Stall-Focused Benchmarks for JVMs on the x86 Architecture

by

Zhuoran Li

Bachelor of Software Engineering, Sun Yat-sen University, 2019

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	David Bremner, Ph.D., Faculty of Computer Science
	Kenneth B. Kent, Ph.D., Faculty of Computer Science
Examining Board:	Eric Aubanel, Ph.D, Faculty of Computer Science, Chair
	Michael Fleming, Ph.D, Faculty of Computer Science
	Eduardo Castillo Guerra, Ph.D.,
	Department of Electrical and Computer Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

June, 2021

Abstract

Java is an important programming language both in industry and in academia. The Java Virtual Machine is the execution platform for applications implemented with Java. The x86 Instruction Set Architecture is widely supported by various modern computer architectures. One of the most important performance bottlenecks on these machines is stalling. Therefore, understanding the effect of stalls on Java applications executed on x86 platforms is important to application development and compiler design. In this thesis, a survey of stalls in Java applications on x86 platforms is carried out to gather critical information about the influence of stalls on various Java applications and the frequencies of different types of stalls in Java workloads. Based on such information, a stall-focused benchmark suite is proposed and validated.

Dedication

To my parents. Thank you for all your love and support.

Acknowledgements

This research was conducted within the Centre for Advanced Studies–Atlantic, Faculty of Computer Science, University of New Brunswick. The author is grateful for the colleagues and facilities of CAS–Atlantic in supporting our research. The author would like to acknowledge the funding support of the Natural Sciences and Engineering Research Council of Canada (NSERC), 536287-18. Furthermore, I would also like to thank the New Brunswick Innovation Foundation for contributing to this project.

I am very grateful to my supervisors, professor David Bremner and professor Kenneth B. Kent, for their support and guidance during my studies. I would like to thank Senior RA Stephen MacKay for your feedbacks and advice. I would also like to thank RA Deverne Jones, Hassan Arafat and other colleagues at CAS-Atlantic Lab for all the help during the studies. Finally, thanks to Andrew Craik and Julian Wang from IBM for their technical input.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The thriving of Java [41] for so many years has proven that "write once, run everywhere" is a common request of software developers. As an implementation of the High-Level Language Virtual Machine (HLLVM) [65], the Java Virtual Machine (JVM) [52] translates architecture-neutral Java bytecode into architecture-specific machine code and executes it. Thus, the developers do not need to handle architecture-specific details and can focus on business logic. Another important feature of the JVM is Just-in-Time compilation [24], which optimizes particular pieces of code automatically to improve application performance. Finally, Java is fundamentally designed for Object-Oriented Programming [34, 67], which is the backbone of many complicated applications.

The x86 Instruction Set Architecture [21, 22] is one of the most common instruction sets supported by computers. One of the most important performance bottlenecks on these machines is stalling. A stall is an undesirable halt in the process of an instruction. Stalls prevent applications from taking full advantage of the high processing speed and decrease the benefits of multi-processing [56, 71].

Given the popularity and special status of Java and JVM, it is vital for developers and compiler designers to understand the effect of stalls on Java applications executed

on x86 platforms. The contributions of this work include the following:

1. A survey of stalls in Java applications on x86 platforms is carried out to gather critical information about the influence of stalls on various Java applications and the frequencies of different types of stalls in Java workloads. This survey is beneficial to the work of researchers, compiler designers and developers.

2. A benchmark suite is proposed to reflect the stalls observed in the survey. These stall-focused benchmarks are validated with experiments utilizing various JVM implementations on different x86 micro architectures. The benchmarks are beneficial to the work of researchers and compiler designers.

The rest of the thesis is organized as follows. Chapter 2 presents the concepts and information necessary to understand the discussion of the thesis. Chapter 3 presents the results and analysis of the survey. Based on such information, Chapter 4 discusses the design of the proposed benchmarks. Then, Chapter 5 illustrates the validity of these benchmarks with experimental results. Finally, Chapter 6 discusses the future work on the project.

Chapter 2

Background

In this chapter, several background concepts are discussed. In Section 2.1, an overview of the Java programming language is provided. Then, a brief description of memory hierarchy in modern computers is presented in Section 2.2. In Section 2.3, the x86 Instruction Set Architecture and the concept of stalls are discussed. Finally, an overview of hardware performance measurement and benchmarking is presented.

2.1 Java Programming Language

The Java programming language [41] is an important programming language both in industry and academia. According to Oracle®, over 10 million developers utilize Java and there are 13 billion devices executing Java applications around the world [3]. Java is utilized to implement a wide range of applications from web applications and database systems to big data applications [2, 11, 57, 64, 76]. From an academic perspective, Java is an example of high-level programming language executed on a virtual machine. Thus, understanding the performance bottlenecks of Java applications provides help for language and compiler designers.

2.1.1 High-Level Language Virtual Machine

The High-Level Language Virtual Machine (HLLVM) [65] is a special category of software systems addressing platform independence and the separation of business logic from implementation details specific to a particular platform. Languages running on HLLVMs are referred to as managed languages [20] in contrast to languages running directly on the platform such as C++ [68]. The latter are referred to as native languages. A HLLVM takes architecture-neutral bytecode as input and translates it into executable code on the specific platform. In addition, HLLVMs also provide automatic memory management such as garbage collection to increase robustness of applications and separate developers from hardware management. Therefore, developers are relieved from platform specific details and are able to focus their efforts on application functionalities. The comparison between managed languages and native languages is shown in Figure 2.1 and Figure 2.2. Applications written in managed languages do not handle specific platform details. Instead, such details are handled by the HLLVM and thus only one implementation of the application is needed.

In the early days, applications running on HLLVMs were slower than applications written in native languages because HLLVMs provided only basic interpretation for the application code and many optimizations were not available. The introduction of the Just-In-Time (JIT) compiler [24] proved to be a successful solution to the issue. Nowadays, application performance is no longer a weakness of managed languages. Instead, more optimizations are available because profile data collected during execution can be utilized by the JIT compiler. Given the portability and performance of HLLVMs, it is not surprising that an increasing number of developers choose managed languages to implement complicated commercial applications.

One of the most widely utilized HLLVMs is the Java Virtual Machine (JVM) [52] which emphasizes the "write once, run everywhere" idiom of the Java Programming Language [41]. Other popular HLLVMs include the Common Language Runtime

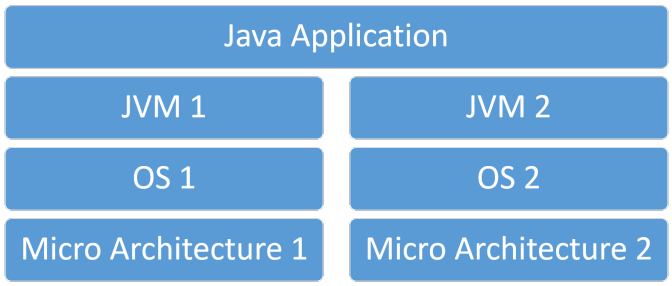


Figure 2.1: Concept of managed languages (using Java as an example).

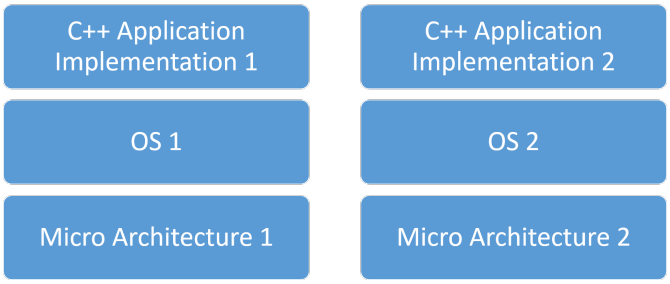


Figure 2.2: Concept of native languages (using C++ as an example).

(CLR) [47] for C# and PyPy [30] for Python.

2.1.2 Java Virtual Machine

The Java Virtual Machine is the execution platform of Java bytecode. Originally the JVMs were built to support the Java programming language. However, there are also multiple other languages such as Scala [58] and Kotlin [46] running on JVMs nowadays. Applications written in these languages are first compiled into Java bytecode before being executed by the JVM. The procedure of the compilation and execution of a Java application is shown in Figure 2.3.

First, the Java application is translated into bytecode. Then, the bytecode is executed by the JVM, possibly on another computer with a different platform from that of the developer's computer. The execution of the application involves interpretation and compilation. The interpreter executes the bytecode directly without performing any optimization. The interpretation procedure is straightforward but

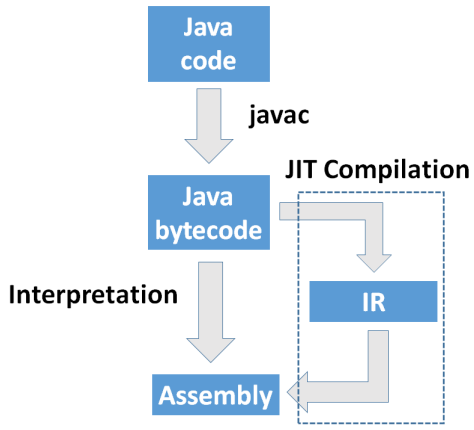


Figure 2.3: Java application compilation and execution procedure.

not efficient. The JIT compiler first transforms the bytecode into Intermediate Representation (IR) which provides more efficiency for applying optimizations. Then, various optimizations are performed on the generated IR structure. Finally, assembly code is generated during the instruction selection process and is executed after minor optimizations. Although the code generated by the JIT compiler is faster, the compilation consumes much more time and memory than the interpreter. Therefore, methods are usually first interpreted and only frequently executed methods are compiled by the JIT compiler. Accordingly, the lifespan of the executed program can be divided into two phases. In the start-up phase, various information, such as the invocation frequencies of methods and the access frequencies of code blocks, is gathered. The JIT compiler actively compiles frequently executed methods based on such information. After the start-up phase, the program enters the steady state. In this phase, most necessary compilations are finished and the JVM focuses its resources on the execution of the program.

2.1.3 Just-In-Time Compiler

As mentioned in Subsection 2.1.2, there are two execution approaches in the JVM. The interpreter immediately executes the bytecode without optimization. The JIT compiler performs time-consuming optimizations and generates much more efficient code. The JIT compiler can perform both static optimizations such as dead code elimination and dynamic optimizations such as code block reordering, when the estimated cost, including the time spent on optimization and the increased code size, is outweighed by the future benefits. The basis for such decisions is the profile data gathered during execution. A typical example is method invocation frequencies, which is utilized to estimate method invocations in the future. Profile data also provides support to the JIT compiler for optimizations based on speculations such as Polymorphic Inline Caching [43, 44], which requires information about utilization frequencies of different types during execution. Such optimizations are difficult to apply during static compilation without hints from the developer. As a result, Java programs sometimes achieve better performance than programs written in native languages [27, 32].

The structure of Java bytecode is not efficient for the analysis and application of optimizations. Therefore, the JIT compiler transforms the Java bytecode into an intermediate representation (IR) [33, 65]. The IR generated from the code snippet in Figure 2.4 and how it enables the compiler to detect and remove dead code are described in Figure 2.5.

Although the structure of the IR varies among different implementations, most JVMs utilize a graph-based structure for simplicity and efficiency. Different support for tracking the compilation procedure of the JIT compiler is provided by different JVMs. This is covered in Subsection 2.1.5 and Subsection 2.1.6.

```
public int add(int a, int b, int c) {
    int d = a / b;
    int e = a + b + c;
    return e;
}
```

Figure 2.4: Java code snippet containing dead code.

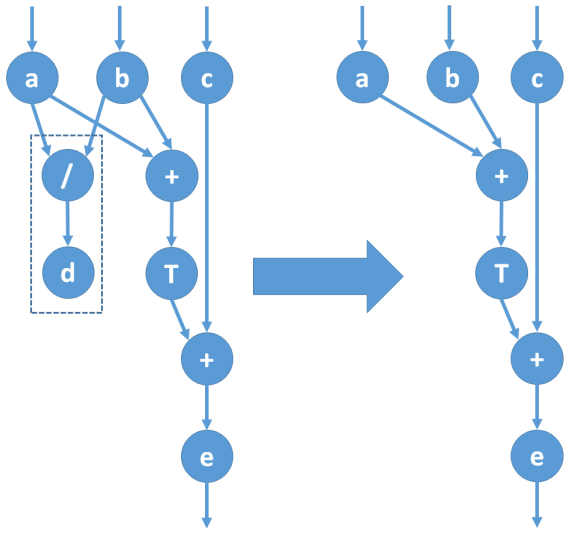


Figure 2.5: Dead code elimination performed on graph-based IR.

2.1.4 Object-Oriented Programming

The support for the popular Object-Oriented Programming (OOP) paradigm [34, 67] is another important feature of Java. The core of OOP is the concept of an object. An object contains data of an individual instance. Such data is referred to as the properties of the object and is accessed via property names and types. Different objects interact with each other by invoking the methods they expose. A class describes a group of similar objects sharing the same property names, property types and methods. A class can provide all such protocols by inheriting another class. To avoid issues caused by multiple inheritance such as the Diamond Problem[1] [31] , every

[1]The Diamond Problem happens when two classes, A and B, both inherit class C, and class D inherits both class A and B. When class D inherits a method from A that is overridden by both B and C, ambiguity arises since there are two versions available.

Java class can inherit at most one class. Java also supports interfaces that specify the signatures of methods and allow each class to provide its own implementation. Each class can implement multiple interfaces. An interface can inherit multiple interfaces because method implementation is completely controlled by the class and thus no name conflict exists when the method is invoked.

Each Java object can be accessed only via its reference. Consider an instance of the Person class described in Figure 2.6. In Java, only the layout described in Figure 2.7a is supported natively. On the other hand, both layouts in Figure 2.7 are supported in C++. This feature regulates the utilization of objects in Java but also requires more dereferencing operations to access a field.

Another regulation is that users are not allowed to place objects on the stack. Therefore, most objects reside in the heap even though the JVM places some objects on the stack as an optimization. The destruction and placement of objects are also performed by the JVM through Garbage Collection (GC) [28]. During GC, inaccessible objects (dead objects) are collected and their heap space is freed. Accessible objects (living objects) are moved for better space efficiency. In this process, the JVM is able to perform some heap layout optimizations [38, 69, 75].

2.1.5 Eclipse OpenJ9

Eclipse OpenJ9 [5] is a JVM implementation originally developed by IBM™ and is now an open source project under the Eclipse Foundation. It provides full support for the features described in the Java Specification [41, 52]. The components provided by the Eclipse OMR project [15] are applied in OpenJ9 for performance and robustness. Many functionalities and optimizations performed by OpenJ9 are provided by such components.

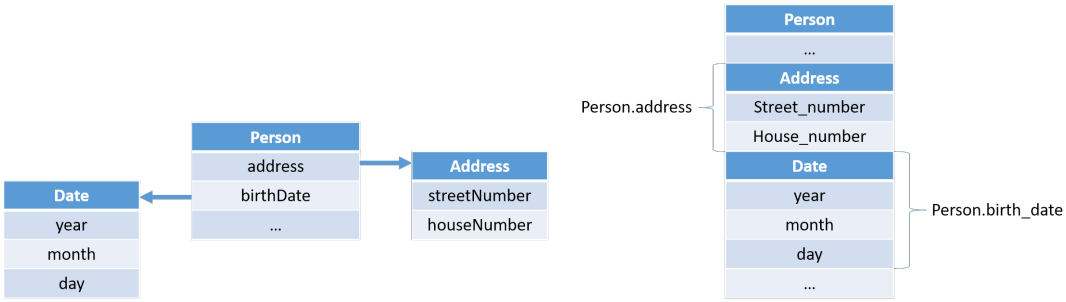
The Testarossa Intermediate Language (Tril) is utilized by OpenJ9 to perform JIT compilation. The bytecode of a method is transformed into a graph where each node

```
public class Person {
    public Address address;
    private Date birthDate;
    ...
}

public class Address {
    public int streetNumber;
    public int houseNumber;
}

public class Date {
    public int year;
    public int month;
    public int day;
}
```

Figure 2.6: Java code snippet for Person, Address and Date class.



(a) Layout supported by Java (b) Layout not supported by Java

Figure 2.7: Layouts for Person Class.

is an instruction from the instruction set of Tril and directed edges connecting these nodes describe the data flow. Optimizations are applied by modifying the graph. For instance, code block reordering is performed by moving subgraphs representing different code blocks in the method.

There are five optimization levels in OpenJ9: cold, warm, hot, very-hot and scorching. By default, all methods are compiled to the warm level. If a method is invoked sufficiently frequently, dynamic profiling is applied to provide more support for optimizations. The higher the optimization level, the more time that is spent on

collecting data to promote it to the next optimization level. Methods with higher optimization levels also consume more compilation time because more optimizations are available and more optimization iterations are executed. Also, optimizations that have the potential to cause performance degradation, such as loop unrolling, are applied cautiously only if the method satisfies a set of conditions.

OpenJ9 provides comprehensive support for tracking the behaviors of its components. In particular, it provides a precise mapping from method signatures to their addresses. This feature enables the utilization of perf [16], a PMU[2]-based sampling profiler, for inspecting specific hardware events such as CPU clock cycles and cache misses happening during execution. Since the generated code of a method often changes during execution due to JIT compilation, the mapping from method signatures to code data addresses is difficult to maintain. OpenJ9 solves this issue by integrating optimization levels into the signatures and thus a mapping exists for each version of the same method. Other information provided by OpenJ9 includes compilation decisions and GC behaviors. Users can obtain all such information by specifying the provided logging options.

2.1.6 HotSpot JVM

Another widely utilized JVM implementation is HotSpot [8], a JVM implementation provided by Oracle™ Corporation. Similar to OpenJ9, HotSpot provides support for both interpretation and JIT compilation. To achieve both fast start-up and steady-state performance, HotSpot utilizes two JIT compilers to perform tiered compilation. The C1 compiler [49] provides faster compilation but generates less efficient code. The C2 compiler [59] performs more optimizations at the expense of longer compilation time. Although the IRs utilized by the two compilers are different in design, they are both graph-based. Tiered compilation enables multiple optimization levels

[2]More discussion about the functionality and mechanism of PMU will be presented in Subsection 2.4.1.

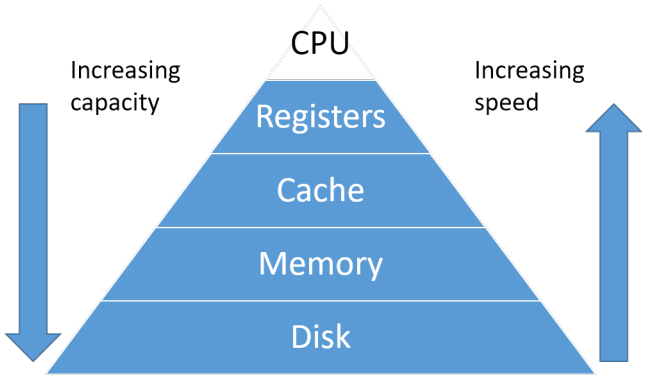


Figure 2.8: Memory hierarchy.

in HotSpot. However, HotSpot does not provide full support for the utilization of perf as OpenJ9 does. For each method, only the mapping of the latest version is provided. Therefore, the mappings of the less optimized versions are unavailable.

2.2 Memory Hierarchy

The processing speed of modern computers is increasing greatly. However, the memory access speed is not increasing accordingly. In addition, the shorter the access latency a memory unit provides, the higher the production cost is. To mitigate the negative effects caused by the discrepancy between processing speed and memory access latency, while maintaining an affordable production cost, a memory hierarchy is introduced in modern computers. In general, a memory hierarchy includes multiple layers of different types of memories including registers, cache, main memory and persistent storage. These layers are arranged in such a structure that the faster the access speed a layer provides, the closer it is to the processors. In this section, different layers in the hierarchy, as depicted in Figure 2.8, are discussed.

2.2.1 Cache

The CPUs interact directly with the registers only. While being the fastest memory unit, the number of registers is limited. Cache is the fastest memory unit with which the CPUs do not interact directly. Cache is divided into two parts.[3] The Instruction Cache (I-Cache) contains instruction data while the Data Cache (D-Cache) contains application data. Such design enables different functionalities in different caches. For instance, instruction fetching requests are handled only in I-Cache and data loading is served only in D-Cache. Both caches are divided into fixed-sized blocks, called cache lines. When the requested data resides in the cache, a cache hit occurs. In contrast, a cache miss happens when the requested data is not in the cache. As a result, the corresponding instruction cannot be executed before the requested data is loaded. Such a procedure often consumes a significant amount of time and causes performance degradation. Since the capacity of the caches is very limited compared to that of main memory, it is possible that two different data blocks are mapped to the same cache line. When a cache conflict occurs, a cache line is swapped out for another cache line. If the swapped-in data is not utilized, then the cache is polluted and performance is degraded since extra time is necessary to load the useful data back from memory. Therefore, ensuring the correct data resides in the cache at the correct time is crucial to application performance.

The multi-processor system supported by modern computers introduces another problem: memory coherency. To guarantee all the processors handle the same version of data, particular hardware protocols are applied. Such protocols force a particular cache line to be flushed to prevent invalid data from being handled. One of the most widely utilized protocols is the MESI protocol [60]. The state transition of MESI is depicted in Figure 2.9.

When processor P reads a data block that does not reside in the cache, it marks

[3] In fact, multi-level caches exist and a unified cache design is often implemented in the lower levels. However, this does not affect our simplified discussion here.

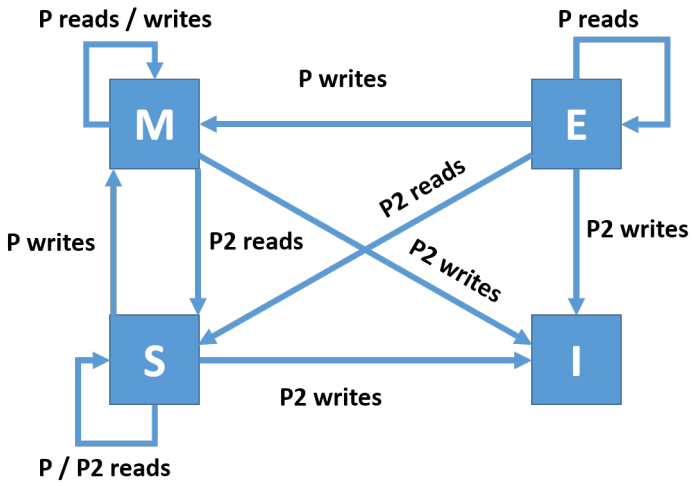


Figure 2.9: MESI state transition from processor P's perspective.

the data as Exclusive (E). When P writes to the block, the cache line is marked as Modified (M). If another processor, P2, reads the same cache line, the block is marked as Shared (S) and P2 obtains a copy of the data. When P writes to the shared block, the copy stored in P2's cache is invalidated. Similarly, the copy in P is marked as Invalid (I) when P2 writes to its copy. Once a cache line is invalidated, it cannot be utilized. One of the problems caused by sharing data among different processors is false sharing. False sharing happens when the data utilized by two processors reside in the same cache line. There is no data sharing between the processors but when one processor writes to its own data it invalidates the cache line and thus the other processor is forced to read the block from memory. False sharing decreases application performance since extra time is spent on loading data even though it is still valid.

2.2.2 Main Memory

Most of the application data is stored in main memory during execution. In general, the application data is stored in either thread-local stacks or the shared heap [52]. As mentioned in Subsection 2.1.4, most Java objects are stored in the heap. Figure 2.10

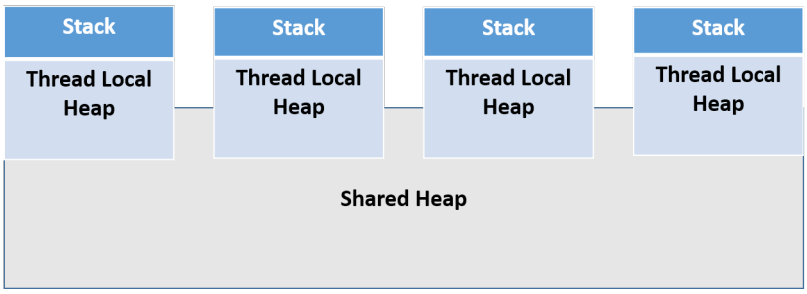


Figure 2.10: Simplified Java memory layout.

describes a simplified Java memory layout based on the implementation in OpenJ9. For each application thread, it is forbidden to access the stack of another thread. Note that the thread-local heaps are still accessible globally despite the speed difference. Therefore, the heap is still shared by all threads conceptually.

2.2.3 Non-Uniform Memory Access

From the users' perspective, each processor accesses different memory sections with the same efficiency. However, this is often not true. Non-Uniform Memory Access (NUMA) [50, 66], as shown in Figure 2.11, is favorable for its simplicity and good scalability. Such a design assigns each processor to a group called a NUMA node. Accessing memory within the same NUMA node is faster than accessing the memory in another NUMA node. It is desirable to place data frequently utilized by a particular thread within its corresponding NUMA node [61].

2.3 x86 Instruction Set Architecture

The x86 Instruction Set Architecture [21, 22] is one of the most widely used architectures. As an implementation of the Complex Instruction Set Computer (CISC) design, each instruction in the instruction set varies in length and must be decoded into micro operations (uOps) before execution. Although originally developed by

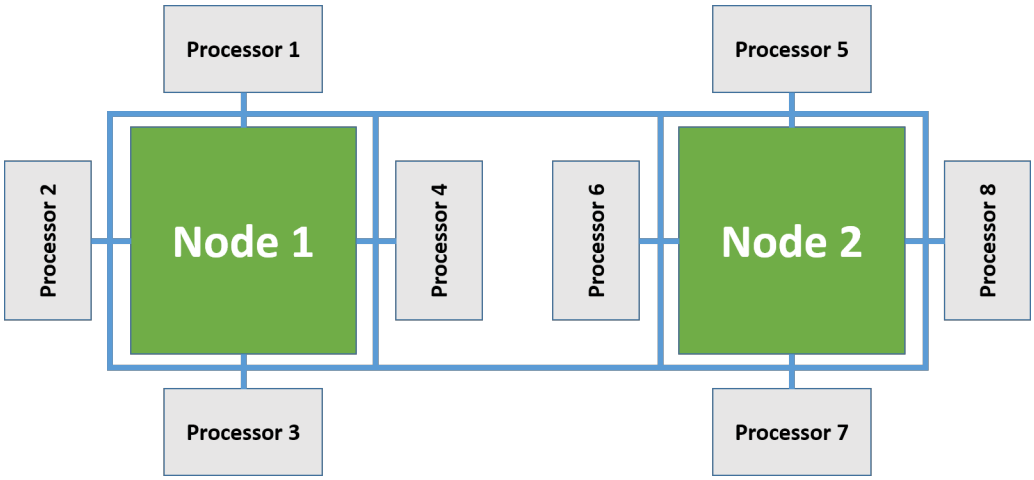


Figure 2.11: NUMA with 2 NUMA nodes.

