Impact of Garbage Collection Policies on Load Stalls on AArch64 in OpenJ9

by

Jonas Rouven Schönauer

Bachelor of Science (Applied Mathematics and Computer Science),
FH-Aachen, Germany, 2021

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

Supervisors: David D. Bremner, PhD., Computer Science
Kenneth B. Kent, PhD., Computer Science

Examining Board: André Hinkenjann, PhD., Computer Science, Chair
Gerhard W. Dueck, PhD., Computer Science
Brent R. Petersen, PhD., Electrical and Computer Engineering

This thesis is accepted by the
Dean of Graduate Studies

The University of New Brunswick

April, 2024

Abstract

The Java Virtual Machine (JVM) is used on many devices worldwide including the Reduced Instruction Set Computer (RISC) architecture ARM AArch64. However, AArch64—like many other modern architectures—suffers from load stalls, i.e., the delays when the Central Processing Unit (CPU) is waiting for data to be fetched from memory. This work evaluates the impact of Garbage Collection (GC) algorithms on load stalls as the algorithms can change the position of objects, which could increase locality. To analyze this impact, an algorithm to classify load stalls in Just-In-Time (JIT) compiled code is developed and, using a wide range of benchmarks, data is collected from performance counters. Comparing the obtained values between different GC algorithms and collection methods reveals that the impact of object position changes on cache performance is not as high as expected. Furthermore, the observed stalls indicate that the out-of-order pipeline is not able to keep up with the memory requests.

Acknowledgments

This research was conducted within the Centre for Advanced Studies–Atlantic, Faculty of Computer Science, University of New Brunswick. I am grateful for the colleagues and facilities of CAS–Atlantic in supporting our research. I would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Furthermore, I would also like to thank the New Brunswick Innovation Foundation for contributing to this project. I thank my supervisors Dr. David Bremner and Dr. Kenneth Kent for their guidance that helped me to get a positive, first impression of academic work. They are both outstanding supervisors. Furthermore, I thank the examiners Dr. André Hinkenjann, Dr. Gerhard Dueck and Dr. Brent Petersen for their comments and suggestions on my thesis.

I am also incredibly grateful for the support from Julian Wang from IBM Canada. His great technical input and expertise was immensely helpful in carrying out this research. I thank our project manager Stephen MacKay for his feedback to improve my writing, hopefully making this work more enjoyable for the reader. I am also thankful for the technical support by our research assistant DeVerne Jones, who helped me getting familiar with AArch64 and supported me in setting up the infrastructure. I would like to extend my gratitude to Hassan Arafat and Georgiy Krylov who listened to my ideas and challenged them—it helped me push my boundaries.

Table of Contents

List of Tables

List of Figures

List of Code Listings

List of Algorithms

Abbreviations

ALU	Arithmetic Logic Unit
CFG	Control-Flow Graph
CPU	Central Processing Unit
DMB	Data Memory Barrier
EX	Execute; Pipeline Stage
GC	Garbage Collection
ID	Instruction Decode; Pipeline Stage
IF	Instruction Fetch; Pipeline Stage
IL	Intermediate Language
IPC	Instructions Per Cycle
IQR	Interquartile Range
ISA	Instruction Set Architecture
JBC	Java Byte Code
JIT	Just-In-Time
JRE	Java Runtime Environment
JVM	Java Virtual Machine
LOA	Large Object Area; Heap Region
LSU	Load/Store Unit
MEM	Memory; Pipeline Stage
NUMA	Non-Uniform Memory Access

OS Operating System

PMU Performance Monitor Unit

RISC Reduced Instruction Set Computer

SIMD Single Instruction, Multiple Data

SOA Small Object Area; Heap Region

STW Stop-The-World; Garbage Collection Pause

TLB Translation Lookaside Buffer

VM Virtual Machine

WB Write Back; Pipeline Stage

Chapter 1

Introduction

Since the instruction set architecture AArch64 was developed, it has had a high impact on the smartphone market [1]. In recent years efforts were made to adopt AArch64 on server platforms [2].

Even though both environments sound fundamentally different, they make extensive use of Java Virtual Machine (JVM) implementations to execute code [3], [4]. On one hand, JVMs offer an abstraction layer that allows platform independent development of applications for diverse consumer devices [5]. On the other hand, the Just-In-Time (JIT) compilation available in most JVM implementations uses profile guided optimizations to generate highly optimized code [6]. This is particularly useful in server environments where applications run for a long time and optimal performance is desired. Additionally, both environments highly benefit from the managed memory system used in the JVM, because it allows for faster development and less error prone software [7].

However, on modern platforms—including AArch64—the execution time of programs is highly impacted by the speed difference between Central Processing Units (CPUs) and the main memory [8]. This gap results in load stalls and the processor must wait for data from memory before execution can continue [9].

1.1 Problem Statement

The aforementioned speed difference has a large impact on program execution time, including JVMs. It is thus especially important to understand the underlying causes of load stalls in JVMs running on AArch64 to mitigate performance losses. This includes overall JVM load stall behavior. In addition, this work will also investigate load stalls in JIT compiled code, because it is particularly beneficial to get optimal performance there. Moreover current JVM implementations offer different Garbage Collection (GC) policies, which might change the position of an object in the heap. Consequently, the proposed work furthermore aims at identifying their impact on the stalling behavior of applications. In all, the following research question arises.

What is the impact of different GC policies on the load stall behavior of applications written for a JVM on AArch64?

To answer this broad question in regard to the previously mentioned subtopics, the question is subdivided and further specified by the following three questions.

1. *How can load stalls be observed in JIT code on AArch64?*
2. *What is the impact of GC policies on the load stall behavior as well as the overall performance?*
3. *Are there machine code or Java Byte Code (JBC) instruction sequences that often lead to load stalls? Do they differ for different GC policies?*

1.2 Structure

Finding an answer to these questions is the main objective of this thesis and this section will give an overview of the thesis structure and how it relates to the research questions. To gain in-depth knowledge of the thesis domain Chapter 2 will discuss

the background with an emphasis on AArch64, load stalls, JVMs and opportunities to measure and quantify the impact of load stalls.

Based on this knowledge a literature review is conducted focusing on related work in Chapter 3. In particular, works that include comparisons between different GC algorithms as well as the impact of GCs on locality are presented.

Next, the methodology of the thesis and the benchmarks are given in Chapter 4, setting the general outline for the following two chapters. In addition, hypotheses regarding the performance behavior of different GC algorithms are formulated.

Then, to answer the first question, a previously developed framework is extended in Chapter 5 to automatically classify instructions—i.e., determining if they are load stalls—in JIT compiled code. The framework is also adapted to help answer the second question by collecting less granular data.

In Chapter 6 the data collected is analyzed and presented, answering the second and third question. In particular the hypotheses that are developed and presented in the methodology chapter are verified.

Finally, the findings of the thesis are summarized in Chapter 7, relating back to the original research question. In the same chapter, potential future work is presented and discussed.

Chapter 2

Background

In this chapter, the background of the thesis is presented. Starting with the AArch64 hardware layout and pipelines, load stalls are introduced. This is followed by information about the JVM and GC. Lastly, benchmarks and tools that can be used to measure the impact of load stalls are presented.

2.1 AArch64

AArch64 is an execution state[1] introduced by the ARMv8-A instruction set architecture. It is developed by ARM Ltd. and defines a Reduced Instruction Set Computer (RISC) which is capable of processing 64-bit data as well as addressing memory with 64-bit addresses [10]. The most common area of application for AArch64 is in mobile devices, however there are laptops, desktop computers and servers that use AArch64 [1], [2].

[1] An execution state defines the registers and instruction set that is used by the CPU. A CPU can thus execute programs written for multiple targets, e.g., 32-bit and 64-bit platforms.

```
1 ; See 1.
2 ldr w3, [x1]
3 ldr w4, [x1, 4]
4
5 ; See 2.
6 mul w5, w3, w4
7
8 ; See 3.
9 str w5, [x2]
```

Code Listing 2.1: Example ARMv8-A Assembly

2.1.1 Instructions

Because ARMv8-A is a RISC architecture, the available instructions have a fixed 32-bit length and only describe basic operations [11]. The operations include loading data from memory into registers, data manipulation using registers and storing data from the registers to memory. As a result of the reduced instruction count of the Instruction Set Architecture (ISA) and that each instruction describes one basic operation, the CPUs that execute the instructions can be highly optimized [12]. Code Listing 2.1 shows an example of ARMv8-A assembly code that 1. loads two values from memory into registers, 2. multiplies the loaded values and saves the result in a new register and 3. saves the result back to memory.

In the given example the mnemonics *ldr* for load, *mul* for multiply and *str* for store are used. After each mnemonic the appropriate operands are given. ARMv8-A lists zero or more output registers first, followed by zero or more input registers, depending on the instruction [11]. In line six, the registers *w3* and *w4* are multiplied and stored in *w5*.

The example also shows the usage of addresses and constant offset calculation. In general, all arguments given in square brackets are addresses. An address can consist of just a register (see line two) or a register with an immediate offset (see line three). Addresses can be constructed in many ways to cover common use cases [11].

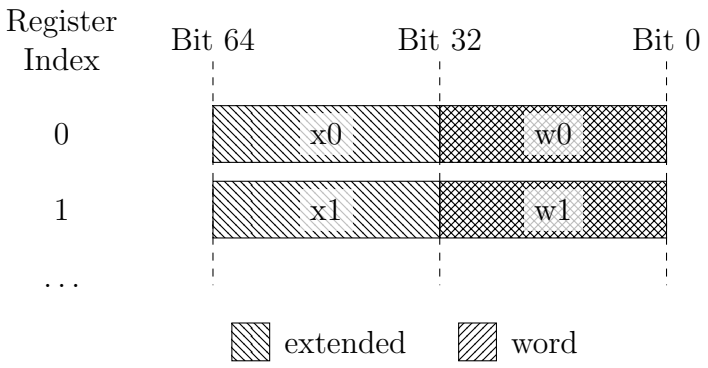


Figure 2.1: Register Layout
Source: Adapted from [13]

Finally, the example shows that ARMv8-A uses different register types. For example, line two uses the registers *w3* and *x1*. More information on the registers can be found in the following section.

2.1.2 Register Layout

To correctly process data of different lengths the general purpose and floating-point registers can be addressed with specific sizes. In ARMv8-A assembly registers are referenced using the prefixes w and x [13]. They are abbreviations for $w$ord and e$x$tended respectively.

An extended register has a capacity of 64 bits whereas a word register only uses 32 bits. However, the lower 32 bits of *x0* are *w0*, i.e., they overlap as shown in Figure 2.1. ARMv8-A defines many registers—a selection of commonly used registers is listed in Table 2.1. In particular the registers that are used for exception handling and managing the processor state are omitted [14].

As previously noted, ARMv8-A also supports Single Instruction, Multiple Data (SIMD) instructions. SIMD instructions need special registers to hold multiple elements of the same size to have a big performance impact. In AArch64, the registers are physically the same as the floating-point registers. However, *d0* to *d31* only address the lower 64 bits of 128-bit registers [13].

Table 2.1: ARMv8-A Registers
Source: Adapted from [13], [15]

Mnemonic	Size (bits)	Description	Note
w0 to *w30*	32	General purpose registers	The lower 32 bits of *x0* to *x30* respectively
x0 to *x30*	64	General purpose registers	
wzr	32	Zero register	The lower 32 bits of *XZR*
xzr	64	Zero register	
wsp	32	Stack pointer	The lower 32 bits of *SP*
sp	64	Stack pointer	
pc	64	Program counter	
h0 to *h31*	16	Floating point	The lower 16 bits of *s0* to *s31* as well as *d0* to *d31* respectively. *h* is short for *h*alf.
s0 to *s31*	32	Floating point	The lower 32 bits of *d0* to *d31* respectively. *s* is short for *s*ingle.
d0 to *d31*	64	Floating point	*d* is short for *d*ouble.

2.2 Code Analysis

To analyze the application code in later chapters different techniques are used. The underlying foundation is presented in this section.

2.2.1 Control-Flow Graph

To enable optimizations, compilers construct a Control-Flow Graph (CFG) denoting the different traversal paths possibly taken when the program is executed. The CFG is generated from source code where the nodes of the graph correspond to basic blocks and the edges indicate the possible control-flow paths taken during execution. A basic block is a linear sequence of one or more instructions $i_0, \ldots, i_n$, that are all executed in logical order[2] if the first instruction i_0 is executed. Thus a basic block

[2]Actual execution order might differ for example due to instruction reordering. However, the new order has to yield the same result as if the instructions where executed in the given sequential order.

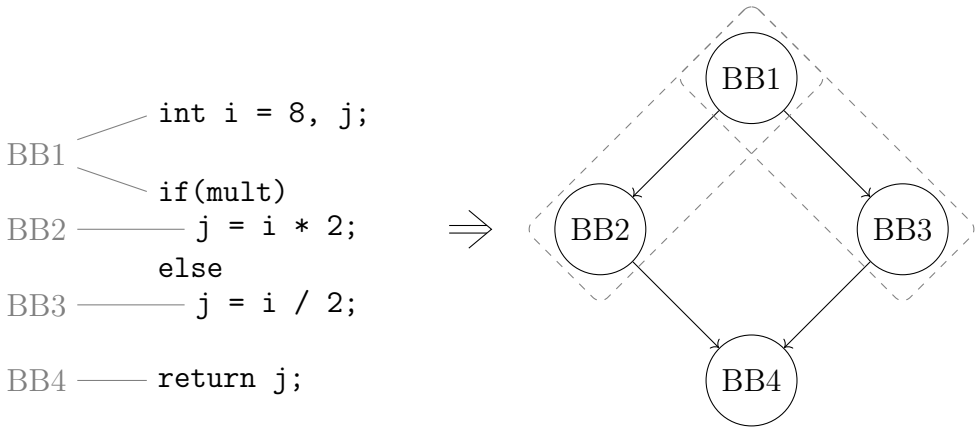


Figure 2.2: Control-Flow Graph Example: The dashed boxes in the CFG indicate the extended basic blocks in the graph.

has exactly one entry point and one exit point. Edges that connect the basic blocks correspond to conditional branch instructions [16].

Because the instruction order is well defined within a basic block, many optimizations use the basic blocks of the CFG as a basis. To improve optimization capabilities, extended basic blocks are often considered where executed basic blocks are two or more basic blocks B_0 to B_n and each B_i has exactly one input edge from B_{i-1}—except for B_0 which can have more than one input edge from any other block in the graph. When a the block B_i is analyzed for optimization opportunities, the CFG demands that B_j was executed directly before, enabling more opportunities [17]. Figure 2.2 shows an example of a CFG constructed from source code and highlights the extended basic blocks.

In the given example, constant replacement can be performed within the basic blocks *BB2* and *BB3* for variable i, because there is only one possible value assigned to i in the corresponding extended basic blocks. However, the variable j cannot be replaced in block *BB4* because it has two predecessors that both set j to a different value.

2.2.2 Data Dependency

Section 2.1.1 describes ARMv8-A assembly, where the instructions can take input from registers and save output to registers. Based on this information the dependency of two instructions can be defined.

Given are the two instructions I_1 and I_2 that will be analyzed, where I_1 is executed before I_2. Furthermore, the function $i(x)$ and $o(x)$ are defined, where $i(x)$ takes an instruction as input and yields a set of locations[3] the instruction x uses as input. Similarly, the function $o(x)$ also takes an instruction as input but yields a set of locations that x uses as output.

Then three different dependency types are defined as

Flow dependency The output of instruction I_1 is part of the input needed by I_2: $o(I_1) \cap i(I_2) \neq \emptyset$ [18]

Output dependency The output of instruction I_1 is overwritten by the output of I_2: $o(I_1) \cap o(I_2) \neq \emptyset$ [18]

Anti-dependency The input of instruction I_1 will be changed by the output of I_2: $i(I_1) \cap o(I_2) \neq \emptyset$ [18]

Together they form the Bernstein condition [18]. The two instructions I_1 and I_2 depend on each other if

$$[o(I_1) \cap i(I_2)] \cup [o(I_1) \cap o(I_2)] \cup [i(I_1) \cap o(I_2)] \neq \emptyset \tag{2.1}$$

2.3 Pipelines

Modern CPUs including those implementing ARMv8-A use pipelines to increase their instruction throughput [12]. Instead of executing each instruction one by one and

[3] A location can be a register but could also be a location in memory.

strictly after another, their execution is split up into independent parts, called stages, which are common to all instructions. Each instruction now propagates through all the independent stages at a much higher frequency [12].

Pipelines in RISCs have multiple stages. For example, the ARM Cortex-A53 has eight to eleven stages whereas the ARM Cortex-A57 has fifteen or more. The aforementioned CPU core architectures implement AArch64. In general, the stages are derived from five base stages and by refining them into multiple specialized stages, the instruction throughput can be increased even more [12].

The following presented concepts use pipelines with the aforementioned five base stages, resulting in an easier representation while still being valid in current RISC implementations.

2.3.1 Stages

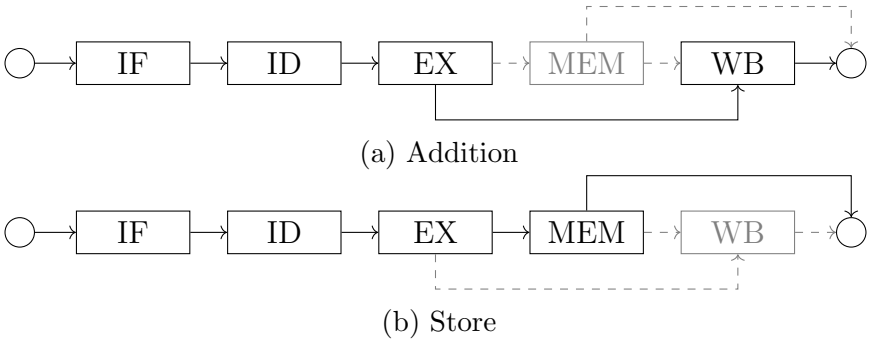
The five stages mentioned above are described in this section in order of execution, based on the model described by Patterson and Hennessy [12].

1. The first stage is called Instruction Fetch (IF). In this stage the instruction pointed to by the program counter is loaded from memory.

2. As soon as the instruction is loaded, the instruction needs to be analyzed to extract its operands and plan its further propagation through the subsequent stages. This is done in the Instruction Decode (ID) stage.

3. The next stage is executing arithmetic operations using the Arithmetic Logic Unit (ALU) and is called Execute (EX). For example, the addition of two register values would be performed.

4. Depending on the instruction, data is saved to or loaded from memory in the Memory (MEM) stage. This could be the main memory, storage or other input/output devices, *excluding* registers.



(a) Addition

(b) Store

Figure 2.3: Example Execution Paths: The edges indicate possible instruction propagation through the stages. In the two graphs the solid edges are taken by the particular instruction.
Source: Adapted from [19]

5. In the last stage, which is called Write Back (WB) stage, results of previous instructions are stored in registers for use in subsequent instructions.

In addition to the previously mentioned benefits, instructions that do not need the resources from a specific stage, can skip them as shown in Figure 2.3 [12]. This can greatly reduce the total number of cycles an instruction needs.

2.3.2 Load Stalls

Because of techniques like pipelining and multi-core processing, instruction throughput increased in recent years. However, due to hardware limitations the memory speed, more precisely the latency at which data can be transferred from the main memory to the CPU, has not kept up [9]. This divergence is called processor-memory performance gap and is shown in Figure 2.4.

This speed difference results in major slowdowns if the processor must wait for data from the memory. In the worst case the CPU stalls while waiting for the data—also referred to as load stall [9].

During a load stall no other instructions can be executed and *nop*[4]instructions are inserted into the pipeline [12]. Those *nop* instructions are called bubbles to

[4] *nop* is the assembly mnemonic for an instruction that does nothing [20].

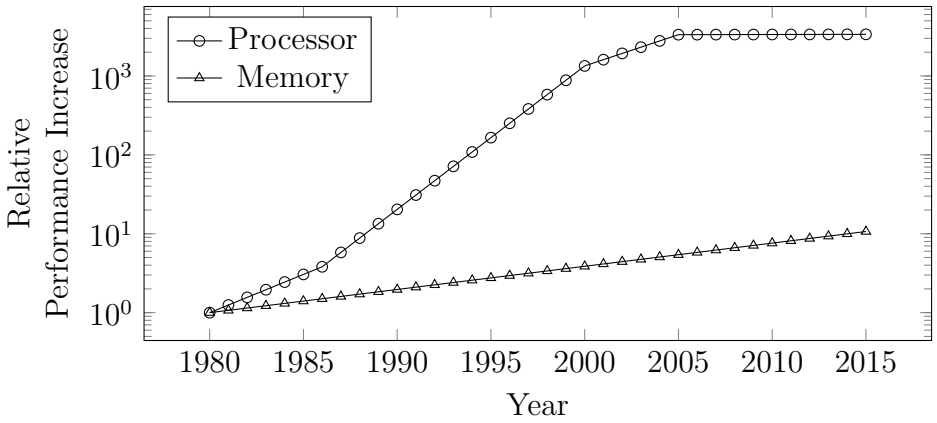


Figure 2.4: Processor-Memory Performance Gap: The relative performance increase is given on a logarithmic scale.
Source: Adapted from [9]

differentiate between *nop* instructions that are part of the program and ones that are inserted due to stalls [12].

In the given pipeline model two stages access the memory, which could result in stalls. Both stalls are discussed in further detail below.

The first stage that accesses the memory is the IF stage, to read the next instruction. In the following, a load stall in the IF stage is described using the example given in Figure 2.5. In the first two cycles the execution is normal and the pipeline fills with the instructions A and B. In cycle three the next instruction cannot be loaded in time—for example because B is a branch instruction and the target is predicted incorrectly—resulting in a stall. As a result, in cycle four a bubble is inserted in the ID stage, while the instruction is still not loaded from memory. This leads to the insertion of another bubble in cycle five. In the same cycle instruction C is finally loaded and—as seen in cycle six—the stall is resolved. Normal execution continues but the pipeline contains two bubbles occupying two pipeline stages until they progressed through the remaining stages. For the purpose of this work stalls occurring in the IF stage are referred to as frontend stalls[5].

[5]A frontend stall occurring in an x86 CPU can have more causes as the frontend is more complex [21].

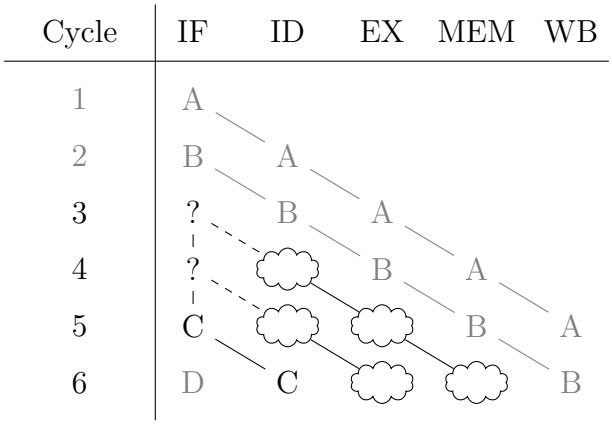


Figure 2.5: Load Stall in the Instruction Fetch Stage: A solid edge describes the path taken by an instruction through the stages and a dashed line indicates that a bubble is created by the connected stalling instruction. The execution of instructions that are colored in gray is not directly influenced by the stall.
Source: Adapted from [12]

Similarly to the previous example, a memory access in the MEM stage can also lead to stalls as shown in Figure 2.6. In the example, the instructions progress through the pipeline as expected until cycle five. In cycle five instruction B accessing the memory does not finish in time. Like in the previous example a bubble is inserted in the WB stage in cycle six. All previous instructions stay in the same stage. Because the memory request of instruction B is not completed at the end of cycle six a new bubble is inserted and no further progress is made in cycle seven. However, at the end of cycle seven the stall is resolved and normal execution resumes. Stalls occurring in the MEM stage are referred to as backend stalls and can be caused by load as well as store instructions.

As seen in the examples each stalling cycle defers the execution of all subsequent instructions by one cycle. In addition, the inserted bubbles are not doing work towards program completion, decreasing the execution efficiency and increasing the execution time.

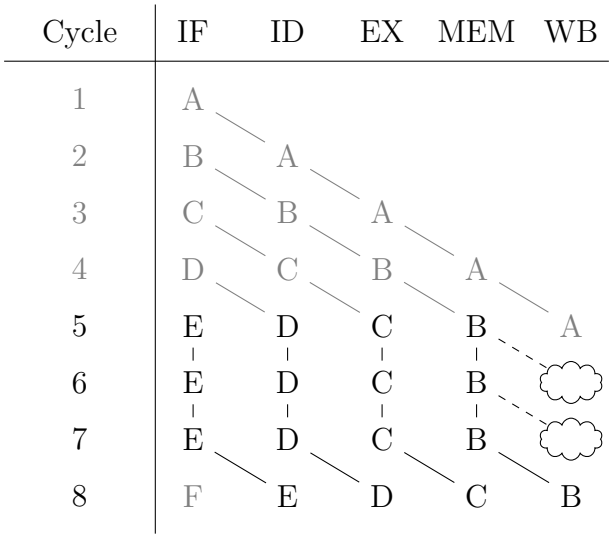


Figure 2.6: Load Stall in the Memory Stage: A solid edge describes the path taken by an instruction through the stages and a dashed line indicates that a bubble is created by the connected stalling instruction. The execution of instructions that are colored in gray is not directly influenced by the stall.
Source: Adapted from [12]

2.3.3 Execution Order

All previously shown examples execute each instruction one by one in the order specified by the program that is currently executing. This is called in-order execution and is, for example, implemented in the aforementioned ARM Cortex-A53 [22].

To reduce the impact of load stalls, some processors, including the ARM Cortex-A57, allow for out-of-order execution of the program instructions [23]. This allows the processor to move instructions forward in the pipeline that are independent from the stalling instruction [12]. For example, in Figure 2.6 instruction C could progress further into the MEM stage while B is stalling, if they are independent. Because this can only be applied to subsequent instructions, and during a frontend stall the next instruction is unknown, this technique can only reduce the impact of backend stalls. To implement out-of-order execution, instructions are buffered in a queue and only instructions with their input data ready can proceed to the subsequent stages. The results are also stored in a queue or register map to recover from branch

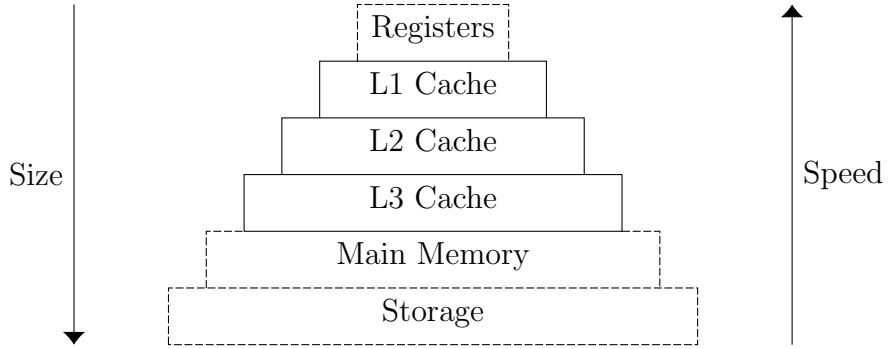


Figure 2.7: Memory Hierarchy: A solid outline indicates that the corresponding memory is a cache.
Source: Adapted from [9]

mispredictions and exceptions and make changes to the registers available in order of program execution [12]. If there is no more space available in the queue, out-of-order pipelines must stall until the next instruction can progress [12]. Because of the usage of queues, out-of-order pipelines can be characterized by the maximum queue length. The queues are also referred to as instruction reorder buffers.

2.4 Caches

Aside from out-of-order execution a commonly used technique to reduce the processor-memory gap is the usage of caches [9], [12]. A cache is a very fast unit of memory that is also physically closer to the CPU and tries to hold data that is likely to be used next by the CPU [12]. Figure 2.7 shows how caches are incorporated in the memory hierarchy of a computer.

When data from the main memory is accessed, first the L1 cache is checked if the data is available. In case the data is available—also referred to as a *cache hit*—the value is made available to the CPU at a high speed. If the data is not available—referred to as a *cache miss*—the next cache level is checked in the same manner. The advantage is that data that is available in a cache can be accessed much faster compared to accessing data that resides in the main memory, thus closing the processor-memory

```
1 void square(int *val) {
2     (*val) = (*val) * (*val);
3 }
```

Code Listing 2.2: Example of Temporal Locality

```
1 ldr w1, [x0]      ; (*val)
2 ldr w2, [x0]      ; (*val)
3
4 mul w3, w1, w2    ; (*val) * (*val)
5
6 str w3, [x0]      ; (*val) = (*val) * (*val)
```

Code Listing 2.3: Example of Temporal Locality (Assembly): The translation could be optimized to only consist of one load operation. However, if compiled with *-O0* or if *val* is volatile the described output is achieved. The code assumes that the pointer *val* is saved in *x0*.

performance gap.

As mentioned previously, caches use data access characteristics of applications to hold data that is likely to be accessed next. Their design is mostly based on the two concepts presented in the next sections.

2.4.1 Temporal Locality

Temporal locality assumes that data at address a that is accessed in cycle i is likely to be accessed again in the near future [12]. An example of temporal locality is the C function shown in Code Listing 2.2.

The function *square* takes a pointer *val* as its argument calculates the square of the integer pointed to by *val* and saves the result back to the location pointed to by *val*. Compiling this code to ARMv-A8 assembly yields Code Listing 2.3.

The compiled code shows that the location *val* is accessed three times. In line one and two the value is loaded into registers and in line six the result is written back.

```
1 size_t null_arr_len(void **arr) {
2     size_t ret = 0;
3     while (*arr != NULL) {
4         ++arr;
5         ++ret;
6     }
7     return ret;
8 }
```

Code Listing 2.4: Example of Spatial Locality

```
1     mov x1, xzr          ; size_t ret = 0;
2
3 loop:
4     ldr x2, [x0]         ; *arr
5     cbz x2, end_loop     ; (*arr != NULL)
6
7     add x0, x0, 8        ; ++arr
8     add x1, x1, 1        ; ++ret
9
10    b loop               ; Next loop iteration
11 end_loop:
12    ret                  ; return ret
```

Code Listing 2.5: Example of Spatial Locality (Assembly): The code assumes that the pointer *arr* is saved in *x0* and the result is saved in *x1*. Furthermore, the while loop is not implemented using a backward branch to make the comparison between the C code and the assembly straight forward.

2.4.2 Spatial Locality

Another characteristic regarding data access patterns is described by spatial locality. It assumes that the probability that the value at address $x + n$ will be accessed is high if address x was accessed in the near past[6] [12]. Code Listing 2.4 is an example for code that complies with spatial locality and like before, Code Listing 2.5 shows the matching ARMv8-A assembly.

