URL: https://manpages.ubuntu.com/manpages/trusty/man1/cset-shield.1.html.

[44] Espen Volnes, Stein Kristiansen, and Thomas Plagemann. "Improving the accuracy of timing in scalable WSN simulations with communication software execution models." eng. In: *Computer networks (Amsterdam, Netherlands : 1999)* 188 (2021), p. 107855. ISSN: 1389-1286.

[45] Espen Volnes et al. "Modeling the Software Execution of CEP in DCEP-Sim." In: *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*. DEBS '19. Darmstadt, Germany: Association for Computing Machinery, 2019, pp. 244–247. ISBN: 9781450367943. DOI: 10.1145/3328905.3332508. URL: https://doi.org/10.1145/3328905.3332508.

[46] Andreas Weissel and Frank Bellosa. "Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management." In: *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. CASES '02. Grenoble, France: Association for Computing Machinery, 2002, pp. 238–246. ISBN: 1581135750.

DOI: 10.1145/581630.581668. URL: https://doi.org/10.1145/581630.581668.

[47] Sean Wentworth and David D Langan. "Performance Evaluation: Java vs C++." In: *Computer Science Department, University of South Alabama* 36688 (2000).

[48] Christian Wimmer et al. "Initialize Once, Start Fast: Application Initialization at Build Time." In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360610. URL: https://doi.org/10.1145/3360610.

[49] Rafael J. Wysocki. *CPU Performance Scaling — The Linux Kernel documentation*. (Accessed on 05/02/2023). 2017. URL: https://www.kernel.org/doc/html/v5.10/admin-guide/pm/cpufreq.html.

[50] Rafael J. Wysocki. *intel_pstate CPU Performance Scaling Driver — The Linux Kernel documentation*. (Accessed on 05/02/2023). 2017. URL: https://www.kernel.org/doc/html/v5.10/admin-guide/pm/intel_pstate.html.

[51] Victor Yodaiken et al. "The rtlinux manifesto." In: *Proc. of the 5th Linux Expo*. 1999.