Intel®, the instruction set is now also supported by a variety of micro architectures produced by different manufacturers. Such micro architectures will be referred to as x86 architectures in the remainder of this thesis. A simplified description of the Nehalem micro architecture [70], one of the x86 architectures produced by Intel, is shown in Figure 2.12.

Conceptually, the life cycle of an instruction can be divided into the following five stages:

1. Fetch. In this stage, a chunk of data containing instructions is fetched from the instruction cache. In order to fetch as many useful instructions as possible, special units, such as the branch prediction unit, are employed to fill the pipeline with prefetched instructions. Due to the complexity and depth of the pipelines in modern micro architectures, the accuracy of instruction prefetching is of crucial importance and remains a priority for manufacturers. Many details of such units are still commercial secrets and not publicly documented.

2. Decode. In this stage, x86 instructions are extracted from instruction data and decoded into micro operations. As mentioned above, the length of an x86 instruction varies according to the type of the instruction. Therefore, the

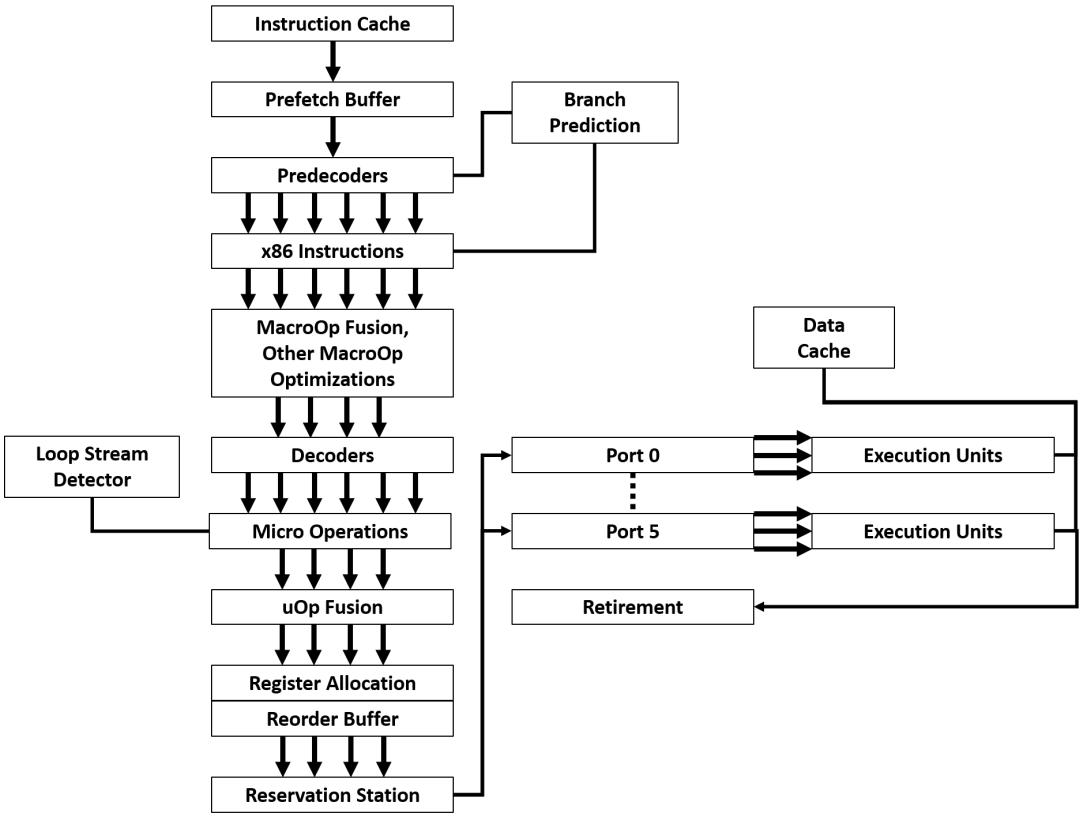


Figure 2.12: A simplified description of the Nehalem architecture.

instructions might not align with the boundaries of cache lines. As a result, the data fetched in the previous stage might contain multiple instructions, junk data for alignment or even part of an instruction. In order to extract the correct x86 instructions, several decoders are employed to process the data. After such instructions are extracted, they are decoded into micro operations by other decoders. In this phase, some optimizations are employed for further processing. For instance, Macro Fusion [70] merges several instructions into one micro operation to save slots and trackers in the Reorder Buffer.[4] Similarly, Micro Fusion [70] merges consecutive micro operations into one. Then, the registers utilized by these micro operations will be reallocated, allowing the

[4] The Reorder Buffer is an essential component for Out-of-Order execution, an optimization achieving instruction level parallelism by adjusting the order in which uOps are executed. To ensure the correctness of execution results, trackers are built within the buffer to track different uOps stored in different slots.

Reorder Buffer to contain and reorder the uOps for Out-of-Order execution[4]. Note that in some micro architectures, both Fetch and Decode stages can be omitted if previously utilized micro operations in the same loop are buffered.

3. Execute. In this stage, micro operations are issued to different ports and dispatched to the corresponding execution units. Each port is connected to different numbers of various execution units. There are more of certain execution units than others. For instance, there is only one divider while multiple adders exist. In addition to the number of available execution units, the number of micro operations delivered from the previous stage also limits instruction level parallelism. Note that different uOps of the same instruction can be executed at the same time as well as uOps of different instructions.

4. Write back. In this stage, the results of micro operations are written back to the corresponding registers or cache lines. Such results can be forwarded to waiting operations so that they can be delivered before the current operations are retired.

5. Retire. In this stage, the correct execution of a micro operation is confirmed and its results are visible as if it was executed in order. An instruction is regarded as finished when all its micro operations are retired. Note that instructions merged by Macro Fusion is treated as separate instructions by performance units despite being merged into one micro operation.

In addition to basic instructions such as arithmetic instructions and branching instructions, the instruction set also provides optional prefixes to add additional features to certain instructions. For instance, the Repeat String Operation Prefix repeats an instruction for a certain number of times. The prefixed instructions are decoded differently by the decoders.

Given the fact that various complex internal optimizations are performed by the processor and the disconnect between x86 instructions and micro operations, abstractions, even though imperfect, are necessary to simplify performance analysis when possible. Apart from the instruction level abstraction above, pipeline slots are utilized to abstract micro operation level behaviors. A machine with a pipeline width of x micro operations, referred to as an x-wide machine, is assumed to feed x micro operations to the ports in one CPU clock cycle. Thus, each clock cycle consists of x pipeline slots, each of which is available for an issued micro operation.

2.3.1 Stalls

A stall happens when the process of an instruction is undesirably halted because certain conditions are not satisfied and thus the program fails to fully employ the resources provided by the architecture. The life cycle of an instruction can be divided into two sections. The front end includes instruction fetching and decoding stages and the back end includes the remaining stages. Accordingly, a stall can be classified as a front-end stall or a back-end stall.

Figure 2.13 describes a stalled 2-wide pipeline. Instruction F is not fetched as expected in cycle 2 due to ineffective instruction fetching and therefore the stall caused by Instruction F is classified as a front-end stall. In addition, instruction E is stalled in cycle 4 because instruction C is still being executed. The stall of instruction E can be caused by dependencies on instruction C (e.g., one of the operands of instruction E is loaded from memory by instruction C) or by resource conflicts (e.g., both instruction C and E perform division, but there is only one divider).

Ahmad et al. [74] proposed a top-down method for performance analysis based on the pipeline model. Compared to the previous model, the top-down method focuses on micro-operation-level behaviors and thus provides better granularity. Figure 2.14 depicts seven stalled pipeline slots during the execution of 24 micro operations on a

Cycle	0	1	2	3	4	5	6	7
Fetch	A	C	E					
	B	D		F				
Decode		A	C	E	E			
		B	D		F			
Execute			A	C	C	E		
			B	D		F		
Write back				A		C	E	
				B	D		F	
Retire					A		C	E
					B	D		F

Figure 2.13: Stalled 5-stage pipeline.

uOp	A1		A2	A3	A4	A5		A6
uOp	B1	B2	B3		B4	B5		B6
uOp	C1	C2	C2		C3	C4		C5
uOp	D1	D2	D3	D4	D5	D6		D7
Cycle	0	1	2	3	4	5	6	7

Figure 2.14: Stalled slots in 4-wide pipeline.

4-wide machine. For each CPU cycle, there are four pipeline slots available. However, some pipeline slots are not filled with useful micro operations and are thus classified as stalled pipeline slots. Note that a micro operation can take multiple cycles to finish and the pipeline slot is not stalled as long as it is performing useful work. Figure 2.14 does not indicate the cause of the stalls. In fact, the causes of stalls are derived from the record of a series of Performance Monitoring Units (PMUs) [21, 22]. Such PMUs are added to the Intel x86 processors after the Ivy Bridge micro architecture [74]. In addition to front-end stalls and back-end stalls, particular PMUs are provided to detect stalls caused by bad speculations such as branch mispredictions.

2.4 Benchmarking

In this section, a discussion about the mechanism and accuracy of hardware performance measurement is presented in Subsection 2.4.1. Then, a brief discussion about benchmarking is presented in Subsection 2.4.2. Most work in this research is related to hardware performance measurement and benchmarking. Thus, an understanding of such topics is necessary.

2.4.1 Hardware Performance Measurement

As mentioned in Subsection 2.3.1, Performance Monitoring Units (PMUs) are built into processors to provide information about hardware behaviors. Although such units record a variety of different information from cache misses to retired pipeline slots, they are mostly implemented with two types of Model Specific Registers (MSRs). The Performance Event Select Registers are utilized by programmers to configure the hardware events measured by the PMU while the Performance Monitoring Counters (PMCs) perform the event counting. Some PMUs are fixed to measure particular events to reduce overhead while others are configurable to save resources inside the processor. Note that PMUs are model specific and differ across different CPU models, even if they are produced by the same manufacturer.

Counting and sampling are the two basic types of measurements supported by PMUs. Counting involves three steps. First, the PMCs are set to 0. Then, the program starts execution and the values stored in the PMCs are incremented whenever the desired events happen. When the program finishes execution, the values are read from the counters. The overhead of counting is low and the results are close to the actual values, but further details about the program are not revealed. In addition, event multiplexing is required when more events than the total number of PMCs are measured. Event multiplexing divides such events into different groups and each

event within the same group is measured by a different PMU. Each group of events is measured in a round-robin manner. To gather information about all required events, longer execution time or multiple runs are required.

Compared to simple counting, sampling provides more details about program behaviors. Sampling also includes three steps. First, the PMCs are set to specific values configurable by the programmer. An interrupt is triggered whenever the MSRs overflow during the execution of the program. A CPU routine handles the overflow by reading and storing the context information such as the program counter and then resets the PMCs. By adjusting the initial values in the PMCs dynamically, the measurements can be executed with a fixed frequency over time regardless of the fluctuations of CPU frequency. When the program finishes execution, the records of each overflow can be mapped to instructions of the program and thus performance bottlenecks can be investigated. The main challenge faced by event-based sampling is the inaccuracy caused by the delay between the overflow and the interrupt. Such a delay can cause an event being mistakenly mapped to an instruction not triggering the event. Such an error is called an event skid. In particular circumstances, an event skid does not influence the profiling result. For instance, analysis of Xu et al. [73] demonstrates that the profiling results of cycle-based events in a simple loop (i.e., a loop without conditional branching, except the one for loop control) are not affected by event skid because the number of events mapped to each instruction within the loop remains unchanged. To mitigate event skid, modern Intel x86 processors support Precise Event Based Sampling (PEBS) [21, 22] for particular events so that the processor can handle the overflow without an interrupt and thus the delay between the event and the read of the program counter is reduced. While not being entirely accurate, sampling remains the most reliable approach for developers to gain insight of performance bottlenecks in various applications.

Given the model-specific feature of PMUs, efforts have been put into abstracting the

details at hardware level and simplifying the performance measurement procedure. Embedded as part of the Linux kernel, Perf [16] is a profiling tool developed and maintained by the Linux community. Internally, Perf maps generic events such as CPU cycles to specific hardware events available on the platform and thus provides uniform interfaces for performance measurement across various micro architectures. Corresponding to the two basic measurements of PMUs, Perf provides event counting and event sampling functionalities. When sampling mode is utilized, developers can choose between a fixed sampling period and a fixed average sampling frequency. As mentioned, the former is achieved by setting a fixed initial value in the PMCs for every overflow while the latter requires adjusting such values dynamically so that overflows happen with a stable frequency. Perf also allows users to choose the hardware events directly.

Intel® VTune™ Profiler [6] is developed and maintained by Intel® as the profiling tool specialized for its processors. With knowledge about the hardware unknown to outsiders and a series of internal mechanisms to improve profiling accuracies, VTune provides more robust results utilizing the same set of PMUs. Compared to Perf, VTune provides further abstractions and thus users can focus more on the analysis of profiling results. For instance, VTune provides automatic event multiplexing so that a vast number of events can be profiled so long as the execution time is sufficiently long or multiple executions of the same application are available. VTune introduces top-down analysis as a feature in recent versions so that stall analysis is available if required PMUs are supported in the micro architecture.

2.4.2 Benchmarking for JVM Performance

Benchmarking is a common method for testing the performance of particular software implementations. A JVM benchmark, or a JVM workload, refers to a specific piece of code that is purposely designed or selected to run on the experimental JVM to

measure its performance or expose its potential problems. There are many indicators of performance. Among these indicators, throughput[5], memory usage[6] and response time[7] are commonly utilized. In general, there are two types of benchmarks: Micro benchmarks and Application benchmarks. Micro benchmarks are simple benchmarks measuring or illustrating the performance of the given JVM on a small piece of code. Application benchmarks are complex benchmarks emulating the complicated correlations between different components of the same application.
The difficulties of writing JVM benchmarks include the following:

1. Java is not the only programming language supported by the JVM. Many programming languages supporting different language features could also be translated into Java bytecodes and executed on the JVM. Such languages include Scala, Kotlin and Clojure. Even if a specific language is picked, there are various application frameworks. A view from the application level is not convincing enough to illustrate the generality of a benchmark.

2. Dynamic optimization increases the difficulty of predicting the runtime behaviors of the JVM. Certain benchmarking tools such as JMH [10] use extra code to increase the predictability of the JVM behaviors, but such interference also introduces noise into the analysis of low level information. Such interference might also reduce the performance of the JVM and generate misleading results.

Therefore, a view from the hardware level, provided by stall-focused benchmarks and measured by PMUs, improves the generality and efficiency of the JVM developers' work.

[5]Throughput is often measured by execution time.
[6]Memory usage is measured by the maximum amount of memory allocated during the execution of the Java application.
[7]Response time is measured by the amount of time between a request to the application is sent and the corresponding response is received.

Chapter 3

A Survey of Stalls in Java Applications

Stalls are important bottlenecks of modern computer programs, because they prevent taking full advantage of the CPU's high processing speed and decrease the benefits of multi-processing. To know about the types, frequencies and effects of stalls in Java applications, a survey on potential stalls in a variety of Java application benchmarks is carried out. This chapter is organized as follows: Section 3.1 describes the investigated benchmarks and methods to identify and categorize potential stalls. Section 3.2 shows information about the benchmarking machine and configurations of application frameworks. Section 3.3 discusses the experimental results.

3.1 Methodology

Since the objective of this survey is to provide foundations for the stall-focused benchmark design, it is vital to test a variety of programs exhibiting different features. Some of these benchmarks are still considered even though they were not originally designed to test JVM performance. The survey relies on PMUs to provide hardware information. The tested benchmarks include the following:

1. The HiBench benchmark suite [7, 45] includes a series of big data programs. It is widely utilized to measure performance of distributed systems. All benchmarks in HiBench are based on the popular MapReduce [35] programming model and are implemented with the Apache™ Hadoop® [64] and the Apache Spark™ [57, 76] frameworks. In this survey, the programs in the machine learning category are tested because they feature computation-intensive workloads with frequent memory accesses. In these applications particular methods are much more frequently invoked than others and are thus referred to as hot methods. The JVM focuses its effort to optimize these methods to a high optimization level to achieve better steady-state performance.

2. The DayTrader7 benchmark [9] is a Java EE application [11] running on IBM® WebSphere® Liberty [19] server. It implements an online stock trading system, which supports user login/logout, viewing user information, searching stocks, buying and selling stocks. It is frequently utilized to evaluate Java platform performance. Internally, the DayTrader7 benchmark utilizes Apache Derby [2], a relational database implemented in Java, to store data.

3. The Acme Air benchmark [1] is another Java web application running on the same platform. It implements a fictitious airline application supporting some fundamental functionalities such as user login/logout, flight search, flight booking and cancellation. The workload is built with key business requirements for the cloud such as scalability. It utilizes MongoDB [13], a document-oriented database implemented in C++, to store data. Both the DayTrader7 and Acme Air benchmarks feature Java web applications where most methods are compiled to a low or medium optimization level. Such workloads are referred to as flat workloads. For such applications the JVM is required to compile as many methods as fast as possible within the limited time of the start-up phase.

4. Renaissance Suite [17, 62] is a benchmark suite designed specially for JVM performance measurement. Renaissance focuses on concurrent programs and includes multiple open source applications written in Java and Scala. A variety of up-to-date concurrent programming models, such as fork-join [51] and actors [42], are utilized in these programs. These models are implemented with several real-world application frameworks. Renaissance is included in this survey for its diverse and up-to-date workloads.

5. The SPECjvm® 2008 benchmark suite [63] focuses on different aspects of JVM performance including start-up time, compilation time and steady-state performance. The performance of such applications depends on the JVM's ability to manage processors and memory rather than file input/output and network throughput. The benchmark suite has been widely utilized for optimization validation and evaluation because of its real life and area-focused workloads.

In the survey, Perf is utilized to obtain the overhead, measured in the number of consumed CPU cycles, of each method and instruction when the programs are executed by OpenJ9. To reduce profile overhead and increase accuracy, the number of CPU cycles is the only measured hardware event. To acquire the compilation information of a set of methods, symbols consisting of the methods' signatures and the corresponding optimization levels must be passed explicitly as compilation arguments to the JVM. Depending on a variety of factors affecting the compiler's decisions, the same method can be optimized to different optimization levels in different executions of the same program. In addition, a method might be inlined, causing Perf to attribute the overhead of the method to its caller. The profile results of such executions are very different from those where the method is not inlined. Therefore, every tested program is executed without any compilation arguments for five times to obtain the most frequent result as follows. For each execution, a list of the top 10 CPU-cycle consuming symbols is recorded. For programs with hot methods

consuming more than 10% of the total CPU cycles, the size of the list is reduced. Only the symbols appearing in such lists for more than three times are tracked in the sixth execution where the CPU cycle consumption information from Perf, along with the compilation information provided by OpenJ9, are acquired. For flat workloads, only methods consuming more CPU cycles than the interpreter are tracked. For the Acme Air benchmark, a candidate to be tracked should also consume more than 0.4% of the total CPU cycles, which is the average median overhead of methods consuming more CPU cycles than the interpreter. This additional condition is set because more methods in the Acme Air workload are compiled and oversampling should be avoided.

After a series of records is obtained, instructions consuming more CPU cycles than others, referred to as hot instructions, are inspected. The number of instructions inspected varies from method to method depending on the compilation. For instance, loops can be unrolled by the compiler and thus hot instructions are duplicated. After the hot instructions are extracted, a procedure depicted in Procedure 1 is applied to classify each of them as one of the four categories: REP_STRING_OPERATION, POTENTIAL_BACK_END_STALL, POTENTIAL_FRONT_END_STALL and OTHER. Note that this classification is not definitive but rather speculation based on the instructions before the hot instruction.

First, the hot prefixed-instructions are filtered out. An x86 instruction can contain multiple prefixes, but the only two prefixes observed in the experiment results are the LOCK prefix for atomic operations and REP prefix for repeated string operations. Among such instructions, repeated string operations are classified as a separate category because they possibly cause stalls depending on the situation. The reason is that such instructions are decoded by a special unit called the micro sequencer [21] and the switches to and from this unit introduce bubbles in the pipeline. Such penalties can be ignored when the execution time of the instruction, determined by the

Procedure 1 Classification of an Instruction.

Input: *the* X^{th} *instruction* $Inst_X$
Output: *the category of* $Inst_X$

```
procedure CLASSIFY(Inst_X)
    if Inst_X is prefixed then
        if Inst_X is repeat string operation then
            return REP_STRING_OPERATION
        else
            return OTHER
        end if
    else
        B ← the Basic block of Inst_X
        isUnexpected ← isUnexpected(B)
        i ← 0
        repeat
            Y ← X − i
            Inst_Y ← the Y^th instruction
            if dependsOn(Inst_X, Inst_Y) and Inst_Y reads from memory then
                return POTENTIAL_BACK_END_STALL
            else if Inst_Y is entry of B and isUnexpected then
                return POTENTIAL_FRONT_END_STALL
            end if
        until i = C or Inst_Y is entry of B
        return OTHERS
    end if
end procedure
```

operands of the instruction, is sufficiently long. However, such information is missing in the profile and thus these instructions are classified as an independent category. Atomic operations are not associated with stalls in this survey because the execution halt is a desired functionality.

For instructions without prefixes, the procedure repeatedly checks whether they are potential back-end stalls or potential front-end stalls. An instruction Inst_X is classified as a potential back-end stall if there is another instruction Inst_Y such that Inst_X depends on Inst_Y and Inst_Y reads from memory. The logic is depicted in Procedure 2. For instructions without prefixes, it is assumed that front-end stalls are caused only when access of basic blocks are unexpected and instruction prefetching fails. To

Procedure 2 Output whether Instruction $Inst_X$ depends on Instruction $Inst_Y$.

Input: *Instruction* $Inst_X$, *Instruction* $Inst_Y$
Output: **TRUE** *if* $Inst_X$ *depends on* $Inst_Y$, **FALSE** *otherwise*

```
procedure DEPENDSON(Inst_X, Inst_Y)
    if Inst_X = Inst_Y then
        return FALSE
    else
        Src_X ← source operands of Inst_X
        Dest_X ← destination operands of Inst_X
        Src_Y ← source operands of Inst_Y
        Dest_Y ← destination operands of Inst_Y
        if Src_X ∩ Dest_Y ≠ ∅ or Dest_X ∩ Src_Y ≠ ∅ or Dest_X ∩ Dest_Y ≠ ∅ then
            return TRUE
        else
            return FALSE
        end if
    end if
end procedure
```

determine whether an instruction is a potential front-end stall, information provided by the control flow graph, where the access frequencies and branching of different code blocks are recorded, is required. In the control flow graph implementation of OpenJ9, each vertex represents a code block that can be reordered and each edge represents a branch. Note that internal control flows exist in some code blocks. These control flows do not split the vertex. A basic block, which starts from single entry and ends with single exit, can appear in an outlined snippet or main line code. If a basic block resides in a snippet then its access is estimated to be unexpected since snippets are utilized to handle rare situations such as exceptions. When a basic block resides in main line code, it is either represented by a node in the control flow graph or within an internal control flow. If the information about the access frequency of the basic block is available then the frequency of instruction prefetching failures can be estimated based on the assumption that the hardware always prefetches the most frequently accessed code after a conditional branch. If the basic block is correctly prefetched then an instruction cache hit happens. In contrast, an instruction cache

miss happens when the basic block is predicted to not be accessed but actually is accessed. In this case a front-end stall happens since extra time is spent in fetching the correct code. When the basic block is inside an internal control flow of a fixed routine, then the compiler does not track nor reorder the basic block. Therefore, information about access frequencies is not available. In this case the estimation is based on the static heuristics that the target of a backward branch is always correctly prefetched. A backward branch refers to a conditional branch targeting an address lower than the address of the branch. Such branches often indicate the existence of loops and thus are predicted by the hardware to be taken. Procedure 3 describes the logic to estimate whether the access of a basic block is unexpected.

3.2 Experimental Environment

The specifications of the experimental machine are listed in Table 3.1. For all experiments, different executions of the same program are pinned to the same NUMA node using the numactl [14] command when possible. For all experiments, OpenJ9 JVM with OpenJDK version 1.8.0_232 is utilized to execute the program.

For the HiBench benchmarks, Hadoop version 2.7.7 and Spark version 2.3.3 are utilized. Since there is only one experimental machine, the Hadoop cluster contains only one node where all Hadoop and Spark Java processes reside together. Therefore, there is no network related overhead within the system. The executor process of Spark is the only process measured by perf since it performs all computations and consumes most resources of the cluster. Since every executor process is an independent Java process unaware of other executor processes, only one executor process is created. Four cores and 4GB memory are assigned to the executor as the maximum amount of available resources. For each benchmark, there are six configurable input dataset sizes available: tiny, small, large, huge, gigantic and bigdata.

Procedure 3 Estimate whether the access of a basic block is Unexpected.

Input: *A Basic Block* B *, Control Flow Graph* $CFG = (V, E)$
Output: **TRUE** *if the access of* B *is unexpected,* **FALSE** *otherwise*

```
procedure isUnexpected(B)
    if B is outlined then
        return TRUE
    else if B is represented by node s in CFG then
        C ← {pair(u, v) | u, v ∈ V ∧ (u, s), (u, v) ∈ E}
        for all pair(u, v) ∈ C do
            if CFG.getFrequency(u, s) ≤ CFG.getFrequency(u, v) then
                S_miss ← S_miss + CFG.getFrequency(u, s)
            else
                S_hit ← S_hit + CFG.getFrequency(u, s)
            end if
        end for
        if S_hit > 4S_miss then
            return FALSE
        else
            return TRUE
        end if
    else
        if B is target of backward branch or unconditional branch then
            return FALSE
        else
            return TRUE
        end if
    end if
end procedure
```

To ensure a sufficiently long execution time so that the program spends most of its lifetime in the steady state, only the large or the huge dataset size is utilized for each benchmark. The tested programs are: Bayesian Classification (Bayes), K-means clustering (KMeans), Logistic Regression (LR), Alternating Least Squares (ALS), Gradient Boosted Trees (GBT), Linear Regression (Linear), Latent Dirichlet Allocation (LDA), Principal Components Analysis (PCA), Support Vector Machine (SVM) and Singular Value Decomposition (SVD). The descriptions of these programs can be found on the website of the benchmark suite [7].

For the DayTrader7 benchmark and the Acme Air benchmark, Apache JMeter™, a

Table 3.1: Specifications of experimental machine.