The code contains one load instruction in line four. This instruction is part of the loop body and iterates over the subsequent elements of the array *arr* until a *NULL*

[6] n is small, i.e., in the order of bytes.

value is reached. The increment of the address is shown in line seven. The address is incremented by eight bytes in each iteration because one address occupies eight bytes.

2.4.3 Implementation

Because of temporal locality, caches save data that was accessed and if the same address is accessed again the cache can provide the previously saved data. To make use of spatial locality, caches save bordering data in cache lines, which consist of multiple bytes. Thus, if nearby addresses are queried, the data may already be stored in the cache [12].

Cache lines can be saved in a fixed-sized hash map where the individual data is organized and accessed with the address. First the least significant bits of the address define the offset within the cache line, i.e., at what position the accessed data is located. Thus the size of the offset depends on the size of the cache line. The subsequent bits are used as an index to reference a specific cache line where the number of used bits corresponds to the cache size. Because the extracted index is not unique each cache line also saves a tag that is used for collision detection. It consists of the remaining unused bits [12]. An example for such a hash map implementation is shown in Figure 2.8

As previously mentioned, collisions can occur in caches and because they have a fixed size this means that previous data is evicted from the cache to make space for the new cache line. As a result, if the program references two addresses that have the same key the load time does not decrease because each access evicts the previous cache line [12]. To mitigate this problem associative caches are used. An associative cache saves a bucket for each index, where a bucket consists of a fixed number of cache lines [12]. Now the cache can hold multiple cache lines with the same key. If each bucket can hold at most n different cache lines, the cache is called n-way set

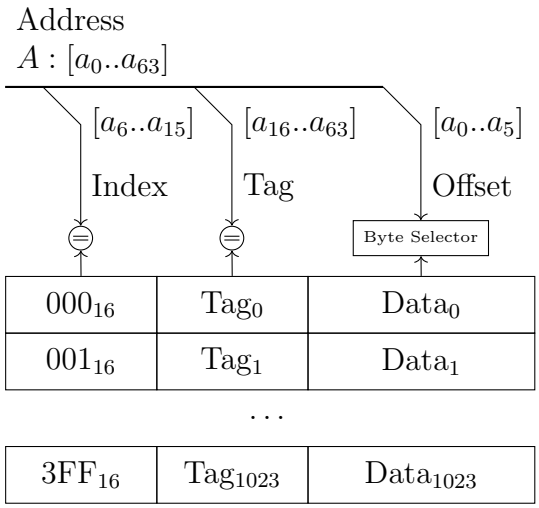


Figure 2.8: Cache Lookup: This cache uses comparators to check if index and tag are equal. If that is the case the appropriate byte is selected from the saved data. The shown cache stores 1024 cache lines where each line consists of 64 bytes, yielding a total capacity of 64 KiB. The depicted cache type is also known as a direct mapped cache.
Source: Adapted from [12]

associative. An example is shown in Figure 2.9.

As previously shown in Figure 2.7 computers typically have more than one cache and multiple caches of the same class can be present within the same system. For example, many CPUs have one L1 cache for each core [12]. Because L1 caches are commonly organized per core, they must communicate with one another because two cores could work on the same data or the same cache line. This means that a write on one core must invalidate the cache line on another core.

In addition, the L1 cache is commonly split into a data and instruction cache [9]. The L1 data cache contains the data that might be processed by the CPU whereas the L1 instruction cache only contains instructions of the current program. Differentiating between data and instruction caches reduces the impact of frontend stalls that would otherwise occur when instructions are evicted from the L1 cache by the data that is processed.

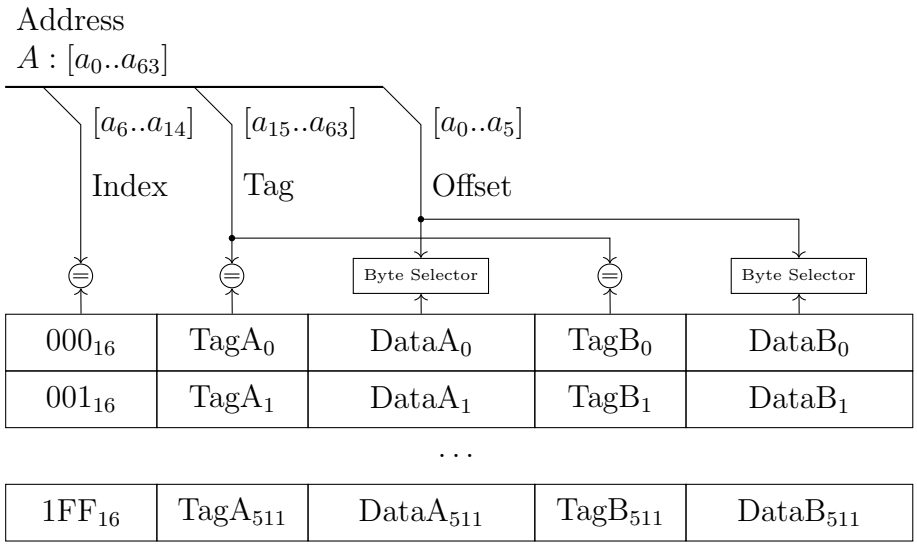


Figure 2.9: Cache Associativity: Like in Figure 2.8 the index is matched using a comparator. However, each index saves two cache lines and two comparators are needed to select the appropriate data based on the tag. Lastly the appropriate byte is selected. The cache stores 512 sets with two cache lines each, where each cache line consists of 64 bytes. Thus, the cache has a total capacity of 64 KiB. Because each set stores two cache lines the cache falls into the category of 2-way set associative caches. Source: Adapted from [12]

2.5 Java Virtual Machine

With the emergence of the internet the need to execute programs on multiple heterogenous platforms, without recompiling for the different targets, arose—also known as the *Write Once, Run Anywhere* paradigm [24]. During that time the Java programming language together with the Java Virtual Machine (JVM) was introduced, claiming to solve the aforementioned problem [25]. To achieve this goal, Java code is first compiled to a platform independent instruction set called JBC. JBC is the instruction set of the JVM, that executes the program [5]. The general interaction between applications compiled for the JVM and native applications directly compiled for the target hardware is shown in Figure 2.10.

As shown, the JVM provides an abstract interface to work with the Operating System (OS) on the hardware. Instead of developing the application for different platforms, the application can be executed if a JVM is installed on the machine [5].

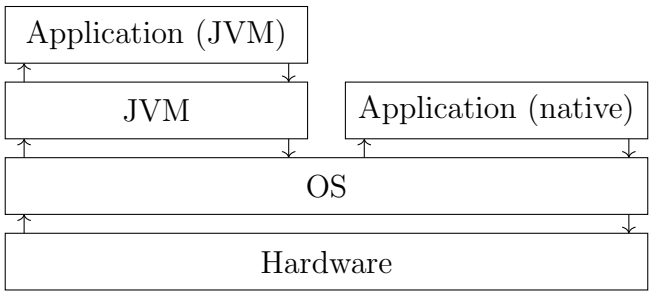


Figure 2.10: Layers of Computing Systems
Source: Adapted from [26]

Further interaction with the OS and the hardware is provided through the Java Class Library and the Java Native Interface.

To execute a given JBC program, the JVM interprets the instructions and simulates their effect on the Virtual Machine (VM) state, allowing the same results to be obtained among different platforms. The interpreter could be implemented using a *while*-loop and a *switch*-block where each branch maps to one JBC instruction and manipulates the VM state according to the actions specified in the JVM standard [27].

2.5.1 Garbage Collection

The previously described VM state includes a heap that is used for object allocation. Due to safety concerns, direct memory manipulation or access is not possible with the instructions defined by the JVM standard [5]. Thus, a clear distinction between references and other data is made and they cannot be mixed.

Additionally, the heap is managed by the JVM—objects that are not reachable from user code are freed automatically to reclaim memory [5]. Objects that are reachable and unreachable are commonly referred to as live and dead objects respectively. The process of freeing dead objects is called Garbage Collection (GC) [5]. Different algorithms are used to collect dead objects [28]. They use different strategies to reduce their impact on the runtime by executing steps in parallel or by splitting up the heap into regions. Sometimes the algorithms require exclusive access to the heap

and all program threads need to be halted—this is called a Stop-The-World (STW) pause [29]. Depending on the algorithm the live objects might be reorganized to defragment the available space.

However, most GC algorithms first must identify all objects that are no longer reachable. This is typically done in a mark step [29]. First a set of object references that are currently directly reachable by the user is created—this set is called the root set. Objects that are directly reachable are, for example, saved on the program stack, either as local variables or as operands in each stack frame. Some object references are not part of the stack, for example static variables, and are separately added to the root set [29].

With the root set completely populated, each referenced object is marked as alive, as it is reachable by the user. Subsequently all fields that are references to objects are also transitively reachable and are marked too. This is performed recursively until the root set is empty [29].

All reachable objects are now marked as alive. Depending on the GC algorithm different steps are performed to reclaim the memory.

2.5.2 Just-In-Time Compilation

As mentioned previously the JVM interprets programs that should be executed. However, this is not very efficient since the interpreter can only behave within the constraints of VM states, i.e., it has to be generic to correctly execute the individual JBCs with little optimizations done. This difference becomes even more notable if the execution time is compared to native applications. To mitigate this problem JVMs use different techniques, including JIT compilation [27], [30].

The basic idea is to compile the JBC to native machine code to eliminate the slow translation process. However, the compilation takes time on its own and the JVM must decide whether the compilation of code—yielding a possible speedup in runtime

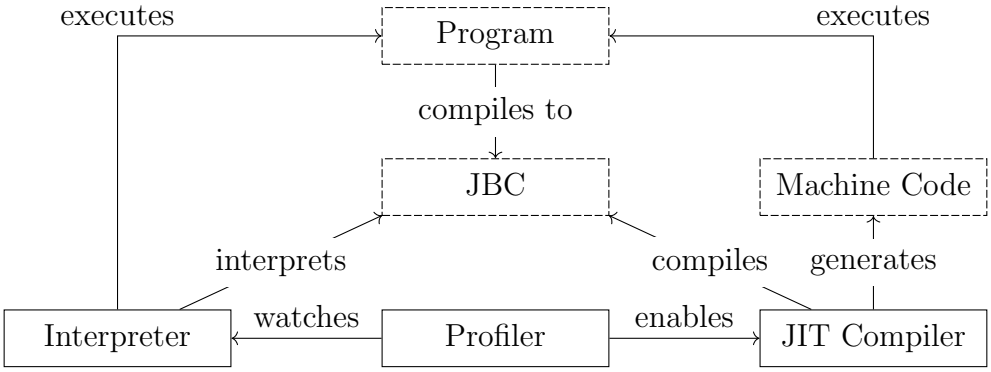


Figure 2.11: Program Execution with Just-In-Time Compilation: A solid outline indicates that the given component is part of the JVM. An edge between two components is a relationship where the type of the relation is specified by the edge label.

at the cost of initially sacrificing runtime—is more beneficial than interpreting. That is why some JVMs include profilers that collect data about the executed code. This information is used to make informed decisions on whether code should be compiled and how much time should be spent on compiling, which is directly influenced by the type and the number of optimizations applied. In addition, the profiling data is also used to guide optimizations yielding code tailored for the specific application run [6]. This interaction between different JVM components is showcased in Figure 2.11.

2.5.3 Eclipse OpenJ9

Eclipse OpenJ9 is a JVM that emerged from the J9 JVM developed by IBM [31]. It is commonly used to run business applications and IBM continues to use OpenJ9 for many of their products. Under the auspices of the Eclipse foundation the project is now open source. In this section parts of OpenJ9 are discussed in more detail.

2.5.3.1 Garbage Collection Policies

OpenJ9 defines multiple different GC algorithms—also called policies. The GC algorithm that should be used for a specific program can be specified using the

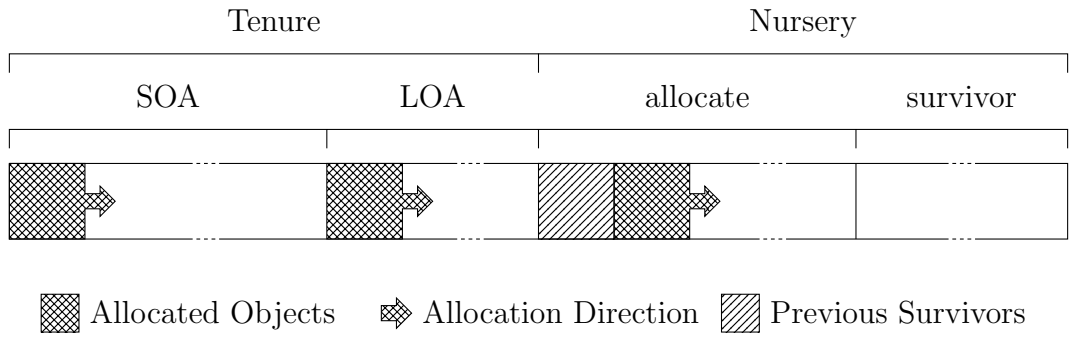


Figure 2.12: Heap Layout (*gencon*)
Source: Adapted from [32]

command line [32]. This section discusses the six available algorithms in OpenJ9 in detail.

gencon The GC policy *gencon* divides the heap into two regions, the nursery, where newly created objects reside, and a tenure area where objects of a specific age are managed. Only objects in the nursery have an age assigned, which corresponds to the number of GC cycles the object survived. The nursery is further subdivided into an allocate space—newly created objects are allocated from here—and a survivor space. The survivor space is used during the scavenge operation and holds objects that survived the current GC cycle but are not old enough to get promoted to the tenure area yet. The tenure area consists of two regions, one for regular sized objects—called Small Object Area (SOA)—and one for large objects—called Large Object Area (LOA). Differentiating between small and large objects aims to prevent fragmentation in the tenure area [32]. The overall layout is shown in Figure 2.12. The general allocation and GC algorithm is described in the following steps.

1. New objects are allocated in the allocate area of the heap with age zero.

2. As soon as the allocate area runs out of memory a local GC is triggered—referred to as scavenge. The live objects in the allocate region are copied to the survivor area if their age is below a given threshold or to the tenure area

if it is greater or equal [33]. At the end the survivor space and the allocate space are swapped—without copying the objects—and allocation can continue as described in step one.

3. Over time, the SOA and LOA will fill with objects that lived long enough to escape the nursery. If the tenure area runs out of space, a global GC cycle is triggered to reclaim memory from objects that died since they were added to the tenure area. A global GC consists of a mark, sweep and optionally a compact operation. A detailed description of the different aforementioned operations is given in Section 2.5.3.1.

The sizes of the allocate and survivor spaces are dynamically adjusted based on space required by the previous cycle. The age objects need to reach to be promoted to the tenure area is also adjusted based on survival rates of the different age classes. Another optimization is to remove the LOA if it is not populated and use the space to extend the SOA [32].
In general, the GC policy *gencon* aims to be particularly useful for programs that use many short-lived objects, as the memory occupied by them is frequently reclaimed. *gencon* is the default policy used by OpenJ9 [32].

balanced The GC policy *balanced* also splits the heap into multiple regions. First the region size x is determined such that $x = 2^y; y \in \mathbb{N}$ and $x \geq 512$ KiB. Furthermore, the number of regions n is constrained to be $n \in [1000 - 2000]$. Exceptions are systems with less than $1000 * 512$ KiB $= 500$ MiB memory available—then fewer regions are used [34].
Each of the given regions can either be empty or it has an assigned age and all objects that are part of this region have the specified age. Newly created objects have an age of zero and they might get as old as 24 after which their age does not increase.

Regions that contain objects of age zero are also referred to as edens [32]. The GC algorithm is described by the following steps.

1. New objects are allocated in eden spaces. If no eden space is available an empty region is added to the eden spaces.

2. If the number of eden spaces reaches a threshold, a partial GC cycle is triggered. The partial GC includes all eden spaces and a list of regions of older age that contain many dead objects—identified by a global mark phase [33]. For each selected region the live objects of age a are copied to a region of the same age that still has space or to an empty region that is then assigned to age a. If all objects were copied to different regions the current region is marked as empty. Finally, the age of all regions that currently contain objects is increased by one—except if their age is already 24.

3. If the partial GC cannot keep up with the allocation rate and not enough memory can be reclaimed a parallel global GC is performed.

The *balanced* GC policy claims to reduce the maximum pause time by managing the heap in smaller units and tries to avoid global GC cycles. It also argues to be beneficial in systems implementing Non-Uniform Memory Access (NUMA) as the different regions are distributed across the different nodes and objects can be allocated in regions that are closest to the issuing processor [33], [34].

optavgpause In contrast to the previously mentioned policies *optavgpause* uses a flat heap, i.e., it is not divided into regions. The GC algorithm consists of a mark, sweep and optionally a compaction operation. First all live objects are marked as described in Section 2.5.1. Then the sweep operation reclaims memory for allocation by saving pointers to regions of unused memory and their size. To defragment the memory, the objects can be compacted by copying them such that

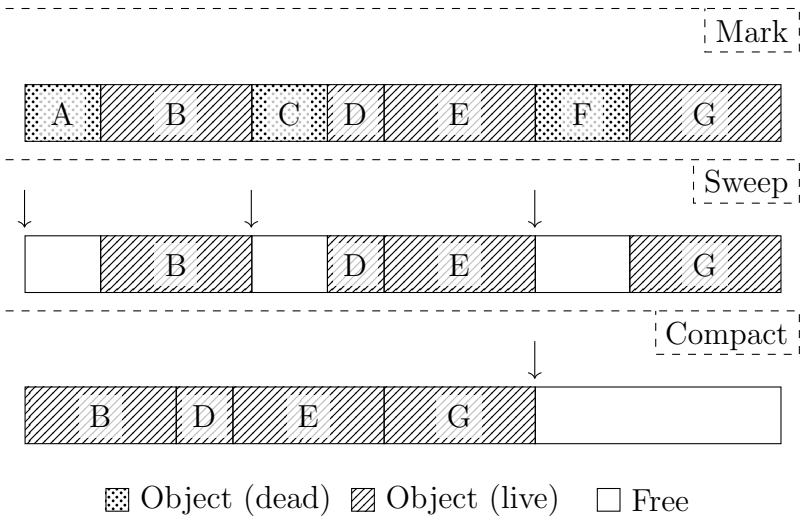


Figure 2.13: Mark-Sweep-Compact-Algorithm: New objects can be allocated starting at the marked pointers.
Source: Adapted from [34]

they are contiguously placed in memory [32]–[34]. The overall procedure is shown in Figure 2.13.

The GC cycle starts as soon as a predefined percentage of the heap is filled. The percentage is based on the object allocation rate and is chosen such that it finishes just before the heap would become exhausted. This is achieved by running the mark operation concurrently [32].

Because *optavgpause* does not split the memory into regions, large objects can be allocated contiguously. In general, *optavgpause* claims to be beneficial in systems with a large heap and for applications that are either short lived or long lived but consist of many short-lived concurrent sessions [32].

optthruput *optthruput* uses the same GC algorithm as *optavgpause*, however the GC cycle is only triggered if an allocation fails due to insufficient memory. Thus, there is no impact due to GC operations as long as there is still space on the heap. However, as soon as the heap is exhausted a long STW pause occurs [32], [33].

metronome The GC policy *metronome* splits the heap into small regions of equal size. The larger the heap is, the more regions will be created. If an object is allocated, it will be placed in a region that manages objects of roughly the same size—16 different size ranges are differentiated. As soon as 50% of the available memory is used, a GC cycle is triggered. The GC is executed in parallel on the different regions and objects that are allocated while the GC is running are always considered to be alive for the duration of the cycle. For the collection, the GC algorithm consists of a mark, sweep and optionally a compact operation [32]–[34].

The usage of the *metronome* policy claims to be soft real-time garbage collection, where the time spent in the GC cycles is bound. This comes at the cost of a high percentage of floating garbage—the heap contains many dead objects, but the memory was not reclaimed yet. In addition, the policy is only available for Linux on x86–64 and AIX [35] platforms [32]. In the future it might be implemented for other platforms if they have access to a high resolution clock [36].

NoGC As the name implies this policy disables the GC and is not trying to reclaim memory. If no more memory is available, the JVM terminates. The *NoGC* policy can be useful when the program will never exceed the memory limits of the system, because the memory will be freed on termination and during program execution no time is spent on GC cycles [32].

2.5.3.2 Scan Order

In addition to the different GC policies, copying GCs can also be adjusted by changing the traversal order and as a result the order of objects in memory. This section describes the three scan orders available in OpenJ9, where all of them expect the root set as input and copy live objects from the *from*-space to the *to*-space [37].

Breadth First All objects from the root set are copied to the *to*-space. Now the

objects in the *to* space are traversed in the order they were added and objects that are referenced by the current object and that are not copied yet are copied to the end of the *to*-space. The traversal of the *to* space is equivalent to a queue in terms of processing order. That is reason why the scan order traverses the object tree in a breadth first order [37]. This algorithm is also referred to as Cheney's algorithm [38].

Dynamic Breadth First The basis of this algorithm is the breath first scan order; however, a partial depth first search algorithm is used to first copy objects based on the field hotness—a measure of how frequently the field is accessed. For the depth first search a stack is required and by limiting its depth it is ensured that the size of the stack is limited. This is the default scan order for the *balanced* GC policy [37].

Hierarchical The hierarchical scan order uses the same idea as the dynamic breath first search algorithm; however, the copying order is not guided by the field hotness but aims to minimize the distance between objects that have a relation based on the class definition. This scan order is the default for the *gencon* GC policy [37].

Because using different scan orders directly influences the order in which the objects are copied to the next region, the scan order affects the order of the objects in memory.

The presented scan orders can be used with the *gencon* or *balanced* GC policies as they incorporate a copy operation. However, the hierarchical scan order is not supported by the *balanced* policy [37].

2.5.3.3 Testarossa

Testarossa is the optimizing JIT compiler used by OpenJ9 to compile JBC to machine code with different preset optimization levels [27]. The optimization levels are chosen based on the method invocation frequency and are listed below [27].

1. cold
2. warm
3. hot
4. very hot
5. scorching

As the level increases, more optimizations are enabled but the compile time also increases.

In general, Testarossa first translates the JBC into the Testarossa-Intermediate Language (IL)—a linked list of trees [27]. This translation is required to enable more optimizations. After the translation is completed, optimization passes are performed based on the selected optimization level. The passes alter the linked list as well as the stored IL trees and finally they are linearized into machine instructions by code generators [27]. The machine code can then be called from the interpreter instead of interpreting the method.

Testarossa can be instructed to produce log files that show how the executed machine code was generated including the used optimizations, the JBC emitting the instruction and profiling data [39].

2.6 Performance Counters

Many CPUs contain Performance Monitor Units (PMUs) that allow the collection of hardware events while a program is running. Examples for events are the elapsed cycles or the number of executed instructions. The available selection of events depends on the implemented PMU. To collect the events, the PMU contains counters that increase every time an event occurs; they are referred to as performance counters. Obtaining this data can be helpful when debugging or profiling code. In the following the ARM PMUv3 technology, which is commonly used in processors implementing ARMv8-A, is further described [40].

The PMUv3 contains six configurable counters as well as a fixed cycle counter [40]. The six counters are configured using special registers and predefined event ids [40]. All six counters could be configured to listen to the same event. Incoming events from other units are then forwarded to the appropriate performance counters. Events that are not selected are ignored. To handle events while they occur, the counters are also connected to an interrupt and overflow unit that can be configured to interrupt execution if a given number of events occurred [40].

2.7 Perf

The aforementioned performance counters can be used manually, however there are tools available that allow easier access and additional analysis. Intel VTune [41] can, for example, be used with Intel CPUs. This section focuses on Perf, which is a command line tool that is available for Linux. The tool is part of the kernel and packages a wide variety of functionalities that can be used to measure, read and analyze performance counters [42]. In general, Perf takes a list of events that should be measured, initializes the performance counters accordingly and then uses one of the two available measurement methods. The two methods are described below.

If the **counting** mode is selected the counter values are accumulated and reported to the user. This is fast and has little impact on the runtime of the program that is executed. However, only basic information is available. For example, the number of executed cycles would be known but not where the cycles are spent during runtime. This yields little information that can be used to optimize or investigate a program but makes it possible to compare the results of different runs. In addition, the obtained values are close to the ones that would occur without monitoring the program. The subcommand *perf stat* uses counting [43].

The second available measurement method is called **sampling**. Instead of just counting the results, the measured results are linked to the instruction that is currently being executed. Additional information like the call stack can also be saved to gain even more information. Obtaining this information has a bigger impact on the program that is currently executed but gives more insight into potential bottlenecks. Perf allows collecting the events in a fixed frequency or period, as saving this data each time the counter increases would be too much overhead. Because of the pipeline optimizations mentioned previously—in particular out-of-order execution—the location the sample is linked to might not be accurate. The subcommand *perf record* uses sampling [44].

In addition to the described measurement methods, Perf also handles challenges that would otherwise complicate the measurements. For example, as discussed in Section 2.6 the CPU has a fixed number of general performance counters that can be used. In cases where the number of events exceeds the available counters Perf switches them in a round-robin fashion and interpolates the missing counts based on the time and average count [45]. Suppose the processor has n general purpose performance counters but $n + c$ events should be monitored where $c > 0$ and all $c + n$ events can only be monitored by general purpose performance counters. Then the

resolution for all measured events decreases, because each event is only measured $\frac{t \cdot n}{c}$ of the total runtime t.

Additionally Perf can either measure events for one program or all currently executed processes. If only one program is observed Perf handles context switches as well as process migrations. During both, the current value needs to be saved and then reused as soon as the process is running again—either in the same or a different context.

2.8 Benchmarks

To measure the performance of the JVM, benchmarks are used. Different benchmarks aim to test different components as well as use cases of the JVM. Below is a collection of benchmarks that were used by Li [21] to develop a stall-focused benchmark suite for OpenJ9 on x86 and subsequently by the author of this work during the development of a framework to automatically collect load stall data [19].

AcmeAir makes use of Open Liberty [46] and MongoDB [47] to simulate an airline web application [48]. It is an application benchmark with a focus on web applications.

Daytrader7 also uses Open Liberty to emulate an online stock trading system [49]. Like AcmeAir it is an application benchmark.

HiBench is a benchmark suite with scientific big data workloads [50]. Previous experiments of the author with HiBench on AArch64 showed that HiBench was difficult to setup due to compatibility issues. In addition, retrieving data proved to be difficult, presumably because the workloads are too large for the available hardware [19].

Stall-Focused Benchmarks are a collection of benchmarks that cause frontend and backend load stalls [21]. They are micro benchmarks as they focus on single

tasks that are likely to cause load stalls.

Renaissance is a benchmark suite for JVM implementations and aims at testing various integrated tools like JIT compilers [51]. It uses multiple different frameworks and focuses on parallel workloads. It incorporates scientific as well as application benchmarks [52].

SPECjvm2008 is a performance measurement benchmark suite for the Java Runtime Environment (JRE), where the JRE includes a JVM implementation with additional libraries to successfully run Java programs. The benchmarks focus on JRE core functionalities and as such SPECjvm2008 includes application benchmarks as well as area-focused benchmarks [53]. The benchmarks are simulating server and client applications and make use of multiple cores [54].

2.8.1 Load Stall Benchmarks

As mentioned, this work uses—among others—the stall-focused benchmarks developed by Li [21]. In previous work, a standalone wrapper was developed, which will be used to obtain results [19].

The wrapper includes a command line interface to run the benchmarks. The benchmarks are configured and executed using the interface. The biggest implementation difference to the benchmarks proposed by Li [21] is the removal of Perf from within the benchmark suite that was used to self monitor the benchmark run. The sample collection is now decoupled and independent from the suite. In this configuration, the samples obtained by Perf also include the benchmark setup—i.e., previously Perf only collected samples during the actual benchmark run.

However, the overhead of the command line interface and benchmark setup in comparison to the actual benchmark run is negligible. The benchmark parameters are chosen such that the most execution time is spent in the benchmark functions

proposed by Li [21].

2.9 Load Stall Benchmarking Framework

The load stall benchmarking framework is a Java application developed to automate the collection of data needed to investigate load stalls in OpenJ9 JIT compiled code [19]. The framework was developed by the author of this work and is based on the procedures described by Li [21]. As such it supports the benchmarks listed in Section 2.8 and uses Perf to collect data from performance counters [19]. In this section a small overview of the framework as well as its capabilities is given.

The framework can be considered to be a wrapper around the supported benchmarks and Perf. It sets up the benchmarks as well as the JVM executing them and attaches Perf to the appropriate JVM instance. After a benchmark run, the output from the JVM, in particular the Testarossa log files, are parsed and merged with the data collected by Perf. After the data is collected, the resulting output is then manually inspected [19]. The general relation of the different components is shown in Figure 2.14 and the data collection process is described in further detail.

To generate the output, first a set M_{hot} of methods is determined where each method in M_{hot} is JIT compiled during the benchmark execution and consumes more cycles than the interpreter. To obtain M_{hot}, the benchmark is executed five times and each time Perf is attached to the JVM process. The Perf output is first filtered to only contain data for JIT compiled methods, then all methods that consume fewer cycles than the interpreter are discarded. Now all methods that occur more than three times across the five collected runs are chosen to be part of M_{hot} [19], [21].

The methods that are part of M_{hot} are likely to be compiled in the next run of the benchmark and they will probably consume a substantial amount of the overall runtime. Thus, in the sixth run the JVM is instructed to write log files for the JIT

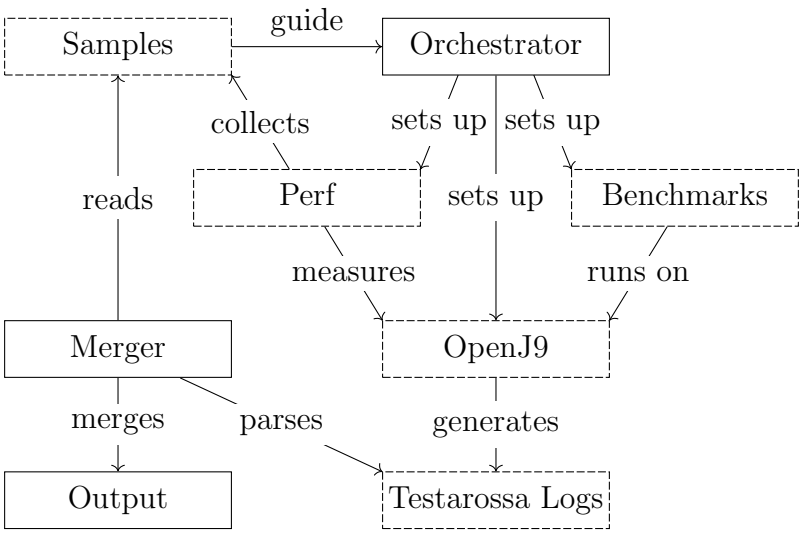


Figure 2.14: Stall Benchmarking Framework Overview: The edges between different components are relations where the type of relation is specified by the edge label. A solid box indicates that the component is part of the Load Stall Benchmarking Framework.