Architecture	x86_64
CPU op-mode	64-bit
NUMA nodes	4
CPUs per NUMA node	8
Model Name	Intel(R) Xeon(R) CPU E7520@1.87GHz
Code Name	Nehalem EX
CPU maximum Frequency	1862MHz
CPU minimum Frequency	1064MHz
Threads per core	2
L1d cache size	32KB
L1i cache size	32KB
L2 cache size	256KB
L3 cache size	18432KB
memory size	148.52 GB
SIMD	MMX, SSE, SSE2, SSSE3, SSE4_1, SSE4_2
Operating System	Linux Ubuntu 18.04.5 LTS
Kernel Version	#127-Ubuntu SMP Fri Nov 6 10:54:43 UTC 2020
Kernel Release	4.15.0-124-generic

load testing tool, is utilized to simulate users accessing the websites. The frequencies of various types of client requests are embedded in the benchmark design and thus are not configured in the experiments. Each user is simulated by a Jthread[8] and multiple requests are sent during its lifetime. There are 50 Jthreads for the DayTrader7 program and 100 Jthreads for the AcmeAir program.

For Renaissance and the SPECjvm2008 benchmarks, the programs are packed in executable .jar files. The executed workload can be specified simply by passing its name as an argument to the .jar file. The tested programs of Renaissance are: akka-uct, fj-kmeans, future-genetic, mnemonics, rx-scrabble and finagle-http. The tested programs of SPECjvm2008 are: compress, crypto.aes and derby. The complete descriptions of these programs can be found in the documentation of the suites [17, 63].

[8]JMeter terminology.

Table 3.2: Survey results.

	HiBench	DayTrader7	Acme Air	Renaissance	SPECjvm2008
total hot methods	19	6	13	21	5
total hot instructions	151	48	49	97	27
total potential stalls	71	36	30	55	15
potential back end stalls					
total count	57	34	28	46	14
affected methods	14	6	11	16	5
average impact	37.29%	39.19%	17.87%	25.44%	30.75%
maximum impact	92.13%	64.50%	41.49%	78.15%	83.83%
minimum impact	4.80%	18.99%	3.10%	3.38%	12.81%
potential front-end stalls					
total count	14	2	2	9	1
affected methods	5	2	2	7	1
average impact	7.12%	4.09%	5.71%	7.36%	2.87%
maximum impact	27.15%	4.16%	7.97%	15.46%	2.87%
minimum impact	1.33%	4.01%	3.44%	3.87%	2.87%
rep string operations					
total count	6	0	7	1	0

The impact of an instruction on its method is measured in the proportion of the instruction's overhead to the method's overhead. Both overheads are measured in the total number of consumed CPU cycles.

3.3 Experimental Results

The experimental results are summarized in Table 3.2. To measure the influence of a potential stall on its method, the proportion of the overhead of a stall to that of its corresponding method is calculated. The overhead of each instruction and method is measured in the number of consumed CPU cycles. Since the overhead of bubbles caused by repeat string operations remains unknown, such instructions are not regarded as stalls and their influence on the corresponding methods are not calculated.

As we can see from Table 3.2, a non-trivial proportion of hot instructions are potential stalls. Such potential stalls affect a majority of hot methods in every tested benchmark suite. While the frequency and impact of potential stalls vary from case to case, potential back-end stalls are more frequently observed and exert more influence on the methods' overhead when compared to potential front-end stalls.

In total, there are 19 hot methods and 151 hot instructions inspected in the HiBench machine learning benchmarks. We can see 73 out of such 151 instructions are classified as potential back-end stalls. These potential stalls exist in 14 out of 19 hot methods and their impact on the methods ranges from 4.8% to 92.13% with an average of 37.19%. A further investigation in the compilation information reveals that such stalls are caused by traversing arrays and dereferencing objects. Five hot methods are affected by 14 potential front-end stalls which cause 1.33% to 27.15% of the methods' overhead. An average 7.12% overhead indicates front-end stalls are not as influential as the back-end stalls.

Six hot methods are investigated in the DayTrader7 program. There are 36 potential stalls within the 48 inspected hot instructions. Among them, 34 are classified as potential back-end stalls. The impact of such instructions ranges from 18.99% to 64.5%. Compared to the HiBench programs, some potential back-end stalls are observed to be caused by the type-checking routine. This is no surprise considering that polymorphism is heavily utilized in web applications. Similar experimental results exist in the Acme Air program, where 30 out of 49 of the hot instructions are stalls, among which 28 stalls are potential back-end stalls. The impact ranges from 3.1% to 41.49%. In both programs, a majority of hot methods are influenced by potential back-end stalls. In contrast, only a few potential front-end stalls are observed in both benchmarks. None of them exert significant impact compared to the potential back-end stalls.

For the Renaissance programs, 55 out of 97 instructions are classified as potential stalls. Among them, 46 are classified as potential back-end stalls. They cause 3.38% to 78.15% overhead in 16 methods. On the other hand, nine potential front-end stalls are observed in seven methods with overhead ranging from 3.87% to 15.46%.

For the SPECjvm2008 benchmarks, five hot methods are inspected where 15 potential stalls are observed among 27 hot instructions. Most of the potential stalls are

potential back-end stalls which cause 12.81% to 83.83% of overhead in the methods. With the above experimental results, we can draw the following conclusions:

1. Potential stalls are very common. In each benchmark, stalls are observed to exist in a significant proportion of hot methods.

2. Stalls account for a non-trivial part of overhead measured in the number of CPU cycles.

3. Potential back-end stalls are more frequently observed than potential front-end stalls.

4. Potential back-end stalls exert more influence to methods than potential front-end stalls.

5. The causes of potential back-end stalls vary according to the programs' behaviors. For computation-intensive workloads such as the HiBench machine learning programs, traversing arrays and dereferencing objects cause potential back-end stalls. In web applications where polymorphism is heavily utilized, type checking routines cause potential back-end stalls.

With the results from this survey, we are able to obtain an overall view about the types, frequencies and effects of stalls in Java applications. In Chapter 4, a set of benchmarks are designed with the guidance of the conclusions in this survey.

Chapter 4

Design

Chapter 3 provides an overview of the types, frequencies and effects of stalls in Java applications. In this chapter, the design of stall-focused benchmarks is discussed. This chapter is organized as follows. Section 4.1 discusses the advantages and disadvantages of Application benchmarks and Micro benchmarks and the motivations to implement stall-focused benchmarks as Micro benchmarks. Section 4.2 describes the basic utilities shared by all stall-focused benchmarks. Section 4.3 presents different memory access patterns and the corresponding benchmarks. Section 4.4 shows the benchmark addressing stalls caused by type checking. Section 4.5 discusses the benchmark addressing stalls caused by the utilization of huge objects. Section 4.6 discusses the utilization of repeated string operations in array initialization.

4.1 Why Micro Benchmarks?

As mentioned in Subsection 2.4.2, there are two categories of benchmarks: Application benchmarks and Micro benchmarks. Application benchmarks simulate complex applications in the real world where multiple modules coexist and interact with each other during execution. The benchmarks tested in the survey of Chapter 3 are typical examples of application benchmarks. For instance, both the DayTrader7 and

the Acme Air benchmarks are implemented with layered architectures[9]. Each layer interacts with the adjacent layers to generate a response to each user request. The advantage of Application benchmarks is obvious: they are implemented with commonly utilized architecture patterns and share many other similarities with real life applications. Thus, Application benchmarks are considered practical. In addition, Application benchmarks are whole applications, within which many integral modules are built. Such benchmarks provide a perspective on how different modules affect each other during execution. The disadvantage of Application benchmarks is that their complexity increases the difficulty to obtain reproducible results. As mentioned in Section 3.1, the same method might be optimized to different levels in different executions. During some executions the method might be inlined while in others remain separate. Another problem of Application benchmarks is that they are no longer practical when their programming models become outdated. For instance, one of the motivations for the Renaissance benchmarks is to include programming models not implemented in previous benchmarks [62].

In contrast to Application benchmarks, Micro benchmarks simulate the simple logic of a single module or even a single function. Their simplicity reduces efforts to obtain stable, reproducible results. Another advantage of Micro benchmarks is the predictability of program behaviors when inputs are changed. Since Micro benchmarks do not simulate the complex interactions between different modules, unexpected influence of input changes is avoided. Therefore, various program behaviors, such as memory consumption and execution time, can be controlled by a set of independent parameters. The main issue of Micro benchmarks is that they might not reflect the real situation of an application. For instance, a module might not be frequently accessed in applications and thus is not highly optimized by the compiler in such cases. In addition, Micro benchmarks cannot provide a global perspective of an applica-

[9]The layered architecture is a software architecture pattern where business logic is implemented in loosely coupled layers and each layer can only interact with the adjacent layers.

Table 4.1: Comparison of Application benchmarks and Micro benchmarks.

	Application benchmarks	**Micro benchmarks**
logic	business logic	single functionality
scope	complete application	single module
interactions between modules	abundant	no/low
result reproducibility	low	high
result predictability	black box	white/grey box
practicality	close to particular real-world applications	single module within application
lifespan	depends on the application model	depends on the illustrated problem

tion and thus cannot be utilized to test certain abilities of the JVM. For instance, a Micro benchmark can hardly test a JVM's ability to prioritize different methods to optimize. Given such weaknesses, a clear understanding of the problem illustrated by the benchmark and sufficient background research to guide the benchmark design is crucial to Micro benchmarks. The comparison of Application benchmarks and Micro benchmarks is summarized in Table 4.1.

Considering the advantages and disadvantages of the two categories of benchmarks, it is vital to understand the granularity of the problem and what information the benchmarks are to provide. The experimental results from Chapter 3 show that the cause of a stall can be identified within the five instructions before the stalled one in many cases. In addition, modules performing similar functionalities share similar stalls in different benchmarks. For instance, the type checking routine causes stalls in both the web application benchmarks and the Renaissance benchmarks. Another example is that traversing integer arrays in the HiBench programs cause the same stalls as those caused by traversing byte arrays in the SPECjvm2008 programs. Therefore, it is reasonable to assume that the perspective of a module is sufficient to illustrate stalls. The survey also provides information about the types, frequen-

cies and effects of stalls in a variety of Java applications. Therefore, the designed benchmarks, driven by the survey's results, reflect stalls in real life applications.

4.2 Common Framework

This section describes the common framework shared by all stall-focused benchmarks. Subsection 4.2.1 describes the benchmarking procedure and a light-weight framework to execute and measure stall-focused benchmarks. Subsection 4.2.2 discusses how parameters are utilized to control the behaviors of benchmarks. In Subsection 4.2.3, a discussion about the proper utilization of randomization is presented.

4.2.1 Benchmarking Procedure

In general, the benchmarking procedure can be divided into three phases: the set-up phase, the execution phase and the end phase. During the set-up phase, preparations, such as creating the environment where the benchmark is executed and generating inputs, are performed. In this phase, the profiler does not collect performance data. Warm-up executions can also be included in the set-up phase so that the program reaches the steady state before the measurement begins. When the set-up phase ends, a profiler, such as perf, is activated and the benchmark workload is executed. In this phase, performance data is gathered by the profiler. If multiple iterations of the same workload are measured in this phase, the inputs for each iteration should be modified so that the results differ from iteration to iteration. Thus, the workload is guaranteed to be executed for exactly the desired number of iterations. Finally, the profiler stops collecting data when the execution phase ends and the results of the workload are consumed in the end phase. Validation for the program's correctness can be performed in the end phase. Note that the results of the workload must be consumed in this phase so that the execution of the benchmark workload is not

```
public abstract class Benchmark {
    ...
    protected abstract void setUpUtil();
    public abstract void run();
    protected abstract void endUtil();
    public void setup() {
        setUpUtil();
        startProfile();
    }
    public void end() {
        stopProfile();
        endUtil();
    }
    ...
}
```

Figure 4.1: Benchmark abstract class.

treated as dead code and is thus not omitted by the JVM.

To apply the benchmarking procedure above to all experiments, the Benchmark abstract class serves as the base class of all benchmarks in the designed benchmark suite. It possesses three abstract methods, which should be overridden by its derived classes, to wrap the mentioned three phases. The sketch of this abstract class is depicted in Figure 4.1.

To activate the profiler, a process is built by the program with the command to attach the profiler to the benchmark JVM process. Both perf and VTune are able to attach to an already running process when the process id is passed as argument. When the execution phase ends, the benchmark process sends a signal to terminate the profiling process so the profiler stops collecting data and generates the corresponding reports. This procedure is shown in Figure 4.2.

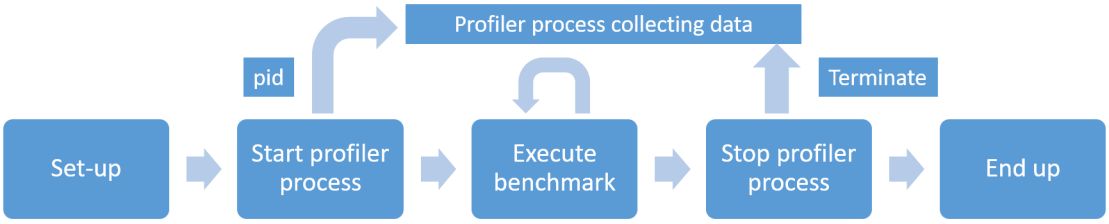


Figure 4.2: Benchmarking procedure.

4.2.2 Parameters

As mentioned in Section 4.1, one of the benefits of the Micro benchmark design is the parameterization of program behaviors. For the stall-focused benchmarks, the parameters of each workload are divided into the following categories:

1. Functional parameters control the logic performed by the workload. In the type-checking benchmark, for example, a shared interface method is invoked for each object in an array. Each object is an instance of a different class implementing the same interface. The proportions of each class in the array are controlled by a set of parameters. Adjusting the values of such parameters can change the behavior of the program when type checking is applied to each instance.

2. Iteration parameters control the number of times each workload is repeatedly executed. Therefore, the time span of the execution phase is configured without consuming more hardware resources, such as memory. For some benchmarks, where setting up particular inputs between different iterations is time-consuming, an extra iteration parameter is utilized so that the workload can reuse such inputs multiple times. Thus, time spent on modifying the inputs is reduced and the workload can be repeated for additional iterations within a plausible time span.

3. Footprint parameters are utilized to adjust the amount of particular hardware

resources consumed by the workload. For instance, configuring the size of matrices in a matrix algebra benchmark changes the memory consumption of the workload. Therefore, the benchmark can adapt to different machines with various cache capacities. In addition, a combination of iteration parameters and resource parameters ensures that the workload is optimized to the desired level within a reasonable warm-up time.

Ideally, each of these parameters affects only one of the program behavior features during execution. However, side effects still exist. For instance, increasing memory consumption often increases execution time inevitably. Despite such side effects, we still try to utilize these parameters to control the benchmark program in an intuitive manner.

4.2.3 Randomization

Randomization is often utilized in various benchmarks. The use cases can be described as the following:

1. The benchmark program itself implements a nondeterministic algorithm and thus involves randomized factors. For instance, many machine learning algorithms, such as logistic regression, require a random initial point where the program starts an iterative procedure until the result converges. Depending on the initial point and other factors, such as the convergence criteria, the number of iterations varies. The original purpose of such randomization is to prevent the learning result from overfitting. When the learning program is utilized as a benchmark, such randomness increases difficulties in reproducing benchmarking results. Therefore, analysis of the results and evaluation of the tested system become more difficult and less convincing.

2. The benchmark workload requires random inputs to simulate real-time situa-

tions. For instance, the TPC Benchmark™ H (TPC-H) [18], a decision support benchmark utilized to measure database system performance, randomizes its queries to simulate real-life use cases.

3. The values of particular inputs are not influential and thus can be randomly generated. In this case, the benchmark workload itself is often not sensitive to these inputs and the randomized inputs serve as place-holders. For instance, the names of users in the DayTrader7 program are randomly generated. These names do not influence the evaluation so long as they do not cause variations in the program behavior and the evaluated systems are not able to hack the benchmark.

In the design of stall-focused benchmarks, the effect of randomization needs to be considered. In particular, randomized inputs affect the access frequencies of code blocks and the performance of the front end. For instance, the branch prediction in function *f*, as depicted in Figure 4.3, is extremely inaccurate and thus the function is frequently stalled to fetch the correct code when arrays A and B are randomized. Such stalls cannot be optimized and affect the evaluation for other optimizations.
To solve such issues, two approaches can be considered. Firstly, the seed to generate random values in a and b can be fixed. Thus, array A and B remain unchanged in different executions of the function. Although the predictions of the branch prediction unit still vary, the access frequencies of the two code blocks remain unchanged and the experimental results become more reproducible. Secondly, the elements of A can be modified such that one branch is always taken. Thus, the impact of branching can be minimized. Both approaches are available so long as branching is required to be excluded from the measurement and the tested workload is not aimed at real-world simulation.

```
public void f(int[] A, int[] B){
    int i = 0; int j = 0;
    while(i < A.length && j < B.length){
        if(A[i] <= B[j]){
            i++;
            ...
        }
        else{
            j++;
            ...
        }
    }
}
```

Figure 4.3: A function heavily affected by randomized inputs.

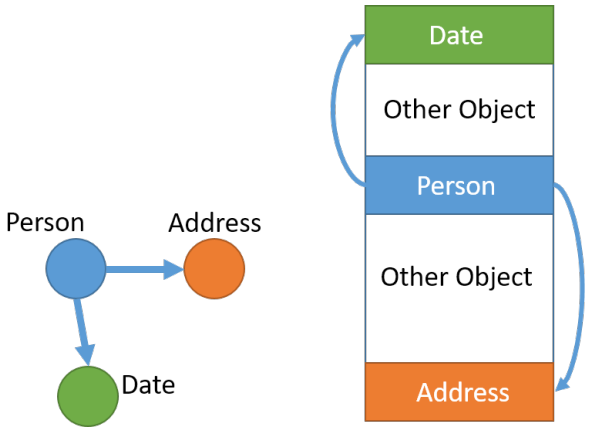
4.3 Object Access

From this section, we start the discussion of each stall-focused benchmark workload's design. In this section, a set of benchmarks reflecting back-end stalls caused by various object accesses in Java programs is presented. As the experimental results in Section 3.3 suggest, back-end stalls are common and influential in various types of Java applications.

The section is organized as follows: a discussion about object access patterns is shown in Subsection 4.3.1. Subsection 4.3.2 presents a matrix calculation workload implemented with primitive types in Java. Then, the program performing the same calculation on wrapper objects is depicted in Subsection 4.3.3. Subsection 4.3.4 discusses a workload performing calculation on sparse matrices. Finally, Subsection 4.3.5 presents a program searching a particular node in an N-ary tree.

4.3.1 Object Access Patterns

One of the difficulties in designing JVM benchmarks is the diversity of Java objects. Depending on the application, various types of objects are utilized with different



(a) Object graph (b) Possible memory layout

Figure 4.4: An object graph example and its possible memory layout.

frequencies. For instance, primitive arrays are heavily utilized in HiBench workloads while a series of complicated Bean[10] objects serve as components in the web application benchmarks. Therefore, abstraction should be utilized to depict various types of objects and the relationships between them. In particular, the reference relationship is of special interest because of its effect on cache efficiency and back-end performance.

Object graphs is a representation commonly utilized in Garbage Collection. Each vertex in the graph represents an object. A directed edge from vertex A to vertex B exists if and only if object A references object B. Therefore, the objects depicted in Figure 2.6 in Subsection 2.1.4 can be represented by the object graph shown in Figure 4.4a. A possible layout of the graph in memory is shown in Figure 4.4b. Note that the three objects are not adjacent to each other and thus cache misses are likely to happen when Address or Date are accessed from Person. Such cache misses cause back-end stalls and the pipeline is forced to halt.

With the graph representation of objects, we can categorize the memory access from one object as one of the following categories:

[10] A J2EE Enterprise Bean encapsulates particular business logic in a J2EE application.

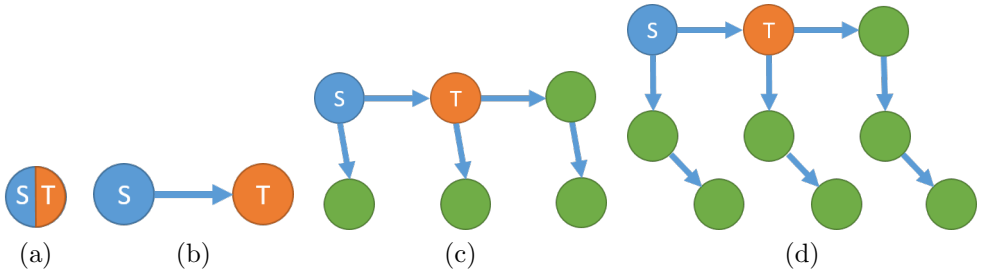


Figure 4.5: Access object T from object S.

1. Accessing the data within the same vertex, as shown in Figure 4.5a. Both accessing an element within the same array and accessing a primitive type (e.g., int, float, etc.) field within the same object fall into this category.

2. Accessing a sink, which is a vertex with zero out-degree, as presented in Figure 4.5b. There is no edge from such vertices because the objects they represent do not reference other objects. A typical example of such objects is wrapper objects, such as the Integer objects provided by Java language package.

3. Accessing a vertex with only one edge to a non-sink vertex. This is depicted in Figure 4.5c. A typical example is accessing a linked list of Integer objects.

4. Accessing a vertex with multiple edges to non-sink vertices, as shown in Figure 4.5d. Accessing a hierarchical structure, such as a tree, falls into this category.

4.3.2 Dense Matrix Algebra using Primitive Types

This benchmark is a single-threaded program executing matrix calculations as described in Figure 4.6. It represents pattern 1 described in Subsection 4.3.1 where the data within the same object is accessed. For the discussion here, we mainly focus on traversing an array of primitive type elements. A primitive type variable is not an

```
class PrimitiveMatrix extends DenseMatrix{
    private double[][] values;
    public double get(int i, int j){
        return values[i][j];
    }
    public void set(int i, int j, double v){
        values[i][j] = v;
    }
}
...
DenseMatrix M1, M2, M3;
public void matMultiplyAdd() {
    for (int i = 0; i < m; i++) {
        double sum = 0;
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < r; k++){
                double a = M1.get(i, k) * 0.2;
                double b = M2.get(j, k) * 0.125;
                sum += (a + b);
            }
            double v = M3.get(i, j) * 0.4
                        + sum * 0.6;
            M3.set(i, j, v);
        }
    }
}
...
```

Figure 4.6: Dense matrix algebra using primitive types.

object and thus is not accessed via a reference. Instead, it is embedded in its containing object or placed on the stack. When an array of primitive type elements is traversed, cache misses are less likely to occur since the elements are placed adjacent to each other. Prefetching is also very efficient because the data block to be accessed is highly predictable and does not contain redundant data such as object headers. Note that the DenseMatrix class is utilized as an abstraction in the workload because the same calculation is applied to another implementation of dense matrices, which is discussed in Subsection 4.3.3. Since the JIT compiler can detect that only one implementation is utilized in the workload and perform optimizations accordingly,

the abstraction does not exert noticeable influence on the workload after the execution reaches the steady state. Also note that matrix M2 is transposed in advance because a 2-dimensional array in Java is implemented as an array of references to 1-dimensional arrays. Thus, accessing elements within the same column does not comply with pattern 1 and the matrix is transposed.

After each iteration, M2 is modified. To ensure reproducible results, a seeded random generator can be utilized to generate the elements in each matrix. Apart from iteration parameters controlling the number of repeated executions, parameters are exposed to adjust the size of M1 and M2 so that memory consumption is configured. Another use case representing pattern 1 in Subsection 4.3.1 is discussed in Section 4.5.

4.3.3 Dense Matrix Algebra using Wrapper Objects

Although primitive types consume less memory and allow more compact layout, there are multiple short-comings. For instance, a primitive variable cannot be locked and cannot be utilized as type arguments of generic types. A generic type is a class or interface that accepts types as parameters [41]. Many Java packages utilize generic types to support generic functionalities. Internally, type arguments of generic types are implemented as objects. Thus, the utilization of primitive types are limited. In addition, Java is an Object-Oriented programming language implementing the "everything is an object" idiom. Therefore, data is often wrapped in objects in Java applications. In fact, the Java language package provides wrapper objects for each primitive type. For instance, an Integer object encapsulates a primitive integer into an object and thus it can be passed as type arguments to generic types such as ArrayLists.

To reflect such use cases and provide a workload for pattern 2 described in Subsection 4.3.1, the program of Subsection 4.3.2 is applied to dense matrices implemented with wrapper objects, as shown in Figure 4.7.

```
... class ArrayList<E>...{
    ... Object[] elementData;
    ...
}
... class Double...{
    ... final double value;
    ...
}
...
public class WrapMatrix extends DenseMatrix{
    private ArrayList<Double[]> values;
    public double get(int i, int j){
        values.get(i)[j];
    }
    public void set(int i, int j, double v){
        Double wv = Double.valueOf(v);
        values.get(i)[j] = wv;
    }
}
...
```

Figure 4.7: Dense matrix implementation using wrapper object.

This implementation stores the reference to each row of a matrix within an instance of ArrayList. The generic class ArrayList encapsulates an array of references and provides a series of common operations for abstract lists such as inserting, removing and modifying list elements. A Double object encapsulates an immutable primitive double-precision floating point value and provides methods to handle it in an object-oriented manner.

Compared to the implementation depicted in Subsection 4.3.2, this implementation consumes more memory because of the references and object headers. The cache efficiency is also lower even if the wrapper objects are placed adjacent to each other, because part of the cache block is occupied by object headers. Prefetching is more difficult because the reference to the wrapper object must be read before fetching the encapsulated primitive type value.

Object inlining [36, 37, 72], which replaces object references with the correspond-

ing object's data, is a possible optimization to mitigate back-end stalls caused by dereferencing wrapper objects. Unlike prefetching, a series of prerequisites must be satisfied before object inlining can be applied. For instance, the inlined object must not be an instance of an inheritable class. While object inlining can be instructed by the developer with support from the JVM, such as the packed-object support provided by OpenJ9, automatic object inlining complies more with the paradigm of JVMs. Although efforts [36, 72] have been applied, automatic object inlining is still not a feature in most industrial JVM implementations.

4.3.4 Sparse Matrix Algebra using Linked Lists

The matrix implementations described in Subsection 4.3.2 and Subsection 4.3.3 are both dense matrix implementations. A dense matrix is a matrix in which most of the elements are not 0s. On the other hand, a sparse matrix is a matrix in which most of the elements are 0s. In practice, a sparse matrix often contains a huge number of elements. Therefore, utilizing arrays or ArrayLists to store each element in the matrix will unnecessarily consume a significant amount of memory. A more memory efficient implementation is storing elements in linked lists, as depicted in Figure 4.8. A sparse matrix contains a linked list of rows. Each row is identified with a unique row number and contains a linked list of Pair objects. Each Pair object represents a non-zero element in the row with the column number and the value of the element. The implementation is a typical example of pattern 3 described in Subsection 4.3.1. Each node in the linked list is a vertex with only one edge to a non-sink vertex (i.e., the next node in the list) in the object graph. Corresponding to the change of data structure, the calculation workload is modified as shown in Figure 4.9. Note that a significant number of branch mispredictions are yielded in the workload if the column numbers of M1 and M2 are generated with different random seeds. The front-end stalls caused by such mispredictions can hardly be mitigated. The same

```
class Node<E>{
    private E value;
    private Node<E> nextNode;
    ...
}
class List<E>{
    private Node<E> head;
    ...
}
class Pair{
    public final int columnNumber;
    public final double pValue;
    ...
}
class Row{
    public final int rowNumber;
    private List<Pair> elements;
    ...
}
class SparseMatrix{
    private List<Row> mat;
    ...
}
...
```

Figure 4.8: Sparse matrix implementation using linked lists.

random seed can be shared by M1 and M2 to exclude such effects.

Compared to dense matrix implementations, the placement of elements in a sparse matrix is more flexible because a large consecutive memory segment is not needed. However, traversing a sparse matrix requires more dereferencing. For each access to the next element in the row from the current element, multiple steps are required as follows:

1. The nextNode field of the current Node, which stores the reference to the next node in the linked list, is read.

2. The offset of the value field is utilized to calculate the address where the reference to the corresponding Pair object is stored.

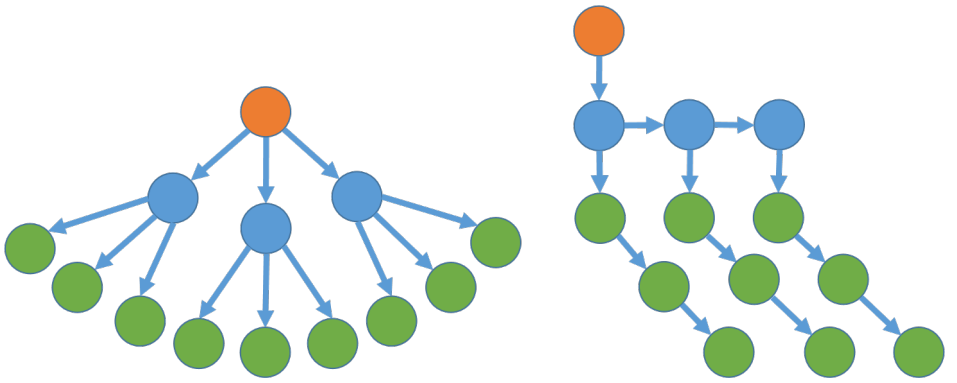
```
...
for(i1=M1.getHead(); i1!=null; i1=i1.next()){
    Row r1 = i1.getValue();
    for(i2=M2.getHead(); i2!=null; i2=i2.next()){
        Row r2 = i2.getValue();
        double s = 0;
        Node<Pair> n1 = r1.getHead();
        Node<Pair> n2 = r2.getHead();
        while(n1 != null && n2 != null){
            Pair p1 = n1.getValue();
            Pair p2 = n2.getValue();
            if(p1.columnNumber==p2.columnNumber){
                double a = p1.pValue * 0.2;
                double b = p2.pValue * 0.125;
                s += (a + b);
                n1 = n1.next();
                n2 = n2.next();
            }
            else if(p1.columnNumber<p2.columnNumber)
                n1 = n1.next();
            else
                n2 = n2.next();
        }
        result = result * 0.6 + s * 0.4;
    }
}
...
```

Figure 4.9: Sparse matrix algebra.

3. The reference to the Pair object is read and the address of the pValue field is calculated.

4. Finally, the pValue field is read.

Such a procedure is a typical example of Pointer Chasing[55]. Pointer Chasing refers to a series of repeated instructions accessing irregular memory sections, each of which depends on previously accessed data. In this case, neither the reference to the next node in the linked list nor the reference to the value of a node can be resolved in advance without reading the reference. Thus, prefetching data is more difficult even



(a) A full 3-ary tree

(b) A Left-Child Right-Sibling representation of (a)

Figure 4.10: Left-Child Right-Sibling representation example.

though the access still follows a certain order. Placing nodes adjacent to each other to increase locality is a plausible optimization. However, the effects of object headers still need to be considered.

4.3.5 Depth First Search on N-ary Tree

Pattern 4 described in Subsection 4.3.1 is reflected by this benchmark workload. The workloads presented in Subsection 4.3.2, Subsection 4.3.3 and Subsection 4.3.4 all perform operations on linear data structures, where each data element is connected sequentially. In contrast, this workload handles a hierarchical data structure, where each data element is connected to multiple other data elements. In particular, the program searches for specific keys in an N-ary tree. An N-ary tree is a tree where each node can connect to at most N children. One implementation of an N-ary tree utilizes the Left-Child Right-Sibling representation [39]. In this representation, each node contains the reference to its first child and the reference to the next sibling of the current node. Figure 4.10 shows the Left-Child Right-Sibling representation of a full 3-ary tree. In the object graph, each vertex representing a node in the tree connects to two non-sink vertices. The implementation is shown in Figure 4.11.

The workload applies Depth First Search (DFS) for each key in the predefined ran-

```
class TreeNode{
    private final long key;
    private TreeNode firstChild;
    private TreeNode nextSibling;
    ...
}
class Tree{
    private TreeNode root;
    ...
}
```

Figure 4.11: Left-Child Right-Sibling tree implementation.

```
public TreeNode dfs(long key, TreeNode node){
    if(node.getKey() == key)
        return node;
    TreeNode result = null;
    TreeNode n = node.getFirstChild();
    while(n != null){
        result = dfs(key, n);
        if(result != null)
            return result;
        n = n.getNextSibling();
    }
    return null;
}
```

Figure 4.12: DFS on Left-Child Right-Sibling tree.

dom sequence to a full N-ary tree. The search is implemented with recursion, as shown in Figure 4.12, since it is more concise and does not include other data structures at the application level.

The height of the tree and the degree of each node control the memory consumption of the workload and are thus parameterized. A configurable set of threads perform the task in parallel. During one iteration, each thread in the set searches a different set of keys independently. After each iteration, a random part of the tree is rebuilt. In such a hierarchical structure, efficient data prefetching cannot be accomplished without proper speculation [55]. The same conclusion also applies to object place-

ment related optimizations. For both optimizations, information about past field access is required.

4.4 Type Checking

In this section, a workload reflecting back-end stalls caused by the type-checking routine is presented. As is shown in the survey in Section 3.3, such stalls are observed to occur more frequently in applications where polymorphism is heavily utilized. Polymorphism, which is the ability to utilize various types with a unified interface, is an important feature of Object-Oriented Programming because it enables the "code to interfaces rather than implementations" [40] practice in software engineering. However, polymorphism also limits the information available to the compiler because the implementation is unknown until execution time. Therefore, type-checking routines are inserted to ensure robustness of Java programs. There are two common situations where a type-checking routine is triggered:

1. Unsafe casting such as down-casting, which is the casting from a base class to a derived class. One of the causes of such casting is type erasure [41] of generic types, which replaces the type arguments in generic types with Object types. An example is shown in Figure 4.13.

2. Method invocation. Since a Java class can inherit all interface methods from its parent class and can implement multiple interfaces itself, checking whether a Java class implements a certain interface is time-consuming. Without any optimization, a list of implemented interfaces must be traversed until the required interface is found. Similar to Polymorphic Inline Caching (PIC) [43, 44], which inlines the most frequently invoked implementations of the same virtual method, the most frequently utilized implementation of an interface can be inlined. This is shown in Figure 4.14. When no implementation is significantly

```
public class List<E>{
    // E is replaced by Object type
    E getFirst()...
    ...
}
...
List<Person> lst = new List<Person>();
...
public void method(){
    ...
    // cast from Object to Person
    Person p = (Person)lst.getFirst();
    ...
}
```

Figure 4.13: An example of Type Erasure and down-casting.

```
if(object instance of frequentConcreteClass1)
    call frequentConcreteClass1.method;
else if(object instance of frequentConcreteClass2)
    call frequentConcreteClass2.method;
else
    call normalTypeCheckingProcedure
    call dynamicDispatchMethod;
```

Figure 4.14: An optimization of type-checking routine.

more frequently utilized than the others, the performance improvement of such inlining is reduced since speculations are often incorrect. In this case, traversing the interface list is still required if the number of inlined implementations cannot be increased. This is often the case since increasing the number of inlined implementations increases code size and is likely to cause degradation of the pipeline performance.

In this benchmark, object arrays are traversed and each element in the same array is utilized as an implementation of the same interface. The elements of the same object array are instances of different concrete classes. Therefore, speculations mentioned above are not always correct and the interface lists are frequently traversed. This

benchmark includes four interfaces. There are four concrete classes and each of them implements all four interfaces. Not all interfaces are utilized with the same frequency for an individual concrete class. For example, DummyClass1 is most frequently utilized as an implementation of DummyInterface1 and least frequently serves as an implementation of DummyInterface4. The parameters of the benchmark include the iteration parameters, the size of the arrays to control memory consumption, and the functional parameters to control the frequencies with which each concrete class is utilized as an implementation of a specific interface.

When the interface list is traversed, the data structure of the list is vital to the performance of the type-checking routine. In addition to the time consumption of such a procedure, the memory required to store the list is also an issue. Since a class implements all interfaces of its ancestor classes and the inheritance chain can be very long with every class in the chain implementing multiple interfaces, storing all the implemented interfaces in a separate list for each individual class consumes a significant amount of memory.

4.5 Huge Objects

Recall that in Subsection 4.3.1 and Subsection 4.3.2, a discussion about accessing data within the same vertex in the object graph was presented. In addition to traversing arrays, accessing data in the same object also follows this pattern. When the object is large enough to occupy multiple cache lines, it is likely the accessed object field does not reside in cache and thus back-end stalls are caused. Unlike arrays, the data within an object can be reordered to achieve better cache locality. When different data of the same object is modified by two threads, false sharing, where the two threads frequently invalidate the cached data copy of the other when no data is actually shared, is likely to happen.

The Java specification does not state a universal object model. Therefore, various object models are utilized by different JVM implementations. In general, each object can be divided into two parts: object header and object data. For every class, there is a class object containing the information shared by all instances of the class. A pointer in the object header references this shared class object. In addition to the pointer, there is a lock word [26] storing the synchronization information of the object. This field is optional if lock nursery [25] is supported. When the class does not have any method that requires exclusive access to an instance, the lock word is not included in the object header. Instead, the mapping between the object and its lock is stored in a hash table. The lock word is added to the object as a hidden field during GC only if its utilization is sufficiently frequent. Another optional field is the hash code serving as the identity of the object. Two bits represent the three states of an object: unhashed, hashed and moved-after-hashed [25]. The hash code of the object is included as a hidden field only if the object is in the 3rd state. Before the garbage collector moves the object, the starting address serves as the hash code and thus no extra space is required for identity comparison. In summary, JVM implementations utilize different object models with various object header sizes. The header can be modified to include optional fields during the execution of the program.

Object fields are placed after object headers. The possible types of object fields in Java include byte (8 bits), short (16 bits), int (32 bits), long (64 bits), float (32 bits), double (64 bits), char (16 bits), boolean (size not specified) and reference (32 bits when compressed or on a 32-bit machine, 64 bits when not compressed on a 64-bit machine). Some JVM implementations expand the size of short, char and boolean values to four bytes so that they can fit on the operand stack. The simplest approach to place the fields in an object is locating them in the order declared by the programmer. Some JVM implementations align the fields for more efficient accesses at the expense of increased memory consumption. In order to reduce wasted memory

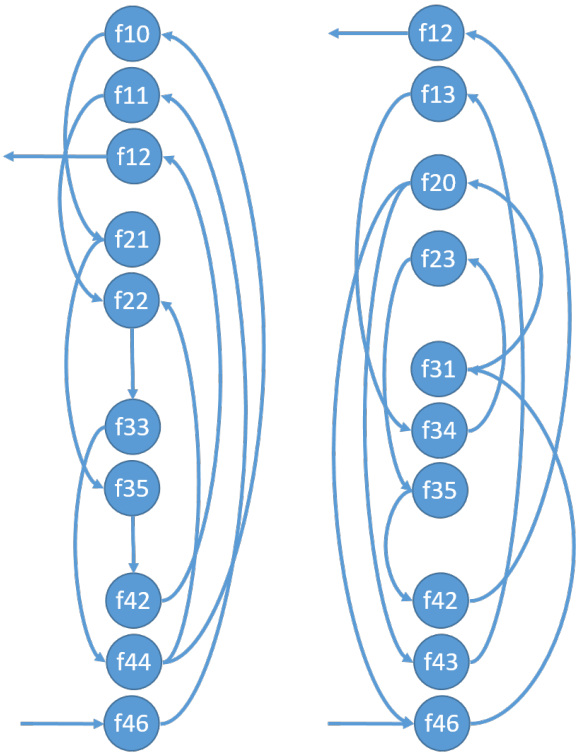
space for padding, fields can be reordered in JVMs [38].

In this benchmark, an array of huge objects is shared among two threads and each thread calls a different method to access the fields in each object. There are 40 fields in each instance. The fields are divided in five groups and each group consists of eight fields. For each object, only 13 fields are accessed in the program. All fields are 8-byte primitive long values so that the influence of alignment is excluded and the benchmark does not overfit a particular JVM implementation. The fields are accessed in such an order that cache performance is inefficient and false sharing is possible. Similar to control flow graphs, we can utilize directed graphs [38] to represent the access order of different fields in the same object. Each vertex in the graph represents a particular field and there is an edge from vertex F_A to vertex F_B if and only if field F_B is accessed immediately after field F_A. With such an abstraction, we can describe the order in which the two methods access the fields in Figure 4.15. Note that the field-declaration order in the class definition is reflected in Figure 4.15. Field fxy denotes the y^{th} field in the x^{th} group. Each cycle in the graph represents the access within a loop. All loops in both methods are executed for three iterations. Each field accessed within the loops are modified in each iteration.

4.6 Repeat String Operations

As mentioned in Section 3.1 and Section 3.3, repeat string operations are likely to cause front-end stalls because of the switches between the legacy pipeline (MITE) and the micro sequencer [21]. There are two common use cases of repeated string operations in Java applications: copying and initialization of arrays. In this section, a benchmark workload reflecting front-end stalls generated during array initialization is presented.

The Java language specification states that every variable and object field must



(a) handle1 (b) handle2

Figure 4.15: Field access order of handle1 and handle2 methods.

obtain a specific initial value. For instance, every element in an integer array must be initialized with 0. This specification ensures that every read of a variable or a field causes a legitimate and predictable result. Other languages, such as C and C++, support fast memory allocation operations that do not provide such guarantees and therefore reading an uninitialized variable introduces a garbage value in the program and causes unpredictable behavior. Although such a guarantee improves the security and robustness of Java programs, it also yields extra overhead. Thus, the implementation of an efficient initialization routine is important.

To provide support for faster memory initialization operations and smaller code size, repeat string operations are included in the x86 Instruction Set. Take the REP STOSD operation as an example. From the user's perspective, the instruction performs the functionality described in Procedure 4.

Before the execution of the instruction, the original data stored in registers EAX,

Procedure 4 REP STOSD operation.

Input: EAX, ESI, ECX

procedure REP STOS(EAX, ESI, ECX)
  $i \leftarrow 0$
  repeat
    $[ESI + i * 4] \leftarrow EAX$
    $i \leftarrow i + 1$
  until $i > ECX$
end procedure

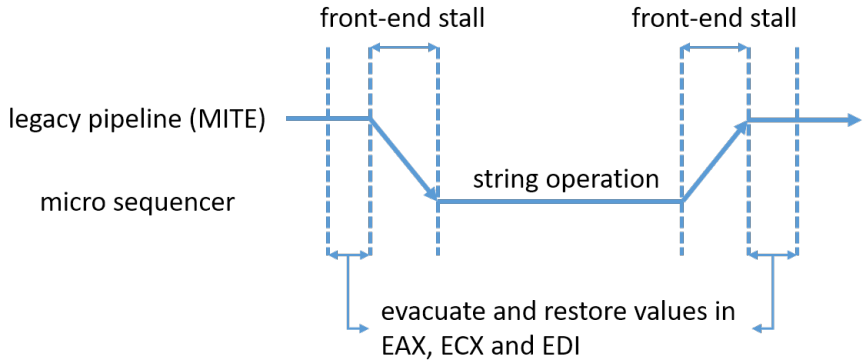


Figure 4.16: Switches during repeat string operation.

EDI and ECX shall be moved to other registers. When no registers are available, the values are pushed to the stack. Internally, the REP prefix triggers a switch from the legacy pipeline to the micro sequencer. During the execution of the instruction, micro operations are generated by the micro sequencer and fed to the execution units. After the instruction is finished, the source of micro operations switches back to the legacy pipeline. When such switches are frequent, the yielded front-end stalls cannot be ignored. The procedure is shown in Figure 4.16.

In this benchmark, a merge-sort [48] procedure is executed by multiple threads on different sections of a shared array. The results of each thread are then merged. The merge-sort implementation is shown in Figure 4.17.

A simple model explains the effects of repeat string operations in the merge-sort program as follows. Assume the input array to each thread contains 2^{n+1} elements

```
public void mergesortLocal(int l, int h, int[] a){
    if(l < h){
        int mid = (l + h) / 2;
        mergesortLocal(l+1, mid, a);
        mergesortLocal(mid+1, h, a);
        int[] t = new int[h - l + 1];
        int i = l, j = mid+1, k = 0;
        while(i <= mid && j <= h){
            if(a[i] <= a[j])
                t[k++] = a[i++];
            else
                t[k++] = a[j++];
        }
        while(i <= mid)
            t[k++] = a[i++];
        while(j <= h)
            t[k++] = a[j++];
            System.arraycopy(t, 0, a, l, h-l+1);
    }
}
```

Figure 4.17: Merge sort implementation.

and the front-end stalls caused by switches are insignificant when the array contains more than 2^k elements. Although the value of k remains unknown, related surveys have been carried out to measure the overhead of repeat string operations [12, 23]. In addition, the value does not affect the discussion below. Applying the merge-sort program on the input array yields 2^{k+1}-1 times of initializations where switch penalties can be ignored. For the remaining $2^{k+1}(2^{n-k}-1)$ initializations, the front-end stalls are influential.

Chapter 5

Validation Results

All benchmarks require validation to guarantee that the benchmarks behave as expected in the design and thus fulfill their purpose. For the stall-focused benchmarks, the validation results are especially important because they are designed with a clear purpose to illustrate the stalls generated in particular cases.

In this chapter, the validation results of the stall-focused benchmarks are presented. Section 5.1 presents the methodology for validation. In Section 5.2, the environment in which the validation experiments are carried out is shown. Finally, the results and their implications are displayed in Section 5.3.

5.1 Validation Methodology

Since the purpose for the benchmarks is to reflect stalls caused by specific behaviors of Java programs executed on x86 architectures, the benchmarks should exhibit similar behaviors across different JVM implementations on different x86 micro architectures. To avoid overfitting, the benchmarks should be validated for two JVM implementations on two micro architectures. Note that in these experiments we are not comparing the performance of the JVMs. The only reason to include different JVM implementations is to prove the generality of our benchmarks.

The measurement should be carried out only after the workloads are at an equivalent optimization level on both JVM implementations. Since different JVM implementations adopt various strategies to optimize the program at different optimization levels, the safest approach is to test the benchmarks only at the most optimized level. In addition, the measurement should focus on specific pieces of code and stalls designated by the design.

As mentioned in Subsection 2.4.1, the support to measure stalls varies according to micro architectures. While recent micro architectures provide better support for stall measurement, top-down analysis is not an available feature in some micro architectures. Therefore, different profilers are utilized on the two micro architectures. For architectures that support top-down analysis PMU events, VTune is utilized to measure the following five PMU events [6, 21, 22, 74]:

1. IDQ_UOPS_NOT_DELIVERED.CORE counts the number of stalled pipeline slots caused by no micro operations being delivered from the instruction queue.

2. UOPS_ISSUE.ANY reports the total number of micro operations being issued to the Reservation Station (RS). The Reservation Station then schedules these micro operations for execution.

3. UOPS_RETIRED.RETIRE_SLOTS counts the number of pipeline slots filled with micro operations that are eventually retired.

4. INT_MISC.RECOVERY_CYCLES measures the number of CPU cycles spent on waiting for the recovery of the pipeline from events such as branch mispredictions.

5. CPU_CLK_UNHALTED.THREAD reports the total number of CPU cycles spent when the thread is not halted. Note that a thread is halted only when the halt (HLT) instruction is executed.

The proportion of each type of stall calculated by VTune is defined as follows [6, 21, 22, 74], where pipeline_width is 4 in the experiments.

$$
\begin{aligned}
\text{N} &= \text{pipeline\_width} * \text{CPU\_CLK\_UNHALTED.THREAD} \\
\text{E}_1 &= \text{IDQ\_UOPS\_NOT\_DELIVERED.CORE} \\
\text{E}_2 &= \text{UOPS\_ISSUE.ANY} \\
\text{E}_3 &= \text{UOPS\_RETIRED.RETIRE\_SLOTS} \\
\text{E}_4 &= \text{INT\_MISC.RECOVERY\_CYCLES} \\
\text{front\_end\_stall} &= \text{E}_1/\text{N} \\
\text{bad\_speculation} &= (\text{E}_2 - \text{E}_3 + \text{pipeline\_width} * \text{E}_4)/\text{N} \\
\text{retiring} &= \text{E}_3/\text{N} \\
\text{back\_end\_stall} &= 1 - (\text{front\_end\_stall} + \text{bad\_speculation} + \text{retiring})
\end{aligned}
$$

CPU_CLK_UNHALTED.THREAD and UOPS_RETIRED.RETIRE_SLOTS are the minimum events required to calculate stalls. When other events are not available, perf is utilized to count the number of the following alternative events [21, 22]:

1. INST_QUEUE_WRITE_CYCLES counts the number of CPU cycles during which instructions are written to the instruction queue. These instructions are then decoded into micro operations and executed.

2. INST_QUEUE_WRITES reports the number of instructions written to the instruction queue. Dividing the number of INST_QUEUE_WRITES by the number of INST_QUEUE_WRITE_CYCLES provides an estimation of front-end efficiency.

3. BRANCHES and BRANCH-MISSES are utilized to calculate branch-miss rate so the effect of branches can be estimated.

4. CACHE-REFERENCES and CACHE-MISSES are utilized to calculate cache-miss rate so back-end efficiency can be estimated.

The measurement estimates the cause of the stalls, which are calculated by the numbers of CPU_CLK_UNHALTED.THREAD and UOPS_RETIRED.RETIRE_SLOTS, rather than measuring the number of each type of stall. The front-end efficiency is estimated as the following:

$$\text{front\_end\_efficiency} = \frac{\text{INST\_QUEUE\_WRITES}}{\text{INST\_QUEUE\_WRITES\_CYCLES} \times \text{pipeline\_width}} \quad (5.1)$$

INST_QUEUE_WRITES and INST_QUEUE_WRITES_CYCLES are not supported on the second experimental machine (see Table 5.1) because it has already provided all the top-down analysis PMU events.

To depict the effect of stalls on each benchmark, two approaches are employed. For the data gathered by VTune, we calculate the total number of each type of stall as well as the proportion of measured x86 instructions affected by various proportions of specific stalls. The latter is utilized to estimate how each instruction is affected by stalls. Note that the two measurements serve as complements of each other because the former is less fine-grained, but more accurate, and the latter provides instruction-granularity information, but is affected by event skid. When the code of interest is only a part of a routine snippet, alternative measurement, as discussed in Subsection 5.3.7, is adopted to present the effect of such code on the routine.

Similarly, we can obtain both method-granularity and instruction-granularity information from the data gathered by perf, except we cannot distinguish one type of stall from another. In addition, we do not process the events recorded to estimate the cause of stalls in a per-instruction manner. This is because such events possess a clearer meaning at a multi-instruction level.

Table 5.1: Specifications of the second experimental machine.

Architecture	x86_64
CPU op-mode	64-bit
NUMA nodes	1
CPUs per NUMA node	12
Model Name	Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
Code Name	Coffee Lake
CPU maximum Frequency	4600 MHz
CPU minimum Frequency	800 MHz
Threads per core	2
L1d cache size	192 KB
L1i cache size	192 KB
L2 cache size	1.5 MB
L3 cache size	12 MB
memory size	32.68 GB
SIMD	MMX, SSE, SSE2, SSSE3, SSE4_1, SSE4_2
Operating System	Linux
Kernel Version	#74-Ubuntu SMP Wed Jan 27 22:54:38 UTC 2021
Kernel Release	5.4.0-66-generic

For each benchmark, variations in parameters are employed to increase the robustness of the results when possible. To reduce the effect of external factors, the experiment in each parameter group on each machine is repeated four times.

5.2 Experimental Environment

The experiments are carried out on two machines. The first experimental machine is the experimental machine of the survey in Chapter 3, the specifications of which are shown in Table 3.1. The specifications of the second experimental machine are shown in Table 5.1.

Only the second experimental machine supports all the five PMU events required for top-down analysis mentioned in Section 5.1. The first experimental machine supports CPU_CLK_UNHALTED.THREAD and UOPS_RETIRED.RETIRE_SLOTS, which is the minimum requirement for our experiments. Also note that NUMA is not provided on the second machine. No explicit configurations about NUMA are applied

Table 5.2: Parameters for the workload in Subsection 5.3.1.

	Group	m	n	r	iteration
machine 1	1	1000	1800	1800	100
	2	1000	2000	2000	100
	3	1000	2200	2200	100
	4	1000	2400	2400	100
machine 2	1	1000	1000	1000	100
	2	1000	1200	1200	100
	3	1000	1300	1300	100
	4	1000	1500	1500	100

to the experiments.

The JVM implementations utilized in the experiments are OpenJ9 and HotSpot, both with OpenJDK 8.

5.3 Experimental Results

In this Section, the experimental results are presented. For each subsection from 5.3.1 to 5.3.7, the input parameters are first presented. Then, the result of each experimental group is shown, followed by an analysis of the benchmark.

5.3.1 Dense Matrix Algebra using Primitive Types

As mentioned in Subsection 4.3.2, the parameters for matrix algebra workloads include the iteration parameter controlling the execution time and the footprint parameters controlling the size of the two input matrices (i.e., an m $\times$ n matrix M1 and an n $\times$ r matrix M2). The parameter groups on each machine are presented in Table 5.2.