Table 2.2: Instruction Information (Load Stall Framework)

Field	Testarossa	Perf
Method	×	×
Address	×	×
Machine Code Instruction	×	
Source JBC	×	
Basic Block	×	
Elapsed Cycles		×

compilation of the methods that are part of M_{hot} and Perf is again attached to the JVM [19], [21]. By merging the output information about the program flow, the generated instructions, its JBC source—logged by the Testarossa compiler—and the cycle consumption per instruction obtained from Perf, further manual investigation can be used to identify instructions that consumed more cycles due to a frontend or backend stall [19]. The information available for merged instructions as well as the source the data originated from is listed in Table 2.2.

As mentioned in Table 2.2 instructions are associated with a basic block which is declared by the Testarossa compiler. The merged output also contains information

about the CFG including branch taken probabilities that are extracted from the Testarossa log files that are based on profiling data [19].

2.10 Summary

In this chapter the background that this thesis is built upon was discussed. In particular load stalls were introduced for the AArch64 architecture. Furthermore, different cache technologies were shown that are used in many CPUs. Because this thesis focuses on load stalls observed in OpenJ9, the general concepts of the JVM are described. Based on this the GC policies and the JIT compiler used in OpenJ9 are introduced. Finally, different ways to observe load stalls in JVM implementations were described, including Perf, benchmark suites and the existing load stall framework.

Chapter 3

Related Work

This chapter discusses work conducted in areas related to this thesis. First locality aware GC algorithms are presented. Then work evaluating memory managed languages and systems is introduced. Finally, work focusing on load stall analysis in JIT compiled code is presented.

3.1 Locality Aware Garbage Collection

Moon [55] describes different GC techniques to mitigate challenges faced in large systems using a high-performance Lisp implementation. Included is a description of an algorithm developed for virtual memory systems to increase the locality of data structures. They claim that the GC algorithm is also responsible for increasing page locality. To improve the object locality, they present a GC algorithm that copies objects in approximate depth-first order. The *scavenger*, responsible for finding references to objects that should be copied, tries to fill virtual memory pages with objects referenced by objects from the same page. Additionally, they note that GC algorithms themselves increase the probability of page faults as they traverse the objects in memory, which in turn leads to thrashing. A generational GC algorithm that divides objects into three classes *static*, *ephemeral* and *dynamic* to change the

frequency of GCs with a focus on areas where a large number of dead objects is expected [55].

To improve locality Huang et al. [56] uses profile-driven class analysis to identify hot object fields that are then used to guide the GC copying order. First, the object fields that are accessed within a method are identified using static analysis conducted when compiling the method. Then, sampling the runtime helps to identified hot methods and for each sample that is collected for a hot method, the object fields accessed within the method are considered to be hot and their heat metric is updated. This makes it possible to adapt the GC order when different access patterns occur, for example due to a program phase change. The proposed online class analysis introduces a low overhead in the existing infrastructure of Jikes RVM [57] since many used techniques are build on top of the existing architecture. The GC algorithm is evaluated using DaCapo [58], SPECjvm98 [59] and pseudojbb, a variation of SPECjbb2000 [60]. The results indicate, that copying GC algorithms can yield better object locality [56].

The work by Siegwart and Hirzel [61] proposes a new parallel hierarchical GC algorithm that copies objects in hierarchical order in parallel environments by generalizing the hierarchical GC algorithm proposed by Wilson et al. [62]. The different blocks used to implement the hierarchical copy order are classified based on their state and a thread pool polls blocks from a work queue. Based on the block class the thread continues the GC operation on the block. The paper concludes that the introduced GC algorithm increases locality for the measured benchmarks using performance counters and by calculating and counting the distances between an object and the stored object fields [61].

Chilimbi and Larus [63] propose a generational GC algorithm to enable cache aware data placement. Copying is guided by collected profiling data to build an object affinity graph. The authors assume that due to small object size, the object fields are not as important as the overall relation between two objects. The affinity graph

is created for each generation using profiling data collected for object accesses. A directed edge in the graph indicates an object access and the weight how frequent the access was. Then objects of the root set are copied greedily based on the traversal of the affinity graph, favoring edges that indicate frequent accesses. Objects that are in the root set but not represented in the affinity graph are then copied using Cheney's algorithm [38]. The copying step is then repeated for the objects in the $from$ space that are referenced by the objects residing in the *to* space until all live objects are copied. The algorithm is evaluated by measuring the L2 cache miss rate and the results showcase that object reordering can lead to fewer cache misses than Cheney's algorithm [38] or the algorithm proposed by Wilson et al. [62] [63].

Chen et al. [64] use profiling data to increase cache and page locality during garbage collection. They conclude that sampling provides the necessary data to improve the overall performance with little overhead. The new GC algorithm determines hot objects and uses a list of the latest accessed objects to place them on the same page to reduce page faults. In comparison to GC algorithms that solely focus on either cache or page locality their solution performs better. Additionally, the proposed algorithm also triggers layout optimizations—even if GC cycles would not be needed—to continuously match the object layout to the current application demands [64].

3.2 Evaluation of Memory Managed Systems

Lengauer et al. [65] provide an analysis of different GC algorithms using DaCapo [58], DaCapo Scala [66] and SPECjvm2008 [54]. They use the allocation count, allocations per second, as well as object size, object-array ratio and average array size to quantify the allocation behavior of the different benchmarks. Additionally, the GC behavior is reported for the benchmarks including the number of GC cycles, the relative GC

run time as well as pause time. Finally they report object reference characteristics, i.e., the number of references observed for each GC cycle, the average number of references per cycle, the ratio of `null` objects and the ratio of references from a younger generation to an older generation [65].

In their work Blackburn et al. [67] compare different GC algorithms with a focus on total time, GC pause time and cache miss behavior of different benchmarks from the SPECjvm98 [59] and pseudojbb benchmark suites. They conducted their experiments on different architectures including Athlon, Pentium 4, and Power PC with the Jikes RVM [57]. The measured metrics include execution time, GC time, L1 cache, L2 cache and Translation Lookaside Buffer (TLB) misses. Additionally, the metrics are compared for different heap sizes. They conclude that GCs can offer a performance advantage over free list allocations with manual reclamation because locality is improved and becomes more important with higher processor throughput [67]. In contrast to this thesis, the experiments were conducted on a different CPU architecture with a different JVM implementation.

Grgic et al. [68] compare different garbage collection algorithms of the HotSpot JVM [69] on x86–64 using DaCapo [58]. Their work focuses on the overall run time as well as the number of GC cycles observed for different GC algorithms in comparison to the G1 GC, which is the default algorithm in the HotSpot JVM. They conclude that the G1 GC does not perform significantly better than previous GC algorithms that are already available in the HotSpot JVM. However, according to the authors, a bigger impact might be observed in environments utilizing a large number of threads [68].

The impact of garbage collection on cache performance is studied by Reinhold [70]. Their work focuses on Scheme [71] programs and uses a trace driven cache simulator. The main metric considered is runtime as it consists of cycles spent executing the instructions and stalled cycles. The work concludes that simple linear storage

allocation performs well for the functional Scheme programs. This conclusion is extended to programs written in a similar style in other languages, which will likely experience the same behavior. Furthermore, the author claims that sophisticated collectors that improve cache performance are unlikely to be effective in those cases as locality is already very good. Lastly, the work suggests that garbage collected languages can have a performance advantage over languages without garbage collection due to improved locality [70].

Papadakis et al. [72] use DaCapo [58] and Renaissance [52] to collect different metrics of the applications executed with MaxineVM [73]. The work includes the collection of high level metrics like object allocations, object layout as well as object accesses with NUMAProfiler [74] which is a NUMA aware profiler for MaxineVM. Furthermore, low level metrics are collected using Perf to observe the cycles per instruction and cache misses for different levels. Based on the data, the benchmarks are classified into groups. Finally, the observed data is compared between MaxineVM and the HotSpot JVM to test if the used methodology is applicable for other managed runtime environments. The authors conclude, that the multi-facet profiling approach is transferable to other JVMs [72]. In comparison to this thesis, a different CPU architecture and JVM implementation is tested.

3.3 Load Stalls in Just-In-Time Compiled Code

Li [21] proposed a new algorithm to detect and classify load stalls in JIT compiled code on the x86 architecture. Perf is used to sample the program. The stalling behavior of JIT compiled code is observed by focusing on samples collected within methods that are JIT compiled. Furthermore, JIT compiler log files are used to collect detailed data about the methods. The results are used to develop a stall-focused benchmark suite for JVMs. One of the major findings is that backend stalls have a

much larger impact than frontend stalls in JIT compiled code [21].

The author of this work has already conducted preliminary experiments on AArch64 and tried to replicate the classification procedure described by Li [21]. At the same time a benchmarking framework was developed to automate the data collection and stall classification. The observed impact of the different load stall categories is not as significant as detected by Li [21] on x86. The obtained results suggest that a different methodology to select and classify hot instructions in comparison to the algorithms presented by Li [21] is needed for AArch64 because the overall overhead of a single instruction is not as big as measured on x86 [19].

3.4 Summary

The discussed related work consists of papers describing locality aware GC algorithms outlining the importance and opportunities to achieve better locality in garbage collected systems. Furthermore, evaluations of garbage collected systems that focus on locality are presented. Finally, work that focuses on load stall detection in JIT compiled code was introduced.

Chapter 4

Methodology

This chapter presents the methodology used to measure the impact of GC algorithms on load stalls in OpenJ9. First the experimental setup and the benchmark selection is described. Then the data collection for different granularity levels is motivated and explained.

4.1 Experimental Setup

To conduct the experiments, a Raspberry Pi 4B [75] with the corresponding Pi OS Lite (64-Bit) [76] version based on Debian 11 with the 6.1 Linux kernel is used. The installation is headless and the machine is accessed over the network. While running the experiments exclusive access to the hardware is granted to minimize effects of other applications. A list of features of the Raspberry Pi 4B platform is shown in Table 4.1.

The Raspberry Pi Model 4B was chosen to conduct the experiments due to its availability. However, its processing power is not comparable to computers with high performance CPUs with larger cache sizes and bigger pipelines. It is assumed that those hardware differences will have an impact on the measured metrics. Larger caches will likely decrease the occurrence of stalls caused by memory accesses. Additionally,

Table 4.1: Raspberry Pi 4B Hardware Characteristics
Source: [77]–[80]

Property	Value
CPU	ARM Cortex-A72
Cores	4
Cluster	1
Frequency	1.8 GHz
L1I Cache	48 KiB; 3-way set-associative; 64 B Lines; per core
L1I Cache TLB	48-entry; fully associative
L1I Cache Line Eviction	Last Recently Used
L1D Cache	32 KiB; 2-way set-associative; 64 B Lines; per core
L1D Cache TLB	32-entry; fully associative
L1D Cache Line Eviction	Last Recently Used
L2 Cache	1 MiB; 16-way set-associative; 64 B Lines; per cluster
L2 Cache TLB	1024-entry; 4-way set-associative
L2 Cache Line Eviction	Pseudo Last Recently Used or Pseudo Random
Main Memory	8 GB
Main Storage	32 GB (OS); 420 GB (Data and Programs)
Out-Of-Order Pipeline	Yes
Reorder Buffer Size	128 Entries

Table 4.2: Program Versions

Tool	Version
PI OS	Bullseye (6.1.21-v8+)
IBM Semeru Runtime	1.8.0_362
Scala	2.11.12
Perf	6.1.57.gba13c3924a2a
Apache JMeter	5.5
MongoDB	4.4.18
Apache Hadoop	2.10.2
Apache Spark	2.3.4

Benchmark	Version
AcmeAir	2.0.0
Daytrader7	1.0-SNAPSHOT
HiBench	7.1.1
Renaissance	0.14.1 (GPL licensed)
SPECjvm2008	1.01
Load Stall Benchmarks	– (see Section 2.8.1)

larger reorder buffer sizes will reduce the impact of stalls further. Nevertheless, the conclusions drawn by comparing the GC algorithms are expected to be alike on similar systems, since the memory access patterns are comparable[1]. Thus, if comparing the stall behavior of different GC algorithms, it is likely caused due to the same object layout.

To run the experiments the OpenJ9 JVM for Java 8 bundled with the Open JDK class libraries in the IBM Semeru Runtime [81] will be used. The Java version is chosen to ensure that all benchmarks are compatible with the same JVM. In addition, choosing the Open J9 JVM enables the usage of the previously developed framework to collect load stall data while running the benchmarks. The different versions of the installed programs are shown in Table 4.2.

[1]In this context similar refers to the implementation of the different components, e.g., not the specific cache size but the eviction policies used by the different cache levels.

4.2 Data Collection

Data to evaluate the JVM performance is collected using three different granularity levels aiming to answer different question regarding the load stall performance of the JVM.

4.2.1 Coarse Granularity

Collecting coarse granularity data makes it possible to obtain general information that helps guide the GC algorithm choice depending on different workloads on AArch64 machines. In addition, the collection of the data should have minimal impact on execution to ensure that the results are as accurate as possible. This should be achieved by counting events with Perf. The events that are measured should give insights on the general application performance as well as the impact of load stalls. The event counts will be collected ten times for each benchmark. The minimum, maximum and average value are calculated. This makes it possible to compare different GC algorithms and different benchmarks. Additionally, the difference between average and minimum or maximum makes it possible to assess how scattered the results are.

4.2.1.1 Observed Events

The data collected by Perf is constrained to events that are collected by performance counters—also referred to as hardware counters—and software counters. Software counters are counters that are built into and collected by the kernel, such as context switches [82]. Table 4.3 lists the events collected by Perf in the coarse granularity setting, where *HW* is used for hardware events and *SW* is used for software events. The *task-clock*, *cycles* and *instructions* events are collected to obtain general runtime information that can be compared between different runs of the same benchmark.

Table 4.3: Collected Events for Coarse Granularity: All Events are only measured while at least one thread of the observed program is assigned to a CPU core. Where possible the events are also only counted if they occurred on a core currently executing a program thread.

Name	Type	Description
task-clock	SW	CPU time
cycles	HW	Elapsed cycles
instructions	HW	Executed instructions
l1i_cache	HW	L1 instruction cache accesses
l1i_cache_refill	HW	L1 instruction cache line refills
l1d_cache	HW	L1 data cache accesses
l1d_cache_refill_rd	HW	L1 data cache line refills caused by read operations
l1d_cache_refill_wr	HW	L1 data cache line refills caused by write operations
l2d_cache	HW	L2 data cache accesses
l2d_cache_refill_rd	HW	L2 data cache line refills caused by read operations
l2d_cache_refill_wr	HW	L2 data cache line refills caused by write operations

The *task-clock* event measures how many nanoseconds of CPU time the task used—a program running for one second on four cores would have a *task-clock* count of $4 * 10^9$ ns. The same is true for *cycles* and *instructions*, they are also measured for each core.

Based on these events more metrics can be calculated. This includes the Instructions Per Cycle (IPC) metric defined as the instructions per clock cycle as shown in Equation 4.1 [9].

$$IPC = \frac{instructions}{cycles} \tag{4.1}$$

In general, a higher IPC value is better as more instructions were executed in the same number of cycles. The more load stalls that occur while running the program, the lower will be the IPC value because more cycles are consumed in which no instruction could be executed. In addition, the CPU frequency is calculated with Equation 4.2

to check the average frequency of the CPU while running the program.

$$f_{CPU} = \frac{cycles}{time_{CPU}} \tag{4.2}$$

This metric is not influenced by the number of load stalls. However, in systems that dynamically adjust the CPU frequency, applications that stall for a significant amount of time might have a lower frequency [83].

The remaining counters are used to calculate the cache hit and cache miss ratios for the *L1i*, *L1d* and *L2d* caches. The cache misses for the *L1d* and *L2d* caches are split between cache misses that were caused by read and write operations. Because directly writing to the *L1i* cache is not permitted, only cache misses that were caused by read operations can be collected. To calculate the aforementioned values Equation 4.3 is used and changed accordingly based on the observed cache.

$$L1dCacheMissRd = \frac{l1d\_cache\_refill\_rd}{l1d\_cache} \tag{4.3a}$$

$$L1dCacheMissWR = \frac{l1d\_cache\_refill\_wr}{l1d\_cache} \tag{4.3b}$$

$$L1dCacheHit = 1 - L1dCacheMissRd - L1dCacheMissWR \tag{4.3c}$$

The ratio between cache misses compared to cache hits is particularly interesting because an application with more cache misses is more likely to stall. In addition, the cache misses that occur in the *L2d* cache will yield longer stalls because the data is fetched from the main memory.

In total, eleven performance counters are used to collect the execution data. The *task-clock* is a software event and is calculated by the kernel—it does not use a performance counter. As specified by the PMUv3 the *cycles* event is always counted and thus has a fixed performance counter. The remaining nine events must share six general purpose hardware counters. Perf will use multiplexing to obtain the data and

for programs with a long runtime the results will be close to the real values. This trade-off is acceptable to collect the relevant data. In systems with fewer hardware counters, this might lead to a large divergence between the real and collected event counts. To overcome this challenge, some events might need to be removed or the data can be accumulated over multiple runs.

4.2.2 Medium Granularity

The medium granularity level guides the selection of benchmarks to be used for fine granularity measurements. The collected data should differentiate between JIT compiled code and the remaining part of the JVM, because the fine granularity level focuses on JIT compiled code. Additionally, the difference between coarse and medium granularity levels makes it possible to assess the validity of the results obtained by sampling events. The events specified in Section 4.2.1.1 are sampled with Perf. Also, to accumulate the results, the same procedure as described for the coarse granularity level is used.

4.2.3 Fine Granularity

The fine granularity level should reveal the details of load stall behavior in JIT compiled code. To achieve this level of granularity the procedures described by Li [21] are used to determine hot methods—thus sampling is used. When classifying the instructions, pipeline effects need to be taken into account. The results from this granularly level should help determine optimization possibilities for the JIT or guide the development of new GC algorithms.

A method is considered hot if it is JIT compiled and consumes more cycles than the interpreter over multiple independent runs [21]. To further select instructions that offer optimization potential, hot instructions are chosen. A hot instruction consumes more than 5% of the method runtime and also consumes a significant number of cycles

in comparison to other instructions of the basic block it resides in. To determine if an instruction is consuming significantly more cycles, the Interquartile Range (IQR) fence method as described by Tukey [84] is used. Equation 4.4 shows how an interval is created—all data points outside of that range are considered to be significantly far away from the sample median [84].

$$[Q_1 - k(Q_3 - Q_1), Q_3 + k(Q_3 - Q_1)] \tag{4.4}$$

In the above equation, Q_n refers to the nth quartile and the parameter k is used to adjust the interval boundaries. Tukey [84] considers $k = 1.5$ to show outside data and if a value is outside the $k = 3$ interval it is far out [84]. An instruction is considered to be hot if it is outside of the $k = 3$ interval. Additionally, only the upper fence is used, because stalling instructions are considered to consume more cycles than other instructions.

This procedure of method local and block local filtering tries to reduce the instructions that are evaluated. This in turn helps focus on instructions where optimization might lead to big a performance increase.

4.3 Benchmarks

In this section the selected benchmark suites and the specific workloads used to collect the data are listed. The specific benchmarks were chosen because they were used by Li [21] to conduct stall-focused tests on the x86 architecture. Additionally, the benchmarks include a wide application range to collect data for different use cases. Support for the benchmarks was also already implemented in the load stall benchmarking framework [19].

The benchmarks AcmeAir and Daytrader7 are used, because they aim at implementing a web server dealing with operations commonly encountered in real life business

applications. They are evaluated using Apache JMeter [85] and the default workload is executed for two hours. Running the benchmark that long assures that a steady state will be reached and the obtained data resembles the pressure observed for long-running web applications.

Moreover, the HiBench suite is used to test the load stall behavior for long running applications that process large datasets with the MapReduce paradigm. A single node cluster is used to reliably benchmark the Spark executer and obtain the same results because the work is always distributed to the same backend. In total seven workloads from the machine learning category are included as they frequently access memory. The considered workloads are Alternating Least Squares (*als*), Bayesian Classification (*bayes*), Gradient Boosted Trees (*gbt*), K-means clustering (*kmeans*), Latent Dirichlet Allocation (*lda*), linear regression (*linear*) and Logarithmic Regression (*lr*). All benchmarks are executed with the size setting *large* except *lr* which uses size *small* as larger settings fail on the given hardware due to insufficient resources. As a benchmark for the popular Hadoop and Spark framework, the chosen workloads represent a mixture of scientific and real-world applications [50], [86].

Additionally, all workloads specified by the stall-focused benchmarking suite, developed by Li [21], are used to obtain stall information of the JVM implementation while performing micro benchmarks. The workloads include dense matrix multiplication with primitives (*dm*), dense matrix multiplication with *java.lang.Double* wrapper objects (*dwm*), sparse matrix multiplication based on linked lists (*sm*), n-ary tree search (*ts*), object interface checking resulting in a class hierarchy traversal (*ic*) and lastly operations on large (*ol*) objects [21].

Furthermore, workloads from the renaissance suite are used to cover scientific workloads as well as workloads designed to test JVM implementations. The included benchmarks are unbalanced cobwebbed tree computations with Akka (*akka-uct*), server loads with Twitter Finagle (*finagle-http*), the K-means clustering algorithm

(*fj-kmeans*), genetic function optimization (*future-genetic*), phone mnemonics computation (*mnemonics*) and scrabble puzzle solving (*rx-scrabble*) [52].
Finally, workloads from the SPECjvm2008 benchmarking suite are chosen to complete the data collection with workloads that also aim to test JVM implementations. The tested workloads include data encryption and decryption with the AES and DES protocols (*crypto.aes*), database operations with a focus on *java.lang.BigDecimal* (*derby*) and data compression with the Lempel-Ziv Method (*compress*) [54], [87].

4.4 Ranking Garbage Collection Algorithms

When the coarse and medium granular data is obtained for a given benchmark the performances of the GC algorithms need to be compared to draw conclusions. This section will discuss the different metrics used to rank the different strategies, based on different performance measures.

4.4.1 Metric Ranking

Due to the different requirements used when selecting a GC algorithm, it is not possible to define overall GC performance that would objectively consider all the different use cases. For example, running real-time applications requires a GC algorithm that is at most using a time interval x within a period of y cycles, where $x < y$. It is desirable but not necessarily important that x, or in other words the GC pause time, is small. Thus, in the example average GC pause time is traded for a hard upper limit maximum GC pause time. There are also applications that do not need that strict upper limit bound but benefit more from a lower average GC pause time and thus a lower average runtime.
The aforementioned examples showcased that the GC algorithm with the best possible overall performance highly depends on the specific application executed by the JVM.

As such this work cannot and does not aim to provide a ranking indicating that GC algorithm X is objectively better than algorithm Y. However, it is possible to rank the different benchmarks based on predefined metrics. This data and the drawn conclusions can then be used by the application user to select and test GC algorithms based on the needs of their specific application.

This work will use the term *metric* performance when comparing different GC algorithms based on a single metric. The procedure first assigns a metric performance value $mpv(c, b)$ to the combination of GC algorithm c and benchmark b, such that a lower value indicates a better performance specific to that metric. For example, $mpv(c, b)$ can be defined as the cycles consumed when executing benchmark b with GC algorithm c. The rank $r_b(c)$ of a GC algorithm $c \in C$ for a fixed benchmark b is then defined as

$$E_c = C \setminus \{c\} \tag{4.5a}$$

$$R_{c,b} = \{c_j \mid c_j \in E_c \wedge mpv(c_j, b) < mpv(c, b)\} \tag{4.5b}$$

$$r_b(c) = |R_{c,b}| + 1 \tag{4.5c}$$

In other words, the rank assigned to a GC algorithm for a given benchmark is its placement where the first place is assigned to the best performing GC and the nth place is assigned to the worst performing GC. If two GC algorithms c_i and c_j where $i \neq j$ tie, i.e., $mpv(c_i, b) = mpv(c_j, b)$, they are both assigned the same value.

This work will use the metrics CPU time and IPC as well as the L1D, L1I and L2 cache miss ratios. The values are calculated using the formulas described in Equations 4.1, 4.2 and 4.3. In the next section the different metrics used to compare the GC algorithms and collection strategies are presented.

4.4.2 Comparison Methodology

To present and discuss the coarse and medium granularity data, the minimum, mean and maximum value for the ten conducted runs for each benchmark is used. Reporting the mean and standard deviation was considered, however the samples did not indicate normal distribution and for a small sample size of ten the Central Limit Theorem cannot be applied [88].

When comparing the GC performance, the normalized values for each algorithm for the given metric are used to form the sample that is analyzed further. Normalization needs to be performed to accumulate the results across different benchmarks. The goal is to calculate $nmpv(c, b)$—a normalized version of $mpv(c, b)$—for configuration $c \in C$ and benchmark $b \in B$ as shown in the following.

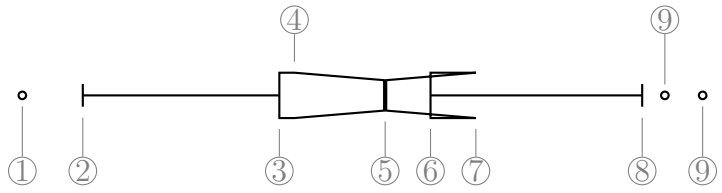
$$mpv_{min}(b) = \min \{mvp(c_j, b) \mid c_j \in C\} \tag{4.6a}$$

$$mpv_{max}(b) = \max \{mvp(c_j, b) \mid c_j \in C\} \tag{4.6b}$$

$$nmpv(c, b) = \frac{mpv(c, b) - mpv_{min}(b)}{mpv_{max}(b) - mpv_{min}(b)} \tag{4.6c}$$

The resulting values of $nmpv(c, b)$ are in $[0, 1]$, where a lower value indicates better performance for a given benchmark. To present the collected $nmpv(c, b)$ values, notched box-plot diagrams are used. In addition to standard box plot diagrams, they include a notch around the median. When two median values are compared and the notch intervals do not overlap, the distribution medians are statistically different. This work uses the equations from McGill et al. [89] resulting in an approximately 95% confidence interval [89]. Figure 4.1 shows a notched box-plot diagram and indicates how the different values are calculated for the diagrams used in this thesis.

Furthermore, when comparing the values collected from coarse and medium granularity the Kendall rank correlation coefficient τ is used [91]. The coefficient expresses



①	$\{v \mid v \in V \wedge v > Q_3 + 1.5 \cdot \mathrm{IQR}\}$
②	$\max\{v \mid v \in V \wedge v \leq Q_3 + 1.5 \cdot \mathrm{IQR}\}$
③	Q_3
④	$Q_2 + 1.7\frac{1.25 \cdot \mathrm{IQR}}{1.35\sqrt{\lvert V \rvert}}$
⑤	Q_2 (Median)
⑥	Q_1
⑦	$Q_2 - 1.7\frac{1.25 \cdot \mathrm{IQR}}{1.35\sqrt{\lvert V \rvert}}$
⑧	$\min\{v \mid v \in V \wedge v \geq Q_1 - 1.5 \cdot \mathrm{IQR}\}$
⑨	$\{v \mid v \in V \wedge v < Q_1 - 1.5 \cdot \mathrm{IQR}\}$

Figure 4.1: Example Notch Box Plot Diagram: The given formulas are taken from McGill et al. [89]. In the figure V is the set of all collected samples, Q_i is the i-th quartile and $\mathrm{IQR} = Q_3 - Q_1$.
Source: Adapted from [90]

similarity between two measured ordered where $\tau \in [-1, 1]$ and a $\tau = 1$ indicates perfect correlation. In addition, the probability ρ is given, where ρ describes the chance that the sample was randomly chosen and achieved the same correlation or better. If $\rho \leq 0.05$ the given τ is accepted for the measurement.

4.5 Hypotheses

Related to the research question and the data collected for the different granularity levels the following hypotheses are formulated.

1. The default GC algorithm *gencon* with the default hierarchical scan order performs best on average regarding execution time and instruction throughput.

2. For the GC policy *balanced* the dynamic breadth first scan order will yield fewer load stalls.

3. Regarding the object scan order for *gencon* the hierarchical scan order will have the best load stall performance.

4.6 Summary

In this chapter the methodology used to obtain load stall related results was presented. The collection process is split into three different granularity levels that allow for the drawing of different conclusions and help guide the selection of GC algorithms and reveal further optimization potential. Additionally, the benchmark suites and the workloads used to obtain the data were presented and categorized. Then different metrics to rank various aspects of the GC performance were presented. Finally, hypotheses regarding the research question were presented, that will be validated with the metrics and methods described in this chapter.

Chapter 5

Load Stall Benchmarking Framework

This chapter of the thesis shows and discusses the changes that were made to the previously developed load stall benchmarking framework. The changes allow the collection of load stall related data as discussed in Chapter 4. First, the addition of two new data collection paradigms is introduced and then the automatic classification procedure, including the description of most implementation changes, is presented. Finally, the changes that were made to support different GC policies and their scan orders—if the GC algorithm supports multiple scan orders—are described.

The changes made to the framework are tailored to the OpenJ9 JVM compliant with the Java Standard Edition 8 [92] running on a Linux system on top of an AArch64 CPU. Changing any of the aforementioned requirements has to be implemented in the framework, however the same procedures may be reused because the implemented algorithms are encapsulated interfaces. The following sections will highlight which additions are platform dependent and how they might be solved for different platforms.

The load stall benchmarking framework is modified to consist of three subcommands that can be used to collect data from the benchmarks. Each subcommand corresponds

to one of the granularity levels listed in Section 4.2. In the subsequent sections the different commands are described in further detail.

5.1 Coarse

As described in Section 2.7 counting refers to a technique used when collecting performance counters with minimal impact on the program execution. This section focuses on adding a target that counts event occurrences during a benchmark run.

5.1.1 Requirements

The procedures used to execute benchmarks as implemented for the previous version of the benchmark using sampling should stay the same, i.e., they should follow the same discover, setup, run, teardown, and forget cycle [19]. Thus, the infrastructure that was previously implemented to execute the supported benchmarks can be reused. Furthermore, data can be collected from benchmarks that will be added to the implementation without additional development overhead.
The implementation should use *perf stat* as Perf is already supported by the framework. However, the implementation should allow the usage of different performance measurement tools in later versions of the framework without the need to change the program structure.