Note that the values of m, n and r differ on two machines. This is because the size of caches vary on the two experimental machines and thus the memory consumption is reconfigured. The parameters are chosen based on a series of factors including execution time and memory consumption. Ideally, the parameters should cause

expected stalls while the program can be finished with reasonable time and memory. While this indicates the parameters need to be updated based on the hardware, most of our experiments show the parameters utilized are portable.

The effect of back-end stalls is measured for the loop executing the matrix algebra logic. The results collected by VTune are shown in Table 5.3 and Figure 5.1. The errors of Figure 5.1 are calculated based on the multinomial distribution while the standard deviations in Table 5.3 are calculated for each parameter group in Table 5.2. Note that the multinomial distribution model is chosen because the proportion of stalled pipeline slots of each measured instruction falls in one of the 10 exclusive categories and the probabilities of such events are considered to be fixed. Each measured instruction is viewed as an independent trial. The results collected by perf are shown in Table 5.4 and Figure 5.2.

The results collected by VTune show that the majority of instructions in the measured code are affected heavily by back-end stalls and a significant number of pipeline slots consumed are stalled by the back end. The results gathered by perf show that the cache-miss rate is unusually high for the measured code, which is hypothesized to be the cause of the observed stalls.

Table 5.3: Proportions of pipeline slots consumed by measured routine (dense matrix algebra using primitive types).

(a) OpenJ9

	Retiring		Front-end stall		Bad speculation		Back-end stall	
Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
1	23.39%	0.98%	1.45%	0.12%	0.51%	0.03%	74.74%	0.90%
2	23.68%	0.76%	1.49%	0.20%	0.47%	0.02%	74.40%	0.66%
3	23.59%	0.87%	1.20%	0.32%	0.40%	0.05%	74.88%	0.97%
4	23.65%	0.77%	1.86%	0.22%	0.41%	0.07%	74.12%	0.66%
Total	23.58%	0.86%	1.50%	0.33%	0.45%	0.07%	74.54%	0.86%

(b) HotSpot

	Retiring		Front-end stall		Bad speculation		Back-end stall	
Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
1	24.10%	0.69%	5.65%	0.65%	0.51%	0.06%	69.76%	1.24%
2	24.07%	0.60%	5.43%	0.42%	0.50%	0.09%	70.01%	0.86%
3	23.97%	0.86%	5.22%	0.13%	0.41%	0.13%	70.31%	1.02%
4	23.96%	0.61%	5.51%	0.16%	0.50%	0.08%	70.04%	0.74%
Total	23.97%	0.70%	5.44%	0.43%	0.48%	0.10%	70.10%	0.99%

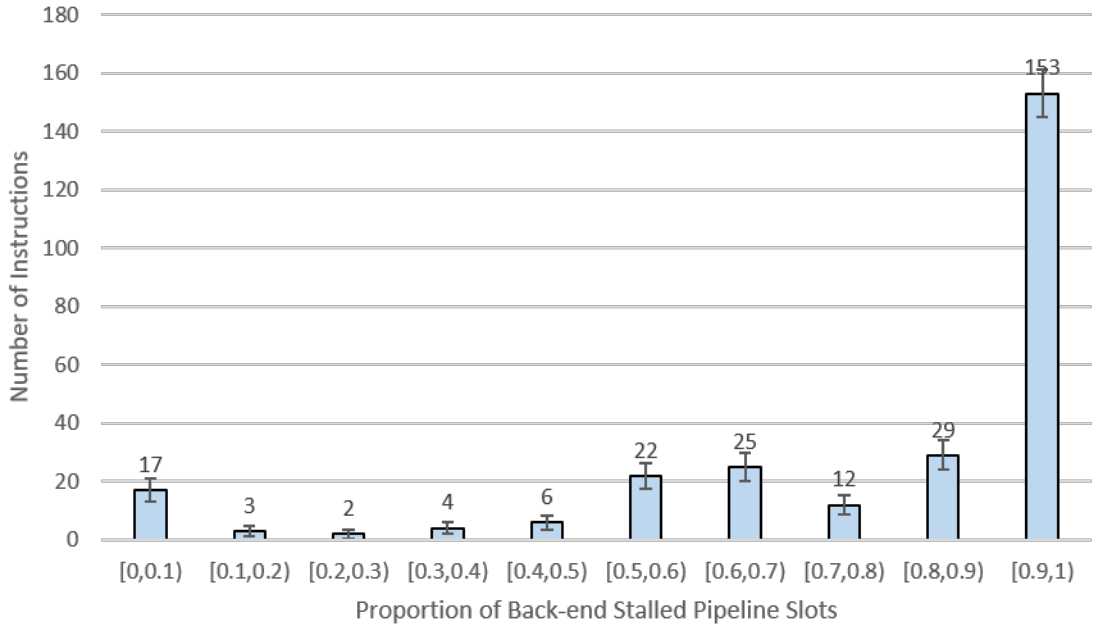
Table 5.4: Perf measurement results (dense matrix algebra using primitive types).

(a) OpenJ9

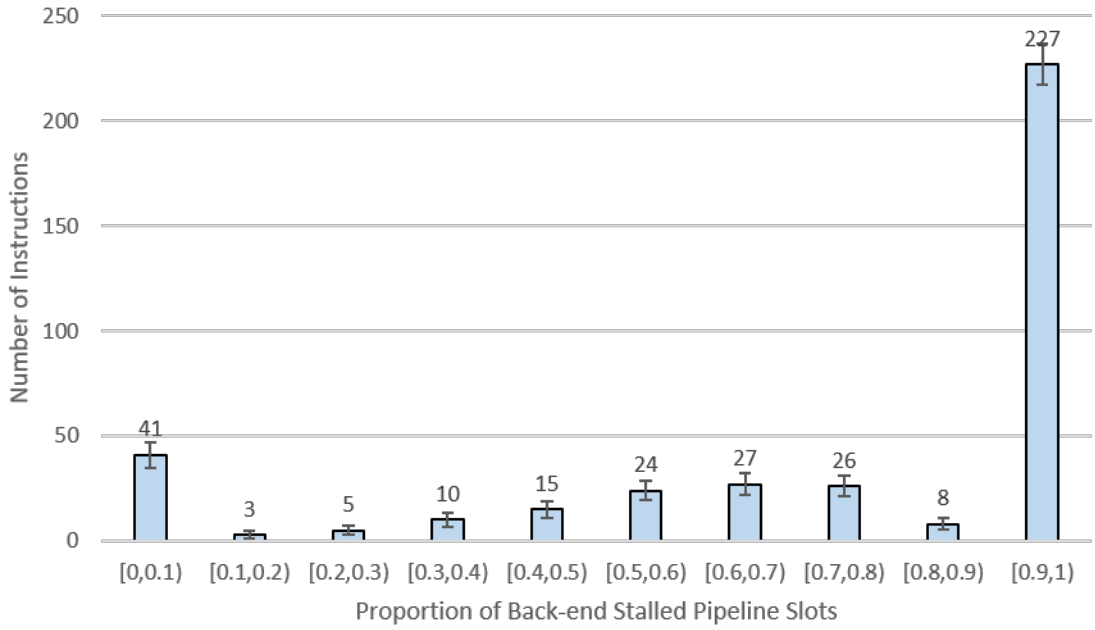
	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev
Group	**1**		**2**		**3**		**4**	
Front-end efficiency	80.87%	0.01%	80.93%	0.02%	80.70%	0.19%	80.70%	0.03%
Branch-miss rate	0.25%	0.01%	0.22%	0.01%	0.20%	0.01%	0.18%	0.00%
Cache-miss rate	96.56%	0.24%	97.78%	0.21%	98.93%	0.17%	99.07%	0.11%
Stalled pipeline slots	73.15%	2.37%	73.83%	3.17%	75.63%	2.12%	74.82%	2.01%

(b) HotSpot

	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev
Group	**1**		**2**		**3**		**4**	
Front-end efficiency	69.55%	0.24%	69.59%	0.33%	69.20%	0.53%	68.91%	0.14%
Branch-miss rate	0.23%	0.01%	0.21%	0.00%	0.20%	0.00%	0.18%	0.00%
Cache-miss rate	97.91%	0.16%	98.75%	0.04%	99.58%	0.02%	99.64%	0.07%
Stalled pipeline slots	73.40%	1.63%	74.79%	1.16%	75.18%	0.74%	76.06%	0.64%

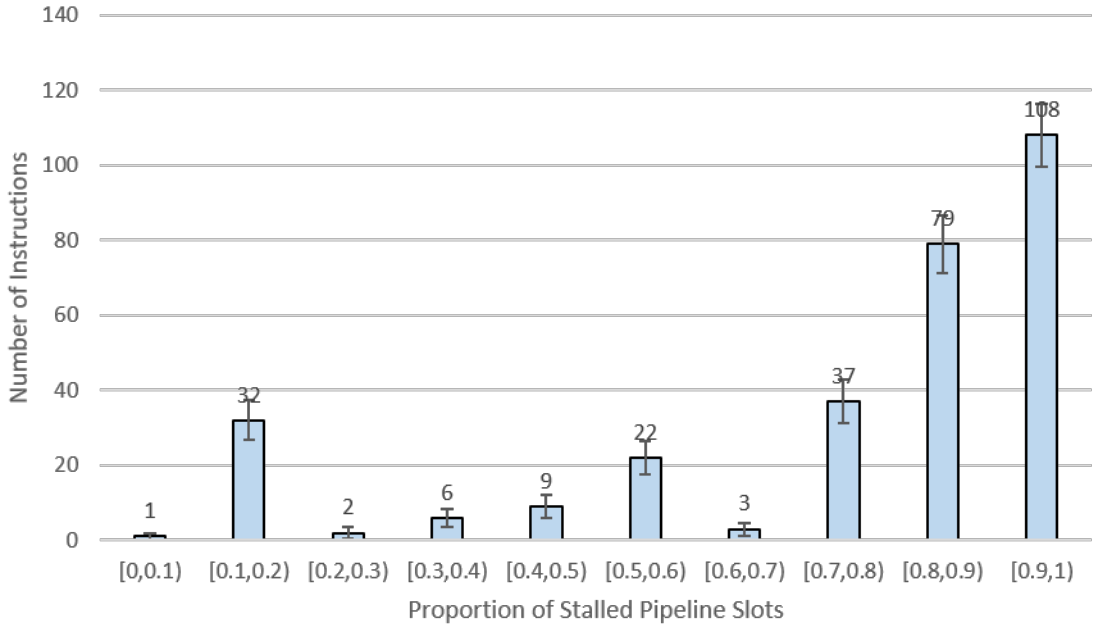


(a) OpenJ9

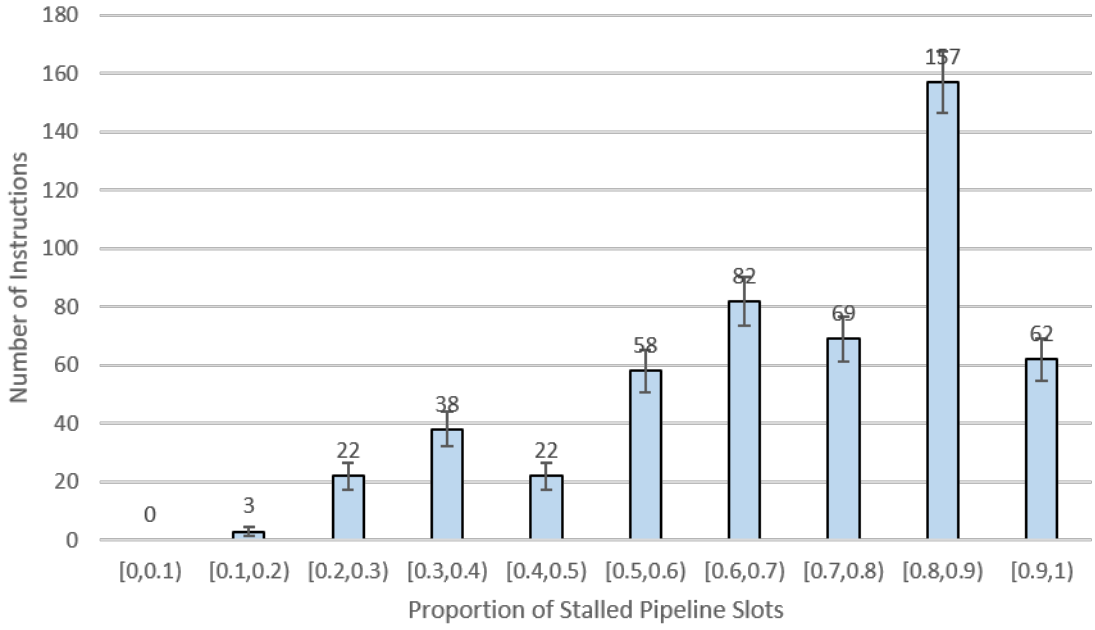


(b) HotSpot

Figure 5.1: Distribution of instructions by proportion of back-end stalled pipeline slots (dense matrix algebra using primitive types) measured by VTune on machine 2.



(a) OpenJ9



(b) HotSpot

Figure 5.2: Distribution of instructions by proportion of stalled pipeline slots (dense matrix algebra using primitive types) measured by perf on machine 1.

Table 5.5: Parameters for the workload in Subsection 5.3.2.

Group	m	n	r	iteration
1	1000	1000	1000	100
2	1000	1200	1200	100
3	1000	1300	1300	100
4	1000	1500	1500	100

5.3.2 Dense Matrix Algebra using Wrapper Objects

The inputs of the workload are shown in Table 5.5. Unlike the workload in Subsection 5.3.1, the parameters are applied to both machines, based on the hypothesis that the generated stalls cannot be mitigated by increased resources.

The results measured by VTune are shown in Table 5.6 and Figure 5.3. The results gathered by perf are shown in Table 5.7 and Figure 5.4.

The results collected by VTune show that the majority of instructions in the measured code is affected heavily by back-end stalls and a significant number of pipeline slots consumed are stalled by the back end. The results gathered by perf show that the cache-miss rate is unusually high for the measured code, which is estimated to be the main cause of the observed stalls. Note that the number of instructions measured varies according to JVM implementations. This reflects the different compilation strategies and optimizations adopted by different JVMs. For instance, aggressive loop unrolling generates more instructions in a loop. Also note that the same parameters cause significant stalls on both machines despite the difference in available resources. The hypothesized cause is the increased dereferencing required by the utilization of wrapper objects.

Table 5.6: Proportions of pipeline slots consumed by measured routine (dense matrix algebra using wrapper objects).

(a) OpenJ9

	Retiring		Front-end stall		Bad speculation		Back-end stall	
Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
1	22.75%	1.14%	3.32%	0.52%	0.63%	0.27%	73.30%	1.39%
2	23.87%	0.80%	5.37%	1.22%	0.36%	0.10%	70.41%	0.64%
3	24.18%	1.50%	3.21%	1.43%	0.35%	0.07%	72.26%	0.56%
4	24.42%	0.31%	5.34%	1.29%	0.41%	0.13%	69.84%	1.69%
Total	23.80%	1.21%	4.31%	1.57%	0.44%	0.20%	71.45%	1.82%

(b) HotSpot

	Retiring		Front-end stall		Bad speculation		Back-end stall	
Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
1	24.83%	0.13%	1.19%	0.05%	0.24%	0.12%	73.75%	0.13%
2	25.19%	0.05%	1.17%	0.18%	0.33%	0.03%	73.30%	0.21%
3	25.05%	0.23%	1.20%	0.08%	0.32%	0.11%	73.43%	0.11%
4	25.07%	0.10%	1.96%	0.63%	0.34%	0.03%	72.64%	0.67%
Total	25.03%	0.19%	1.38%	0.47%	0.31%	0.09%	73.28%	0.54%

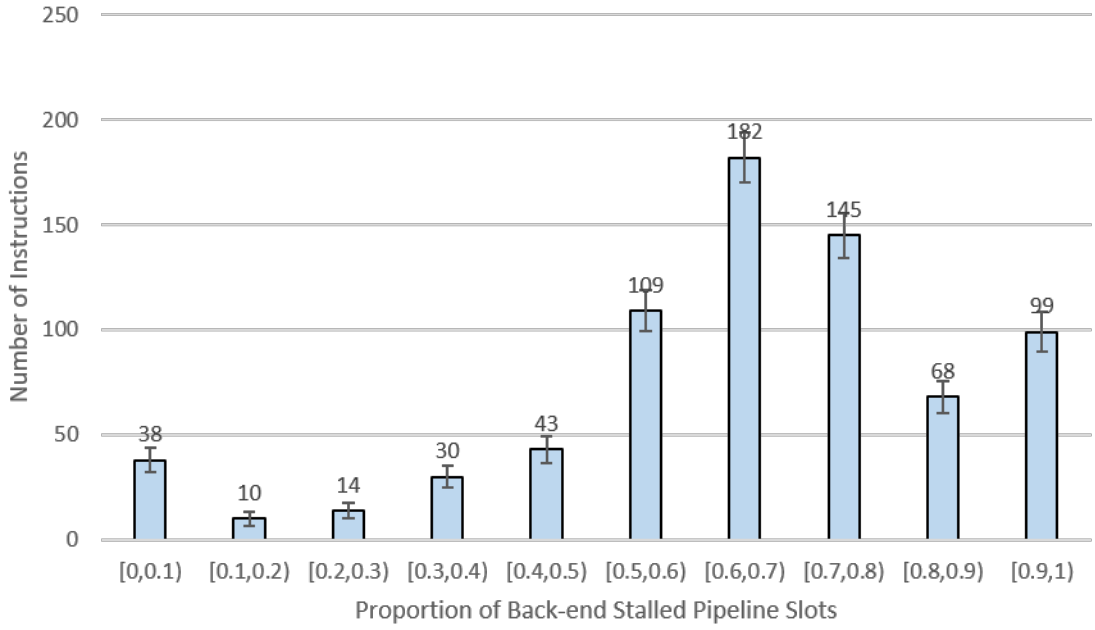
Table 5.7: Perf measurement results (dense matrix algebra using wrapper objects).

(a) OpenJ9

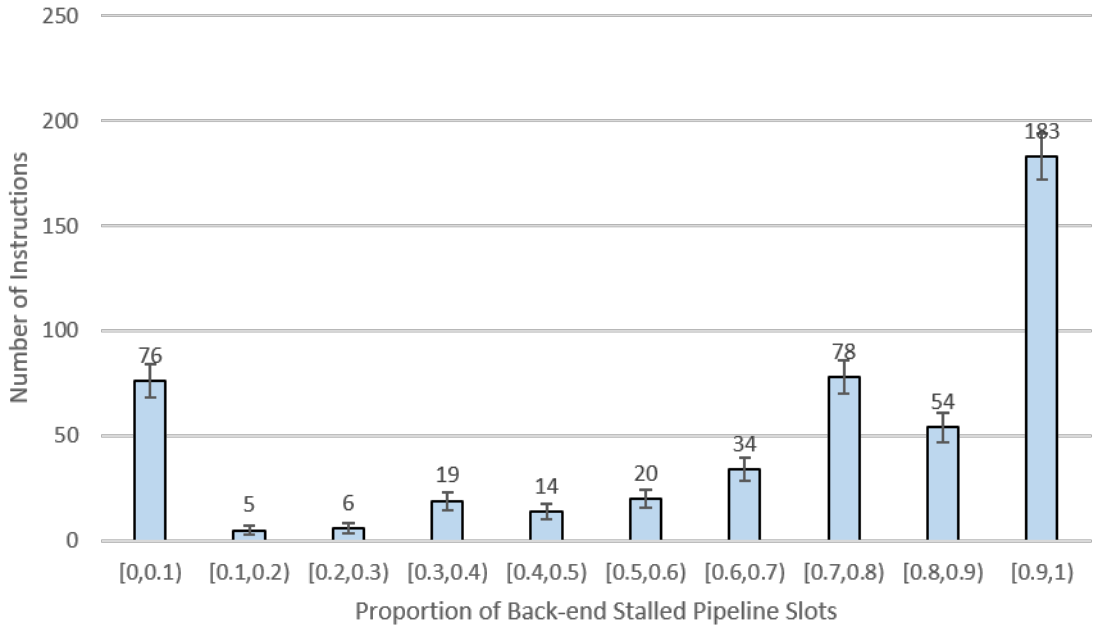
	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev
Group	**1**		**2**		**3**		**4**	
Front-end efficiency	80.84%	0.09%	82.62%	0.06%	81.23%	0.05%	82.81%	0.05%
Branch-miss rate	0.13%	0.01%	0.13%	0.00%	0.11%	0.00%	0.11%	0.00%
Cache-miss rate	81.79%	1.57%	93.77%	0.45%	96.03%	0.37%	96.01%	0.21%
Stalled pipeline slots	63.69%	2.01%	61.32%	4.35%	69.05%	1.58%	59.36%	1.60%

(b) HotSpot

	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev
Group	**1**		**2**		**3**		**4**	
Front-end efficiency	78.43%	0.69%	76.99%	2.30%	74.53%	4.52%	77.53%	0.17%
Branch-miss rate	0.19%	0.02%	0.17%	0.00%	0.17%	0.01%	0.14%	0.00%
Cache-miss rate	97.44%	0.37%	97.22%	0.14%	97.05%	0.18%	96.65%	0.18%
Stalled pipeline slots	83.56%	1.05%	84.07%	0.90%	83.02%	0.79%	83.21%	0.40%

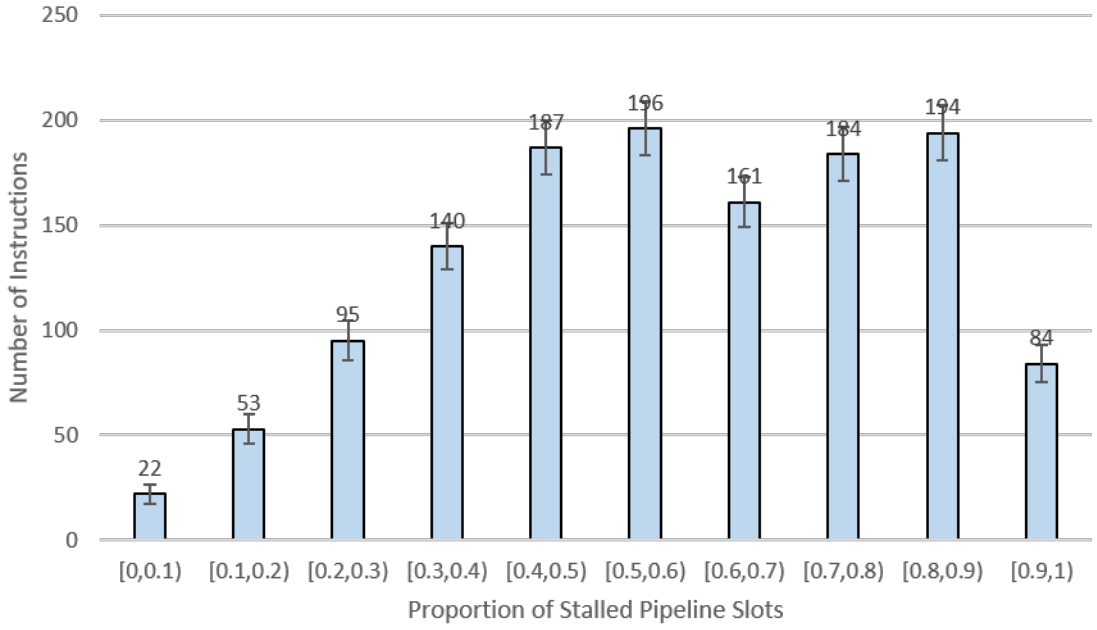


(a) OpenJ9

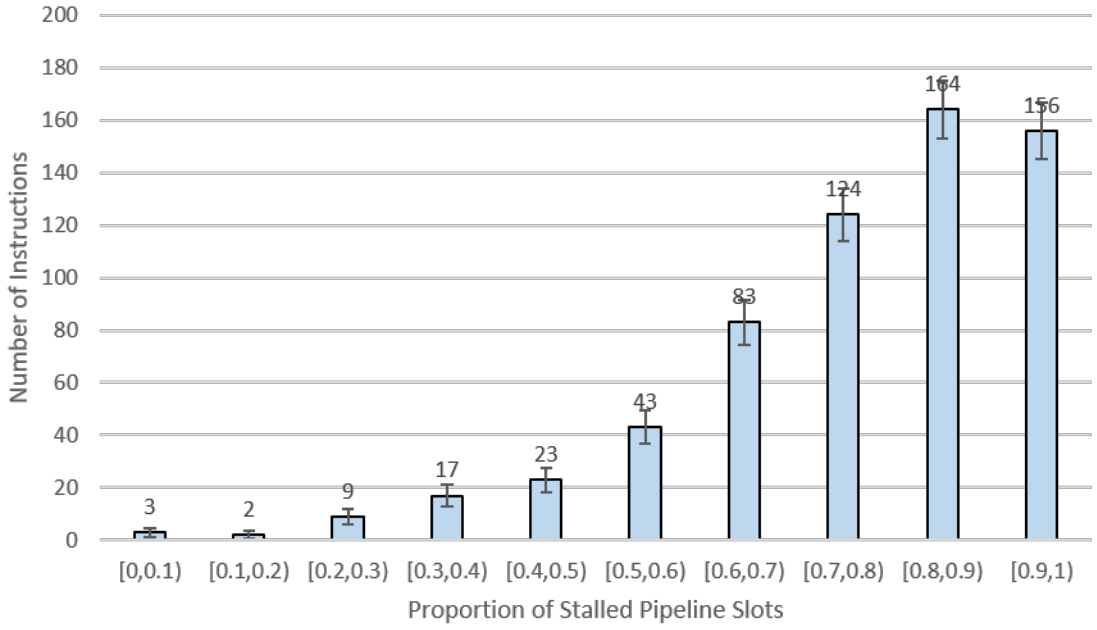


(b) HotSpot

Figure 5.3: Distribution of instructions by proportion of back-end stalled pipeline slots (dense matrix algebra using wrapper objects) measured by VTune on machine 2.



(a) OpenJ9



(b) HotSpot

Figure 5.4: Distribution of instructions by proportion of stalled pipeline slots (dense matrix algebra using wrapper objects) measured by perf on machine 1.

Table 5.8: Proportions of pipeline slots consumed by measured routine (sparse matrix algebra).

(a) OpenJ9

	Retiring		Front-end stall		Bad speculation		Back-end stall	
Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
1	25.32%	0.38%	24.83%	0.39%	0.40%	0.03%	49.44%	0.65%
2	24.86%	0.14%	23.36%	1.85%	0.39%	0.01%	51.38%	1.72%
3	24.54%	0.16%	23.34%	1.06%	0.23%	0.01%	51.89%	1.16%
4	25.32%	0.33%	25.12%	0.40%	0.30%	0.05%	49.26%	0.73%
Total	25.01%	0.43%	24.16%	1.37%	0.33%	0.07%	50.49%	1.63%

(b) HotSpot

	Retiring		Front-end stall		Bad speculation		Back-end stall	
Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
1	23.83%	0.87%	1.10%	0.52%	0.26%	0.05%	74.81%	1.22%
2	24.50%	0.04%	1.09%	0.41%	0.25%	0.03%	74.16%	0.38%
3	24.08%	0.21%	0.80%	0.12%	0.19%	0.03%	74.93%	0.14%
4	24.63%	0.47%	1.71%	0.43%	0.22%	0.03%	73.45%	0.70%
Total	24.26%	0.60%	1.17%	0.52%	0.23%	0.05%	74.34%	0.94%

5.3.3 Sparse Matrix Algebra using Linked Lists

The inputs of the workload are the same as in Subsection 5.3.2. Note that the calculations are performed on sparse matrices where half of the elements are 0s. Thus, the number of instantiated elements is much lower than in Subsection 5.3.2. The parameters are applied to both machines. The results measured by VTune are shown in Table 5.8 and Figure 5.5. The results gathered by perf are shown in Table 5.9 and Figure 5.6.

On both machines, stalls are observed for a significant proportion of instructions. The results of VTune show a non-trivial proportion of pipeline slots stalled by the back end for both JVM implementations. According to perf's results, cache misses

Table 5.9: Perf measurement results (sparse matrix algebra).

(a) OpenJ9

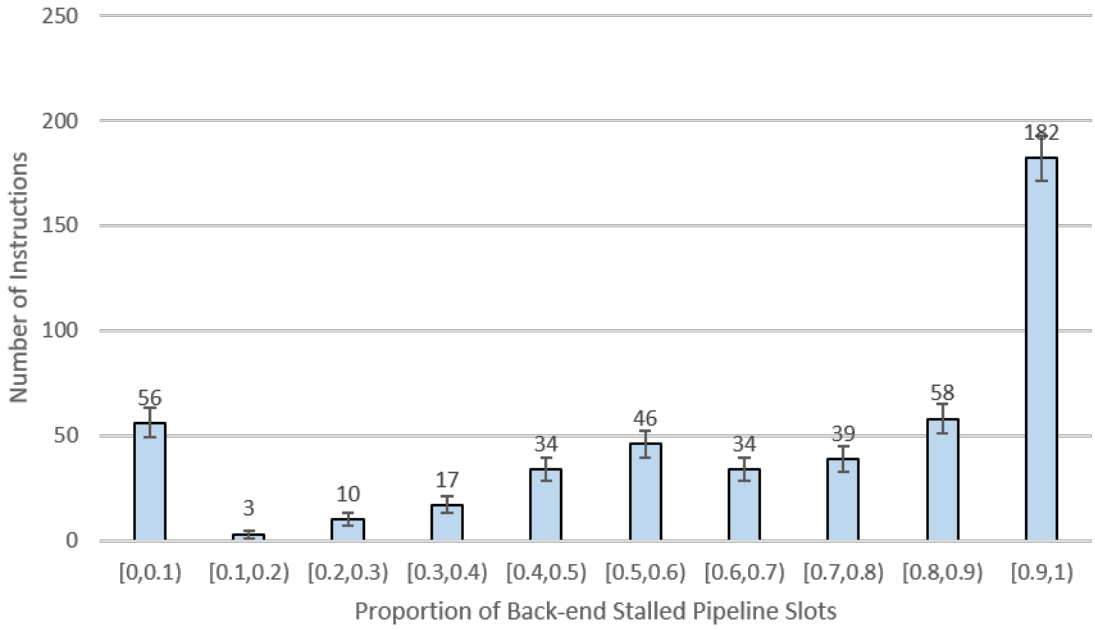
	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev
Group	**1**		**2**		**3**		**4**	
Front-end efficiency	83.63%	0.15%	78.12%	0.02%	88.00%	0.09%	78.11%	0.03%
Branch-miss rate	0.03%	0.00%	0.03%	0.00%	0.03%	0.00%	0.02%	0.00%
Cache-miss rate	85.67%	4.59%	95.93%	0.31%	96.51%	0.12%	96.83%	0.09%
Stalled pipeline slots	64.24%	3.05%	67.57%	0.95%	69.82%	0.68%	68.33%	1.44%

(b) HotSpot

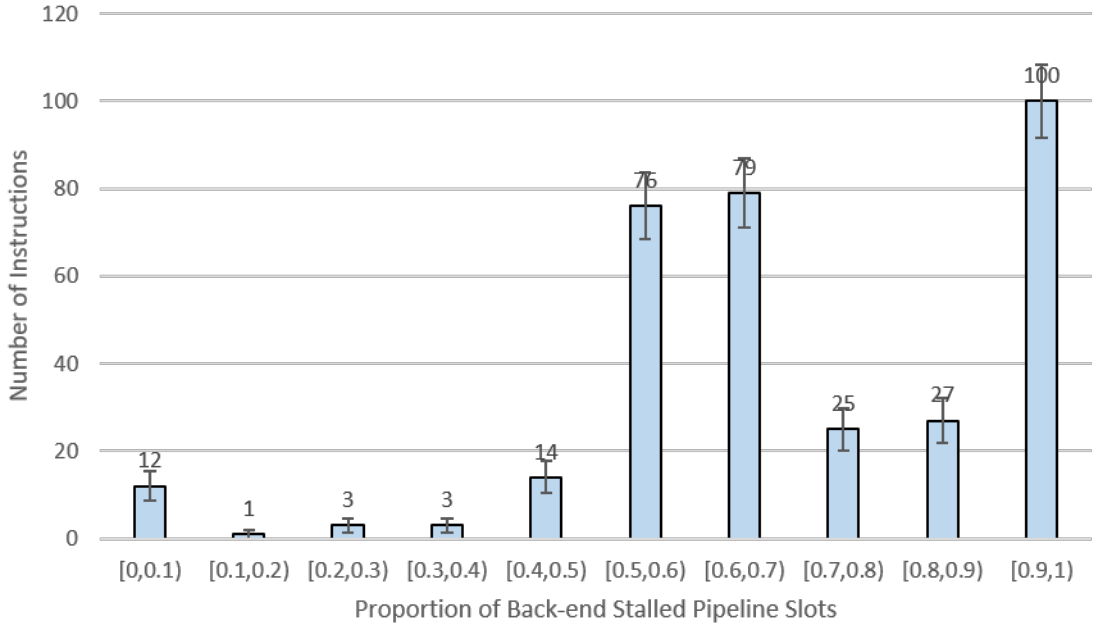
	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev
Group	**1**		**2**		**3**		**4**	
Front-end efficiency	84.44%	1.86%	83.56%	1.86%	82.44%	1.75%	84.07%	2.50%
Branch-miss rate	0.07%	0.00%	0.07%	0.00%	0.07%	0.00%	0.06%	0.00%
Cache-miss rate	99.47%	0.07%	99.31%	0.06%	99.30%	0.07%	98.81%	0.25%
Stalled pipeline slots	92.61%	0.43%	92.31%	0.12%	86.61%	0.16%	88.84%	0.22%

are the main cause of stalls.

Compared to dense matrices implemented with primitive types, the results of sparse matrices are more similar to those of dense matrices implemented with wrapper objects. The same parameters cause non-trivial back-end stalls on both experimental machines. Judging from the results of this subsection and Subsection 5.3.2, the data prefetching is not performed by the hardware when data blocks are not accessed in a consecutive manner. This observation indicates compiler designs should not be based on the assumption that hardware prefetchers can handle complex memory access patterns.

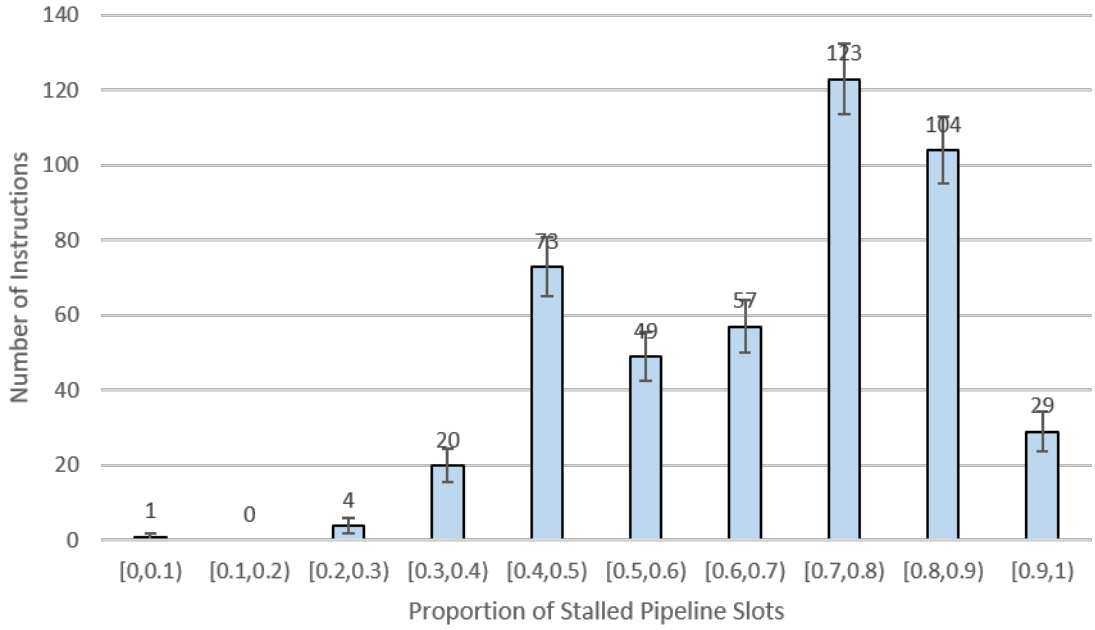


(a) OpenJ9

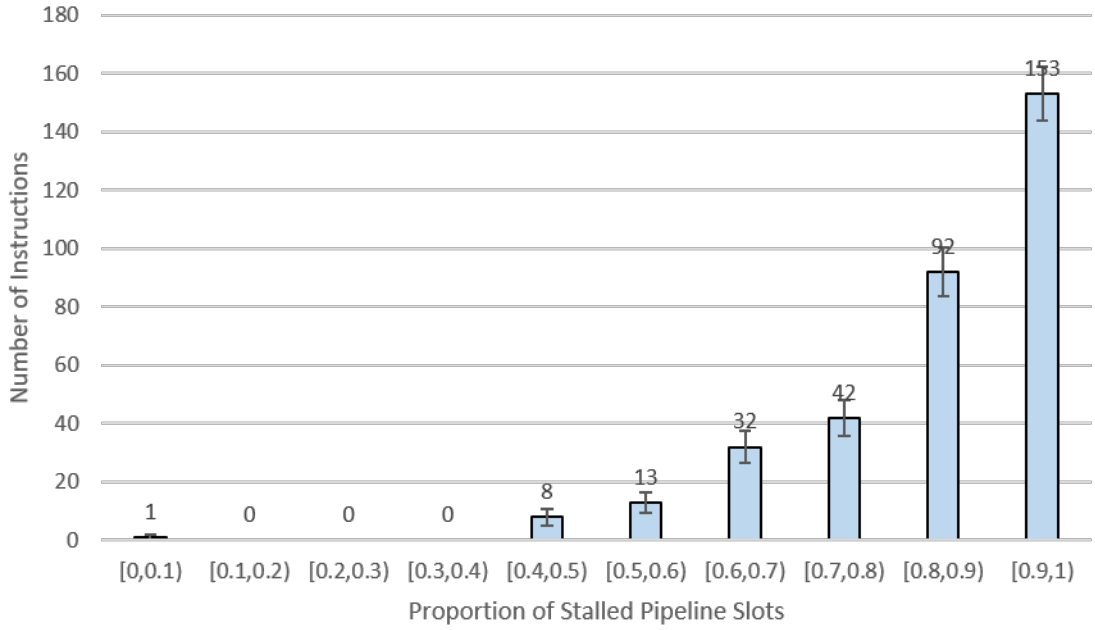


(b) HotSpot

Figure 5.5: Distribution of instructions by proportion of back-end stalled pipeline slots (sparse matrix algebra) measured by VTune on machine 2.



(a) OpenJ9



(b) HotSpot

Figure 5.6: Distribution of instructions by proportion of stalled pipeline slots (sparse matrix algebra) measured by perf on machine 1.

Table 5.10: Parameters for the workload in Subsection 5.3.4.

Group	InnerIter	OuterIter	H	T	R
1	1000	100	5	4	69905
2	1200	100	5	4	69905
3	1300	100	5	4	69905
4	1500	100	5	4	69905

5.3.4 Depth First Search on N-ary Tree

For this workload, there are five parameters. Iteration parameter *InnerIter* controls the number of keys each thread searches individually before the search results of each thread are merged. Iteration parameter *OuterIter* controls the number of iterations this procedure is repeated. H configures the height of the tree. T controls the number of threads. R configures the range of the key to be searched. The search is carried out on a 16-ary tree. The parameter sets are shown in Table 5.10. The results measured by VTune are shown in Table 5.11 and Figure 5.7. The results gathered by perf are shown in Table 5.12 and Figure 5.8.

Each instruction performing operations on fields of the tree nodes is measured separately for each thread. The results from VTune show that a non-trivial proportion of measured instructions are affected by back-end stalls to a non-trivial extent for both JVMs. The results from perf show non-trivial cache-miss rates for execution on both JVMs. A significant number of instructions are heavily affected by stalls. These stalls are estimated to be mainly caused by the observed cache misses.

Note that the cache-miss rate of this workload is lower than that of the matrix algebra workloads. A possible reason is that the key of a tree node resides in the same object as the references to other tree nodes. Thus, these fields are more likely to reside in the same cache line.

Table 5.11: Proportions of pipeline slots consumed by measured routine (DFS on N-ary tree).

(a) OpenJ9

	Retiring		Front-end stall		Bad speculation		Back-end stall	
Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
1	31.91%	1.18%	24.84%	1.27%	2.70%	0.38%	40.54%	2.66%
2	30.55%	1.10%	24.05%	1.26%	2.32%	0.46%	43.08%	2.55%
3	33.39%	2.47%	26.48%	2.39%	2.66%	0.61%	37.48%	5.39%
4	29.35%	1.85%	22.97%	2.84%	2.28%	0.26%	45.41%	4.84%
Total	31.30%	2.30%	24.58%	2.43%	2.49%	0.49%	41.63%	5.02%

(b) HotSpot

	Retiring		Front-end stall		Bad speculation		Back-end stall	
Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
1	32.65%	2.16%	6.13%	0.94%	1.74%	0.28%	59.48%	3.24%
2	30.35%	0.90%	5.27%	0.38%	1.53%	0.23%	62.85%	1.31%
3	32.95%	1.51%	6.04%	0.75%	1.72%	0.31%	59.29%	2.41%
4	29.98%	1.71%	5.16%	0.71%	1.96%	0.85%	62.90%	1.97%
Total	31.48%	2.11%	5.65%	0.85%	1.74%	0.51%	61.13%	2.92%

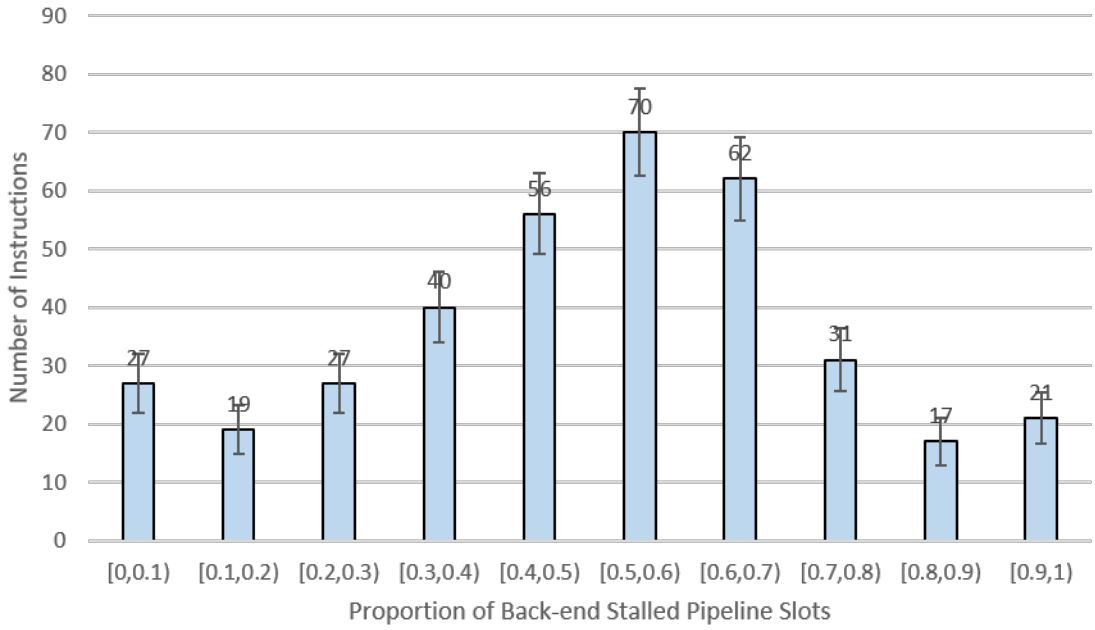
Table 5.12: Perf measurement results (DFS on N-ary tree).

(a) OpenJ9

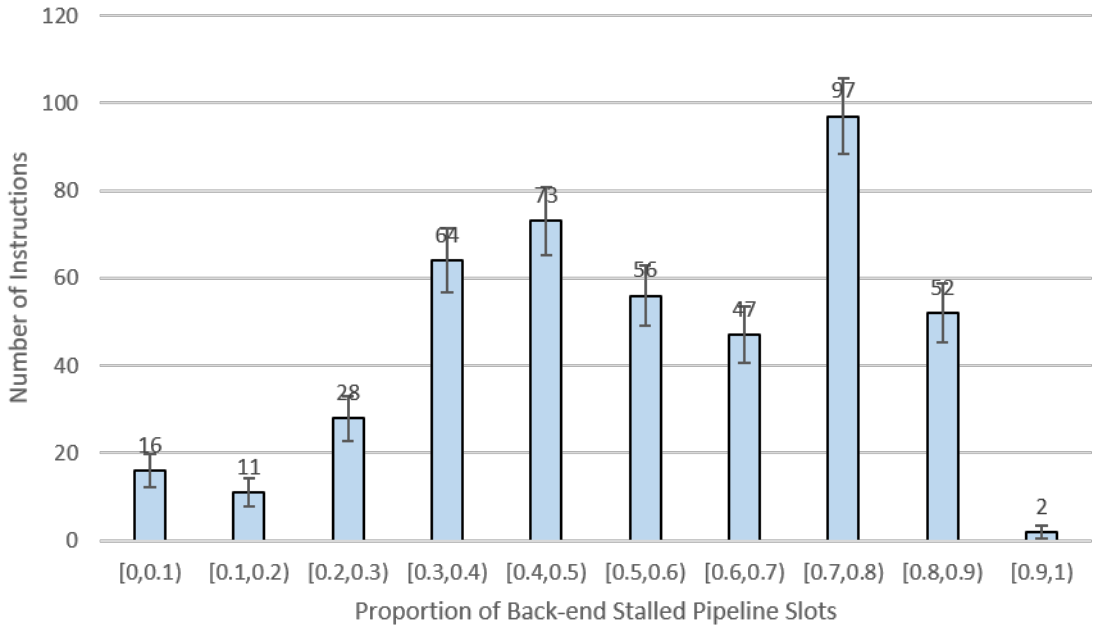
	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev
Group	**1**		**2**		**3**		**4**	
Front-end efficiency	76.95%	2.43%	78.79%	0.72%	75.02%	2.06%	76.47%	2.08%
Branch-miss rate	0.19%	0.03%	0.22%	0.02%	0.19%	0.00%	0.18%	0.02%
Cache-miss rate	57.08%	2.32%	57.30%	2.14%	59.35%	3.14%	60.68%	4.60%
Stalled pipeline slots	83.99%	0.98%	82.52%	0.32%	82.35%	1.25%	81.38%	0.64%

(b) HotSpot

	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev
Group	**1**		**2**		**3**		**4**	
Front-end efficiency	85.61%	0.14%	85.62%	0.29%	85.72%	0.20%	85.92%	0.11%
Branch-miss rate	0.22%	0.00%	0.21%	0.01%	0.22%	0.00%	0.22%	0.00%
Cache-miss rate	61.05%	1.69%	58.58%	3.40%	61.06%	1.71%	65.34%	3.86%
Stalled pipeline slots	82.10%	0.05%	81.02%	0.07%	81.80%	0.10%	81.00%	0.08%

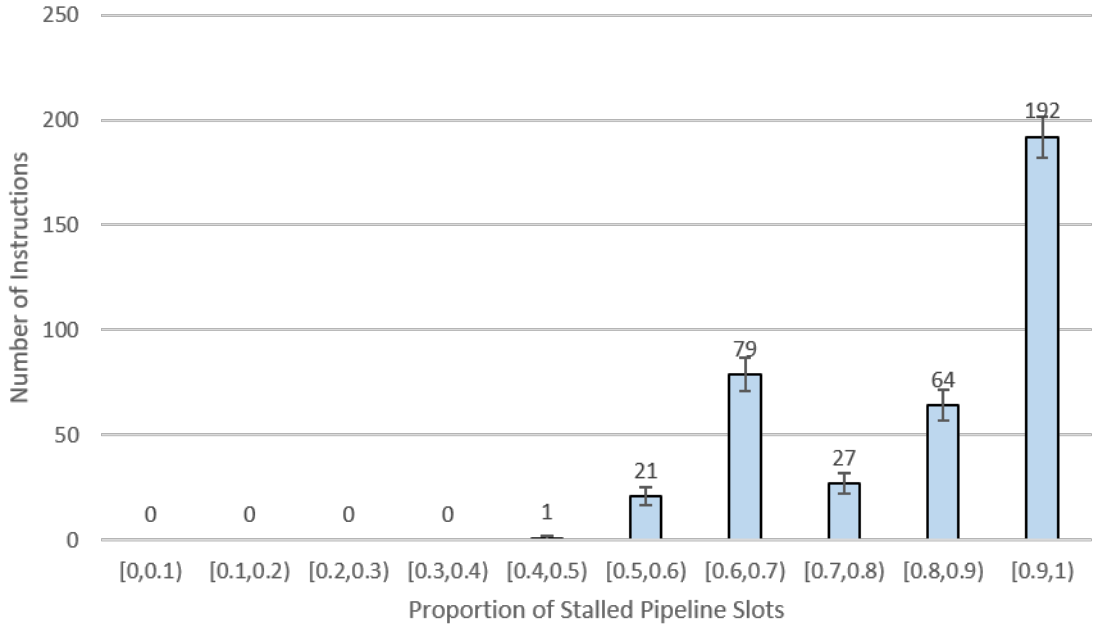


(a) OpenJ9

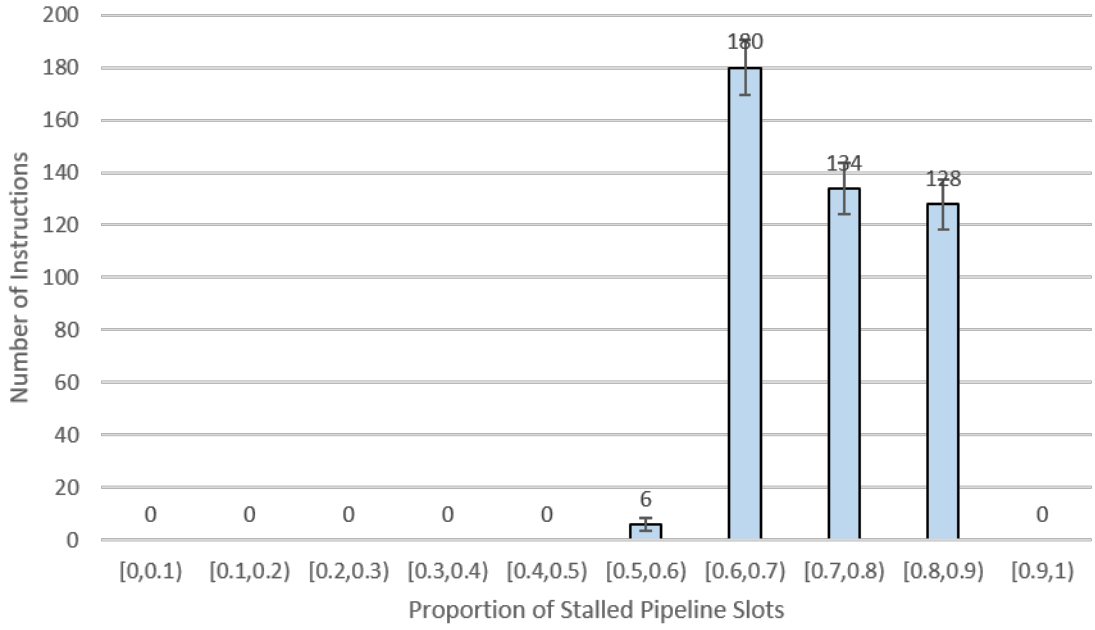


(b) HotSpot

Figure 5.7: Distribution of instructions by proportion of back-end stalled pipeline slots (DFS on N-ary tree) measured by VTune on machine 2.



(a) OpenJ9



(b) HotSpot

Figure 5.8: Distribution of instructions by proportion of stalled pipeline slots (DFS on N-ary tree) measured by perf on machine 1.

Table 5.13: Parameters for the workload in Subsection 5.3.5.