5.1.2 Design

The major addition to the framework is a new abstraction layer for the *WorkloadOrchestrator* and *MeasurementTool*, where the data that is measured is made generic. In addition, the benchmark suites now call the workload orchestrator with the available benchmarks. This makes orchestrating easier and removes the filter as an orchestrator function parameter. Furthermore, the benchmarks can then use the

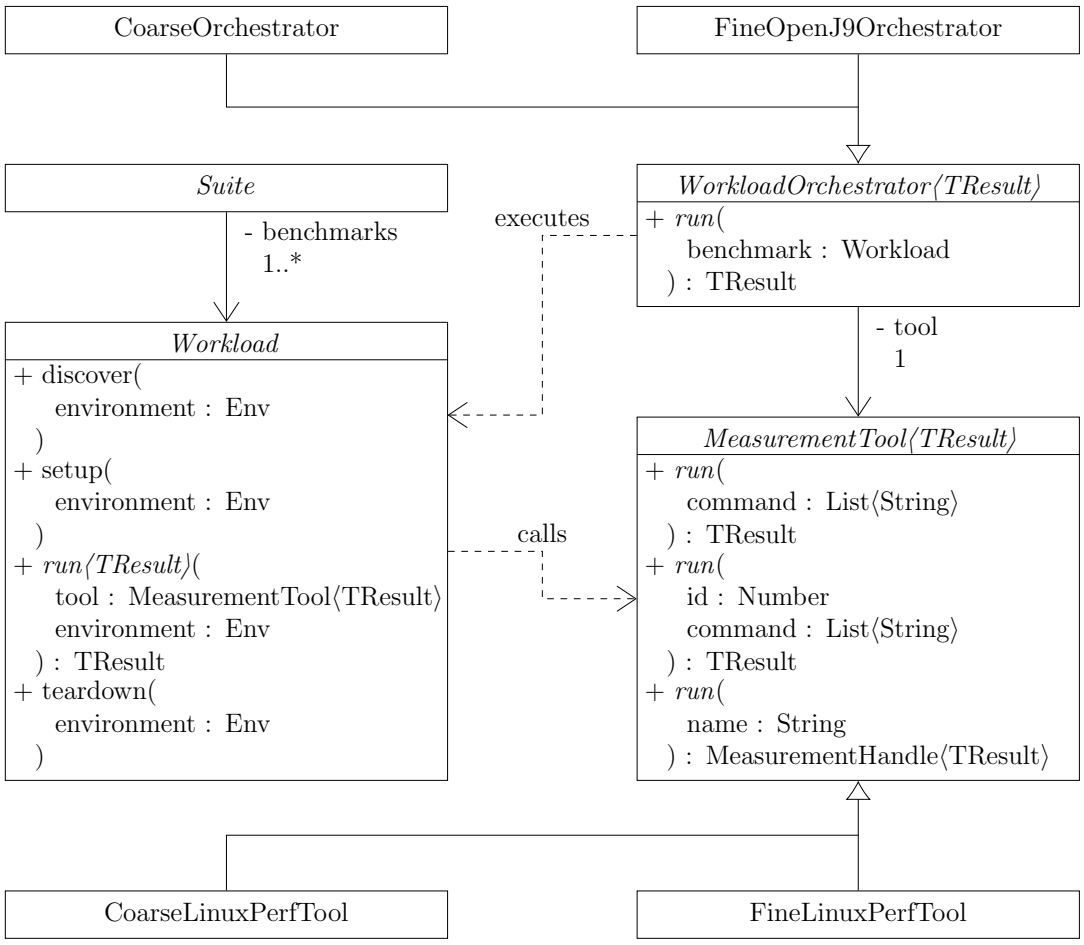


Figure 5.1: Benchmarking Framework Class Structure: The shown class diagram is not complete; not all classes and functions are included. The main difference is a new abstraction layer using generics. As a result, the workloads are not aware of the collected data—this is now defined by the *MeasurementTool* that runs and collects data for the benchmark.

abstract and generic functions of the measurement tool to obtain results without knowing what information is collected specifically. Figure 5.1 shows the new class structure.

The already existing orchestrator and measurement tool are adapted to inherit from the new abstract parent classes. A new orchestrator is added that collects the counting results for a specific benchmark—Figure 5.2 describes the collection procedure. Each benchmark first initializes the environment when the *discover* function is called. Then the benchmark is executed n times, where n can be adjusted using the available

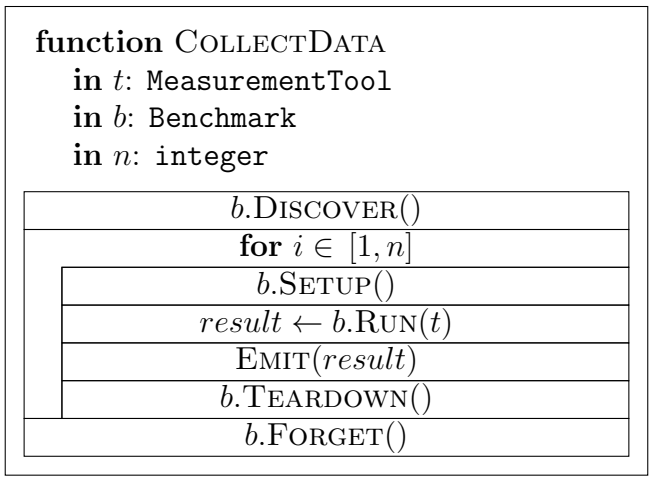


Figure 5.2: Benchmarking Procedure

configuration mechanism. In each iteration the benchmark environment is prepared for the next run. Then, while running the benchmark, the event occurrences are counted using the selected measurement tool. The individual results are accumulated to form the overall result. After running the benchmark, the environment is reset by calling the *teardown* function. Finally, after n iterations the *forget* function ensures that environmental changes that were made by previous calls are reverted.

As previously mentioned, a new measurement tool is added to the framework to count event occurrences. The tool is implemented using *perf stat*. The same interface definition that was used in the previous version of the framework is used, ensuring interoperability between the different implementations and the benchmarks. By implementing the abstract interface, other collection tools like Intel VTune could be used in the future to count the events.

To save the collected data two formats XML [93] and SQLite [94] are supported—both store the same information. The formats were chosen because they were already supported by the framework to save output. In particular the SQLite output is generated such that common information—for example suite and benchmark data—is stored in the same table for different collection strategies.

The command line interface is changed accordingly to support subcommands. The

user can select between the subcommands *coarse*, *medium* and *fine*. They initialize the application accordingly and run the benchmarks using the correct orchestrator, measurement tool and output writer based on the user selection. A complete definition of the command line interface can be found in Appendix A.

5.2 Medium

In addition to counting, a sampling subcommand is added to the framework. The design mostly follows the one discussed in the previous section—a new orchestrator, measurement tool and output writer are added accordingly. The benchmarking procedure is the same as the one implemented for the coarse subcommand. Thus, the results of the medium target can be compared to the results obtained with the coarse subcommand to investigate the impact of sampling on the application execution.
Moreover, the sampling data contains more information to filter samples that are associated with JIT compiled code. To collect the data *perf record* instead of *perf stat* is used.

5.2.1 Extraction of Data for Just-In-Time Compiled Code

While collecting samples of a program using *perf record*, a Perf data file is created that contains all samples. Each sample consists of the specific event name as well as information about the name of the symbol the program counter pointed to when the sample was collected. Because all methods that are JIT compiled with Testarossa follow a strict naming convention, the symbols and thus the samples can be attributed to JIT compiled code for further processing. An example for the described filtering as well as the naming convention is shown in Figure 5.3.
The divided samples are aggregated yielding two results, one for the samples that are attributed to JIT compiled code and one for all other events. The filtering and

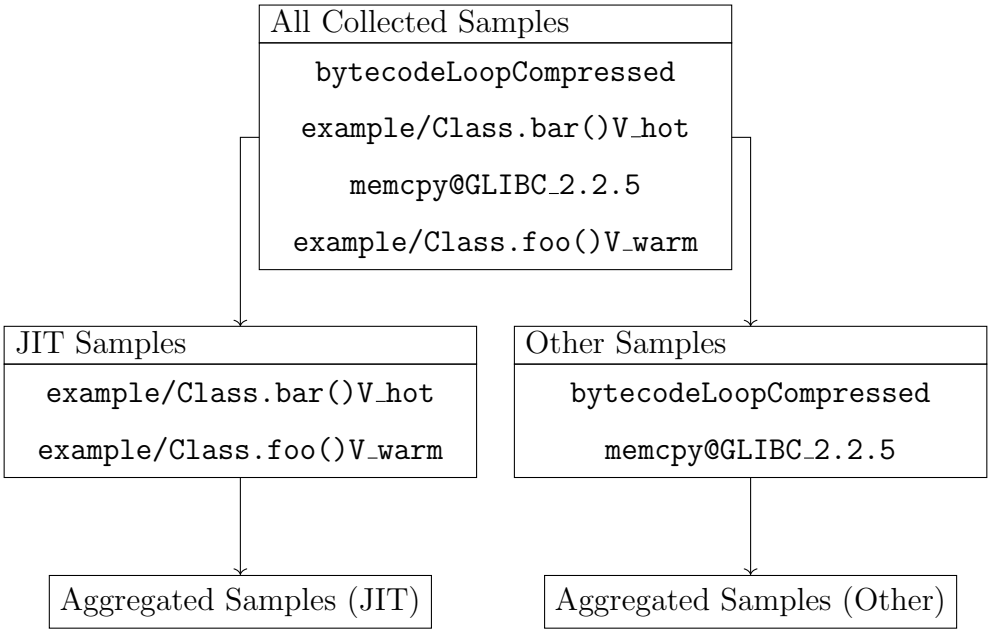


Figure 5.3: Just-In-Time Compiled Method Filtering

aggregation process is implemented using *perf script* together with the *dlfilter* option that filters the objects using a dynamically loaded shared object file and passes the result to a Perl [95] or Python [96] script. The shared object file is compiled from C++ [97] and must be provided on each benchmarking system, i.e., it is not automatically compiled by the framework.

As mentioned, the symbols are considered to be JIT compiled code if the observed symbol name matches the pattern for JIT compiled methods. As a result, the implemented filter is OpenJ9 and Testarossa specific. To change to different JVMs the filter script needs to be changed and selected appropriately when calling *perf script*.

5.3 Automated Classification

In this section the automatic classification of instructions that might cause load stalls as well as the implementation for the framework as part of the fine subcommand is discussed. The collection strategy was already implemented in a previous version of the

framework. First the classification procedure is described, then the implementation of the instruction parsing is presented.

5.3.1 Procedure

As described in Section 2.3.2 load stalls can be distinguished into frontend and backend stalls. In this section the procedure to distinguish between the different stall types is discussed. First, the characteristics of the stalls are given, based upon which a classification strategy is developed. To obtain accurate results while collecting the data the procedure only uses the cycle event. This also enables platform independence between different systems, because an accessible cycle counter is implemented in many CPUs.

5.3.1.1 Frontend Stall Classification

A frontend stall on AArch64 refers to a pipeline stall caused by bubbles inserted due to branch mispredictions or cache misses in the IF stage. The classification process developed by Li for x86 defines an instruction as a potential frontend stall, if the basic block the instruction resides in is unlikely to be called [21]. Based on this premise a micro benchmark was developed that causes L1 cache misses. The resulting program is sampled using Perf and the data is analyzed starting at hot instructions—i.e., instructions with a higher cycle count than expected.

The benchmark first defines six functions—*addOne*, *addFive*, *add100*, *subtractOne*, *subtractFive* and *subtract100*—that take an integer parameter and return an integer. The functions perform basic arithmetic operations implemented in assembly using the *add* and *sub* mnemonics—the amount of performed work is not important as the focus is on the function call itself. Furthermore, the functions can be aligned such that the function address distance is a multiple of X by using padding functions that only consist of *nop* instructions. The achieved function memory layout is shown in

```
0x5F       mov mov add ret   ; addOne
0x6F       nop nop nop nop   ; padAddFive
                   ...
0x4F+X     nop nop nop ret   ; padAddFive
0x5F+X     mov mov add ret   ; addFive
0x6F+X     nop nop nop nop   ; padAdd100
                   ...
0x4F+2X    nop nop nop ret   ; padAdd100
0x5F+2X    mov mov add ret   ; add100
                   ...
```

Figure 5.4: Frontend Stall Benchmark Function Layout: The given offset X refers to a value determined at compile time such that the functions reside in the same cache line.

Figure 5.4. If the value X is chosen such that the functions are stored within the same instruction cache line, eviction will result in more stalls. For example, for a 48 KiB 3-way set associative cache $X = 48/3$ KiB $= 16$ KiB would be chosen.

Furthermore, the benchmark defines objects with a pointer to such a function. When creating the object, one of the previously mentioned functions is randomly assigned to the objects. Then an array of such objects is created and the functions are called by iterating over the array. If the functions align with the cache, it is expected that the runtime will be slower and the number of L1 instruction cache misses increases. The obtained benchmark results are listed in Appendix B. They indicate that a frontend stall is only attributed to the first instruction of a basic block that is either unexpected or was not loaded in time resulting in a cache miss. This is particularly evident when looking at the number of L1 instruction cache misses between a program with aligned and unaligned functions. In addition, the frontend stalls also consume proportionally more cycles when comparing aligned function calls with unaligned function calls.

The procedure to classify frontend stalls based on the obtained results is described in Algorithm 5.1. It is largely comparable to the procedure described by Li [21], however due to the observed stalling behavior for the previously described benchmark, only

Algorithm 5.1: Frontend Stall Classification: Determines if instruction X could be a frontend stall. The function IsUnexpected refers to the procedure developed by Li [21].

```
1: procedure IsFrontendStall(in X: Instruction)
2:     BB ← BasicBlockOf(X)
3:     return FirstInstructionOf(BB) = X and IsUnexpected(BB)
4: end procedure
```

the first instruction of a block is considered to be a frontend stall.

5.3.1.2 Backend Stall Classification

The second category of stalls that is further examined are backend stalls during which the processor waits for completion of a memory access. This classification process also relies on sampling the application and retrieving the cycles needed to execute each instruction. However, because of out-of-order execution, the samples might be attributed to an instruction that is after the real stalling instruction in program order, since out-of-order execution changed the execution order such that the program counter—and thus the current instruction—already progressed while the memory access is performed in the background. When classifying instructions this effect needs to be taken into account.

The general idea of the classification procedure is to check if instruction X is a memory access or if it depends on instruction Y that is a memory access. Li [21] used the Bernstein condition as described in Section 2.2.2 to determine if two instructions depend on each other [21]. For the dependency analysis this work only considers the flow dependency—that is a part of the Bernstein condition—as described in Algorithm 5.2. This difference to the procedure proposed by Li [21] was considered due the use of techniques such as register renaming and result reordering that are implemented in many modern CPUs and that allow discovery of more instruction-level parallelism by renaming variables—thus removing output or anti dependencies [9].

The procedure described in Algorithm 5.2 can be used to test if the instructions A and

Algorithm 5.2: Dependency Check: Checks if instruction X depends on instruction Y.

```
1: procedure DependsOn(in X: Instruction, in Y: Instruction)
2:     if Precedes(Y, X) then
3:         in_X ← InputOperands(X)
4:         out_Y ← OutputOperands(Y)
5:         return in_X ∩ out_Y ≠ ∅
6:     end if
7:     return False
8: end procedure
```

$f(a, b, c) = \frac{2c}{a^2-b}$; where $a = 5 \land b = 7 \land c = 11$

translates to

```
1.  mov w0, 5        ; a = 5
2.  mov w1, 7        ; b = 7
3.  mov w2, 11       ; c = 11
4.  lsl w2, w2, 1    ; 2c
5.  nop              ; -
6.  mul w0, w0, w0   ; a^2
7.  sub w0, w0, w1   ; a^2 − b
8.  div w0, w2, w0   ; 2c / (a^2 − b)
```

Figure 5.5: Backend Stall Classification Program: The *nop* instruction is artificially inserted into the assembly to showcase graph creation and traversal in upcoming examples.

B directly depend on each other. However, a strategy needs to be developed, to check if two instructions transitively depend on each other. For example, if DependsOn(A, B) and DependsOn(B, C) is true and instruction C accesses the memory, then an increased cycle count attributed to instruction A could be due to a backend stall caused by instruction C. To discuss this problem further the example given in Figure 5.5 is considered.

The given assembly code describes the function $f(a, b, c)$ where a, b and c are fixed to create a self-contained example. The assembly code is implemented using arithmetic instructions and a *nop* instruction is inserted in line five. The *nop* instruction is inserted to better showcase the creation of the dependency graph as well as its

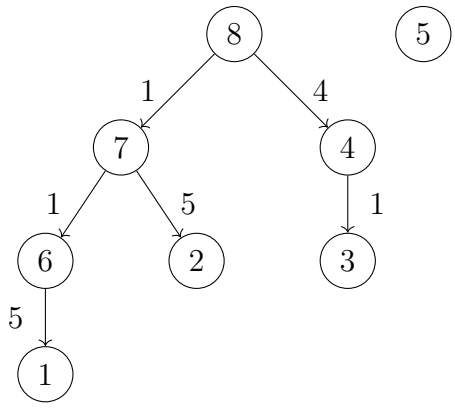


Figure 5.6: Backend Stall Classification Dependency Graph: The graph is based on the example shown in Figure 5.5 and uses the line numbers to refer to the different instructions.

traversal and would not be part of the direct translation of the function into assembly. Based on this input a weighted directed acyclic graph is built for each basic block where an edge from vertex A to vertex B indicates that the instruction represented by A depends on the instruction represented by B and the edge weight corresponds to the distance in instructions between A and B. Because not all instructions depend on each other the resulting graph is not necessarily connected. Figure 5.6 shows such a graph generated from the example program. The graph is constructed solely based on the instructions present in a basic block because control flow for the block is well defined. To improve dependency analysis capabilities the extended basic block is considered.

To traverse the graph a priority queue is used where the nodes are sorted based on the distance to the instruction that should be classified. Thus, the classification procedure can consider the instructions in the same order they were fetched by the CPU and before they were reordered. The distance between two dependant instructions is equivalent to the sum of the edge weights of the dependency path. Additionally, the traversal depth can be constrained by limiting the lookup distance or the dependency skip count. The lookup distance limits how many instructions the classification procedure will look back at, to make its decision and roughly translates to the cycles

Start	Visited Nodes	Lookup Distance	Dependency Skip
8	7, 6, 4, 3, 2, 1	∞	∞
8	7, 6, 4	∞	3
8	7, 6	3	∞

Table 5.1: Backend Classification Traversal: Shows the traversal of the dependency graph shown in Figure 5.6 including the effects of constraining the lookup distance as well as the dependency skip count.

a main memory request—i.e., the worst case—takes to complete. If instruction A depends on instruction B, directly or transitively, and B is a memory request but completed before A was called, then A should not be classified as a potential backend stall. The aforementioned dependency skip count limits the number of visited nodes or dependencies skipped to search for the stall cause. This limit translates to the reorder buffer size and limits the traversal to look at as many other instructions as there is space in the buffer for reordering. The difference between both limits and the general traversal order is shown in Table 5.1. If there is more than one path from one instruction to another, the procedure ensures that the instruction will only be processed once.

The overall procedure to determine if a given instruction is a potential backend stall is given in Algorithm 5.3. When calling the function, the dependency graph is already built and the constraints *lookup distance* and *dependency skip* are function parameters.

5.3.1.3 Overall

When combining the previously described classification procedures an instruction can be assigned one of four distinct categories. If IsFrontendStall(X) for instruction X is true, the category of X is FRONTEND_STALL. However, even if X is a potential frontend stall, if it also accesses the memory, it could be both a frontend and backend stall. This is indicated by the FRONTEND_OR_BACKEND_STALL

Algorithm 5.3: Backend Stall Classification: Determines if instruction X could be a backend stall.

```
1: procedure IsBackendStall(in X: Instruction,
                            in LookupDist: Integer,
                            in MaxSkip: Integer)
2:     q ← PriorityQueue()
3:     visted ← Set()
4:     Enqueue(q, X, 0)
5:     while not IsEmpty(q) do
6:         c ← DequeueMin(q)
7:         if not Contains(visted, c) and Size(visted) < MaxSkip then
8:             Add(visited, c)
9:             if IsMemoryAccess(c) then
10:                return True
11:            end if
12:            for dep ∈ Dependencies(c) do
13:                if Distance(X, dep) < LookupDist then
14:                    Enqueue(q, dep, Distance(X, dep))
15:                end if
16:            end for
17:        end if
18:    end while
19:    return False
20: end procedure
```

category. For all other instructions the function IsBackendStall(X) is called and if it returns true, they belong to the category BACKEND_STALL. Finally, if X is neither a frontend stall nor a backend stall it is classified as OTHER. The described procedure is shown in Algorithm 5.4.

Algorithm 5.4: Classification Procedure: Determines if instruction X might consume more cycles than usual due to a stall.

```
1: procedure ClassifyInstruction(in X: Instruction,
                                 in LookupDist: Integer,
                                 in MaxSkip: Integer)
2:     if IsFrontendStall(X) then
3:         if IsMemoryAccess(X) then
4:             return FRONTEND_OR_BACKEND_STALL
5:         end if
6:         return FRONTEND_STALL
7:     end if
8:     if IsBackendStall(X, LookupDist, MaxSkip) then
9:         return BACKEND_STALL
10:    end if
11:    return OTHER
12: end procedure
```

5.3.2 Instruction Parsing

As described in the previous chapter the dependency analysis that is used to determine if a given instruction is a backend stall or not, needs the input and output operands of each instruction to build the dependency graph. The implementation of the load stall benchmarking framework was extended to parse the assembly generated by Testarossa that describes the JIT compiled code. The instructions are available in their binary form and as assembly, where the assembly consists of a mnemonic and arguments and additional information. The following considerations are important to develop the instruction parser:

Correctness All used instructions need to be recognized correctly including their operands, otherwise the classification will yield incorrect results.

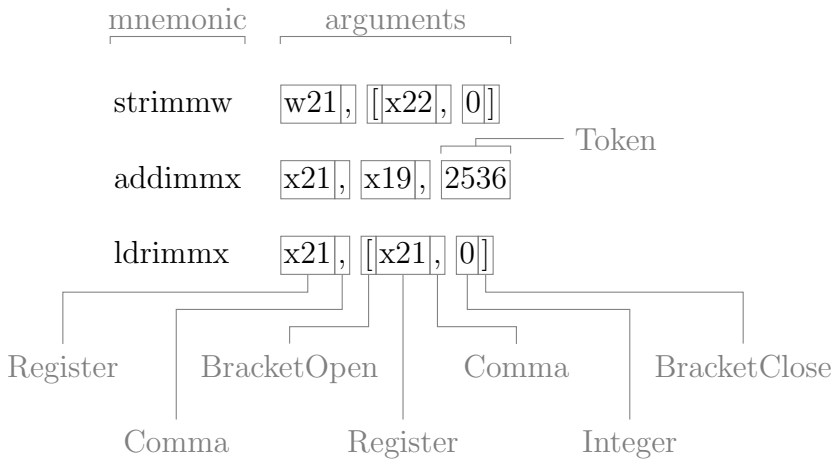


Figure 5.7: Assembly Instruction Example

Dependency Optimized The parsing of the data needs to be aware of the overlapping registers—defined in Section 2.1.2. For example, an instruction that uses *w0* depends on an instruction that writes to *x0*.

Extendable The parser or its output should be easily extendable to support new instruction sets. Ideally this option supports loading a language definition from a file that might be supplied by the user of the framework.

Based on the aforementioned considerations the parser was developed. The parser reads the mnemonics with their arguments and uses a language definition to recognize the instructions as well as their operands. The rules to parse an assembly language are given in a YAML [98] file.

In the following, the YAML file structure is explained using the example assembly code given in Figure 5.7. The various parts of an instruction are described in the following.

The definition used by the parser consists of tokens, arguments and instructions. Tokens are used to describe arguments and arguments are used when defining instructions. Each step constrains the order of tokens more and reduces them to exactly describe one instruction. In the following the different defined types are further

explained starting with tokens.

A token refers to the smallest identifiable unit and can consist of one or more characters. Individual tokens are concatenated with one or more separators—in some cases, like commas, no separators are needed. To define if a separator is required between two tokens, each token specifies a glue property that can be *none*, *left*, *right* or *both*. If tokens glue together the separator can be omitted otherwise at least one separator is required. Tokens are distinguished between *literal*, *pattern* and *composed* tokens. Literal tokens describe fixed character sequences. In the example in Figure 5.7 the token *Comma* is a literal token. In contrast, pattern tokens like *Integer* are defined using a regular expression that matches the token. Additionally, tokens can be combined to form composed tokens which helps to define common patterns. For example—by combining *BracketOpen*, *Register*, *Comma*, *Integer*, *BracketClose*—the composed token *Address* can be defined. They are not needed to describe a language but help to group tokens that are almost exclusively used together. To support dependency analysis tokens can be part of a dependency. For example, in general—excluding read only registers—tokens describing registers are important for the dependency analysis. To account for these special tokens, they can be marked as dependable. While parsing, dependable tokens are matched to a string id and instructions using the same string ids are expected to depend on each other. This string id can either be fixed or it can be generated by using a captured group from the token pattern. For example, the two registers *w21* and *x21* need to have the same id because they share the lower 32 bits. Some commonly used tokens are defined in Code Listing 5.1.

With the different tokens, arguments—more precisely argument lists—are defined. Each argument consists of zero or more tokens and arranges them in the order they appear in the assembly code. At this stage the type of dependency is chosen for the dependable tokens. They can be input, output or input-output dependencies. The

```
tokens:
  Whitespace:
    pattern: "\s"
    glue: "both"
  Comma:
    literal: ","
    glue: "both"
  OpenBracket:
    literal: "["
    glue: "right"
  CloseBracket:
    literal: "]"
    glue: "left"
  Integer:
    pattern: "(?:\+|-)?(?:0|[1-9]\d*)"
  Register:
    pattern: "[wx]([0-9]|[12][0-9]|3[01])"
    isDependable: true
    idPrefix: "id_"
  Address:
    composedOf: ["OpenBracket", "Register",
       "Comma", "CloseBracket"]
```

Code Listing 5.1: Parser Definition (Tokens): The defined tokens are created for the example given in Figure 5.7. They are also a subset of the tokens used by the framework to parse AArch64 assembly code.

```
arguments:
  RegisterInputAddress:
    - type: "Register"
      input: true
    - "Comma"
    - "Address"
      input: true
  RegisterOutputAddress:
    - type: "Register"
      output: true
    - "Comma"
    - "Address"
      input: true
  RegisterOutputRegisterInputNumber:
    - type: "Register"
      output: true
    - "Comma"
    - type: "Register"
      input: true
    - "Comma"
    - "Integer"
```

Code Listing 5.2: Parser Definition (Arguments): The defined arguments are created for the example given in Figure 5.7. They are also a subset of the arguments used by the framework to parse AArch64 assembly code.

argument definitions needed to parse the example in Figure 5.7 are given in Code Listing 5.2. As seen in the definition composed tokens like *Address* might also contain dependable tokens and thus, they can be input or output to the instruction too. The dependency type is passed to the *Register* token used by the *Address* token.

As previously described, instructions are defined using the arguments and matching them with one or more mnemonics. This combination of mnemonic and arguments uniquely defines an instruction and it is possible to extract the dependency information for a specific instruction. Furthermore, the backend stall analysis depends on the knowledge of which instructions access the memory. Thus, this information is also given in the definition for a specific list of mnemonics. An example for instruction definitions is provided in Code Listing 5.3.

```
instructions:
  - mnemonics: "strimmw"
    possibleArguments: "RegisterInputAddress"
    isMemoryAccess: true
  - mnemonics: "addimmx"
    possibleArguments:
       "RegisterOutputRegisterInputNumber"
  - mnemonics: "ldrimmx"
    possibleArguments: "RegisterOutputAddress"
    isMemoryAccess: true
```

Code Listing 5.3: Parser Definition (Instructions): The defined instructions are created for the example given in Figure 5.7. They are also a subset of the instructions used by the framework to parse AArch64 assembly.

Most assembly languages also contain a small set of statements that are variable, i.e., do not have a fixed mnemonic they can be mapped to. An example of such a statement is a label. To support parsing labels, the definition allows specifying variable mnemonics; they are described using patterns. By combining all previously listed definitions a complete parser rule set is created that would correctly parse and identify the instructions given in the example.

As previously mentioned, parsing of the arguments and some of the mnemonics is performed using regular expressions. This might have two important disadvantages. Depending on the regular expressions, parsing text can be slow, particularly if the pattern heavily relies on backtracking to match a given string[1]. In addition, regular expressions have the same descriptive power as finite automata—formal languages that can only be recognized by pushdown automata or Turing machines cannot be verified. An example for such a formal language is $L = \{1^n a 1^n | n \geq 0\}$ where on both sides of a exactly n ones are given[2].

However, for this particular application the previously mentioned disadvantages

[1]To match the string *a0123456789* with the regular expression `.*a\d*` 17 steps are needed because the first star operator is greedy. Chaining regular expressions of this type increases the impact.

[2]Examples: $a \in L \wedge 1a1 \in L \wedge 11a11 \in L \wedge 111a111 \in L \wedge \ldots$

are negligible, because the assembly language generated by Testarossa is generally constrained. It consists of mnemonics and the mnemonic narrows the allowed arguments down. There are also no complex constructs like mathematical expressions where parentheses would need to be matched. Overall—for this application—regular expressions are usable as they are easy to define in text form, are widely supported in programming languages and do not require building a more complicated parser based on context-free grammars.

The parser rules for AArch64 were implemented by analyzing the Testarossa component responsible for writing the assembly to the log files. The definition covers all encountered instructions during benchmarking and is listed in Appendix C. Because the instruction definition was created by manually reversing the log file writer the list is likely not complete. Additionally, some optimizations reassign the instruction opcodes resulting in instructions for which the mnemonics do not match the originally defined arguments—such instructions and their new arguments were manually added when they were encountered.

5.4 Supporting Garbage Collection Policies

Because the framework should allow the user to specify the GC and, where applicable, the scan order used when running the JVM, new options are defined within the existing framework. The two options *openj9.benchmark.gc* and *openj9.benchmark.gc-order* are added and can be set using an options file or the command line arguments of the framework. They are used by the abstract OpenJ9 handler class, that already included a facility to supply general JVM arguments to the underlying instance of the JVM. The previously mentioned arguments are incorporated in this procedure and are also passed to the JVM instance running the benchmark.

All other JVMs that are not monitored by Perf and instances that might be started to

run the benchmark—for example client programs creating request on the monitored server side—are not influenced by the GC policy. This ensures that all changes during program execution that occur due to the changed GC policy are monitored by Perf.

5.5 Summary

In this chapter the changes made to the load stall benchmarking framework in the course of this work were presented. They were included to satisfy the data collection requirements presented in Chapter 4. In all, two new targets were added to collect coarse and medium granular data. Additionally, the automatic classification of load stalls was implemented for the fine granularity subcommand and the classification algorithms were discussed and shown.

Chapter 6

Evaluation of Load Stalls

In this chapter the data obtained with the framework is presented and discussed. First the tested configurations are listed, then the data for the different granularity levels is analyzed.

6.1 Configurations

To abbreviate the reference to a specific configuration, the names displayed in Table 6.1 will be used. In addition, all runs are executed with the *-Xjit:perfTool* command line option. As a result, Testarossa generates a map file that helps Perf to map specific addresses of JIT compiled code to their symbolic name.