Group	Iter	F1	F2	F3	S
1	1000	0.4	0.3	0.2	600,000
2	1000	0.4	0.3	0.2	800,000
3	1000	0.4	0.3	0.2	1,000,000
4	1000	0.4	0.3	0.2	1,200,000
5	1000	0.4	0.3	0.2	1,400,000
6	1000	0.32	0.28	0.24	600,000
7	1000	0.32	0.28	0.24	800,000
8	1000	0.32	0.28	0.24	1,000,000
9	1000	0.32	0.28	0.24	1,200,000
10	1000	0.32	0.28	0.24	1,400,000

5.3.5 Type Checking

The inputs, as mentioned in Section 4.4, include the iteration parameter *Iter*, the size of the arrays S and three functional parameters (i.e., *F1*, *F2* and *F3*) controlling the frequencies with which each concrete class is utilized as an implementation of a specific interface. An implicit parameter *F4* is calculated by $1 - F1 - F2 - F3$ and is thus not presented here. Without loss of generality, we further confine $F1 > F2 > F3 > F4$. The values are listed in Table 5.13. Only instructions of the type-checking routine are measured. The results gathered by VTune are shown in Table 5.14 and Figure 5.9. The results collected by perf are presented in Table 5.15 and Figure 5.10. As the results from both VTune and perf show, the measured instructions are affected by stalls. However, the types of stalls are different. For OpenJ9, the instructions are mainly affected by back-end stalls while the front end is the main source of stalls on HotSpot. This might be explained by the different data structures and instructions utilized by the JVMs to perform type checking. Note that the purpose of this thesis is to discuss the effects of stalls only and thus any results do not indicate performance advantage or disadvantage.

Table 5.14: Proportions of pipeline slots consumed by measured routine (type checking).

(a) OpenJ9

	Retiring		Front-end stall		Bad speculation		Back-end stall	
Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
1	16.12%	0.76%	13.13%	2.00%	22.45%	9.42%	48.31%	7.83%
2	15.35%	1.47%	13.77%	6.18%	21.30%	15.99%	49.58%	13.67%
3	15.57%	0.83%	15.39%	4.16%	11.18%	3.34%	57.85%	4.25%
4	15.13%	1.62%	17.82%	9.18%	30.38%	4.50%	36.67%	4.91%
5	15.49%	1.11%	14.11%	5.71%	33.68%	7.53%	36.73%	3.79%
6	16.70%	0.85%	14.26%	1.58%	13.46%	1.20%	55.58%	1.55%
7	20.82%	5.73%	19.88%	6.27%	10.13%	9.14%	49.17%	4.76%
8	14.71%	2.74%	18.13%	9.35%	35.05%	8.42%	32.11%	6.96%
9	16.40%	0.93%	12.91%	2.56%	37.36%	6.91%	33.33%	5.90%
10	16.17%	1.71%	15.35%	4.05%	22.18%	8.79%	46.31%	7.00%

(b) HotSpot

	Retiring		Front-end stall		Bad speculation		Back-end stall	
Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
1	23.96%	0.71%	59.84%	1.06%	8.45%	0.86%	7.75%	1.12%
2	25.04%	1.41%	60.47%	1.69%	7.37%	2.07%	7.12%	1.67%
3	23.78%	1.18%	59.55%	1.52%	9.15%	1.52%	7.53%	1.46%
4	24.02%	1.51%	60.26%	1.55%	8.42%	2.18%	7.30%	1.45%
5	23.90%	0.87%	60.57%	1.04%	9.57%	1.46%	5.95%	1.41%
6	24.15%	0.67%	59.60%	0.66%	7.59%	1.05%	8.67%	0.84%
7	24.89%	0.87%	60.28%	0.91%	7.94%	1.07%	6.88%	1.20%
8	24.76%	0.57%	60.49%	1.29%	7.96%	0.86%	6.80%	0.81%
9	25.95%	1.11%	61.08%	1.11%	7.20%	1.89%	5.77%	1.01%
10	24.82%	1.08%	60.50%	0.86%	7.81%	0.56%	6.87%	1.30%

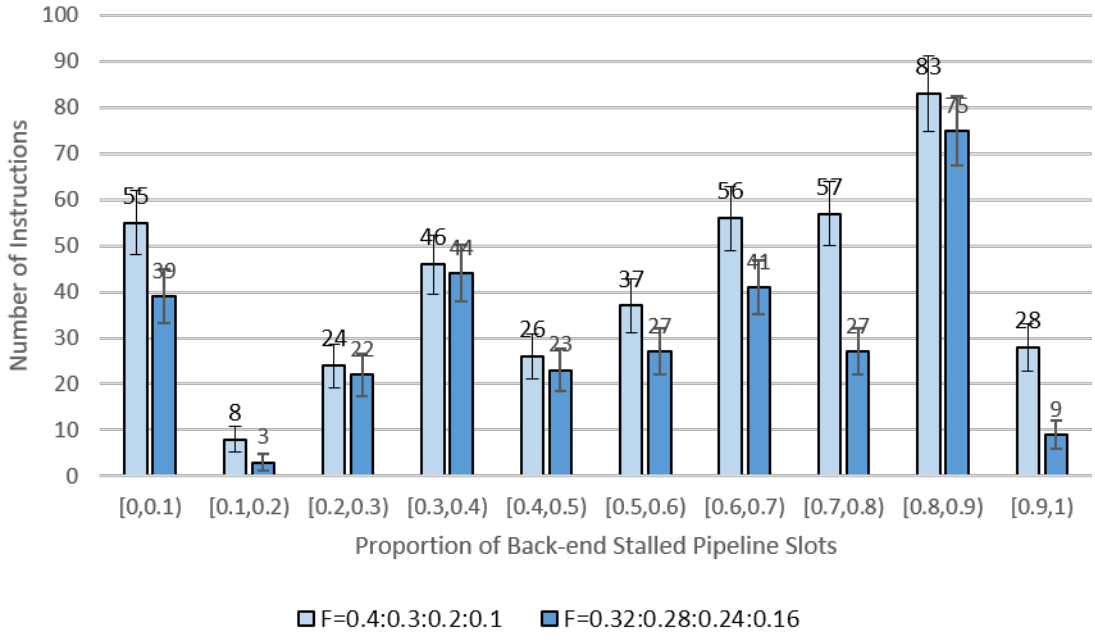
Table 5.15: Perf measurement results (type checking).

(a) OpenJ9

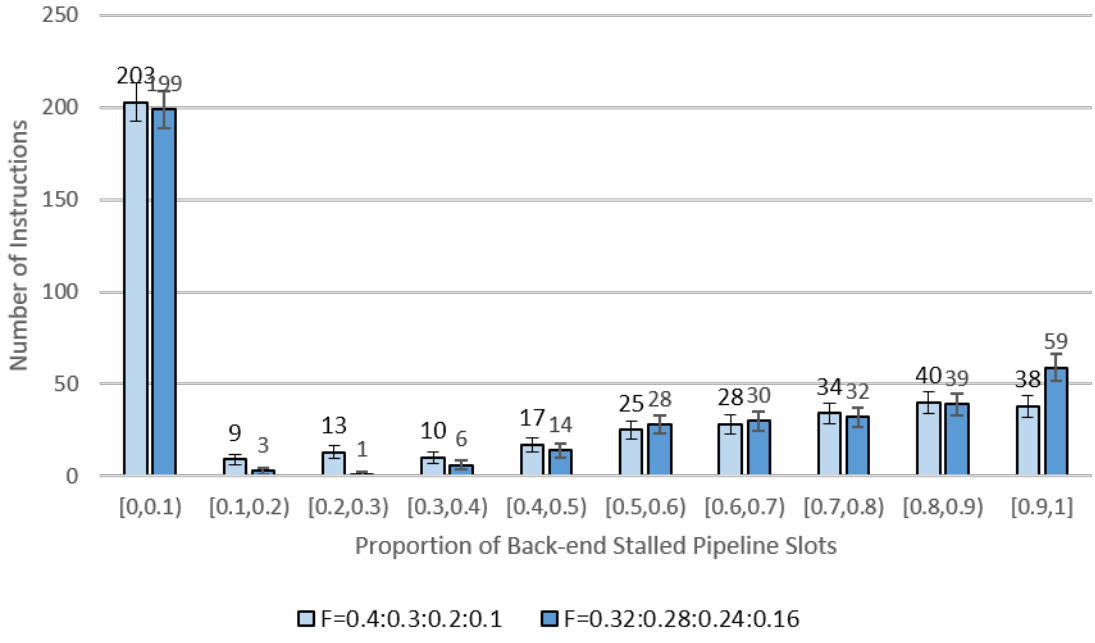
	Group	Front-end efficiency	Branch-miss rate	Cache-miss rate	Stalled pipeline slots
Average	**1**	74.67%	43.29%	56.26%	92.73%
Std Dev		3.52%	1.04%	7.54%	0.19%
Average	**2**	75.06%	36.39%	80.87%	93.14%
Std Dev		2.26%	1.23%	9.14%	0.23%
Average	**3**	72.04%	37.58%	84.19%	92.98%
Std Dev		3.20%	0.68%	7.11%	0.21%
Average	**4**	75.70%	36.16%	85.51%	93.12%
Std Dev		1.12%	1.11%	5.46%	0.15%
Average	**5**	74.54%	37.29%	80.88%	92.98%
Std Dev		1.70%	1.40%	11.20%	0.20%
Average	**6**	74.63%	43.87%	66.14%	92.97%
Std Dev		2.55%	1.15%	13.98%	0.16%
Average	**7**	74.73%	36.62%	80.89%	93.22%
Std Dev		2.42%	1.30%	5.48%	0.20%
Average	**8**	75.90%	38.61%	87.35%	93.19%
Std Dev		1.44%	0.98%	5.71%	0.20%
Average	**9**	76.51%	36.25%	92.35%	93.19%
Std Dev		2.56%	1.10%	6.29%	0.15%
Average	**10**	75.50%	38.38%	87.45%	93.12%
Std Dev		2.13%	1.09%	5.64%	0.20%

(b) HotSpot

	Group	Front-end efficiency	Branch-miss rate	Cache-miss rate	Stalled pipeline slots
Average	**1**	65.52%	31.64%	10.08%	92.30%
Std Dev		1.69%	6.11%	5.08%	0.03%
Average	**2**	69.29%	31.50%	28.02%	92.46%
Std Dev		1.62%	3.50%	4.63%	0.04%
Average	**3**	70.59%	32.71%	40.10%	92.56%
Std Dev		0.93%	3.24%	6.86%	0.05%
Average	**4**	70.30%	34.26%	29.79%	92.50%
Std Dev		1.49%	3.20%	8.42%	0.03%
Average	**5**	70.63%	31.11%	43.29%	92.56%
Std Dev		1.22%	2.69%	8.58%	0.02%
Average	**6**	64.77%	28.73%	12.34%	81.25%
Std Dev		1.97%	8.79%	3.14%	0.34%
Average	**7**	69.22%	27.62%	29.29%	82.86%
Std Dev		1.68%	4.57%	6.17%	0.19%
Average	**8**	70.32%	28.73%	47.05%	83.81%
Std Dev		1.24%	2.45%	8.80%	0.18%
Average	**9**	68.76%	29.15%	29.93%	83.36%
Std Dev		1.35%	1.90%	5.07%	0.32%
Average	**10**	70.33%	28.46%	41.59%	83.94%
Std Dev		1.49%	1.37%	5.31%	0.17%

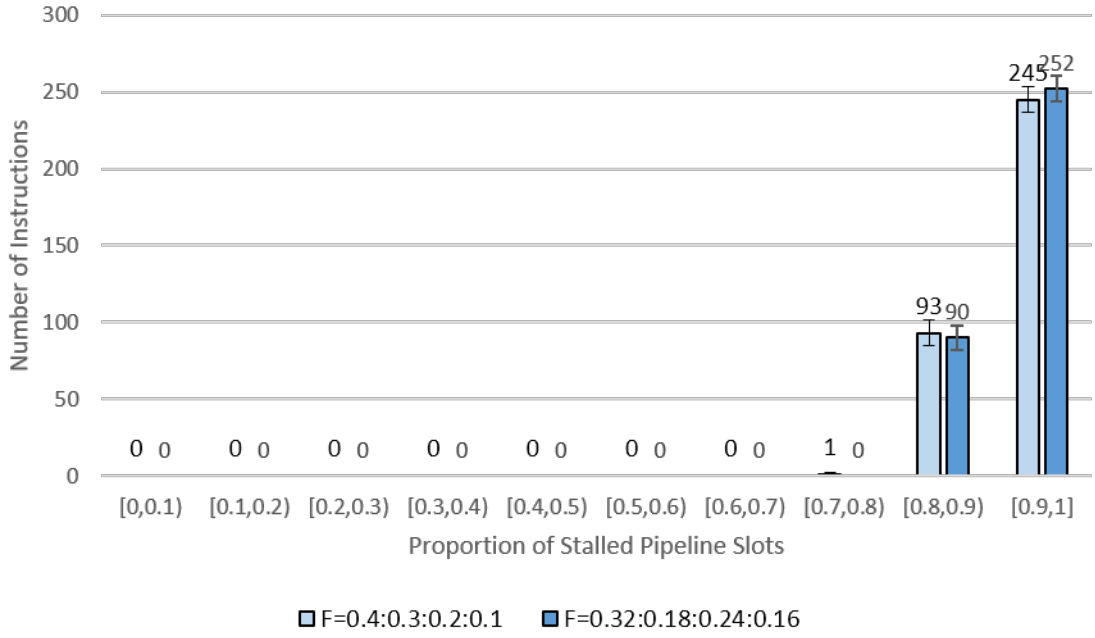


(a) OpenJ9

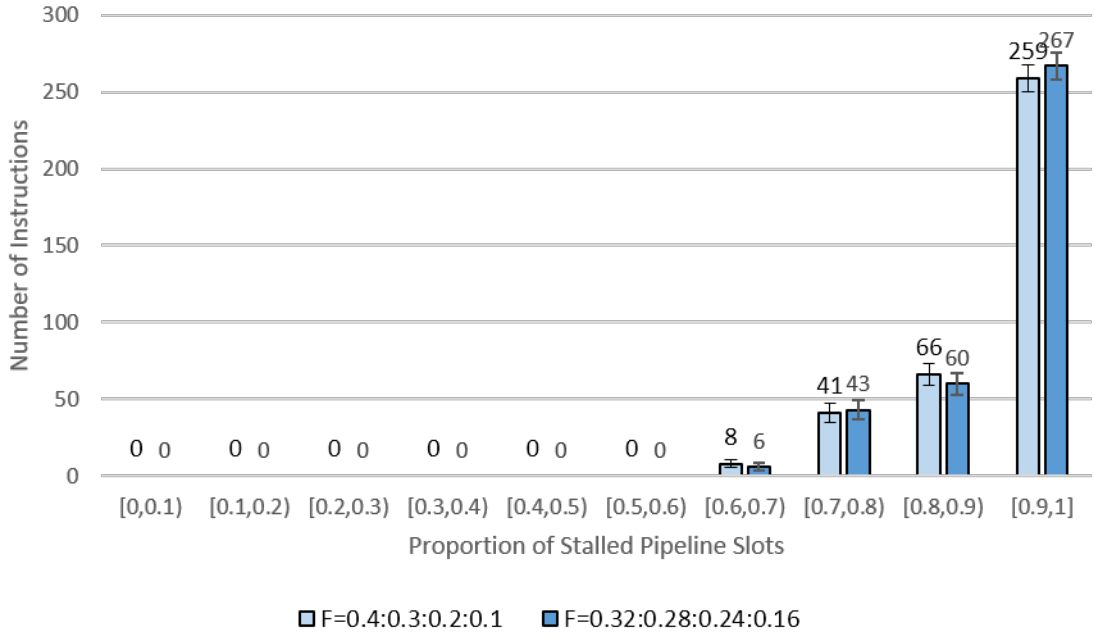


(b) HotSpot

Figure 5.9: Distribution of instructions by proportion of back-end stalled pipeline slots (type checking) measured by VTune on machine 2.



(a) OpenJ9



(b) HotSpot

Figure 5.10: Distribution of instructions by proportion of stalled pipeline slots (type checking) measured by perf on machine 1.

Table 5.16: Parameters for the workload in Subsection 5.3.6.

Group	Iter	S
1	1000	1,000,000
2	1000	1,100,000
3	1000	1,200,000
4	1000	1,300,000

5.3.6 Huge Objects

The input parameters of this benchmark include the iteration parameter *Iter* and the footprint parameter S controlling the number of huge objects, as shown in Table 5.16. Due to the huge object size, the values of S are confined in a relatively small range. The results gathered by VTune are shown in Table 5.17 and Figure 5.11. The results collected by perf are presented in Table 5.18 and Figure 5.12.

The results from VTune show that both threads handling the shared huge objects are heavily affected by back-end stalls. The results from perf show high cache-miss rates in all experiments. These cache misses are estimated to be the cause of the high proportion of stalled pipeline slots observed in the experiments.

Whether the cache misses and the back-end stalls observed in the experimental results are caused by false sharing or low cache locality remains unknown. The impact of false sharing is difficult to estimate [29], even though recent research [53, 54] proposed tools to automatically detect and mitigate false sharing. This is because false sharing is closely related to a series of external factors. However, it is a reasonable hypothesis that the probability of false sharing increases when the number of threads sharing the same memory section increases.

Table 5.17: Proportions of pipeline slots consumed by measured routine (huge objects).

(a) OpenJ9

		Retiring		Front-end stall		Bad speculation		Back-end stall	
	Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
Handle1	**1**	21.85%	1.65%	2.03%	0.94%	1.33%	0.55%	74.79%	0.73%
	2	23.67%	1.63%	3.05%	0.58%	3.03%	1.70%	70.25%	3.08%
	3	24.13%	1.49%	2.90%	0.71%	2.33%	1.79%	70.64%	1.44%
	4	21.78%	0.96%	2.41%	0.76%	2.29%	1.42%	73.51%	2.41%
Handle2	**1**	23.06%	0.60%	2.36%	0.15%	2.24%	1.67%	72.34%	1.22%
	2	23.13%	2.18%	2.94%	0.58%	2.71%	0.68%	71.21%	2.28%
	3	23.33%	1.11%	3.15%	0.54%	1.83%	0.89%	71.68%	0.78%
	4	23.09%	1.22%	2.80%	0.87%	2.58%	1.55%	71.53%	2.42%

(b) HotSpot

		Retiring		Front-end stall		Bad speculation		Back-end stall	
	Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
Handle1	**1**	16.51%	0.61%	2.89%	0.43%	7.03%	0.74%	73.57%	0.20%
	2	16.46%	0.91%	2.50%	0.67%	6.60%	1.37%	74.45%	2.04%
	3	17.80%	0.50%	2.66%	0.15%	7.31%	0.48%	72.24%	1.03%
	4	16.93%	0.65%	2.71%	0.17%	6.53%	0.70%	73.83%	0.51%
Handle2	**1**	16.79%	1.00%	2.60%	0.41%	5.81%	1.30%	74.80%	0.88%
	2	16.99%	0.64%	2.57%	0.11%	5.90%	1.39%	74.54%	1.83%
	3	18.69%	0.37%	2.42%	0.22%	6.47%	0.60%	72.41%	0.97%
	4	17.95%	0.53%	2.55%	0.40%	5.54%	1.12%	73.96%	1.07%

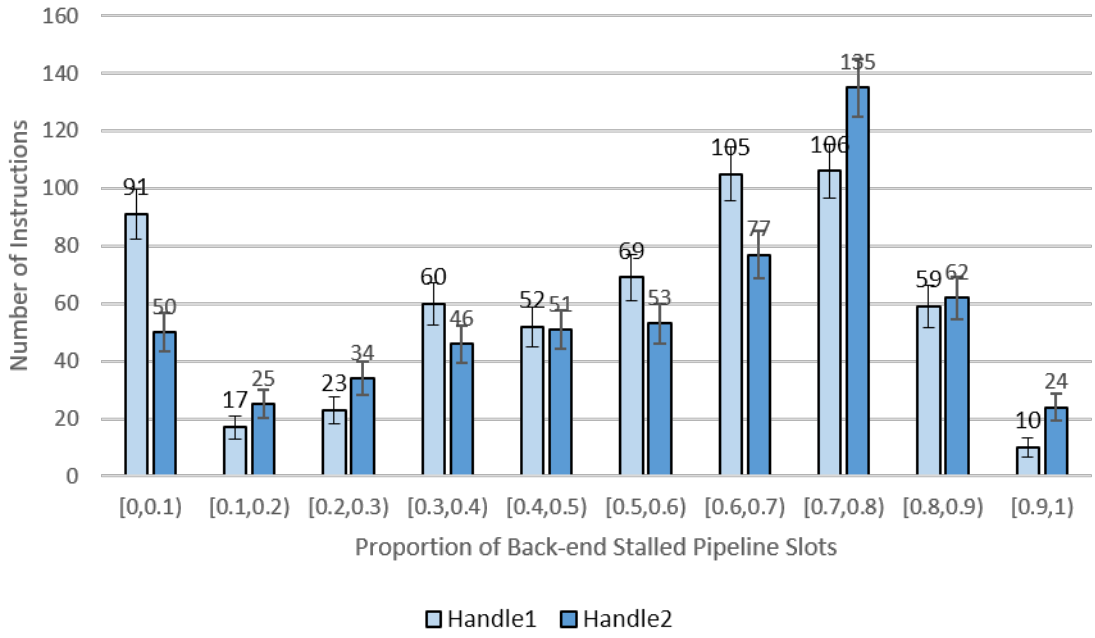
Table 5.18: Perf measurement results (huge objects).

(a) OpenJ9

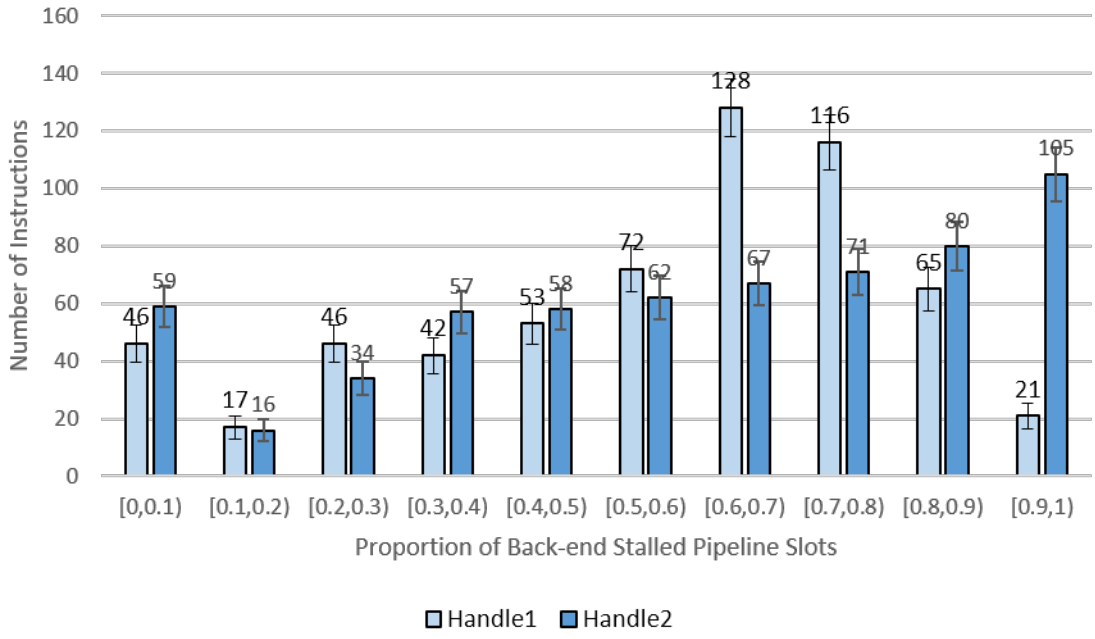
	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev
Group	**1**				**2**				**3**				**4**			
Method	**Handle1**		**Handle2**		**Handle1**		**Handle2**		**Handle1**		**Handle2**		**Handle1**		**Handle2**	
Front-end efficiency	75.24%	1.07%	72.78%	1.23%	75.62%	0.53%	73.42%	0.16%	75.41%	1.45%	72.69%	1.02%	76.39%	0.44%	73.63%	0.73%
Branch-miss rate	0.00%	0.00%	0.02%	0.03%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.01%	0.01%	0.01%	0.00%	0.00%
Cache-miss rate	96.38%	1.36%	95.92%	2.76%	97.12%	0.36%	96.74%	0.69%	97.69%	1.18%	97.55%	0.80%	97.18%	1.22%	96.94%	2.37%
Stalled pipeline slots	95.66%	0.16%	95.56%	0.17%	95.65%	0.13%	95.39%	0.18%	95.75%	0.08%	95.44%	0.18%	95.64%	0.04%	95.44%	0.17%

(b) HotSpot

	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev
Group	**1**				**2**				**3**				**4**			
Method	**Handle1**		**Handle2**		**Handle1**		**Handle2**		**Handle1**		**Handle2**		**Handle1**		**Handle2**	
Front-end efficiency	83.54%	0.35%	79.92%	0.42%	83.86%	0.15%	79.88%	0.36%	83.88%	0.40%	79.53%	0.23%	83.33%	0.46%	79.73%	0.45%
Branch-miss rate	0.22%	0.06%	0.26%	0.03%	0.20%	0.14%	0.14%	0.04%	0.26%	0.04%	0.25%	0.04%	0.33%	0.13%	0.38%	0.12%
Cache-miss rate	90.60%	0.79%	89.68%	0.64%	90.74%	0.30%	90.84%	0.44%	91.50%	0.54%	91.18%	0.34%	90.70%	1.21%	89.62%	1.43%
Stalled pipeline slots	90.64%	0.13%	96.17%	0.25%	91.55%	0.17%	96.40%	0.31%	91.66%	0.10%	96.48%	0.12%	91.45%	0.20%	96.43%	0.27%

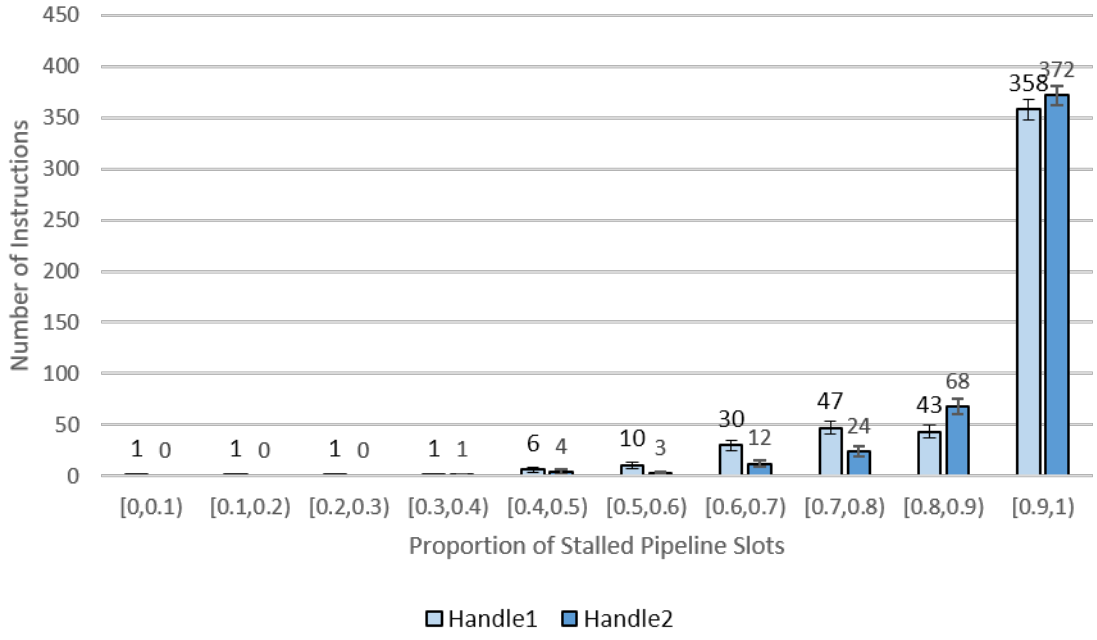


(a) OpenJ9

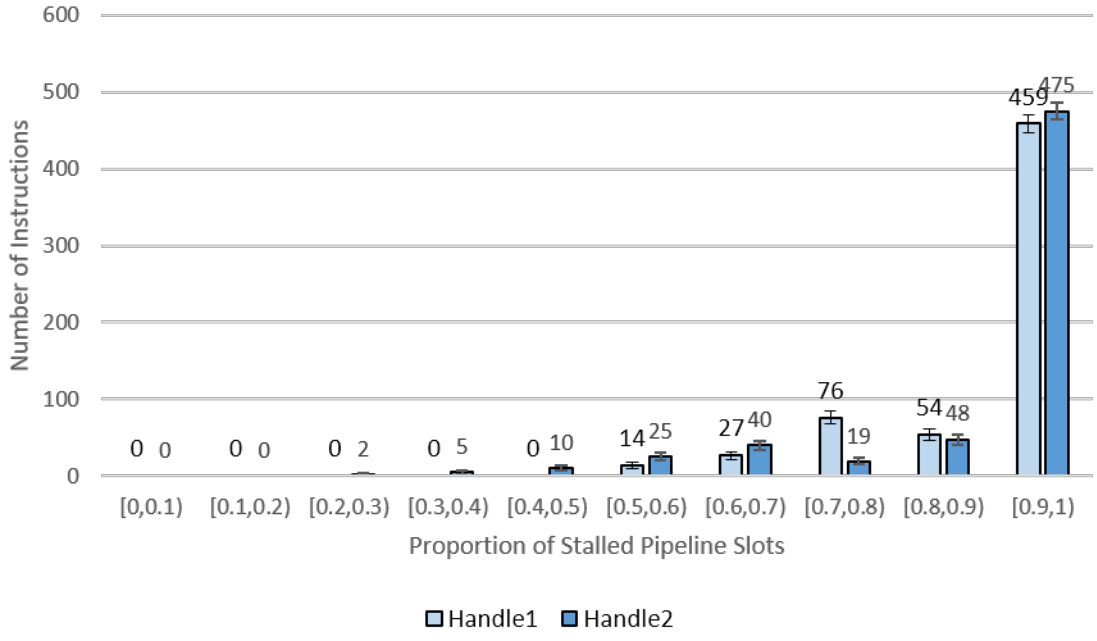


(b) HotSpot

Figure 5.11: Distribution of instructions by proportion of back-end stalled pipeline slots (huge objects) measured by VTune on machine 2.