Table 6.1: Benchmark Configurations: The configurations used to execute the different benchmarks. They focus on the different GC algorithms and their scan order.

Name	GC Policy	Scan Order
B:BF	*balanced*	*breadthFirstScanOrdering*
B:dBF	*balanced*	*dynamicBreadthFirstScanOrdering*
G:BF	*gencon*	*breadthFirstScanOrdering*
G:dBF	*gencon*	*dynamicBreadthFirstScanOrdering*
G:H	*gencon*	*hierarchicalScanOrdering*
PAUSE	*optavgpause*	–
THRU	*optthruput*	–

6.2 Coarse Granularity

This section presents the counting data collected from the benchmarks. Then the hypotheses 1., 2. and 3. from Section 4.5 are tested with the obtained results. The counting data is used to test them, as it is the closest to the real event counts that occur during execution. The data shown in this section was calculated from the collected data samples available in Appendix D.

6.2.1 Frequency

In general, the calculated frequency of the different benchmarks indicates that the CPU executes the programs with an average frequency of 1.78 GHz, a minimum of 1.67 GHz and a maximum of 1.79 GHz. The lowest frequencies are measured for LoadStallSuite *sm*. Based on those observations, it is concluded, that the benchmarks are executed at the default CPU frequency of 1.8 GHz—in case of LoadStallSuite *sm* the short runtime is assumed to be the reason for the small, but nonetheless noticeable, fluctuation. The frequency data is given in Appendix D.1 as well as data supporting the conclusion that with longer runtime of LoadStallSuite *sm* its frequency converges to the average frequency of 1.78 GHz.

6.2.2 Cache Miss Ratios

In the following, the cache miss ratios for the different cache levels are presented.

6.2.2.1 L1I Cache

In general, the obtained L1I cache miss ratios do not indicate many differences between the different configurations. Typically, the values are not more than 2% apart, most of the time, or even less. In addition, the observed values show that the executed programs are cache friendly—the highest observed miss rate was 6.82%.

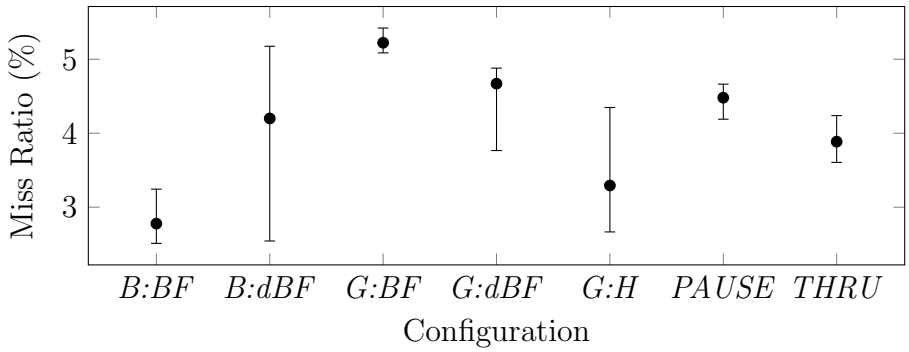


Figure 6.1: AcmeAir L1I Miss Ratio: Shown is the minimum, maximum and average miss ratio calculated from ten runs.

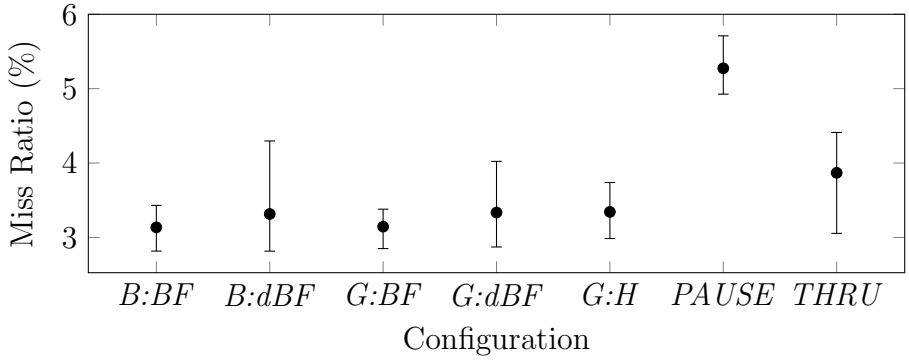


Figure 6.2: SPECjvm2008 *derby* L1I Miss Ratio: The same pattern is also noticeable for Renaissance *akka-uct* and HiBench *kmeans*. Shown is the minimum, maximum and average miss ratio calculated from ten runs.

The data for benchmarks with higher differences between GC algorithms for L1I cache performance are discussed in the next paragraphs.

The AcmeAir L1I cache miss ratio is shown in Figure 6.1 and indicates a relatively good performance for *B:BF* with 2.77% L1I cache misses. It is also noticeable that *B:dBF* occasionally performs as well as *B:BF*, however the performance is not consistent resulting in variation. *G:H* is comparable to *B:BF* as it also occasionally scores particularly good. The worst performance was consistently measured for *G:BF* with a L1I cache miss ratio of 5.22%.

The second observed trend is described using the data for SPECjvm2008 *derby* in Figure 6.2, it is also observed for the benchmarks Renaissance *akka-uct* and HiBench *kmeans*. It is characterized by a large gap between *PAUSE* and all other

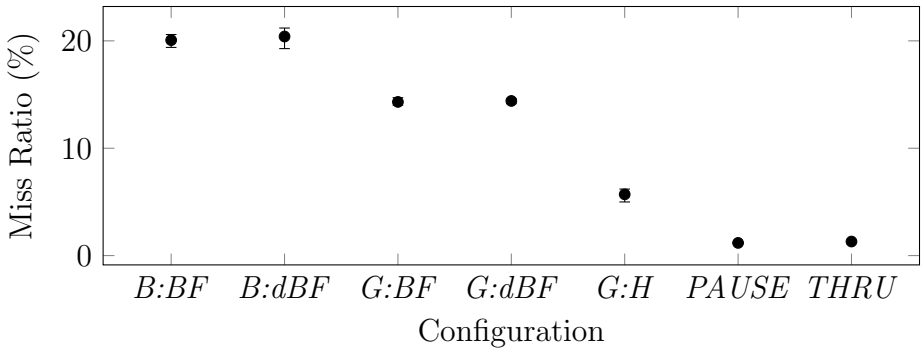


Figure 6.3: LoadStallSuite *ts* L1D Miss Ratio: Shown is the minimum, maximum and average miss ratio calculated from ten runs.

GC algorithms where *PAUSE* is performing the worst. Additionally, the miss ratio of *THRU* is also worse than the ones observed for *balanced* and *gencon*, however it is much closer.

Even though the presented benchmarks show different behaviors regarding L1I misses, the observed results are still close and the observed minimum and maximum values often overlap. Thus, no further conclusions can be extracted from the L1I cache behavior regarding the load stall behavior of the GC algorithms for different workloads. This is to be expected, as the GC is mostly concerned with changing the in-memory object layout and thus is not influencing time it takes to lookup instructions.

6.2.2.2 L1D Cache

The L1D cache miss ratios are, like the L1I miss ratios, relatively close together for the same configuration—again 2% difference at maximum between the highest and lowest ratio is often found. This also indicates that the data that is processed is likely available in the L1D cache. However, the benchmark LoadStallSuite *ts* differs from the general observation with a maximum cache miss L1D cache miss ratio of 20.41%; after that the second highest observed miss ratio is 5.36%.

Figure 6.3 shows the L1D cache miss ratios observed for LoadStallSuite *ts*. It has the highest cache miss ratio with 20.41% for *B:dBF* and also the biggest gap between

observed minimum and maximum between different GC algorithms of 19.23%. The overall results can be divided into four groups, *B:BF* and *B:dBF* performing the worst at around 20% cache misses followed by *G:BF* and *G:dBF* with around 14%. *PAUSE* and *THRU* perform best with cache miss ratios of 1.18% and 1.31% respectively. *G:H* has a cache miss ratio of 5.71% and is thus not considered to be part of the aforementioned groups. The reason for the superior performance of *THRU* and *PAUSE* is given by the implemented allocation pattern. The n children of each node are organized as a linked list, by keeping them relatively close the traversal of the children list is fast, they reside in the same cache line. However, breadth first search scan orders used for *B:BF*, *B:dBF*, *G:BF* as well as *G:dBF* reorganize the tree structures which—looking at the references—is a directed acyclic graph fitted into a treelike order, resulting in worse locality. As the number of siblings is relatively big, i.e., the dynamic breadth first search scan order does not significantly improve the node locality. The measured performance of *G:H* is also caused by a better achieved layout, connected tree node triplets are saved together in closer proximity. The difference between the *gencon* and the *balanced* policies can be explained by the use of regions that further fragment the tree nodes. In all, in this specific case the object structure used by the benchmark is already very cache friendly and the GC reordering techniques worsen the locality cache for child traversal.

6.2.2.3 L2 Cache

In contrast to the L1I and L1D cache misses, the L2 cache misses often show a difference between 4 and 6%. There are also more benchmarks exceeding the range. In general, the cache miss ratio of the L2 cache is higher than the ones observed for L1I and L1D. However, because the L1I and L1D cache already perform relatively well, the measured L2 ratios are considered to be higher as the misses, which occur because the cache is cold, have a relatively big impact on the miss ratios. The next

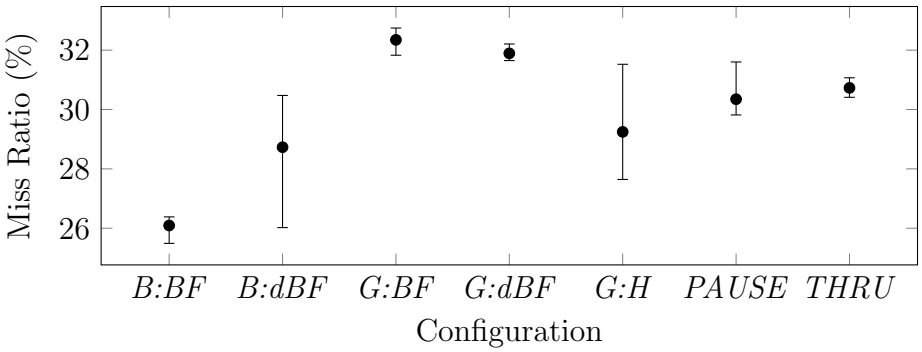


Figure 6.4: AcmeAir L2 Miss Ratio: A similar pattern is also observed for LoadStallSuite *ol*.

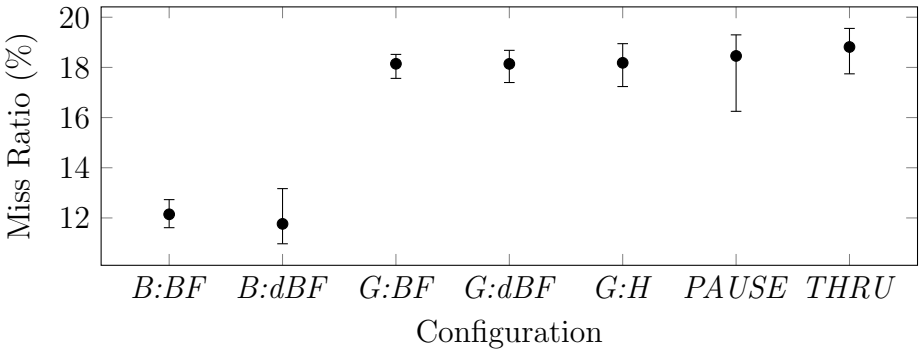


Figure 6.5: HiBench *gbt* L2 Miss Ratio

paragraphs discuss the results that were observed with a relative difference of 6% or higher.

The L2 cache miss ratio of AcmeAir is shown in Figure 6.4, the overall trend was also observed for the L2 cache miss ratio of LoadStallSuite *ol*. The data indicates that *B:BF* has the best cache miss ratio with 26.09%. The other GC algorithms are closer to 30% with *G:BF* yielding the highest results with 32.35%. Additionally, *B:dBF* and *G:H* have a big variation between observed minimum and maximum cache miss ratio.

Figure 6.5 shows the L2 cache miss ratios for HiBench *gbt* and a similar but more extreme behavior as the previously discussed AcmeAir benchmarks. The observed GC algorithms can be split into two groups where *B:BF* and *B:dBF* perform best with 12.14% and 11.76% respectively. All other GC algorithms have a cache miss

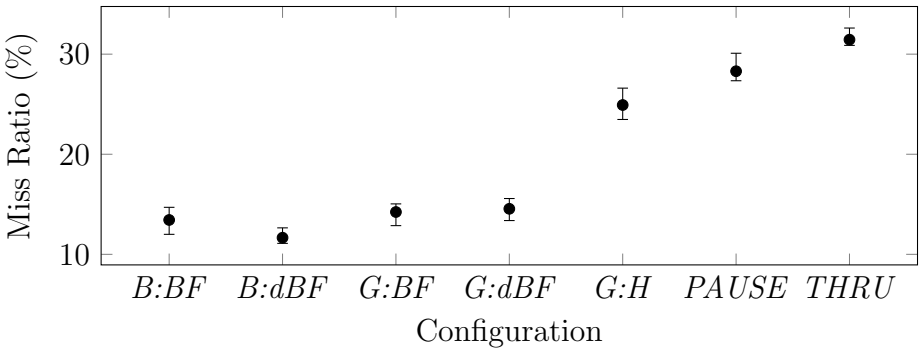


Figure 6.6: LoadStallSuite *ts* L2 Miss Ratio

ratio between 18% and 19%.

The last coarse granularity benchmark discussed is LoadStallSuite *ts*. The observed cache miss ratios show the mirrored cache miss ratios that were observed for L1D cache miss ratios. Thus, the performance of *PAUSE* and *THRU* are the worst with 28.29% and 31.45% respectively. The GC algorithms *B:BF* and *B:dBF* perform best with 13.43% and 11.66% cache miss ratios. Looking at the number of L2D lookups, it is possible to see that for *PAUSE* and *THRU* more cache misses occur, however in addition many fewer overall L2 cache accesses were observed, increasing the cache miss ratio even though *PAUSE* and *THRU* performed better because the L2 cache was accessed less.

6.2.3 Hypothesis 1: Overall Garbage Collection Performance

In this section the first hypothesis, claiming that the metric performance of *G:H* is better in comparison to other GC algorithms, is tested. Based on this statement the following null hypothesis is developed.

H_0 : There is no significant difference between *G:H* and other GC algorithms when comparing CPU time and the IPC metric.

To test the hypothesis the normalized ranking distribution of all GC algorithms is considered. Figure 6.7 shows the observed ranking distributions.

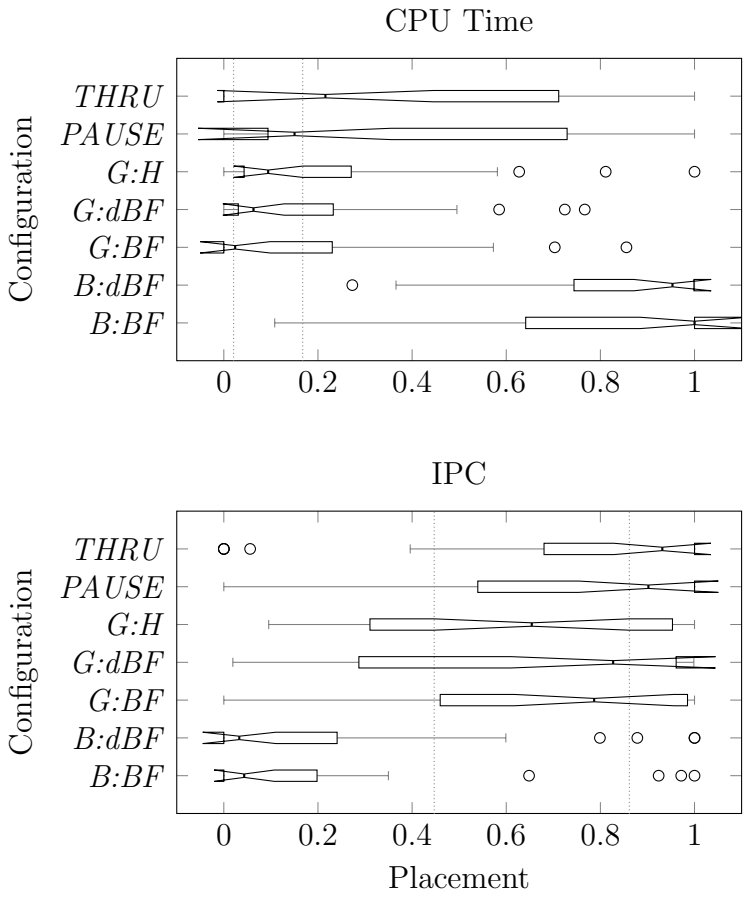


Figure 6.7: Overall Benchmark Ranking: Represented as a notched box plot diagram, its components are described in Figure 4.1. For both diagrams a lower value is better where zero is the perfect score and one the worst. Both diagrams include dotted lines showing the median significance interval for the tested configuration *G:H*.

Regarding CPU time, the observed algorithms *THRU*, *PAUSE*, *G:H*, *G:dBF* and *G:BF* are the closest to the perfect score. However, their notches overlap which indicates not statistically significant differences between the medians of the aforementioned GC algorithms. In addition, the distributions show that *B:dBF* and *B:BF* perform significantly worse than the other configurations. This is likely because balanced uses more threads to manage the GC cycles for the available regions increasing overall CPU time. This is important if the system running the JVM has few cores, i.e., the overhead of managing the regions is high and cannot be distributed across different cores.

For the IPC metric the GC algorithms group in the same way. The medians for *THRU*, *PAUSE*, *G:H*, *G:dBF* and *G:BF* are not significantly different; however, they are now outperformed by *B:dBF* and *B:BF*. The difference is likely due to the region management again. Running the management code on different cores yields more overall executed instructions. If, in addition, the executed instructions are relatively simple, i.e., they do not need multiple cycles, the IPC will indicate higher performance.

The obtained result show that H_0 cannot be rejected; *G:H* performs well in terms of CPU time but is not significantly different from the other GC algorithms. Additionally, regarding IPC, *G:H* does not perform best. As mentioned in Section 4.4.1 the results can be used as a starting point for choosing a GC algorithm that performs best for a given application.

6.2.4 Hypothesis 2: Load Stall Performance of *balanced*

The second hypothesis assumes that out of the available scan orders of the GC algorithm *balanced*, *B:dBF* will perform best in terms of load stall performance. The L1I, L1D and L2 miss ratios are used to approximate the stall behavior. The following null hypotheses is tested.

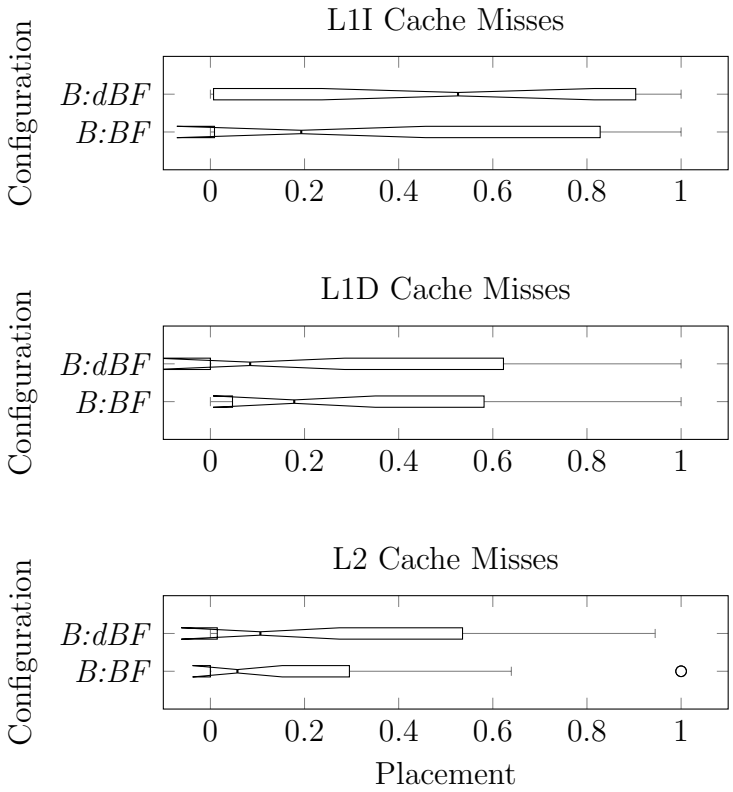


Figure 6.8: Stall Performance of *balanced* Scan Orders: Represented as a notched box plot diagram, its components are described in Figure 4.1

H_0 : There is no significant difference between the GC algorithms *B:dBF* and *B:BF* that would indicate that *B:dBF* has a better load stall performance.

The hypothesis is tested using the normalized rank across the benchmarks for the cache misses. The resulting distributions are shown in Figure 6.8.

By comparing the notch overlap for the different distributions, no significant difference between the medians of *B:dBF* and *B:BF* can be observed and thus, H_0 is accepted again. As a result, if choosing between the scan orders *B:dBF* and *B:BF* both need to be tested as there is no clear scan order that outperforms the respective other.

6.2.5 Hypothesis 3: Load Stall Performance of *gencon*

The third hypothesis is comparable to the second one but focuses on *gencon* and the available scan orders. It assumes that *G:H* has a better load stall performance than

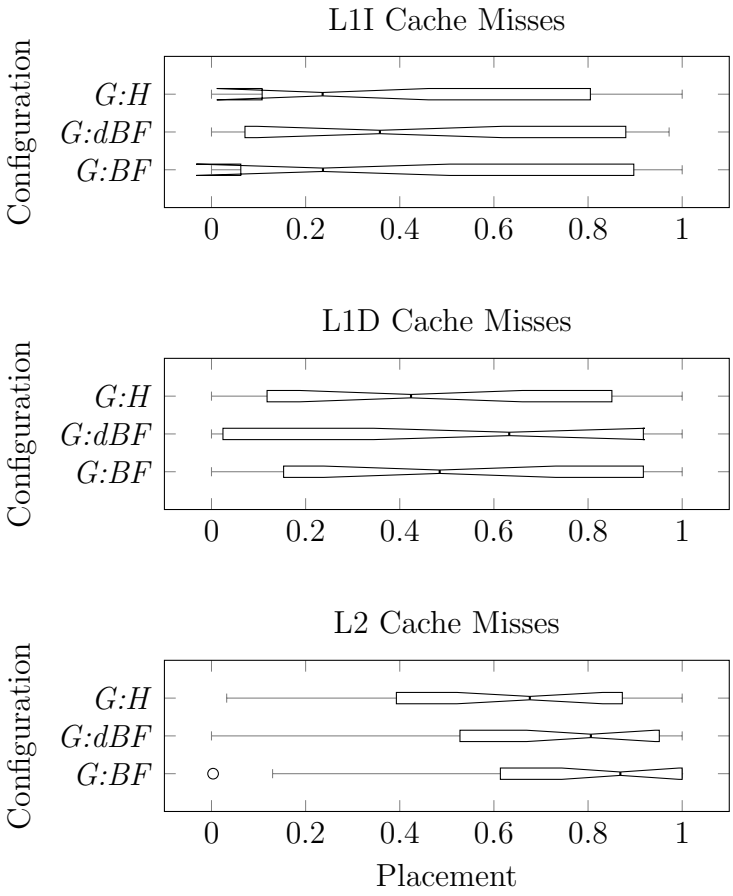


Figure 6.9: Stall Performance of *gencon* Scan Orders: Represented as a notched box plot diagram, its components are described in Figure 4.1

the other available scan orders; the below null hypotheses is used to test the claim.

H_0 : There is no significant difference between the GC algorithms *G:H*, *G:dBF* and *B:BF* that would indicate that *G:H* has a better load stall performance.

Again, the normalized L1I, L1D and L2 cache miss ratio ranking is used to test the load stall performance of the algorithms. The obtained data is shown in Figure 6.9. The observed relative rankings and in particular the overlapping notches for all three GC algorithms do not indicate significant differences between the medians of the different underlying distributions, H_0 is accepted, i.e., there is no significant difference between the tested scan orders regarding cache miss behavior and thus their load stall behavior.

6.3 Medium Granularity

As written in Section 4.2.2, the results of the medium granularity test are used to determine how closely the sampled data correlates to the counted values to make statements about the accuracy of the results obtained using the fine granularity. Additionally, this granularity level makes it possible to gain information about the stalling behavior of JIT compiled code. The obtained data this discussion is based on are listed in Appendix E.

6.3.1 Correlation

This section shows and discusses the measured correlations between the results collected with coarse and medium granularity. For each configuration and benchmark the Kendall correlation coefficient is calculated based on the measured cycles as well as the L1D, L1I, and L2 cache miss ratios. If for all of them the correlation coefficient τ is positive with a significance value of $\rho \geq 95\%$ or greater, it is assumed that both the coarse and medium granular data show the same trend. The resulting null hypothesis tested for each benchmark is then given as:

H_0 : There is no significant correlation between the coarse and medium granularity data regarding cycles, L1I, L1D and L2 cache misses.

Results for the benchmarks, where H_0 is accepted for at least one of the compared metrics, is shown in Table 6.2.

Because the fine granularity measurement samples CPU cycles to infer further information about the load stall behavior, a correlation between the counted and sampled cycles is preferred. As shown previously the L1I and L1D cache miss data are relatively close together with a high range of observed values. As a result, a rank change can likely occur leading to the acceptance of H_0 for some benchmarks.

Table 6.2: Benchmarks without Significant Positive Correlation: A cross indicates that for the given benchmark H_0 is accepted when comparing the results of the medium and fine granularity data.

Suite	Benchmark	Cycles	L1I	L1D	L2
AcmeAir	–	×	×	×	
Daytrader7	–	×			
HiBench	*gbt*		×	×	
HiBench	*linear*	×	×	×	×
HiBench	*lr*	×	×	×	×
LoadStallSuite	*ic*		×	×	×
LoadStallSuite	*sm*		×	×	×
LoadStallSuite	*ts*	×			
Renaissance	*mnemonics*		×	×	
SPECjvm2008	*compress*				×
SPECjvm2008	*crypto.aes*		×	×	×
SPECjvm2008	*derby*	×			

As a result the benchmarks listed in Table 6.2 are not excluded from the fine granularity analysis. However, the fine granularity data is separated between benchmarks that showed a high positive correlation in contrast to the remaining benchmarks. The obtained data can thus give more insights into the underlying causes for differences between correlated and non-correlated benchmarks.

6.3.2 Load Stalls in Just-In-Time Compiled Code

In addition, the obtained data makes it possible to gain insights on the load stall behavior of JIT compiled code. This section will discuss the obtained results based on the observed cache miss rates.

In general, the small differences between the averages for different GC algorithms that were observed for L1I, L1D and L2 cache miss ratios are even smaller when focusing on JIT compiled code. The differences are now often bound by 2% with only some exceptions. As a result, JIT compiled code does not seem to be as affected by the GC algorithm in terms of cache miss ratios. This is to be expected because the overall samples also include the data collected for the GC algorithm itself.

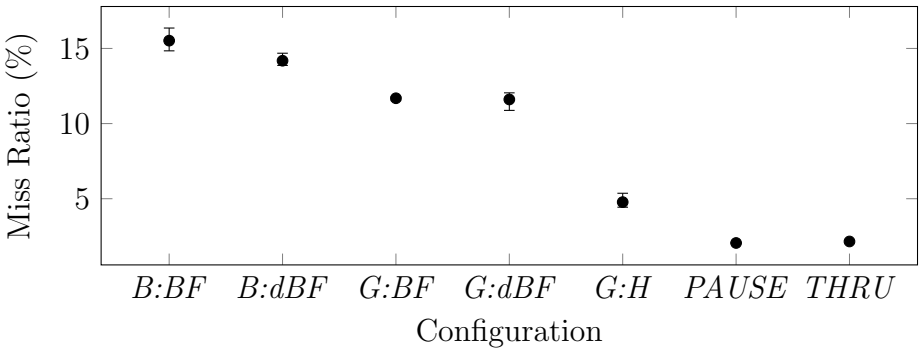


Figure 6.10: LoadStallSuite *ts* L1D Miss Ratio (JIT)

Even though the difference between the averages is smaller, the obtained minimum and maximum values are much wider spread; the obtained samples for JIT compiled code vary between runs. In general, fewer samples are collected for JIT compiled code in comparison to the whole program, which means that outliers are more likely to occur due to the inaccuracies of sampling, as a lost sample has a high impact on the calculates results. This can be observed in particular for short running applications—for example, for LoadStallSuite *sm* the minimum and maximum value differs by up to 62.11% for the L2 cache miss ratio of *THRU*.

However, the observations about the more extreme stall behaviors as described in Section 6.2.2 are also observable in JIT compiled code as shown in Figure 6.10 for LoadStallSuite *ts*. However, the maximum observed stall ratio also decreases by approximately 5% to 15.52% and 14.18% for the GC algorithms *B:BF* and *B:dBF*. The data indicates that the samples collected within Just-In-Time compiled code are less precise than the data collected overall because in comparison less time is spent in the JIT code collecting fewer samples. However, if the counted values indicated large difference between the cache miss ratios, similar trends can be observed in the JIT compiled code.

6.4 Fine Granularity

Finally, the collected fine granularity data is presented by analyzing the different collected stall categories for each GC algorithm. First the observed categories are presented, then the stalling assembly instructions are further evaluated. Next the JBCs that the stalls are ascribed to, are shown and lastly the distance between the sampled instruction and the stalling instruction is discussed. In the following B_c refers to the set of benchmarks for which positive correlation was shown and B_o is used for all other benchmarks. The corresponding data can be found in Appendix F.

6.4.1 Overview

First the different observed stall categories are analyzed for the two benchmark sets. Table 6.3 shows the obtained results. The data indicates that the major stalling instructions observed are backend stalls and instructions that were not classified, i.e., the category *other*. Only very few instructions were classified as frontend/backend stalls. Additionally, no instructions were classified as only frontend stalls, i.e., instructions at the start of an unexpected block that consume many cycles and are not memory accesses. All instructions could be classified, no classification error due to unknown instructions was observed.

Furthermore, the *gencon*, *optavgpause* and *optthruput* algorithms were highly impacted by instructions classified as other. Additionally, the stall data indicates that the different GC algorithms behave similarly regardless of scan order, i.e., all *balanced*, all *gencon* and *optavgpause* together with *optthruput* form three groups. Finally, using the new filtering method introduced in the previous chapter decreased the number of hot instructions in comparison to stalls observed in previous work [19]. This is intended to focus further analysis on instructions with a high impact on overall execution.