(a) OpenJ9



(b) HotSpot

Figure 5.12: Distribution of instructions by proportion of stalled pipeline slots (huge objects) measured by perf on machine 1.

5.3.7 Repeat String Operations

The parameters of the merge sort benchmark include the iteration parameter, the number of threads and the size of the block each thread sorts before the results are merged. The inputs are listed in Table 5.19.

The measurement focuses on the bubbles introduced when the source of micro operations switch between the legacy pipeline and the micro sequencer. Such bubbles cannot be ignored when the switch is frequent. Since the array initialization routine is small in code size and there is only one repeat string operation in each routine, an alternative measurement is utilized instead of the distribution of instructions. The alternative approach measures the effect of the instruction sequence related to the repeat string operation (i.e., storing values of particular registers, performing repeat string operation and restoring the stored values) on the array initialization routine. The effects are calculated by dividing the number of pipeline slots in the instruction sequence with the number of pipeline slots in the whole routine. For instance, Table 5.20 shows an average of 18.96% of pipeline slots are stalled by the front end for experiments in parameter group 1. An average of 98.57% of such pipeline slots are within the sequence. If the measurement is performed by perf, then the effects are calculated by dividing the number of particular events in the sequence with the number of the same events in the array initialization routine. A comparison between the front-end efficiency of the array initialization routine and that of the sequence is also provided. For instance, Table 5.21 shows that the sequence's front-end efficiency is 95.12% of that of the array initialization routine for experiments on OpenJ9 in parameter group 1. This indicates the sequence is less efficient in front end than the rest of the whole routine. The table also shows that 84.04% of the stalled pipeline slots in the array initialization routine happens in the sequence.

The results are shown in Table 5.20 and Table 5.21.

The results from VTune show that the majority of front-end stalls and back-end

Table 5.19: Parameters for the workload in Subsection 5.3.7.

Group	Iteration	Threads	Block Size
1	1000	4	1000000
2	1000	4	1200000
3	1000	4	1400000
4	1000	4	1600000

Table 5.20: VTune measurement results (merge sort).

(a) OpenJ9

		Retiring		Front-end stall		Bad speculation		Back-end stall	
	Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
Array Initialization Routine	1	47.34%	5.93%	18.96%	2.21%	6.30%	1.80%	27.39%	4.88%
	2	43.43%	2.72%	21.49%	1.58%	3.93%	1.39%	31.15%	3.31%
	3	41.51%	6.58%	22.27%	1.92%	4.48%	0.94%	31.74%	7.18%
	4	36.02%	2.19%	26.67%	3.11%	6.87%	2.41%	30.44%	3.65%
Effect of REP operation	1	74.80%	4.67%	98.57%	0.96%	51.87%	13.15%	62.91%	8.31%
	2	80.05%	4.58%	98.50%	0.84%	41.81%	13.25%	71.90%	7.22%
	3	77.76%	2.50%	99.26%	0.25%	57.04%	12.99%	64.53%	5.21%
	4	81.52%	3.83%	98.92%	0.72%	40.87%	9.88%	74.88%	7.40%

(b) HotSpot

		Retiring		Front-end stall		Bad speculation		Back-end stall	
	Group	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**	**Average**	**Std Dev**
Array Initialization Routine	**1**	45.25%	6.27%	23.10%	1.74%	1.17%	0.58%	30.48%	6.00%
	2	39.07%	9.45%	32.03%	12.79%	0.93%	0.45%	27.97%	5.06%
	3	29.76%	8.31%	43.74%	13.84%	0.91%	0.29%	25.60%	6.44%
	4	39.13%	12.03%	34.78%	11.77%	1.15%	0.47%	24.94%	5.30%
Effect of REP operation	**1**	84.28%	3.46%	99.43%	0.65%	3.02%	4.88%	75.68%	7.12%
	2	86.06%	4.70%	99.65%	0.27%	7.71%	11.58%	74.10%	5.98%
	3	90.05%	2.95%	99.75%	0.26%	3.78%	3.34%	77.10%	3.04%
	4	88.47%	3.86%	99.47%	0.33%	8.43%	10.07%	77.45%	8.13%

stalls in the array initialization routine exist in the instruction sequence related to repeat string operation. Both type of stalls wasted a non-trivial proportion of the consumed pipeline slots. The results from perf show that the sequence is the major source of stalls in the array initialization routine and almost all cache misses are caused by the sequence. The results from VTune strongly suggest that the bubbles introduced by the repeat string operation are the major cause of front-end stalls during array initialization. However, the results from perf do not provide strong evidence for this conclusion. Possible reasons include the systematic error of the estimation approach, the difference of the experimental machines, the measurement implementation employed or other factors.

Table 5.21: Perf measurement results (merge sort).

(a) OpenJ9

		Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev
	Group	**1**		**2**		**3**		**4**	
Array initialization routine	**Front-end efficiency**	79.68%	2.06%	80.20%	0.90%	77.06%	3.85%	77.75%	4.16%
	Branch-miss rate	0.09%	0.06%	0.07%	0.05%	2.78%	3.70%	2.64%	3.67%
	Cache-miss rate	45.64%	8.39%	37.95%	2.69%	35.60%	2.60%	39.23%	2.64%
	Stalled pipeline slots	76.46%	0.21%	74.88%	0.22%	75.55%	0.79%	74.19%	0.48%
Sequence	**Front-end efficiency**	75.79%	2.15%	76.24%	1.91%	76.59%	1.51%	76.75%	1.21%
	Branch-miss rate	0.02%	0.05%	0.07%	0.07%	0.05%	0.05%	0.05%	0.04%
	Cache-miss rate	46.14%	8.89%	37.67%	2.99%	35.89%	2.58%	40.28%	2.79%
	Stalled pipeline slots	81.34%	0.19%	80.45%	0.13%	80.69%	0.12%	80.41%	0.16%
Effect of sequence	**Front-end efficiency**	95.12%	0.99%	95.06%	1.42%	99.64%	5.31%	98.93%	4.16%
	Branch misses	13.33%	30.55%	61.67%	43.01%	23.61%	32.60%	32.58%	37.34%
	Cache misses	98.13%	1.33%	97.12%	1.22%	82.09%	22.32%	80.64%	23.80%
	Stalled pipeline slots	84.04%	0.56%	83.53%	0.22%	83.97%	0.99%	84.69%	0.18%

(b) HotSpot

		Average	Std Dev	Average	Std Dev	Average	Std Dev	Average	Std Dev
	Group	**1**		**2**		**3**		**4**	
Array initialization routine	**Front-end efficiency**	78.62%	0.76%	78.03%	0.78%	78.40%	0.85%	79.11%	0.97%
	Branch-miss rate	0.02%	0.02%	0.03%	0.02%	0.02%	0.01%	0.02%	0.01%
	Cache-miss rate	86.49%	4.80%	80.64%	3.13%	77.36%	5.15%	79.26%	4.34%
	Stalled pipeline slots	74.01%	0.18%	74.23%	0.18%	74.04%	0.16%	73.34%	0.17%
Sequence	**Front-end efficiency**	85.33%	1.67%	84.40%	1.01%	84.08%	1.09%	84.81%	1.80%
	Branch-miss rate	0.03%	0.03%	0.04%	0.03%	0.01%	0.02%	0.02%	0.02%
	Cache-miss rate	88.27%	5.01%	83.30%	3.77%	78.44%	5.30%	80.59%	4.18%
	Stalled pipeline slots	75.13%	0.19%	75.28%	0.22%	75.70%	0.17%	75.43%	0.20%
Effect of sequence	**Front-end efficiency**	108.54%	1.49%	108.17%	1.53%	107.26%	1.11%	107.20%	1.08%
	Branch misses	79.17%	33.07%	66.67%	19.25%	29.17%	34.11%	50.00%	43.23%
	Cache misses	98.94%	0.47%	99.17%	0.42%	99.15%	0.44%	99.37%	0.35%
	Stalled pipeline slots	74.59%	0.41%	74.65%	0.43%	76.32%	0.14%	76.74%	0.19%

5.4 Summary

In this chapter the validation results of all proposed benchmarks are presented. Given the considered parameters, all benchmarks display expected stalls and thus are validated. Although we experiment on different JVMs and micro architectures, the purpose is not comparing the performance of any JVM on any platform. Instead, the experiments indicate that the same workload generates stalls across different JVMs on different x86 micro architectures. Such results imply that the proposed benchmarks, as well as the stalls they reflect and generate, are universal. Therefore, the benchmarks are validated and can be utilized for further research on stalls. In the future, more works should be dedicated to detecting and mitigating stalls. Further discussion is included in Chapter 6.

Chapter 6

Conclusion and Future Work

This thesis set out to explore the effects of stalls on various Java applications, and the viability to induce such stalls with micro benchmarks.

The survey in Chapter 3 indicates that stalls are a universal and significant performance bottleneck for Java applications on x86 micro architectures. The observed stalls, especially back-end stalls, happen in various Java applications and cause a significant number of wasted CPU cycles. In addition, the cause of stalls shows similarity in different Java applications. This indicates the possibility to induce such stalls with micro benchmarks.

Based on the results from Chapter 3, micro benchmarks focusing on stalls are designed and presented in Chapter 4. The experimental results in Chapter 5 show that the stalls observed in various complex Java applications can be reproduced by the proposed micro benchmarks. In addition, the generated stalls are common in at least two JVM implementations on two x86 micro architectures.

Our research indicates that more future efforts should be dedicated to detecting and mitigating stalls. In the remaining sections of this chapter, several future paths are discussed.

6.1 Stalls on Other HLLVMs

So far, our discussion focuses on Java Virtual Machines, which is just one category of HLLVMs. The effect of stalls on other HLLVMs, such as CLR and PyPy mentioned in Subsection 2.1.1, remain unknown. However, some speculations based on language features are reasonable. For instance, C# applications may exhibit similar stalls to those in Java applications since both languages are statically typed, object-oriented programming languages. On the other hand, applications executing on PyPy may exhibit stalls in different locations because Python is a dynamically-typed language. This difference introduces more type-checking in Python applications and often causes Python runtimes to employ different type-checking implementations. The applications these languages implement also vary. However, stalls are assumed to be a problem in these applications. In addition, certain program logics, such as accessing referenced data, are universal. Thus, it is possible to develop optimizations for language runtime infrastructures such as Eclipse OMR [4], which is the basis of OpenJ9.

6.2 Stalls on Other Micro Architectures

While our discussion is about x86 ISA and micro architectures, stalls are expected to exist on other micro architectures utilizing different instruction sets. The differences in the cause of stalls and the methods to detect them remain unknown and thus require further experiments in the future. However, as is shown in Chapter 5, estimations based on generic PMU events are possible when certain PMU events are supported by the micro architecture.

6.3 Optimizations

While we know that stalls cause performance degradation and that certain routines cause stalls, we still lack information about the cost and benefits of stall-focused optimizations. For Just-in-Time optimizations to benefit the application, heuristics are necessary so that the improved performance outweighs the cost to apply the optimizations. For HLLVMs developed for general-purpose programming languages, such as Java, this can be even more difficult. Another possible issue of developing Just-in-Time optimizations for stalls is gathering information during runtime. This may require establishing an abstract CPU model with which the JVM can interact. This is especially important in cloud environments since access to hardware-level information might be limited for security.

Bibliography

[1] *Acme Air sample and benchmark (monolithic simple version)*, https://github.com/blueperf/acmeair-monolithic-java, (Accessed on 02/17/2021).

[2] *Apache Derby*, https://db.apache.org/derby/, (Accessed on 02/17/2021).

[3] *Develop your next breakthrough with Java — go.Java — Oracle*, https://go.java/developer-opportunities/, (Accessed on 03/01/2021).

[4] *Eclipse OMR: IBM open sources its runtime technology – IBM developer*, https://developer.ibm.com/depmodels/cloud/projects/eclipse-omr/, (Accessed on 05/19/2021).

[5] *Eclipse OpenJ9*, https://www.eclipse.org/openj9/, (Accessed on 02/16/2021).

[6] *Fix performance bottlenecks with Intel® VTune™ Profiler*, https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html, (Accessed on 02/17/2021).

[7] *GitHub - Intel-bigdata/HiBench: HiBench is a big data benchmark suite.*, https://github.com/Intel-bigdata/HiBench, (Accessed on 02/26/2021).

[8] *The HotSpot group*, https://openjdk.java.net/groups/hotspot/, (Accessed on 02/16/2021).

[9] *Java EE7: DayTrader7 sample*, https://github.com/WASdev/sample.daytrader7, (Accessed on 02/17/2021).

[10] *Java Microbenchmark Harness (JMH)*, https://github.com/openjdk/jmh, (Accessed on 02/17/2021).

[11] *Java Platform, Enterprise Edition (Java EE) — Oracle technology network — Oracle Canada*, https://www.oracle.com/ca-en/java/technologies/java-ee-glance.html, (Accessed on 03/01/2021).

[12] *Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*, https://www.agner.org/optimize/instruction_tables.pdf, (Accessed on 03/25/2021).

[13] *The most popular database for modern apps — MongoDB*, https://www.mongodb.com/3, (Accessed on 02/17/2021).

[14] *numactl(8) - Linux man page*, https://linux.die.net/man/8/numactl, (Accessed on 03/03/2021).

[15] *OMR technology*, https://www.eclipse.org/omr/, (Accessed on 02/16/2021).

[16] *perf(1) - linux manual page*, https://man7.org/linux/man-pages/man1/perf.1.html, (Accessed on 02/16/2021).

[17] *Renaissance Suite, a benchmark suite for the JVM*, https://renaissance.dev/, (Accessed on 02/17/2021).

[18] *TPC-H homepage*, http://www.tpc.org/tpch/, (Accessed on 03/10/2021).

[19] *WebSphere Liberty — IBM*, https://www.ibm.com/cloud/websphere-liberty, (Accessed on 02/17/2021).

[20] *What is managed code? — Microsoft Docs*, https://docs.microsoft.com/en-us/dotnet/standard/managed-code, (Accessed on 03/01/2021).

[21] *Intel® 64 and IA-32 architectures optimization reference manual*, https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html, May 2020, (Accessed on 03/24/2021).

[22] *Intel® 64 and IA-32 architectures software developer manuals*, https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html, April 2021, (Accessed on 05/03/2021).

[23] Andreas Abel and Jan Reineke, *uops.info: Characterizing latency, throughput, and port usage of instructions on Intel microarchitectures*, Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 673–686.

[24] John Aycock, *A brief history of just-in-time*, ACM Computing Surveys (CSUR) **35** (2003), no. 2, 97–113.

[25] David F Bacon, Stephen J Fink, and David Grove, *Space-and time-efficient implementation of the Java object model*, European Conference on Object-Oriented Programming, Springer, 2002, pp. 111–132.

[26] David F Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano, *Thin locks: Featherweight synchronization for Java*, Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, 1998, pp. 258–268.

[27] Didier H Besset, *Object-oriented implementation of numerical methods: An introduction with Java & Smalltalk*, Morgan Kaufmann, 2001.

[28] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley, *Myths and realities: The performance impact of garbage collection*, ACM SIGMETRICS Performance Evaluation Review **32** (2004), no. 1, 25–36.

[29] William J Bolosky and Michael L Scott, *False sharing and its effect on shared memory performance*, Proceedings of the Fourth symposium on Experiences with distributed and multiprocessor systems, Citeseer, 1993.

[30] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo, *Tracing the meta-level: PyPy's tracing JIT compiler*, Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, 2009, pp. 18–25.

[31] Gilad Bracha, *The programming language Jigsaw: Mixins, modularity and multiple inheritance*, Ph.D. thesis, Dept. of Computer Science, University of Utah, 1992.

[32] J Mark Bull, Lorna A Smith, Carwyn Ball, Linday Pottage, and Robin Freeman, *Benchmarking Java against C and Fortran for scientific applications*, Concurrency and Computation: Practice and Experience **15** (2003), no. 3-5, 417–430.

[33] Cliff Click and Michael Paleczny, *A simple graph-based intermediate representation*, ACM Sigplan Notices **30** (1995), no. 3, 35–49.

[34] Brad J Cox, *Object-oriented programming: an evolutionary approach*, (1986).

[35] Jeffrey Dean and Sanjay Ghemawat, *MapReduce: simplified data processing on large clusters*, Communications of the ACM **51** (2008), no. 1, 107–113.

[36] Julian Dolby and Andrew Chien, *An automatic object inlining optimization and its evaluation*, Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, 2000, pp. 345–357.

[37] Julian Dolby and Andrew A Chien, *An evaluation of automatic object inline allocation techniques*, ACM SIGPLAN Notices **33** (1998), no. 10, 1–20.

[38] Taees Eimouri, Kenneth B Kent, and Aleksandar Micic, *Effects of false sharing and locality on object layout optimization for multi-threaded applications*, 2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), IEEE, 2016, pp. 1–5.

[39] Michael L Fredman, Robert Sedgewick, Daniel D Sleator, and Robert E Tarjan, *The pairing heap: A new form of self-adjusting heap*, Algorithmica **1** (1986), no. 1-4, 111–129.

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: elements of reusable object-oriented software*, Pearson Education India, 1995.

[41] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley, *The Java language specification, Java SE 8 edition*, Addison-Wesley Professional, 2014.

[42] Philipp Haller and Martin Odersky, *Actors that unify threads and events*, International Conference on Coordination Languages and Models, Springer, 2007, pp. 171–190.

[43] Urs Hölzle, Craig Chambers, and David Ungar, *Optimizing dynamically-typed object-oriented languages with polymorphic inline caches*, European Conference on Object-Oriented Programming, Springer, 1991, pp. 21–38.

[44] Urs Hölzle and David Ungar, *Optimizing dynamically-dispatched calls with run-time type feedback*, Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, 1994, pp. 326–336.

[45] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang, *The HiBench benchmark suite: Characterization of the MapReduce-based data analysis*, 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010), IEEE, 2010, pp. 41–51.

[46] Dmitry Jemerov and Svetlana Isakova, *Kotlin in action*, Manning Publications Company, 2017.

[47] Andrew Kennedy and Don Syme, *Design and implementation of generics for the .Net common language runtime*, Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, 2001, pp. 1–12.

[48] Donald Ervin Knuth, *The art of computer programming*, vol. 3, Pearson Education, 1997.

[49] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox, *Design of the Java HotSpot™ client compiler for Java 6*, ACM Transactions on Architecture and Code Optimization (TACO) **5** (2008), no. 1, 1–32.

[50] Christoph Lameter, *NUMA (Non-Uniform Memory Access): An overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors.*, Queue **11** (2013), no. 7, 40–51.

[51] Doug Lea, *A Java Fork/Join framework*, Proceedings of the ACM 2000 conference on Java Grande, 2000, pp. 36–43.

[52] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley, *The Java virtual machine specification*, Pearson Education, 2014.

[53] Tongping Liu and Emery D Berger, *Sheriff: precise detection and automatic mitigation of false sharing*, Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, 2011, pp. 3–18.

[54] Tongping Liu and Xu Liu, *Cheetah: detecting false sharing efficiently and effectively*, 2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2016, pp. 1–11.

[55] Chi-Keung Luk and Todd C Mowry, *Compiler-based prefetching for recursive data structures*, Proceedings of the seventh international conference on Architectural support for programming languages and operating systems, 1996, pp. 222–233.

[56] Jonathan Mak and Alan Mycroft, *Limits of parallelism using dynamic dependency graphs*, Proceedings of the Seventh International Workshop on Dynamic Analysis, 2009, pp. 42–48.

[57] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al., *Mllib: Machine learning in Apache Spark*, The Journal of Machine Learning Research **17** (2016), no. 1, 1235–1241.

[58] Martin Odersky, Lex Spoon, and Bill Venners, *Programming in Scala*, Artima Inc, 2008.

[59] Michael Paleczny, Christopher Vick, and Cliff Click, *The Java Hotspot™ server compiler*, Proceedings of the Java Virtual Machine Research and Technology Symposium, vol. 1, 2001, pp. 1–12.

[60] Mark S Papamarcos and Janak H Patel, *A low-overhead coherence solution for*

multiprocessors with private cache memories, Proceedings of the 11th annual international symposium on Computer architecture, 1984, pp. 348–354.

[61] Maria Patrou, Kenneth B Kent, Gerhard W Dueck, Charlie Gracie, and Aleksandar Micic, *Numa awareness: Improving thread and memory management*, 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2018, pp. 119–123.

[62] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, et al., *Renaissance: Benchmarking suite for parallel applications on the JVM*, Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019, pp. 31–47.

[63] Kumar Shiv, Kingsum Chow, Yanping Wang, and Dmitry Petrochenko, *SPECjvm2008 performance characterization*, SPEC Benchmark Workshop, Springer, 2009, pp. 17–35.

[64] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler, *The Hadoop distributed file system*, 2010 IEEE 26th symposium on mass storage systems and technologies (MSST), IEEE, 2010, pp. 1–10.

[65] Jim Smith and Ravi Nair, *Virtual machines: versatile platforms for systems and processes*, Elsevier, 2005.

[66] William Stallings, *Computer organization and architecture: designing for performance*, Pearson Education India, 2003.

[67] Mark Stefik and Daniel G Bobrow, *Object-oriented programming: Themes and variations*, AI magazine **6** (1985), no. 4, 40–40.

[68] Bjarne Stroustrup, *The C++ programming language*, Pearson Education India, 2000.

[69] Abhijit Taware, Kenneth B Kent, Gerhard W Dueck, and Charlie Gracie, *Cold object identification and segregation using page protection and profiling*, 2020 9th Mediterranean Conference on Embedded Computing (MECO), IEEE, 2020, pp. 1–6.

[70] Michael E Thomadakis, *The architecture of the Nehalem processor and Nehalem-EP SMP platforms*, Resource **3** (2011), no. 2, 30–32.

[71] David W Wall, *Limits of instruction-level parallelism*, Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, 1991, pp. 176–188.

[72] Christian Wimmer, *Automatic object inlining in a Java virtual machine*, Trauner, 2008.

[73] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu, *Can we trust profiling results? understanding and fixing the inaccuracy in modern profilers*, Proceedings of the ACM International Conference on Supercomputing, 2019, pp. 284–295.

[74] Ahmad Yasin, *A top-down method for performance analysis and counters architecture*, 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, 2014, pp. 35–44.

[75] Scott Young, Michael Flawn, Gerhard W. Dueck, Kenneth B. Kent, and Charlie Gracie, *Persistent memory storage of cold regions in the openj9 java virtual machine*, Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering, CASCON '18, IBM Corp., 2018, p. 213–223.

[76] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman,

Michael J Franklin, et al., *Apache Spark: a unified engine for big data processing*, Communications of the ACM **59** (2016), no. 11, 56–65.

Vita

Candidate's full name: Zhuoran Li
University attended (with dates and degrees obtained):
Bachelor of Software Engineering, Sun Yat-sen University, 2019
Publications:

1. Z. Li, H.Arafat, Y.Kim, K. B. Kent, D. Bremner, M. Fleming and A. Craik. "A Survey of Stalls in Java". Workshop. 4th Workshop on Advances in Open Runtime Technology for Cloud Computing (AORCPT 2020), 30th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2020), Markham, Canada, November 10-13, 2020.

2. Z. Li, K. B. Kent, D. Bremner, M. Fleming and A. Craik. "Towards Stall Focused Benchmark for JVMs on x86 Architecture". Poster. 30th Annual International Conference on Computer Science and Software Engineering (CASCON x EVOKE 2020), Markham, Canada, November 10-13, 2020.

Conference Presentations: N/A