Table 6.3: Observed Load Stall Categories

Algorithm	Set	Backend	Frontend/Backend	Other
B:BF	B_c	12	1	5
	B_o	30	0	0
B:dBF	B_c	8	0	2
	B_o	27	2	1
G:BF	B_c	22	1	13
	B_o	35	3	5
G:dBF	B_c	26	2	11
	B_o	41	3	8
G:H	B_c	24	3	14
	B_o	36	3	4
PAUSE	B_c	19	2	8
	B_o	33	2	5
THRU	B_c	17	2	9
	B_o	28	2	6

6.4.2 Stalling Instructions

In this section the instructions that caused the stalls are further investigated; it is again differentiated between B_c and B_o. First the observations for B_c are discussed. The results obtained for *G:BF*—shown in Figure 6.11—reveal that the observed instructions classified as *other* are all due to the same instruction mnemonic *dmb*. The instruction is a Data Memory Barrier (DMB) that is issued to ensure that all previous memory accesses are written to memory. The target memory can be selected together with the memory access type. Thus, the *dmb* instruction stalling is expected, it explicitly limits out-of-order execution. Furthermore, the instructions *ldrimmw* and *vldroffd* are often observed to cause stalls—classified as either backend or frontend/backend.

For the benchmarks with B_o a broader range of instructions is observed as shown in Figure 6.12. In addition, the instruction *ldrimmw* is again observed to have caused load stalls. The data furthermore reveals that stalls due to the *dmb* instructions are not as prevalent as observed for B_c.

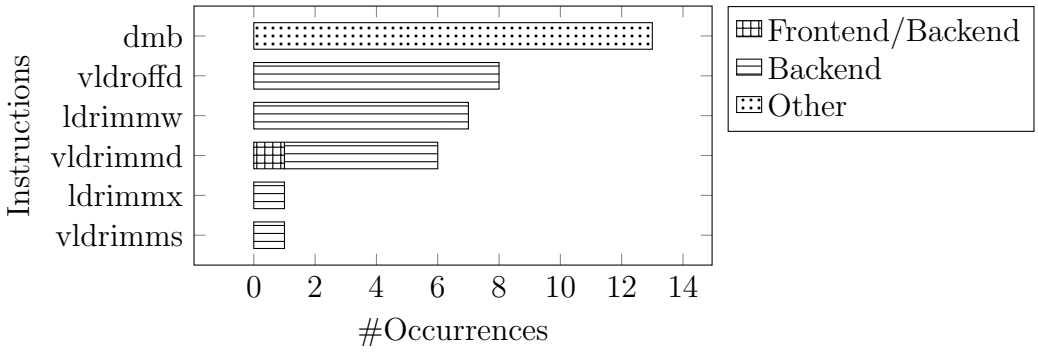


Figure 6.11: Stalling A64 Instructions for B_c (*G:BF*): The instructions are sorted by the total number occurrences and the different observed categories are stacked for each instruction.

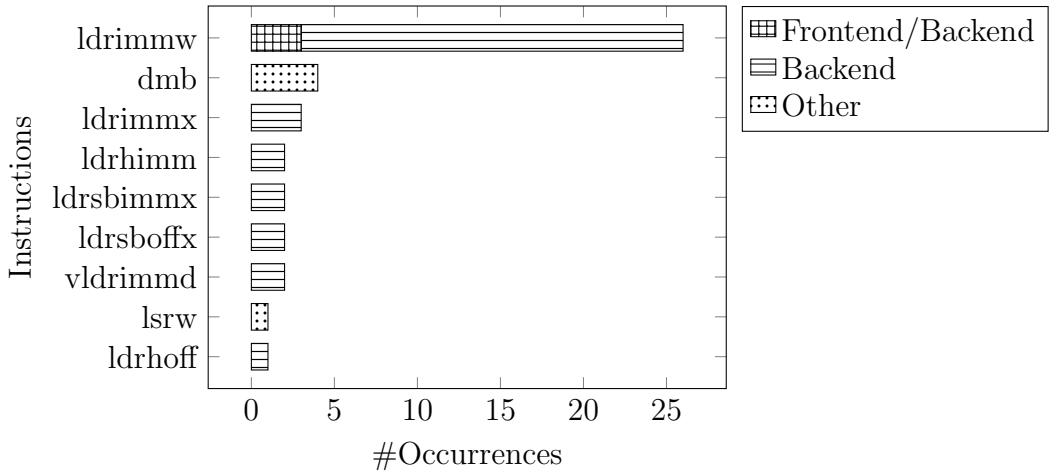


Figure 6.12: Stalling A64 Instructions for B_o (*G:BF*): The instructions are sorted by the total number occurrences and the different observed categories are stacked for each instruction.

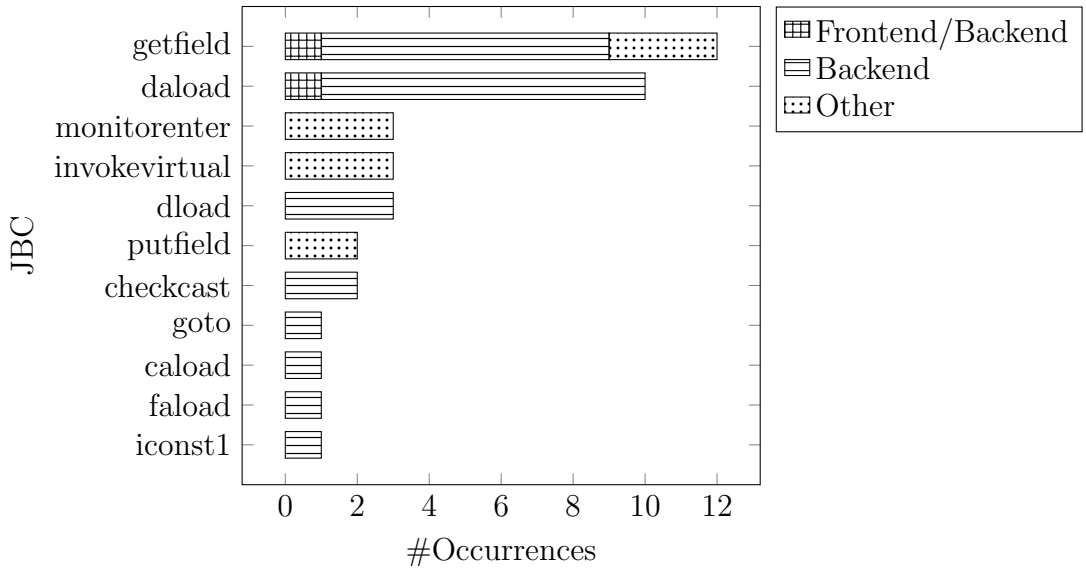


Figure 6.13: Stalling Java Byte Codes B_c (*G:dBF*): The JBCs are sorted by the total number occurrences and the different observed categories are stacked for each instruction.

6.4.3 Stalling Java Byte Codes

This section discusses the JBCs that are associated with the different stall categories for B_c and B_o. The JBCs that are implemented using *dmb* instructions are *invokevirtual*, *monitorenter* as well as *getfield* and *putfield*. In addition, the JBC that causes most backend stalls is *getfield*.

Figure 6.13 shows the observed JBCs for *G:dBF*. The distribution is typical for all GC algorithms except for the *balanced* algorithms as they only consist of backend stalls.

The JBCs observed for the benchmarks in set B_o are similar to the ones observed for B_c. However, instead of *getfield* the JBC *checkcast* can be found to generate the most stalling instructions. The observed JBCs for *G:dBF* are shown in Figure 6.14. This difference is explained by the different benchmark behaviors that were run to obtain the results.

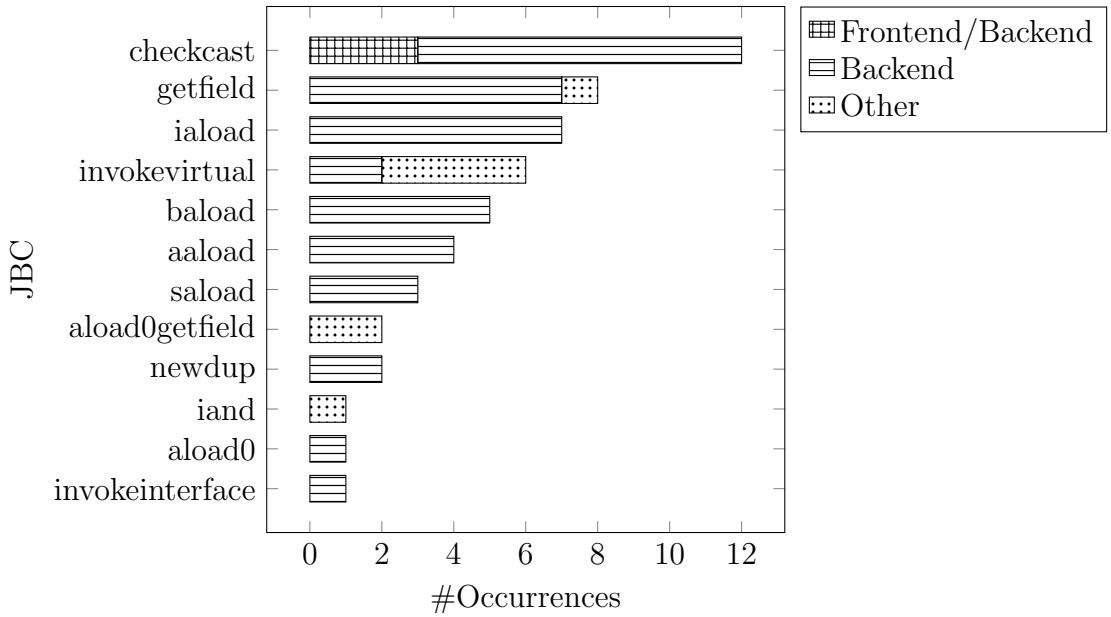


Figure 6.14: Stalling Java Byte Codes for B_o (*G:dBF*): The JBCs are sorted by the total number occurrences and the different observed categories are stacked for each instruction.

6.4.4 Backend Stall Distance

The backend stall distance is the distance between the instruction the *cycles* are attributed to and the instruction that was determined as the cause of the stall. The data shows that none of the backend stalls were observed for instructions that are not also memory accesses—i.e., the stalling distance was always zero. This observation indicates that even though the tested CPU consisted of a 128-entry reorder buffer, it was not able to close the processor memory performance gap in all cases. This can either be caused by issuing too many load or stall instructions at once, filling up the Load/Store Unit (LSU) buffers, or if load or store instructions take a long time to complete. The first problem would indicate suboptimal code structure whereas the second problem is caused by poor locality.

6.5 Summary

In this chapter, the data obtained for the different granularity levels was presented as well as discussed. It was shown, that the GC algorithm choice influences cache miss behavior in specific cases but is mostly grouped together. Furthermore, the hypotheses made in Section 4.5 were tested and rejected. This indicates that the impact of the GC algorithm on load stalls on AArch64 is, on average, not as big as expected. However, for certain workloads—like LoadStallSuite *ts*—the GC algorithm had a big impact on the cache miss ratio, because the object reordering scattered the objects in the heap.

Furthermore, the comparison between coarse and medium granularity data showed that, depending on the workload, no strong positive correlation between the counted and sampled data can be assumed. Moreover, the trends observed in JIT compiled code are less extreme, presumably because fewer samples are collected and the caches already contain the data that is used.

Finally, using the fine granularity data common stalling instructions and byte codes were identified. The results indicated that the out-of-order execution pipeline was not able hide or close the memory processor performance gap.

Chapter 7

Conclusions and Future Work

This work aimed at answering the research question: what impact GC algorithms have on the load stall behavior of JVMs running on AArch64?

To answer the question, Chapter 5 explored the possibilities of detecting load stalls in JVMs using different observation methods. In particular algorithms and data structures to automatically classify hot instructions in JIT compiled code were developed. The discussed procedures only used basic events to obtain results because dedicated stall counters are not available on the available AMR Cortex-A72 CPU.

In Chapter 6 the developed framework was used to collect and analyze the data observed from different benchmarks for the GC algorithms available in OpenJ9. The coarse granularity data revealed benchmarks that are impacted by the GC choice. However, the hypotheses that were made based on the expectation of the load stall behavior were rejected, indicating that on AArch64 and for the chosen benchmarks the impact of the GC algorithm on load stall behavior differs on an individual level but no general trends could be observed. The medium granular data revealed that for JIT compiled code, the GC has less impact on stalling performance and depending on the benchmark sampling yields significantly different results than counting events. Lastly the fine granularity data obtained showed the different instructions that cause

stalls with a high impact for the GC algorithms. It is particularly notable that the highest impact was observed for instructions where the out-of-order execution failed to hide the gap.

In general, if selecting a GC algorithm, *balanced* should be chosen to get good overall cache performance but if the environment is constrained, i.e., it does not have enough cores to run the application and manage the regions, *gencon* should be considered. For programs that use graph-like data structures, *optavgpause* or *optthruput* can be beneficial, in particular if the allocation order highly relates to the traversal order because the other tested GC algorithms were not able to find placement strategies that benefitted locality. Thus this work has unveiled the possibilities of the *optavgpause* or *optthruput* policies to improve locality if the access patterns are hard to predict or the limited depth first search is not able to copy the objects in a locality aware order.

7.1 Future Work

This thesis revealed potential future work based on the obtained results. Different future directions are discussed in this section.

This work focuses on the OpenJ9 JVM, however repeating the experiments with other JVM implementations could uncover differences not only in the general load stall behavior but also in regards to the observed patterns. Additionally, this would extend the knowledge base for other GC algorithms.

As mentioned in Section 4.1 this work uses the Raspberry Pi Model 4B to conduct the experiments and the specific hardware implementation is assumed to have an impact on the observed stalls. In the future—by conducting the experiments on different AArch64 machines—the observed patterns can be validated and new patterns might be discovered. Different results are expected, in particular for hardware that uses different designs to implement components that serve to hide or close the

processor-memory performance gap—e.g., different cache eviction policies.

Currently the detection of hot instructions is primarily based on the collected cycle samples for each instruction with a block-local and a method-global filter. This filter is biased towards instructions that, by definition, consume more cycles than other instructions. Enabling the framework to consider the minimum number of cycles the execution of an instruction takes to normalize the collected samples, allows for more precise detection of stalling operations.

The load stall benchmarks developed by Li [21] try to cause load stalls based on observations from other benchmark applications. The general procedure is to set up the benchmark, create all the needed objects and then repeatedly access the data. Only the LoadStallSuite *ts* benchmark allocates a substantial number of new objects during the aforementioned routines. The future development of benchmarks that also create allocation pressure could be beneficial as they give GC algorithms more opportunities to reorder the objects in memory to improve locality. This, in turn, allows for better testing of GC algorithms and their effectiveness regarding object reordering and locality improvements.

This work has shown that for the given benchmarks the load stalls that occurred were mainly backend stalls. Their impact is particularly noticeable when the out-of-order pipeline fails to reorder more incoming instructions. The results of this work suggest that, for AArch64, JIT compiler optimizations that can improve locality—like prefetching—should be investigated further. As the performance of locality aware GC algorithms can fluctuate; while in some instances can have a positive impact on locality, in other instances they can also negatively impact locality.

Bibliography

[1] ARM Ltd. "Smartphone Processors - Smartphone Technology," [Online]. Available: https://www.arm.com/markets/consumer-technologies/smartphones (visited on 08/27/2023).

[2] IBM Corp. "The future is now: ARM's AArch64 on the rise with IBM Instana." (May 30, 2022), [Online]. Available: https://www.ibm.com/blog/the-future-is-now-arms-aarch64-on-the-rise-with-ibm-instana/ (visited on 08/27/2023).

[3] P. Carbonnelle. "PYPL PopularitY of Programming Language index," [Online]. Available: https://pypl.github.io/PYPL.html (visited on 08/27/2023).

[4] S. Sherry. "What is Java Used For? 10 Major Applications of Java." (Apr. 21, 2022), [Online]. Available: https://www.hostreview.com/blog/220421-what-is-java-used-for-10-major-applications-of-java (visited on 08/27/2023).

[5] *The Java® Virtual Machine Specification*, version 8, T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, Feb. 13, 2015, [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se8/html/index.html (visited on 12/16/2024).

[6] A. W. Wade, P. A. Kulkarni, and M. R. Jantz, "AOT vs. JIT: Impact of profile data on code quality," in *Proceedings of the 18th ACM SIGPLAN/SIGBED Con-*

ference on Languages, Compilers, and Tools for Embedded Systems, ser. LCTES 2017, New York, NY, USA: Association for Computing Machinery, Jun. 21, 2017, pp. 1–10, ISBN: 978-1-4503-5030-3. DOI: 10.1145/3078633.3081037. [Online]. Available: https://dl.acm.org/doi/10.1145/3078633.3081037 (visited on 08/27/2023).

[7] E. Ovadia. "How Memory Safety Approaches Speed Up and Slow Down Development Velocity." (Jan. 16, 2023), [Online]. Available: https://verdagon.dev/blog/when-to-use-memory-safe-part-2 (visited on 08/27/2023).

[8] N. R. Mahapatra and B. Venkatrao, "The processor-memory bottleneck: Problems and solutions," *XRDS: Crossroads, The ACM Magazine for Students*, vol. 5, 2–es, 3es Apr. 1999, ISSN: 1528-4972. DOI: 10.1145/357783.331677.

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Nov. 23, 2017, 939 pp., ISBN: 978-0-12-811906-8. Google Books: cM8mDwAAQBAJ.

[10] ARM Ltd. "ARMv8-A," [Online]. Available: https://developer.arm.com/documentation/den0024/a/ARMv8-A-Architecture-and-Processors/ARMv8-A (visited on 08/27/2023).

[11] ARM Ltd. "Arm A-profile A64 Instruction Set Architecture," [Online]. Available: https://developer.arm.com/documentation/ddi0602/2022-09?lang=en (visited on 01/15/2024).

[12] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan Kaufmann, May 6, 2016, 724 pp., ISBN: 978-0-12-801835-4.

[13] ARM Ltd. "ARMv8-A: Registers," [Online]. Available: https://developer.arm.com/documentation/den0024/a/ARMv8-Registers (visited on 01/05/2024).

[14] ARM Ltd. "AArch64: System Registers," [Online]. Available: https://developer.arm.com/documentation/102374/0102/Registers-in-AArch64---system-registers (visited on 01/05/2024).

[15] ARM Ltd. "AArch64: Other Registers," [Online]. Available: https://developer.arm.com/documentation/102374/0102/Registers-in-AArch64---other-registers (visited on 01/12/2024).

[16] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*, New York, NY, USA: Association for Computing Machinery, Jul. 1, 1970, pp. 1–19, ISBN: 978-1-4503-7386-9. DOI: 10.1145/800028.808479. [Online]. Available: https://dl.acm.org/doi/10.1145/800028.808479 (visited on 04/21/2024).

[17] K. Cooper and L. Torczon, *Engineering a Compiler*, 2 edition. Amsterdam Heidelberg: Morgan Kaufmann, Feb. 21, 2011, 824 pp., ISBN: 978-0-12-088478-0.

[18] A. J. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Transactions on Electronic Computers*, vol. EC-15, no. 5, pp. 757–763, Oct. 1966, ISSN: 0367-7508. DOI: 10.1109/PGEC.1966.264565. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4038883?casa_token=C2_00ID55gIAAAAA:huMUrVgXTUFo2xOQmgzUHjdLPlFjKAtUJb-sshyMt8YHY7zvq-x0shHJxpaaG2Kuy8my3z_Q6A (visited on 12/19/2023).

[19] J. R. Schönauer, "Automated Framework for Stall-Focused Benchmarks for JVMs on AArch64," Hochschule Bonn-Rhein-Sieg, M.S. project, Mar. 3, 2023.

[20] ARM Ltd. "Armasm: NOP," [Online]. Available: https://developer.arm.com/documentation/dui0801/h/A64-General-Instructions/NOP (visited on 01/15/2024).

[21] Z. Li, "Stall-Focused Benchmarks for JVMs on the X86 Architecture," M.S. thesis, University of New Brunswick, Jun. 2021, 125 pp.

[22] ARM Ltd. "Arm Cortex-A53: Features," [Online]. Available: https://developer.arm.com/documentation/ddi0500/j/Introduction/Features (visited on 01/16/2024).

[23] ARM Ltd. "ARM Cortex-A57: Features," [Online]. Available: https://developer.arm.com/documentation/ddi0488/h/introduction/features?lang=en (visited on 01/16/2024).

[24] *Write once, run anywhere*, in *Wikipedia*, Jan. 6, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Write_once,_run_anywhere&oldid=1194053413 (visited on 02/19/2024).

[25] Sun Microsystems Inc. "Javasoft ships java 1.0," [Online]. Available: https://web.archive.org/web/20070310235103/http://www.sun.com/smi/Press/sunflash/1996-01/sunflash.960123.10561.xml (visited on 08/27/2023).

[26] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Elsevier, Jul. 12, 2005, 661 pp., ISBN: 978-0-08-052540-2. Google Books: JPhQw41vD2MC.

[27] M. Stoodley, "Under the Hood of the Testarossa JIT Compiler," presented at the JavaOne 2016 (San Francisco), Sep. 19, 2016. [Online]. Available: https://slideshare.net/MarkStoodley/under-the-hood-of-the-testarossa-jit-compiler (visited on 01/23/2013).

[28] P. Pufek, H. Grgić, and B. Mihaljević, "Analysis of Garbage Collection Algorithms and Memory Management in Java," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2019, pp. 1677–1682. DOI: 10.23919/MIPRO.2019.8756844.

[29] IBM Corp. "Garbage collection." (Aug. 17, 2023), [Online]. Available: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=management-garbage-collection-gc (visited on 12/17/2024).

[30] R. Westrelin. "How the JIT compiler boosts Java performance in OpenJDK." (Jun. 23, 2021), [Online]. Available: https://developers.redhat.com/articles/2021/06/23/how-jit-compiler-boosts-java-performance-openjdk (visited on 01/20/2024).

[31] Eclipse Foundation. "Eclipse OpenJ9," [Online]. Available: https://eclipse.dev/openj9/ (visited on 08/25/2023).

[32] Eclipse Foundation. "Garbage collection policies," [Online]. Available: https://eclipse.dev/openj9/docs/gc/ (visited on 08/27/2023).

[33] K. Briggs. "Memory management in Eclipse OpenJ9." (Jan. 11, 2019), [Online]. Available: https://developer.ibm.com/articles/garbage-collection-tradeoffs-and-tuning-with-openj9/ (visited on 08/27/2023).

[34] C. Gracie, "Deep dive into the Eclipse OpenJ9 GC technologies," presented at the JPoint 2018 (Moscow). [Online]. Available: https://assets.ctfassets.net/oxjq45e8ilak/3d22xuJ2HeQkoAesWkKic0/1ee2d740edcc5ebaf1602ae16a680267/CharlieGracie_DeepDiveIntoTheEclipseOpenJ9Technologies.pdf (visited on 01/23/2013).

[35] IBM Corp. "IBM AIX," [Online]. Available: https://www.ibm.com/products/aix (visited on 01/22/2024).

[36] D. Pivkine. "Debug cmdLineTests on MacOS," Eclipse OpenJ9. (Nov. 1, 2018), [Online]. Available: https://github.com/eclipse-openj9/openj9/issues/3321#issuecomment-435074705 (visited on 04/16/2024).

[37] Eclipse Foundation. "OpenJ9 -Xgc," [Online]. Available: https://eclipse.dev/openj9/docs/xgc/ (visited on 08/27/2023).

[38] C. J. Cheney, "A nonrecursive list compacting algorithm," *Communications of the ACM*, vol. 13, no. 11, pp. 677–678, Nov. 1, 1970, ISSN: 0001-0782. DOI: 10.1145/362790.362798. [Online]. Available: https://dl.acm.org/doi/10.1145/362790.362798 (visited on 08/27/2023).

[39] M. Stoodley. "Eclipse OpenJ9 Verbose JIT logs." (Apr. 26, 2018), [Online]. Available: https://mstoodle.github.io/EclipseOpenJ9JitVerboseLogs/ (visited on 01/17/2024).

[40] ARM Ltd. "ARM Cortex-A72: Performance Monitor Unit," [Online]. Available: https://developer.arm.com/documentation/100095/0003/Performance-Monitor-Unit?lang=en (visited on 01/20/2024).

[41] Intel Corp. "Intel® VTune™ Profiler," [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html (visited on 01/22/2024).

[42] The Linux Kernel Organization. "Perf Wiki," [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page (visited on 08/25/2023).

[43] The Linux Kernel Organization. "Counting with perf stat," [Online]. Available: https://perf.wiki.kernel.org/index.php/Tutorial#Counting_with_perf_stat (visited on 01/07/2024).

[44] The Linux Kernel Organization. "Sampling with perf record," [Online]. Available: https://perf.wiki.kernel.org/index.php/Tutorial#Sampling_with_perf_record (visited on 01/03/2024).

[45] The Linux Kernel Organization. "Multiplexing and scaling events," [Online]. Available: https://perf.wiki.kernel.org/index.php/Tutorial#multiplexing_and_scaling_events (visited on 01/02/2024).

[46] IBM Corp. "Open Liberty," [Online]. Available: https://openliberty.io/ (visited on 01/22/2024).

[47] MongoDB Inc. "MongoDB: The Developer Data Platform," [Online]. Available: https://www.mongodb.com/en-us (visited on 01/22/2024).

[48] *Acme Air Sample and Benchmark (monolithic simple version)*, Aug. 2, 2022. [Online]. Available: https://github.com/blueperf/acmeair-monolithic-java (visited on 08/25/2023).

[49] *Sample.Daytrader7*, IBM Corp., Aug. 24, 2023. [Online]. Available: https://github.com/WASdev/sample.daytrader7 (visited on 08/25/2023).

[50] *HiBench Suite*, Aug. 18, 2023. [Online]. Available: https://github.com/Intel-bigdata/HiBench (visited on 08/25/2023).

[51] "Renaissance Suite," [Online]. Available: https://renaissance.dev/ (visited on 08/25/2023).

[52] A. Prokopec, A. Rosà, D. Leopoldseder, et al., "Renaissance: Benchmarking suite for parallel applications on the JVM," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, New York, NY, USA: Association for Computing Machinery, Jun. 8, 2019, pp. 31–47, ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314637. [Online]. Available: https://doi.org/10.1145/3314221.3314637 (visited on 01/22/2024).

[53] Standard Performance Evaluation Corp. "SPECjvm® 2008," [Online]. Available: https://www.spec.org/jvm2008/ (visited on 08/25/2023).

[54] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko, "SPECjvm2008 Performance Characterization," in *Computer Performance Evaluation and Benchmarking*, D. Kaeli and K. Sachs, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2009, pp. 17–35, ISBN: 978-3-540-93799-9. DOI: 10.1007/978-3-540-93799-9_2.

[55] D. A. Moon, "Garbage collection in a large LISP system," in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, ser. LFP '84, New York, NY, USA: Association for Computing Machinery, Aug. 6, 1984, pp. 235–246, ISBN: 978-0-89791-142-9. DOI: 10.1145/800055.802040. [Online]. Available: https://dl.acm.org/doi/10.1145/800055.802040 (visited on 08/27/2023).

[56] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng, "The garbage collection advantage: Improving program locality," *ACM SIGPLAN Notices*, vol. 39, no. 10, pp. 69–80, Oct. 1, 2004, ISSN: 0362-1340. DOI: 10.1145/1035292.1028983. [Online]. Available: https://doi.org/10.1145/1035292.1028983 (visited on 08/27/2023).

[57] B. Alpern, C. R. Attanasio, J. J. Barton, et al., "The Jalapeño virtual machine," *IBM Systems Journal*, vol. 39, no. 1, pp. 211–238, Jan. 1, 2000, ISSN: 0018-8670. DOI: 10.1147/sj.391.0211. [Online]. Available: https://doi.org/10.1147/sj.391.0211 (visited on 04/03/2024).

[58] S. M. Blackburn, R. Garner, C. Hoffmann, et al., "The DaCapo benchmarks: Java benchmarking development and analysis," *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 169–190, Oct. 16, 2006, ISSN: 0362-1340. DOI: 10.1145/1167515.1167488. [Online]. Available: https://dl.acm.org/doi/10.1145/1167515.1167488 (visited on 02/19/2024).

[59] Standard Performance Evaluation Corp. "SPEC JVM98 Benchmarks," [Online]. Available: https://www.spec.org/jvm98/ (visited on 02/19/2024).

[60] Standard Performance Evaluation Corp. "SPEC JBB2000," [Online]. Available: https://www.spec.org/jbb2000/ (visited on 04/16/2024).

[61] D. Siegwart and M. Hirzel, "Improving locality with parallel hierarchical copying GC," in *Proceedings of the 5th International Symposium on Memory Man-*

agement, ser. ISMM '06, New York, NY, USA: Association for Computing Machinery, Jun. 10, 2006, pp. 52–63, ISBN: 978-1-59593-221-1. DOI: 10.1145/1133956.1133964. [Online]. Available: https://dl.acm.org/doi/10.1145/1133956.1133964 (visited on 02/19/2024).

[62] P. R. Wilson, M. S. Lam, and T. G. Moher, "Effective "static-graph" reorganization to improve locality in garbage-collected systems," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, ser. PLDI '91, New York, NY, USA: Association for Computing Machinery, May 1, 1991, pp. 177–191, ISBN: 978-0-89791-428-4. DOI: 10.1145/113445.113461. [Online]. Available: https://dl.acm.org/doi/10.1145/113445.113461 (visited on 04/04/2024).

[63] T. M. Chilimbi and J. R. Larus, "Using generational garbage collection to implement cache-conscious data placement," in *Proceedings of the 1st International Symposium on Memory Management*, Vancouver British Columbia Canada: ACM, Oct. 1998, pp. 37–48, ISBN: 978-1-58113-114-7. DOI: 10.1145/286860.286865. [Online]. Available: https://dl.acm.org/doi/10.1145/286860.286865 (visited on 02/19/2024).

[64] W.-k. Chen, S. Bhansali, T. Chilimbi, X. Gao, and W. Chuang, "Profile-guided proactive garbage collection for locality optimization," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06, New York, NY, USA: Association for Computing Machinery, Jun. 11, 2006, pp. 332–340, ISBN: 978-1-59593-320-1. DOI: 10.1145/1133981.1134021. [Online]. Available: https://dl.acm.org/doi/10.1145/1133981.1134021 (visited on 02/19/2024).

[65] P. Lengauer, V. Bitto, H. Mössenböck, and M. Weninger, "A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo,

DaCapo Scala, and SPECjvm2008," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17, New York, NY, USA: Association for Computing Machinery, Apr. 17, 2017, pp. 3–14, ISBN: 978-1-4503-4404-3. DOI: 10.1145/3030207.3030211. [Online]. Available: https://dl.acm.org/doi/10.1145/3030207.3030211 (visited on 02/19/2024).

[66] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder, "Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11, New York, NY, USA: Association for Computing Machinery, Oct. 22, 2011, pp. 657–676, ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048118. [Online]. Available: https://dl.acm.org/doi/10.1145/2048066.2048118 (visited on 02/19/2024).

[67] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and realities: The performance impact of garbage collection," *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1, pp. 25–36, Jun. 1, 2004, ISSN: 0163-5999. DOI: 10.1145/1012888.1005693. [Online]. Available: https://doi.org/10.1145/1012888.1005693 (visited on 08/27/2023).

[68] H. Grgic, B. Mihaljević, and A. Radovan, "Comparison of garbage collectors in Java programming language," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2018, pp. 1539–1544. DOI: 10.23919/MIPRO.2018.8400277.

[69] Oracle Inc. "The HotSpot Group," [Online]. Available: https://openjdk.org/groups/hotspot/ (visited on 02/20/2024).

[70] M. B. Reinhold, "Cache Performance of Garbage-collected Programming Languages," Sep. 1993. [Online]. Available: https://dspace.mit.edu/handle/1721.1/149748 (visited on 02/20/2024).

[71] "Scheme," [Online]. Available: https://www.scheme.org/ (visited on 02/20/2024).

[72] O. Papadakis, A. Andronikakis, N. Foutris, et al., "A Multifaceted Memory Analysis of Java Benchmarks," in *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, ser. MPLR 2023, New York, NY, USA: Association for Computing Machinery, Oct. 19, 2023, pp. 70–84. DOI: 10.1145/3617651.3622978. [Online]. Available: https://dl.acm.org/doi/10.1145/3617651.3622978 (visited on 02/19/2024).

[73] C. Wimmer, M. Haupt, M. Van De Vanter, M. Jordan, L. Daynès, and D. Simon, "Maxine: An Approachable Virtual Machine For, and In, Java," presented at the ACM Transactions on Architecture and Code Optimization (TACO), vol. 9, Jan. 20, 2013. DOI: 10.1145/2400682.2400689.

[74] O. Papadakis, "Performance Analysis and Optimizations of Managed Applications on Non-Uniform Memory Architectures," Ph.D. dissertation, The University of Manchester (United Kingdom), England, 2022, 144 pp. [Online]. Available: https://www.escholar.manchester.ac.uk/api/datastream?publicationPid=uk-ac-man-scw:335501&datastreamId=FULL-TEXT.PDF (visited on 04/05/2024).

[75] Raspberry Pi Ltd. "Raspberry Pi 4," [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/ (visited on 08/25/2023).

[76] Raspberry Pi Ltd. "Operating system images - Raspberry Pi," [Online]. Available: https://www.raspberrypi.com/software/operating-systems/ (visited on 08/25/2023).

[77] Paul J. Drongowski. "ARM Cortex-A72 execution and load/store." (Jan. 6, 2021), [Online]. Available: http://sandsoftwaresound.net/arm-cortex-a72-execution-and-load-store/ (visited on 03/15/2024).

[78] Paul J. Drongowski. "ARM Cortex-A72 fetch and branch processing." (Dec. 8, 2020), [Online]. Available: http://sandsoftwaresound.net/arm-cortex-a72-fetch-and-branch-processing/ (visited on 03/15/2024).

[79] Raspberry Pi Ltd. "Raspberry Pi 4 Model B specifications," Raspberry Pi, [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/ (visited on 03/15/2024).

[80] ARM Ltd. "ARM Cortex-A72: Memory Management Unit," [Online]. Available: https://developer.arm.com/documentation/100095/0001/memory-management-unit/ (visited on 03/15/2024).

[81] IBM Corp. "IBM Semeru Runtimes," [Online]. Available: https://developer.ibm.com/languages/java/semeru-runtimes/ (visited on 08/25/2023).

[82] The Linux Kernel Organization. "Events," [Online]. Available: https://perf.wiki.kernel.org/index.php/Tutorial#Events (visited on 01/24/2024).

[83] G. Dhiman and T. S. Rosing, "System-Level Power Management Using Online Learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 676–689, May 2009, ISSN: 1937-4151. DOI: 10.1109/TCAD.2009.2015740. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4838819 (visited on 02/20/2024).

[84] J. W. Tukey, *Exploratory Data Analysis*, in collab. with Internet Archive. Reading, Mass. : Addison-Wesley Pub. Co., 1977, 714 pp., ISBN: 978-0-201-07616-5.

[85] Apache Software Foundation. "Apache JMeter - Apache JMeter™," [Online]. Available: https://jmeter.apache.org/ (visited on 03/15/2024).

[86] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, Mar. 2010, pp. 41–51. DOI: 10.1109/ICDEW.2010.5452747. [Online]. Available: https://ieeexplore.ieee.org/document/5452747 (visited on 03/15/2024).

[87] Standard Performance Evaluation Corp. "SPECjvm2008 Benchmarks," [Online]. Available: https://www.spec.org/jvm2008/docs/benchmarks/index.html (visited on 03/15/2024).

[88] R. V. Hogg, E. A. Tanis, and D. L. Zimmerman, *Probability and Statistical Inference*, Tenth edition. Hoboken: Pearson, 2020, 548 pp., ISBN: 978-0-13-518939-9.

[89] R. McGill, J. W. Tukey, and W. A. Larsen, "Variations of Box Plots," *The American Statistician*, vol. 32, no. 1, pp. 12–16, Feb. 1, 1978, ISSN: 0003-1305. DOI: 10.1080/00031305.1978.10479236. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/00031305.1978.10479236.

[90] F. Schoonjans. "Construction of a Box-and-Whisker plot," MedCalc, [Online]. Available: https://www.medcalc.org/manual/box-and-whisker.php (visited on 03/15/2024).

[91] M. G. Kendall, "A New Measure Of Rank Correlation," *Biometrika*, vol. 30, no. 1-2, pp. 81–93, Jun. 1, 1938, ISSN: 0006-3444. DOI: 10.1093/biomet/30.1

-2.81. [Online]. Available: https://doi.org/10.1093/biomet/30.1-2.81 (visited on 03/16/2024).

[92] Oracle Inc. "Java platform, standard edition (java SE) 8," [Online]. Available: https://docs.oracle.com/javase/8/ (visited on 01/24/2024).

[93] World Wide Web Consortium. "Extensible Markup Language (XML)," [Online]. Available: https://www.w3.org/XML/ (visited on 02/20/2024).

[94] "SQLite," [Online]. Available: https://www.sqlite.org/index.html (visited on 02/20/2024).

[95] Perl Org. "Perl," [Online]. Available: https://www.perl.org/ (visited on 02/20/2024).

[96] Python Software Foundation. "Python," Python.org. (Feb. 15, 2024), [Online]. Available: https://www.python.org/ (visited on 02/20/2024).

[97] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, Jul. 10, 2013, 1366 pp., ISBN: 978-0-13-352285-3. Google Books: PSUNAAAAQBAJ.

[98] "YAML," [Online]. Available: https://yaml.org/ (visited on 02/20/2024).

Appendix A

Load Stall Framework Command Line Interface

This chapter lists the command line flags, options and parameters for the different commands of the command line interface. Parameters that are supplied by the user are enclosed with angle brackets. The given explanation omits the *lsf* and *lsf list* commands because they cannot be used to obtain benchmarking data.

`lsf coarse` Executes given suites and uses a performance measurement tool to observe event occurrences that are related to load stalls.

`<suites>...` The suites that should be executed.

`-c, --conf=<file>` The configuration file, which can be used to set execution parameters.

`--format=<format>` The format used to save the result. Available formats are XML and SQL (default).

`-h, --help` Shows a help message.

`--log-level=<level>` Sets the log level used for logging to the console. Available are DEBUG, INFO (default), WARNING and ERROR.

`-o, --out-directory=<path>` The directory where the output files will be written to.

`-P=<key-value>` Configurations that are passed through the command line. They overwrite configurations given in a file.

`-t, --tool=<tool>` The performance measurement tool that is used to run the individual benchmarks specified by a suite.

`-V, --version` Prints version information.

`lsf medium` Executes given suites and uses a performance measurement tool to sample performance counters that are related to load stalls. The output distinguishes between samples attributed to JIT compiled code and all other symbols.

`<suites>...` The suites that should be executed.

`-c, --conf=<file>` The configuration file, which can be used to set execution parameters.

`--format=<format>` The format used to save the result. Available formats are XML and SQL (default).

`-h, --help` Shows a help message.

`--log-level=<level>` Sets the log level used for logging to the console. Available are DEBUG, INFO (default), WARNING and ERROR.

`-o, --out-directory=<path>` The directory where the output files will be written to.

`-P=<key-value>` Configurations that are passed through the command line. They overwrite configurations given in a file.

`-t, --tool=<tool>` The performance measurement tool that is used to run the individual benchmarks specified by a suite.

`-V, --version` Prints version information.

`lsf fine` Executes given suites and uses a performance measurement tool to sample performance counters that are related to load stalls. Filters out samples related to JIT compiled code and merges them with further information obtained from the running JVM instance. Instructions that consume a large amount of time are—if possible—classified as frontend and/or backend stalls.

`<suites>...` The suites that should be executed.

`-a, --analyzer=<analyzer>` The analyzer, which should be used to observe and classify the benchmark load stall behavior.

`-c, --conf=<file>` The configuration file, which can be used to set execution parameters.

`--format=<format>` The format used to save the result. Available formats are XML and SQL (default).

`-h, --help` Shows a help message.

`--log-level=<level>` Sets the log level used for logging to the console. Available are DEBUG, INFO (default), WARNING and ERROR.

`-o, --out-directory=<path>` The directory where the output files will be written to.

`-P=<key-value>` Configurations that are passed through the command line. They overwrite configurations given in a file.

`-r, --runtime=<orchestrator>` The java runtime that should be used to run the individual benchmarks specified by a suite.

`-t, --tool=<tool>` The performance measurement tool that is used to run the individual benchmarks specified by a suite.

`-V, --version` Prints version information.

Appendix B

Frontend Stall Behavior Benchmark Results

This chapter shows the results obtained by running the benchmark described in Section 5.3.1.1. For a discussion of the results please refer to the aforementioned section.

B.1 Environment

The experiments were conducted on a RaspberryPi 4B with 8GB of RAM running PiOS with version 6.1.21-v8+ of the Linux kernel. The GCC compiler toolchain version 10.2.1 is used to compile the C source code to AArch64 machine code. The installed Perf tool version is 6.1.57 which was built from the kernel source obtained from `https://github.com/raspberrypi/linux`.

B.2 Results

The obtained results are listed in Table B.1 for the *l1i_cache_refill* event and in Table B.2 for the *cycles* event. The observed samples are accounted to one of three

categories. The first category is *Function Entry* and all samples collected for the first instruction of a called function are included. Samples referring to an instruction that a function returns to, are called *Link Register*. All remaining samples are categorized as *Other*.

Table B.1: Obtained Instruction Cache Misses: The measured event is *l1i_cache_refill* and the percentage of samples observed for one of three categories is presented.

(a) Compact

Run	Function Entry (%)	Link Register (%)	Other (%)	Total Events
1	24.62	7.70	67.68	32388000
2	21.38	7.28	71.34	29253000
3	20.53	6.32	73.15	40267600
4	23.28	7.67	69.05	43739200
5	27.42	6.80	65.79	33444800
AVG	23.45	7.15	69.40	35818520
STD	±2.74	±0.59	±2.91	±5979978

(b) 16 KiB Aligned

Run	Function Entry (%)	Link Register (%)	Other (%)	Total Events
1	95.53	3.35	1.12	133065400200
2	95.44	3.35	1.21	130007334200
3	95.57	3.33	1.10	124493330300
4	94.85	3.59	1.56	129935060200
5	95.42	3.34	1.24	129614074100
AVG	95.36	3.39	1.25	129423039800
STD	±0.29	±0.11	±0.19	±3090666756

Table B.2: Obtained Cycles: The measured event is *cycles* and the percentage of samples observed for one of three categories is presented.

(a) Compact

Run	Function Entry (%)	Link Register (%)	Other (%)	Total Events
1	32.98	8.51	58.51	2923910644900
2	32.97	8.26	58.77	2923609643700
3	32.33	8.37	59.30	2923895490200
4	32.98	8.18	58.84	2923631726200
5	33.02	8.41	58.57	2924548633500
AVG	32.86	8.34	58.80	2923919227700
STD	±0.29	±0.13	±0.31	±379238764

(b) 16 KiB Aligned

Run	Function Entry (%)	Link Register (%)	Other (%)	Total Events
1	73.37	8.76	17.87	10898028063700
2	73.29	8.83	17.88	10898382866700
3	73.33	8.82	17.85	10918210835300
4	73.10	8.86	18.04	10900900579100
5	73.16	8.90	17.94	10895749257000
AVG	73.25	8.83	17.92	10902254320360
STD	±0.12	±0.05	±0.08	±9105056734

Appendix C

Supported Instructions Automatic Classification

In the following the instruction mnemonics supported by the automatic classification procedure are listed. If an instruction is present in the list, at least one of its forms is supported—because most instructions have more than one way to supply their arguments, there might be an argument list missing. However, this list includes all instructions that were encountered while running the benchmarks.

addextx	addimmw	addimmx	addp2d
addsimmw	addsimmx	addw	addx
adr	andimmw	andimmx	andsimmw
andsimmx	andsw	andsx	andw
andx	asrvw	asrvx	asrw
asrx	b.al	b.cc	b.cs
b.eq	b.ge	b.gt	b.hi
b.le	b.ls	b.lt	b.mi
b.ne	b.nv	b.pl	b.vc
b.vs	bfmw	bfmx	bicw
bicx	blr	bl	br
b	cbnzw	cbnzx	cbzw
cbzx	ccmnimmw	ccmnimmx	ccmnw
ccmnx	ccmpimmw	ccmpimmx	ccmpw
ccmpx	clzw	clzx	cmnimmw
cmnimmx	cmpimmw	cmpimmx	cmpw

cmpx	cselw	cselx	cset
csincw	csincx	csnegx	dmb
eonw	eonx	eorimmw	eorimmx
eorw	eorx	extrw	extrx
fabsd	fabss	faddd	faddp2d
faddp2s	fadds	fcmpd_zero	fcmpd
fcmps_zero	fcmps	fcseld	fcsels
fcvt_dtos	fcvt_stod	fcvtzs_dtow	fcvtzs_dtox
fcvtzs_stow	fcvtzs_stox	fdivd	fdivs
fmaxp2d	fminp2d	fmov_dtox	fmov_stow
fmov_wtos	fmov_xtod	fmovd	fmovimmd
fmovimms	fmovs	fmuld	fmulelem_2d
fmulelem_4s	fmuls	fnegd	fnegs
fsqrtd	fsqrts	fsubd	fsubs
ldaddalw	ldaddalx	ldppostx	ldrbimm
ldrboff	ldrbpost	ldrbpre	ldrhimm
ldrhoff	ldrhpre	ldrimmw	ldrimmx
ldroffw	ldroffx	ldrprew	ldrprex
ldrsbimmw	ldrsbimmx	ldrsboffw	ldrsboffx
ldrshimmw	ldrshimmx	ldrshoffw	ldrshoffx
ldrswimm	ldrswoff	ldrx	ldurb
ldurh	ldursbw	ldursbx	ldurshw
ldurshx	ldurw	ldurx	ldxrw
ldxrx	lslvw	lslvx	lslw
lslx	lsrvw	lsrvx	lsrw
lsrx	maddw	maddx	movid
movkw	movkx	movnw	movnx
movw	movx	movzw	movzx
msubw	msubx	mulw	mulx
mvnw	mvnx	negw	negx
nop	ornw	ornx	orrimmw
orrimmx	orrw	orrx	prfmimm
prfmoff	rbitw	rbitx	ret
rev16w	revw	revx	rorvw
rorvx	sbfmw	sbfmx	scvtf_wtod
scvtf_wtos	scvtf_xtod	scvtf_xtos	sdivw
sdivx	smaddl	smovwb	smovwh
smovxb	smulh	stlrb	stlrh
stlrw	stlrx	stlxrw	stlxrx
strbimm	strboff	strbpost	strbpre
strhimm	strhoff	strhpost	strhpre
strimmw	strimmx	stroffw	stroffx
strpostw	strpostx	strprew	strprex
strw	strx	sturb	sturh
sturw	sturx	stxrw	stxrx

subimmw	subimmx	subsimmw	subsimmx
subsw	subsx	subw	subx
swpalw	swpalx	sxtbw	sxtbx
sxthw	sxthx	sxtwx	tbz
tstimmw	tstimmx	tstw	tstx
ubfizw	ubfizx	ubfmw	ubfmx
ubfxw	ubfxx	udivw	udivx
umovwb	umovwh	umovws	umovxd
uxtbx	uxthx	vabs16b	vabs2d
vabs4s	vabs8h	vadd16b	vadd2d
vadd4s	vadd8h	vaddv16b	vaddv4s
vaddv8b	vaddv8h	vand16b	vbic16b
vbicimm4s	vbif16b	vbit16b	vbsl16b
vcmeq16b_zero	vcmeq16b	vcmeq2d_zero	vcmeq2d
vcmeq4s_zero	vcmeq4s	vcmeq8h_zero	vcmeq8h
vcmge16b_zero	vcmge16b	vcmge2d_zero	vcmge2d
vcmge4s_zero	vcmge4s	vcmge8h_zero	vcmge8h
vcmgt16b_zero	vcmgt16b	vcmgt2d_zero	vcmgt2d
vcmgt4s_zero	vcmgt4s	vcmgt8h_zero	vcmgt8h
vcmle16b_zero	vcmle2d_zero	vcmle4s_zero	vcmle8h_zero
vcmlt16b_zero	vcmlt2d_zero	vcmlt4s_zero	vcmlt8h_zero
vcmtst16b	vcmtst4s	vcmtst8h	vcnt8b
vdup16b	vdup2d	vdup4s	vdup8h
vdupe16b	vdupe2d	vdupe4s	veor16b
vext16b	vfabs2d	vfabs4s	vfadd2d
vfadd4s	vfaddp4s	vfcmeq2d_zero	vfcmeq2d
vfcmeq4s_zero	vfcmeq4s	vfcmge2d_zero	vfcmge2d
vfcmge4s_zero	vfcmge4s	vfcmgt2d_zero	vfcmgt2d
vfcmgt4s_zero	vfcmgt4s	vfcmle2d_zero	vfcmle4s_zero
vfcmlt2d_zero	vfcmlt4s_zero	vfdiv2d	vfdiv4s
vfmax2d	vfmax4s	vfmaxv4s	vfmin2d
vfmin4s	vfminv4s	vfmla2d	vfmla4s
vfmov2d	vfmov4s	vfmul2d	vfmul4s
vfmulelem_2d	vfmulelem_4s	vfneg2d	vfneg4s
vfsqrt2d	vfsqrt4s	vfsub2d	vfsub4s
vgnop	vinseb	vinsed	vinswb
vinswh	vinsws	vinsxd	vldrimmb
vldrimmd	vldrimmh	vldrimmq	vldrimms
vldroffb	vldroffd	vldroffh	vldroffq
vldroffs	vldrpostq	vldrpreq	vldurb
vldurd	vldurh	vldurq	vldurs
vmovi16b	vmovi2d	vmovi2s	vmovi4s_one
vmovi4s	vmovi8h	vmul16b	vmul4s
vmul8h	vmvni4s_one	vmvni4s	vmvni8h
vneg16b	vneg2d	vneg4s	vneg8h

vnot16b	vorr16b	vorrimm4s	vrev64_4s
vshl16b	vshl2d	vshrn_2s	vshrn_4h
vshrn_8b	vsli16b	vsli2d	vsli4s
vsli8h	vsmax16b	vsmax4s	vsmax8h
vsmaxv16b	vsmaxv4s	vsmaxv8h	vsmin16b
vsmin4s	vsmin8h	vsminv16b	vsminv4s
vsminv8h	vsri2d	vsshr2d	vstpoffq
vstppreq	vstrimmb	vstrimmd	vstrimmh
vstrimmq	vstrimms	vstroffb	vstroffd
vstroffh	vstroffq	vstroffs	vstrpostd
vstrpostq	vstrposts	vstrpreq	vsturb
vsturd	vsturh	vsturq	vsturs
vsub16b	vsub2d	vsub4s	vsub8h
vumlal_2d	vushll2_2d	vushll2_4s	vushll2_8h
vushll_2d	vushll_4s	vushll_8h	vushr16b
vushr2d	vushr4s	vushr8h	vuzp1_4s

Appendix D

Coarse Granularity Benchmark Results

In this appendix the data obtained with coarse granularity is given. First the frequency data for all benchmarks is given, then the cache miss ratios are presented.

D.1 Frequency

The average as well as minimum and maximum frequencies of the benchmark runs are given in Table D.1.

Table D.1: Frequencies

			Frequency (GHz)		
Suite	Benchmark	Configuration	Min	Avg	Max
AcmeAir	–	*B:BF*	1.77	1.77	1.77
AcmeAir	–	*B:dBF*	1.77	1.77	1.77
AcmeAir	–	*G:BF*	1.77	1.77	1.77
AcmeAir	–	*G:dBF*	1.77	1.77	1.77
AcmeAir	–	*G:H*	1.77	1.77	1.77
AcmeAir	–	*PAUSE*	1.77	1.77	1.77
AcmeAir	–	*THRU*	1.77	1.77	1.77
Daytrader7	–	*B:BF*	1.78	1.78	1.78
Daytrader7	–	*B:dBF*	1.78	1.78	1.78

Table D.1: (Continued)

Suite	Benchmark	Configuration	Frequency (GHz) Min	Avg	Max
Daytrader7	–	*G:BF*	1.78	1.78	1.78
Daytrader7	–	*G:dBF*	1.78	1.78	1.78
Daytrader7	–	*G:H*	1.78	1.78	1.78
Daytrader7	–	*PAUSE*	1.78	1.78	1.78
Daytrader7	–	*THRU*	1.78	1.78	1.78
HiBench	*als*	*B:BF*	1.78	1.78	1.78
HiBench	*als*	*B:dBF*	1.78	1.78	1.78
HiBench	*als*	*G:BF*	1.78	1.78	1.78
HiBench	*als*	*G:dBF*	1.78	1.78	1.78
HiBench	*als*	*G:H*	1.78	1.78	1.78
HiBench	*als*	*PAUSE*	1.78	1.78	1.78
HiBench	*als*	*THRU*	1.78	1.78	1.78
HiBench	*bayes*	*B:BF*	1.78	1.78	1.78
HiBench	*bayes*	*B:dBF*	1.78	1.78	1.78
HiBench	*bayes*	*G:BF*	1.78	1.78	1.78
HiBench	*bayes*	*G:dBF*	1.78	1.78	1.78
HiBench	*bayes*	*G:H*	1.78	1.78	1.78
HiBench	*bayes*	*PAUSE*	1.78	1.78	1.78
HiBench	*bayes*	*THRU*	1.78	1.78	1.78
HiBench	*gbt*	*B:BF*	1.78	1.78	1.78
HiBench	*gbt*	*B:dBF*	1.78	1.78	1.78
HiBench	*gbt*	*G:BF*	1.78	1.78	1.78
HiBench	*gbt*	*G:dBF*	1.78	1.78	1.78
HiBench	*gbt*	*G:H*	1.78	1.78	1.78
HiBench	*gbt*	*PAUSE*	1.78	1.78	1.78
HiBench	*gbt*	*THRU*	1.78	1.78	1.78
HiBench	*kmeans*	*B:BF*	1.78	1.78	1.78
HiBench	*kmeans*	*B:dBF*	1.78	1.78	1.78
HiBench	*kmeans*	*G:BF*	1.78	1.78	1.78
HiBench	*kmeans*	*G:dBF*	1.78	1.78	1.78
HiBench	*kmeans*	*G:H*	1.78	1.78	1.78
HiBench	*kmeans*	*PAUSE*	1.78	1.78	1.78
HiBench	*kmeans*	*THRU*	1.78	1.78	1.78
HiBench	*lda*	*B:BF*	1.78	1.78	1.78
HiBench	*lda*	*B:dBF*	1.78	1.78	1.79
HiBench	*lda*	*G:BF*	1.78	1.78	1.78
HiBench	*lda*	*G:dBF*	1.78	1.78	1.78
HiBench	*lda*	*G:H*	1.78	1.78	1.78
HiBench	*lda*	*PAUSE*	1.78	1.78	1.78
HiBench	*lda*	*THRU*	1.78	1.78	1.78
HiBench	*linear*	*B:BF*	1.77	1.77	1.77

Table D.1: (Continued)

Suite	Benchmark	Configuration	Frequency (GHz)		
			Min	Avg	Max
HiBench	*linear*	*B:dBF*	1.77	1.77	1.77
HiBench	*linear*	*G:BF*	1.77	1.77	1.78
HiBench	*linear*	*G:dBF*	1.77	1.77	1.77
HiBench	*linear*	*G:H*	1.77	1.77	1.77
HiBench	*linear*	*PAUSE*	1.77	1.77	1.77
HiBench	*linear*	*THRU*	1.77	1.77	1.77
HiBench	*lr*	*B:BF*	1.77	1.78	1.78
HiBench	*lr*	*B:dBF*	1.77	1.77	1.78
HiBench	*lr*	*G:BF*	1.78	1.78	1.78
HiBench	*lr*	*G:dBF*	1.78	1.78	1.78
HiBench	*lr*	*G:H*	1.78	1.78	1.78
HiBench	*lr*	*PAUSE*	1.78	1.78	1.78
HiBench	*lr*	*THRU*	1.78	1.78	1.78
LoadStallSuite	*dm*	*B:BF*	1.79	1.79	1.79
LoadStallSuite	*dm*	*B:dBF*	1.79	1.79	1.79
LoadStallSuite	*dm*	*G:BF*	1.79	1.79	1.79
LoadStallSuite	*dm*	*G:dBF*	1.79	1.79	1.79
LoadStallSuite	*dm*	*G:H*	1.79	1.79	1.79
LoadStallSuite	*dm*	*PAUSE*	1.79	1.79	1.79
LoadStallSuite	*dm*	*THRU*	1.79	1.79	1.79
LoadStallSuite	*dwm*	*B:BF*	1.79	1.79	1.79
LoadStallSuite	*dwm*	*B:dBF*	1.79	1.79	1.79
LoadStallSuite	*dwm*	*G:BF*	1.78	1.79	1.79
LoadStallSuite	*dwm*	*G:dBF*	1.78	1.79	1.79
LoadStallSuite	*dwm*	*G:H*	1.78	1.79	1.79
LoadStallSuite	*dwm*	*PAUSE*	1.79	1.79	1.79
LoadStallSuite	*dwm*	*THRU*	1.79	1.79	1.79
LoadStallSuite	*ic*	*B:BF*	1.79	1.79	1.79
LoadStallSuite	*ic*	*B:dBF*	1.79	1.79	1.79
LoadStallSuite	*ic*	*G:BF*	1.79	1.79	1.79
LoadStallSuite	*ic*	*G:dBF*	1.79	1.79	1.79
LoadStallSuite	*ic*	*G:H*	1.79	1.79	1.79
LoadStallSuite	*ic*	*PAUSE*	1.79	1.79	1.79
LoadStallSuite	*ic*	*THRU*	1.79	1.79	1.79
LoadStallSuite	*ol*	*B:BF*	1.79	1.79	1.79
LoadStallSuite	*ol*	*B:dBF*	1.79	1.79	1.79
LoadStallSuite	*ol*	*G:BF*	1.79	1.79	1.79
LoadStallSuite	*ol*	*G:dBF*	1.79	1.79	1.79
LoadStallSuite	*ol*	*G:H*	1.79	1.79	1.79
LoadStallSuite	*ol*	*PAUSE*	1.79	1.79	1.79
LoadStallSuite	*ol*	*THRU*	1.79	1.79	1.79

Table D.1: (Continued)

Suite	Benchmark	Configuration	Frequency (GHz) Min	Avg	Max
LoadStallSuite	*sm*	*B:BF*	1.71	1.75	1.78
LoadStallSuite	*sm*	*B:dBF*	1.71	1.75	1.77
LoadStallSuite	*sm*	*G:BF*	1.72	1.75	1.78
LoadStallSuite	*sm*	*G:dBF*	1.67	1.73	1.76
LoadStallSuite	*sm*	*G:H*	1.71	1.75	1.77
LoadStallSuite	*sm*	*PAUSE*	1.71	1.75	1.77
LoadStallSuite	*sm*	*THRU*	1.73	1.75	1.77
LoadStallSuite	*ts*	*B:BF*	1.79	1.79	1.79
LoadStallSuite	*ts*	*B:dBF*	1.79	1.79	1.79
LoadStallSuite	*ts*	*G:BF*	1.79	1.79	1.79
LoadStallSuite	*ts*	*G:dBF*	1.79	1.79	1.79
LoadStallSuite	*ts*	*G:H*	1.79	1.79	1.79
LoadStallSuite	*ts*	*PAUSE*	1.79	1.79	1.79
LoadStallSuite	*ts*	*THRU*	1.79	1.79	1.79
Renaissance	*akka-uct*	*B:BF*	1.78	1.78	1.78
Renaissance	*akka-uct*	*B:dBF*	1.78	1.78	1.78
Renaissance	*akka-uct*	*G:BF*	1.78	1.78	1.78
Renaissance	*akka-uct*	*G:dBF*	1.78	1.78	1.78
Renaissance	*akka-uct*	*G:H*	1.78	1.78	1.78
Renaissance	*akka-uct*	*PAUSE*	1.78	1.78	1.78
Renaissance	*akka-uct*	*THRU*	1.78	1.78	1.78
Renaissance	*finagle-http*	*B:BF*	1.78	1.78	1.78
Renaissance	*finagle-http*	*B:dBF*	1.78	1.78	1.78
Renaissance	*finagle-http*	*G:BF*	1.78	1.78	1.78
Renaissance	*finagle-http*	*G:dBF*	1.78	1.78	1.78
Renaissance	*finagle-http*	*G:H*	1.78	1.78	1.78
Renaissance	*finagle-http*	*PAUSE*	1.78	1.78	1.78
Renaissance	*finagle-http*	*THRU*	1.78	1.78	1.78
Renaissance	*fj-kmeans*	*B:BF*	1.78	1.78	1.78
Renaissance	*fj-kmeans*	*B:dBF*	1.77	1.78	1.78
Renaissance	*fj-kmeans*	*G:BF*	1.78	1.78	1.78
Renaissance	*fj-kmeans*	*G:dBF*	1.78	1.78	1.78
Renaissance	*fj-kmeans*	*G:H*	1.77	1.78	1.78
Renaissance	*fj-kmeans*	*PAUSE*	1.78	1.78	1.78
Renaissance	*fj-kmeans*	*THRU*	1.78	1.78	1.78
Renaissance	*future-genetic*	*B:BF*	1.77	1.77	1.77
Renaissance	*future-genetic*	*B:dBF*	1.77	1.77	1.77
Renaissance	*future-genetic*	*G:BF*	1.76	1.77	1.77
Renaissance	*future-genetic*	*G:dBF*	1.76	1.77	1.77
Renaissance	*future-genetic*	*G:H*	1.77	1.77	1.77
Renaissance	*future-genetic*	*PAUSE*	1.77	1.77	1.77

Table D.1: (Continued)

			Frequency (GHz)		
Suite	Benchmark	Configuration	Min	Avg	Max
Renaissance	*future-genetic*	*THRU*	1.77	1.77	1.77
Renaissance	*mnemonics*	*B:BF*	1.79	1.79	1.79
Renaissance	*mnemonics*	*B:dBF*	1.79	1.79	1.79
Renaissance	*mnemonics*	*G:BF*	1.78	1.79	1.79
Renaissance	*mnemonics*	*G:dBF*	1.79	1.79	1.79
Renaissance	*mnemonics*	*G:H*	1.79	1.79	1.79
Renaissance	*mnemonics*	*PAUSE*	1.79	1.79	1.79
Renaissance	*mnemonics*	*THRU*	1.78	1.79	1.79
Renaissance	*rx-scrabble*	*B:BF*	1.78	1.78	1.78
Renaissance	*rx-scrabble*	*B:dBF*	1.78	1.78	1.78
Renaissance	*rx-scrabble*	*G:BF*	1.78	1.78	1.78
Renaissance	*rx-scrabble*	*G:dBF*	1.78	1.78	1.78
Renaissance	*rx-scrabble*	*G:H*	1.78	1.78	1.78
Renaissance	*rx-scrabble*	*PAUSE*	1.78	1.78	1.79
Renaissance	*rx-scrabble*	*THRU*	1.78	1.79	1.79
SPECjvm2008	*compress*	*B:BF*	1.79	1.79	1.79
SPECjvm2008	*compress*	*B:dBF*	1.79	1.79	1.79
SPECjvm2008	*compress*	*G:BF*	1.79	1.79	1.79
SPECjvm2008	*compress*	*G:dBF*	1.79	1.79	1.79
SPECjvm2008	*compress*	*G:H*	1.79	1.79	1.79
SPECjvm2008	*compress*	*PAUSE*	1.79	1.79	1.79
SPECjvm2008	*compress*	*THRU*	1.79	1.79	1.79
SPECjvm2008	*crypto.aes*	*B:BF*	1.79	1.79	1.79
SPECjvm2008	*crypto.aes*	*B:dBF*	1.79	1.79	1.79
SPECjvm2008	*crypto.aes*	*G:BF*	1.79	1.79	1.79
SPECjvm2008	*crypto.aes*	*G:dBF*	1.79	1.79	1.79
SPECjvm2008	*crypto.aes*	*G:H*	1.79	1.79	1.79
SPECjvm2008	*crypto.aes*	*PAUSE*	1.79	1.79	1.79
SPECjvm2008	*crypto.aes*	*THRU*	1.79	1.79	1.79
SPECjvm2008	*derby*	*B:BF*	1.78	1.78	1.78
SPECjvm2008	*derby*	*B:dBF*	1.78	1.78	1.78
SPECjvm2008	*derby*	*G:BF*	1.78	1.78	1.78
SPECjvm2008	*derby*	*G:dBF*	1.78	1.78	1.78
SPECjvm2008	*derby*	*G:H*	1.78	1.78	1.78
SPECjvm2008	*derby*	*PAUSE*	1.79	1.79	1.79
SPECjvm2008	*derby*	*THRU*	1.79	1.79	1.79

Table D.2: Load Stall Benchmark *sm* Frequency (Revisited)

	Frequency (GHz)		
Configuration	Min	Avg	Max
B:BF	1.78	1.79	1.79
B:dBF	1.78	1.79	1.79
G:BF	1.78	1.78	1.79
G:dBF	1.78	1.78	1.79
G:H	1.78	1.79	1.79
PAUSE	1.78	1.78	1.79
THRU	1.78	1.79	1.79
Overall	1.78	1.79	1.79

D.1.1 Load Stall Benchmark *sm*

In this section the frequency results for the load stall benchmark *sm* are presented in Table D.2. Instead of the default ten iterations with matrix $A \in \mathbb{R}^{10\times3}$ and $B \in \mathbb{R}^{3\times10}$, 200 iterations with $A \in \mathbb{R}^{200\times50}$ and $B \in \mathbb{R}^{50\times300}$ are executed internally to increase the runtime. Each run is repeated ten times while counting the same events that were previously collected.

D.2 Data

In this section the obtained load stall performance data is presented. Table D.3 shows the obtained CPU times and IPC values. It is directly followed by Table D.4 with the according cache miss ratios. For each value the minimum, average and maximum from the ten observed runs is presented.

Table D.3: CPU Time and IPCs

			CPU Time (sec)			IPC		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max
AcmeAir	–	*B:BF*	17305.51	17504.36	17642.10	0.42	0.46	0.49
AcmeAir	–	*B:dBF*	10168.20	13082.92	17538.87	0.29	0.36	0.48
AcmeAir	–	*G:BF*	9667.30	9739.77	9846.74	0.26	0.27	0.27
AcmeAir	–	*G:dBF*	9959.91	10106.19	10420.49	0.28	0.30	0.35
AcmeAir	–	*G:H*	10518.54	14611.96	17023.45	0.32	0.40	0.47
AcmeAir	–	*PAUSE*	10457.85	10763.49	10938.39	0.30	0.31	0.32
AcmeAir	–	*THRU*	10291.21	10358.66	10434.34	0.30	0.32	0.33
Daytrader7	–	*B:BF*	24133.87	24300.69	24457.54	0.30	0.31	0.31
Daytrader7	–	*B:dBF*	23893.86	24172.66	24465.12	0.31	0.31	0.32
Daytrader7	–	*G:BF*	23553.29	23751.97	24010.24	0.26	0.27	0.27
Daytrader7	–	*G:dBF*	23466.37	23704.85	24096.21	0.28	0.29	0.29
Daytrader7	–	*G:H*	23749.28	23837.02	23900.32	0.29	0.29	0.29
Daytrader7	–	*PAUSE*	24140.98	24228.04	24376.33	0.30	0.31	0.32
Daytrader7	–	*THRU*	23997.59	24132.61	24293.10	0.31	0.32	0.32
HiBench	*als*	*B:BF*	1007.89	1040.16	1083.79	0.97	0.99	1.01
HiBench	*als*	*B:dBF*	996.27	1033.21	1062.25	0.96	0.99	1.01
HiBench	*als*	*G:BF*	869.33	896.20	936.40	0.88	0.90	0.91
HiBench	*als*	*G:dBF*	886.55	911.83	960.16	0.86	0.88	0.90
HiBench	*als*	*G:H*	867.61	908.91	955.92	0.87	0.90	0.92
HiBench	*als*	*PAUSE*	878.92	914.96	951.03	0.85	0.87	0.89
HiBench	*als*	*THRU*	863.97	898.36	925.55	0.85	0.88	0.91
HiBench	*bayes*	*B:BF*	896.96	932.85	971.61	0.56	0.57	0.59
HiBench	*bayes*	*B:dBF*	860.44	930.96	1012.05	0.58	0.60	0.66
HiBench	*bayes*	*G:BF*	717.36	764.44	806.84	0.58	0.60	0.62
HiBench	*bayes*	*G:dBF*	734.84	778.87	811.52	0.56	0.59	0.60

Table D.3: (Continued)

Suite	Benchmark	Configuration	CPU Time (sec)			IPC		
			Min	Avg	Max	Min	Avg	Max
HiBench	*bayes*	*G:H*	720.67	772.17	797.50	0.57	0.59	0.61
HiBench	*bayes*	*PAUSE*	844.37	870.85	902.33	0.56	0.57	0.58
HiBench	*bayes*	*THRU*	805.61	846.85	877.11	0.55	0.56	0.57
HiBench	*gbt*	*B:BF*	2179.31	2329.63	2505.78	0.78	0.79	0.81
HiBench	*gbt*	*B:dBF*	2135.64	2257.75	2361.07	0.77	0.79	0.81
HiBench	*gbt*	*G:BF*	1698.92	1779.60	1852.35	0.70	0.74	0.75
HiBench	*gbt*	*G:dBF*	1783.49	1848.85	1978.72	0.72	0.73	0.74
HiBench	*gbt*	*G:H*	1681.94	1807.23	1909.11	0.71	0.72	0.74
HiBench	*gbt*	*PAUSE*	1923.37	2047.58	2152.01	0.71	0.73	0.74
HiBench	*gbt*	*THRU*	1909.45	1991.48	2141.79	0.71	0.73	0.74
HiBench	*kmeans*	*B:BF*	7774.62	7966.08	8198.11	0.99	1.02	1.05
HiBench	*kmeans*	*B:dBF*	7525.72	8132.84	8420.77	1.00	1.02	1.04
HiBench	*kmeans*	*G:BF*	6621.93	6923.41	7355.38	0.94	0.97	1.01
HiBench	*kmeans*	*G:dBF*	6479.45	7009.21	7331.62	0.93	0.99	1.03
HiBench	*kmeans*	*G:H*	6501.24	6934.87	7443.93	0.94	0.98	1.02
HiBench	*kmeans*	*PAUSE*	10599.04	11353.16	11857.36	0.70	0.73	0.78
HiBench	*kmeans*	*THRU*	9457.99	10065.65	10607.42	0.74	0.77	0.81
HiBench	*lda*	*B:BF*	7495.50	12639.67	15855.98	1.20	1.32	1.39
HiBench	*lda*	*B:dBF*	10884.78	14731.41	17590.52	1.19	1.42	1.91
HiBench	*lda*	*G:BF*	10162.08	12003.10	14547.67	0.58	0.68	0.70
HiBench	*lda*	*G:dBF*	10260.28	12575.15	14414.23	0.57	0.67	0.70
HiBench	*lda*	*G:H*	9789.11	12775.35	15831.44	0.57	0.66	0.70
HiBench	*lda*	*PAUSE*	8991.18	12933.42	18128.51	0.58	0.65	0.69
HiBench	*lda*	*THRU*	11411.62	14016.85	17090.02	0.47	0.65	0.69
HiBench	*linear*	*B:BF*	5488.81	5700.73	5832.14	1.00	1.02	1.05
HiBench	*linear*	*B:dBF*	6305.58	6453.50	6601.71	0.87	0.88	0.90

Table D.3: (Continued)

Suite	Benchmark	Configuration	CPU Time (sec)			IPC		
			Min	Avg	Max	Min	Avg	Max
HiBench	*linear*	*G:BF*	5447.08	5774.78	6007.01	0.87	0.90	0.96
HiBench	*linear*	*G:dBF*	5955.42	6103.61	6319.51	0.84	0.86	0.89
HiBench	*linear*	*G:H*	5810.32	5999.70	6357.84	0.84	0.86	0.90
HiBench	*linear*	*PAUSE*	5371.64	5609.34	5899.49	1.00	1.03	1.07
HiBench	*linear*	*THRU*	6207.26	6454.42	6874.02	0.83	0.86	0.91
HiBench	*lr*	*B:BF*	220.39	248.25	311.71	0.67	0.71	0.79
HiBench	*lr*	*B:dBF*	224.05	237.27	268.85	0.63	0.70	0.82
HiBench	*lr*	*G:BF*	220.17	229.66	236.57	0.61	0.82	1.02
HiBench	*lr*	*G:dBF*	216.86	230.18	238.92	0.80	0.85	0.89
HiBench	*lr*	*G:H*	222.52	231.00	243.55	0.58	0.84	1.06
HiBench	*lr*	*PAUSE*	220.41	232.13	260.82	0.78	0.86	1.02
HiBench	*lr*	*THRU*	223.24	229.08	238.46	0.81	0.85	0.90
LoadStallSuite	*dm*	*B:BF*	479.10	480.35	482.56	2.08	2.08	2.09
LoadStallSuite	*dm*	*B:dBF*	479.16	480.07	481.47	2.08	2.09	2.09
LoadStallSuite	*dm*	*G:BF*	275.17	281.53	287.82	1.23	1.27	1.30
LoadStallSuite	*dm*	*G:dBF*	276.41	282.97	290.61	1.23	1.26	1.29
LoadStallSuite	*dm*	*G:H*	269.41	282.81	290.66	1.22	1.26	1.31
LoadStallSuite	*dm*	*PAUSE*	268.71	269.90	271.88	1.31	1.32	1.33
LoadStallSuite	*dm*	*THRU*	269.82	270.78	272.48	1.31	1.31	1.32
LoadStallSuite	*dwm*	*B:BF*	1040.52	1067.55	1082.15	1.69	1.73	1.80
LoadStallSuite	*dwm*	*B:dBF*	1064.21	1081.07	1096.42	1.67	1.69	1.72
LoadStallSuite	*dwm*	*G:BF*	784.40	967.21	1331.30	1.16	1.21	1.26
LoadStallSuite	*dwm*	*G:dBF*	816.22	991.63	1354.09	1.17	1.22	1.25
LoadStallSuite	*dwm*	*G:H*	860.11	1008.70	1380.45	1.20	1.22	1.24
LoadStallSuite	*dwm*	*PAUSE*	665.15	704.06	727.79	1.16	1.19	1.26
LoadStallSuite	*dwm*	*THRU*	678.89	697.55	727.66	1.16	1.22	1.33

Table D.3: (Continued)

Suite	Benchmark	Configuration	CPU Time (sec)			IPC		
			Min	Avg	Max	Min	Avg	Max
LoadStallSuite	*ic*	*B:BF*	1688.24	1750.63	1792.61	0.78	0.80	0.82
LoadStallSuite	*ic*	*B:dBF*	1699.00	1735.12	1775.36	0.79	0.80	0.81
LoadStallSuite	*ic*	*G:BF*	1159.72	1219.52	1285.76	0.43	0.44	0.47
LoadStallSuite	*ic*	*G:dBF*	1212.00	1238.85	1322.99	0.41	0.44	0.45
LoadStallSuite	*ic*	*G:H*	1192.18	1237.91	1261.23	0.43	0.44	0.45
LoadStallSuite	*ic*	*PAUSE*	1190.57	1229.29	1300.20	0.42	0.44	0.46
LoadStallSuite	*ic*	*THRU*	1188.76	1229.89	1266.56	0.43	0.44	0.45
LoadStallSuite	*ol*	*B:BF*	7877.42	8161.19	8889.85	0.88	0.96	0.99
LoadStallSuite	*ol*	*B:dBF*	7887.54	8006.08	8207.34	0.96	0.98	0.99
LoadStallSuite	*ol*	*G:BF*	4947.70	5098.06	5194.54	0.35	0.36	0.37
LoadStallSuite	*ol*	*G:dBF*	4691.52	4775.29	4892.63	0.37	0.38	0.39
LoadStallSuite	*ol*	*G:H*	4482.47	4607.94	4709.55	0.40	0.40	0.42
LoadStallSuite	*ol*	*PAUSE*	3617.27	3693.47	3718.02	0.40	0.41	0.42
LoadStallSuite	*ol*	*THRU*	3287.02	3316.10	3387.16	0.39	0.40	0.40
LoadStallSuite	*sm*	*B:BF*	0.92	1.01	1.19	0.62	0.65	0.67
LoadStallSuite	*sm*	*B:dBF*	0.90	0.98	1.12	0.64	0.66	0.67
LoadStallSuite	*sm*	*G:BF*	0.95	1.04	1.16	0.65	0.66	0.69
LoadStallSuite	*sm*	*G:dBF*	0.93	1.03	1.22	0.64	0.67	0.69
LoadStallSuite	*sm*	*G:H*	0.96	1.06	1.26	0.65	0.66	0.67
LoadStallSuite	*sm*	*PAUSE*	0.88	0.94	1.15	0.64	0.66	0.67
LoadStallSuite	*sm*	*THRU*	0.89	0.93	0.99	0.65	0.67	0.72
LoadStallSuite	*ts*	*B:BF*	7110.01	7424.76	7691.98	0.24	0.24	0.25
LoadStallSuite	*ts*	*B:dBF*	12095.89	12450.03	12717.08	0.14	0.15	0.15
LoadStallSuite	*ts*	*G:BF*	5943.75	6108.27	6306.15	0.29	0.30	0.30
LoadStallSuite	*ts*	*G:dBF*	5946.81	6279.88	6544.01	0.28	0.29	0.30
LoadStallSuite	*ts*	*G:H*	6957.16	7251.62	7662.70	0.24	0.25	0.26

Table D.3: (Continued)

Suite	Benchmark	Configuration	CPU Time (sec)			IPC		
			Min	Avg	Max	Min	Avg	Max
LoadStallSuite	*ts*	*PAUSE*	6801.99	7003.54	7300.75	0.25	0.26	0.27
LoadStallSuite	*ts*	*THRU*	7857.60	7893.37	7954.46	0.23	0.23	0.23
Renaissance	*akka-uct*	*B:BF*	8152.36	8349.98	8485.57	0.77	0.78	0.80
Renaissance	*akka-uct*	*B:dBF*	7775.07	7951.27	8046.65	0.80	0.81	0.82
Renaissance	*akka-uct*	*G:BF*	5797.15	5956.80	6048.92	0.61	0.62	0.63
Renaissance	*akka-uct*	*G:dBF*	5334.21	5499.06	5630.02	0.64	0.65	0.66
Renaissance	*akka-uct*	*G:H*	5304.04	5468.36	5531.01	0.67	0.68	0.69
Renaissance	*akka-uct*	*PAUSE*	5072.82	5162.68	5345.97	0.55	0.56	0.56
Renaissance	*akka-uct*	*THRU*	4665.33	4754.84	4819.44	0.57	0.57	0.58
Renaissance	*finagle-http*	*B:BF*	1993.89	2014.80	2033.50	0.27	0.27	0.27
Renaissance	*finagle-http*	*B:dBF*	1984.62	2002.09	2025.00	0.27	0.27	0.27
Renaissance	*finagle-http*	*G:BF*	1713.32	1737.68	1762.60	0.23	0.23	0.24
Renaissance	*finagle-http*	*G:dBF*	1705.14	1738.59	1775.51	0.23	0.23	0.24
Renaissance	*finagle-http*	*G:H*	1718.03	1734.63	1764.88	0.23	0.23	0.24
Renaissance	*finagle-http*	*PAUSE*	1919.86	1957.67	2061.57	0.24	0.25	0.26
Renaissance	*finagle-http*	*THRU*	1741.17	1776.70	1869.92	0.25	0.25	0.26
Renaissance	*fj-kmeans*	*B:BF*	2833.27	2899.11	2949.21	0.92	0.96	0.99
Renaissance	*fj-kmeans*	*B:dBF*	2875.17	2917.33	2964.51	0.94	0.98	0.99
Renaissance	*fj-kmeans*	*G:BF*	2684.98	2717.28	2760.75	0.80	0.83	0.87
Renaissance	*fj-kmeans*	*G:dBF*	2663.15	2720.72	2797.64	0.81	0.84	0.87
Renaissance	*fj-kmeans*	*G:H*	2496.21	2547.00	2583.18	0.86	0.89	0.92
Renaissance	*fj-kmeans*	*PAUSE*	2806.37	2857.32	2986.71	0.96	0.98	1.01
Renaissance	*fj-kmeans*	*THRU*	2859.58	2907.28	2937.45	0.96	1.00	1.02
Renaissance	*future-genetic*	*B:BF*	1357.15	1389.02	1452.68	0.94	0.97	1.00
Renaissance	*future-genetic*	*B:dBF*	1315.37	1372.48	1416.32	0.90	0.97	1.01
Renaissance	*future-genetic*	*G:BF*	921.19	953.47	1045.37	0.81	0.90	0.97

Table D.3: (Continued)

			CPU Time (sec)			IPC		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max
Renaissance	*future-genetic*	*G:dBF*	924.41	967.43	995.22	0.83	0.94	1.00
Renaissance	*future-genetic*	*G:H*	948.49	980.82	1021.00	0.92	0.95	0.98
Renaissance	*future-genetic*	*PAUSE*	1183.95	1236.13	1283.51	0.80	0.88	0.93
Renaissance	*future-genetic*	*THRU*	1084.33	1135.35	1186.98	0.81	0.89	0.95
Renaissance	*mnemonics*	*B:BF*	639.50	653.76	678.69	0.81	0.84	0.87
Renaissance	*mnemonics*	*B:dBF*	637.86	650.13	666.70	0.83	0.85	0.88
Renaissance	*mnemonics*	*G:BF*	537.62	580.82	610.79	0.70	0.72	0.74
Renaissance	*mnemonics*	*G:dBF*	525.79	560.91	603.49	0.68	0.72	0.74
Renaissance	*mnemonics*	*G:H*	539.85	576.14	607.40	0.71	0.73	0.77
Renaissance	*mnemonics*	*PAUSE*	646.83	704.72	736.08	0.74	0.76	0.79
Renaissance	*mnemonics*	*THRU*	627.61	672.22	742.41	0.66	0.69	0.70
Renaissance	*rx-scrabble*	*B:BF*	268.28	273.71	280.41	0.74	0.75	0.76
Renaissance	*rx-scrabble*	*B:dBF*	265.16	271.55	279.80	0.74	0.74	0.76
Renaissance	*rx-scrabble*	*G:BF*	210.95	220.92	233.93	0.64	0.65	0.66
Renaissance	*rx-scrabble*	*G:dBF*	212.89	223.88	237.58	0.64	0.65	0.66
Renaissance	*rx-scrabble*	*G:H*	214.51	227.25	237.30	0.64	0.65	0.66
Renaissance	*rx-scrabble*	*PAUSE*	265.33	280.31	287.82	0.63	0.64	0.64
Renaissance	*rx-scrabble*	*THRU*	290.44	304.92	330.77	0.58	0.59	0.60
SPECjvm2008	*compress*	*B:BF*	1620.94	1637.67	1655.52	0.59	0.63	0.65
SPECjvm2008	*compress*	*B:dBF*	1624.01	1633.66	1640.48	0.59	0.63	0.65
SPECjvm2008	*compress*	*G:BF*	1581.32	1594.51	1606.60	0.51	0.56	0.58
SPECjvm2008	*compress*	*G:dBF*	1586.12	1592.96	1599.86	0.52	0.56	0.58
SPECjvm2008	*compress*	*G:H*	1572.96	1591.00	1603.83	0.56	0.56	0.57
SPECjvm2008	*compress*	*PAUSE*	1582.39	1594.46	1607.59	0.51	0.55	0.57
SPECjvm2008	*compress*	*THRU*	1579.75	1589.90	1599.26	0.52	0.56	0.57
SPECjvm2008	*crypto.aes*	*B:BF*	1619.55	1633.64	1654.68	2.04	2.12	2.20

Table D.3: (Continued)

			CPU Time (sec)			IPC		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max
SPECjvm2008	*crypto.aes*	*B:dBF*	1611.58	1635.48	1664.45	1.96	2.14	2.21
SPECjvm2008	*crypto.aes*	*G:BF*	1598.85	1610.26	1618.84	2.17	2.21	2.24
SPECjvm2008	*crypto.aes*	*G:dBF*	1581.07	1605.68	1625.23	2.16	2.21	2.24
SPECjvm2008	*crypto.aes*	*G:H*	1592.87	1610.76	1638.65	1.81	2.17	2.23
SPECjvm2008	*crypto.aes*	*PAUSE*	1565.50	1583.46	1599.69	1.89	2.13	2.20
SPECjvm2008	*crypto.aes*	*THRU*	1544.45	1576.46	1590.74	1.71	2.12	2.20
SPECjvm2008	*derby*	*B:BF*	1677.42	1684.76	1699.95	0.72	0.73	0.74
SPECjvm2008	*derby*	*B:dBF*	1678.74	1684.43	1692.85	0.71	0.73	0.74
SPECjvm2008	*derby*	*G:BF*	1638.69	1647.97	1658.15	0.64	0.65	0.67
SPECjvm2008	*derby*	*G:dBF*	1635.93	1647.17	1656.72	0.68	0.69	0.71
SPECjvm2008	*derby*	*G:H*	1638.71	1648.21	1654.40	0.67	0.69	0.71
SPECjvm2008	*derby*	*PAUSE*	1679.73	1693.04	1701.19	0.58	0.59	0.60
SPECjvm2008	*derby*	*THRU*	1657.36	1670.61	1681.45	0.59	0.62	0.66

Table D.4: Cache Miss Ratios

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
AcmeAir	–	*B:BF*	2.51	2.78	3.24	2.34	2.47	2.58	25.49	26.09	26.38
AcmeAir	–	*B:dBF*	2.54	4.20	5.18	2.39	2.65	2.86	26.02	28.73	30.47
AcmeAir	–	*G:BF*	5.09	5.22	5.42	2.62	2.68	2.72	31.83	32.35	32.74
AcmeAir	–	*G:dBF*	3.77	4.67	4.88	2.42	2.59	2.66	31.65	31.89	32.21
AcmeAir	–	*G:H*	2.67	3.29	4.35	2.28	2.41	2.55	27.64	29.25	31.52

Table D.4: (Continued)

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
AcmeAir	–	*PAUSE*	4.19	4.48	4.66	2.50	2.70	2.84	29.82	30.35	31.60
AcmeAir	–	*THRU*	3.61	3.89	4.24	2.44	2.57	2.75	30.41	30.73	31.07
Daytrader7	–	*B:BF*	5.41	5.60	5.67	3.40	3.48	3.63	21.34	21.61	21.82
Daytrader7	–	*B:dBF*	5.45	5.68	5.99	3.33	3.39	3.45	20.47	21.30	21.89
Daytrader7	–	*G:BF*	5.82	5.95	6.10	3.46	3.51	3.55	22.84	23.14	23.38
Daytrader7	–	*G:dBF*	5.83	5.93	6.06	3.45	3.47	3.49	22.43	22.78	22.98
Daytrader7	–	*G:H*	5.77	5.81	5.84	3.40	3.45	3.52	21.66	21.82	22.11
Daytrader7	–	*PAUSE*	5.25	5.56	5.76	3.39	3.49	3.58	21.65	21.98	22.27
Daytrader7	–	*THRU*	5.06	5.20	5.38	3.38	3.42	3.44	20.83	21.16	21.46
HiBench	*als*	*B:BF*	1.07	1.16	1.31	1.61	1.73	1.80	16.08	16.84	18.09
HiBench	*als*	*B:dBF*	1.08	1.23	1.37	1.63	1.71	1.77	15.63	16.29	17.80
HiBench	*als*	*G:BF*	0.75	0.86	0.93	1.63	1.76	1.92	19.66	20.42	21.81
HiBench	*als*	*G:dBF*	0.79	0.95	1.07	1.70	1.80	1.90	18.10	19.64	21.00
HiBench	*als*	*G:H*	0.68	0.83	0.91	1.69	1.76	1.83	19.13	20.17	21.38
HiBench	*als*	*PAUSE*	0.85	0.93	1.08	1.73	1.83	1.91	19.17	20.07	21.31
HiBench	*als*	*THRU*	0.75	0.87	1.00	1.55	1.76	1.88	19.02	19.84	21.16
HiBench	*bayes*	*B:BF*	0.66	0.72	0.93	2.13	2.17	2.21	22.35	23.01	23.59
HiBench	*bayes*	*B:dBF*	0.55	0.70	0.86	1.91	2.11	2.19	21.87	22.51	22.92
HiBench	*bayes*	*G:BF*	0.70	0.74	0.78	2.05	2.13	2.22	19.96	20.74	21.57
HiBench	*bayes*	*G:dBF*	0.69	0.73	0.82	2.01	2.08	2.14	20.24	20.74	21.66
HiBench	*bayes*	*G:H*	0.70	0.74	0.86	2.01	2.10	2.18	19.81	20.40	20.86
HiBench	*bayes*	*PAUSE*	0.99	1.04	1.12	2.24	2.37	2.49	18.91	19.52	20.07
HiBench	*bayes*	*THRU*	0.72	0.76	0.81	2.21	2.28	2.36	20.42	21.04	21.82
HiBench	*gbt*	*B:BF*	0.94	1.33	1.76	1.65	1.79	1.99	11.61	12.14	12.73
HiBench	*gbt*	*B:dBF*	0.97	1.40	1.83	1.73	1.89	2.00	10.97	11.76	13.17
HiBench	*gbt*	*G:BF*	0.60	0.66	0.75	1.51	1.58	1.68	17.56	18.14	18.52

Table D.4: (Continued)

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
HiBench	*gbt*	*G:dBF*	0.61	0.69	0.85	1.45	1.52	1.62	17.40	18.14	18.68
HiBench	*gbt*	*G:H*	0.59	0.65	0.71	1.48	1.57	1.65	17.24	18.18	18.95
HiBench	*gbt*	*PAUSE*	0.61	0.69	0.84	1.52	1.62	1.95	16.25	18.46	19.30
HiBench	*gbt*	*THRU*	0.55	0.66	0.89	1.49	1.58	1.88	17.74	18.81	19.55
HiBench	*kmeans*	*B:BF*	1.16	1.38	1.62	0.94	0.99	1.05	10.93	11.52	11.93
HiBench	*kmeans*	*B:dBF*	1.18	1.40	1.98	0.91	0.95	1.01	10.98	11.74	12.36
HiBench	*kmeans*	*G:BF*	1.09	1.41	1.76	0.86	0.98	1.12	10.04	11.44	12.15
HiBench	*kmeans*	*G:dBF*	0.93	1.36	1.82	0.85	0.94	1.06	10.55	11.77	13.33
HiBench	*kmeans*	*G:H*	1.03	1.31	1.77	0.88	0.96	1.07	10.48	11.81	12.94
HiBench	*kmeans*	*PAUSE*	2.43	2.75	3.47	1.75	1.89	2.03	10.28	11.43	11.89
HiBench	*kmeans*	*THRU*	1.46	1.78	1.97	1.42	1.53	1.61	13.70	14.25	15.08
HiBench	*lda*	*B:BF*	0.11	0.13	0.15	0.88	0.92	0.96	20.11	20.66	20.98
HiBench	*lda*	*B:dBF*	0.08	0.12	0.15	0.34	0.80	1.00	20.09	20.68	21.08
HiBench	*lda*	*G:BF*	0.22	0.23	0.27	1.12	1.14	1.15	21.41	21.72	21.92
HiBench	*lda*	*G:dBF*	0.21	0.24	0.29	1.14	1.15	1.15	21.61	21.74	21.83
HiBench	*lda*	*G:H*	0.21	0.24	0.29	1.13	1.14	1.15	21.44	21.67	21.95
HiBench	*lda*	*PAUSE*	0.24	0.27	0.31	1.21	1.22	1.23	21.44	21.87	22.26
HiBench	*lda*	*THRU*	0.23	0.25	0.33	1.23	1.25	1.26	21.48	21.86	22.30
HiBench	*linear*	*B:BF*	0.36	0.48	1.02	0.88	1.02	1.21	15.35	16.27	17.12
HiBench	*linear*	*B:dBF*	0.40	0.57	1.09	0.82	0.93	1.08	16.00	17.81	18.96
HiBench	*linear*	*G:BF*	0.38	0.39	0.41	0.71	0.84	0.93	17.78	18.44	19.32
HiBench	*linear*	*G:dBF*	0.37	0.39	0.47	0.71	0.82	0.93	18.01	18.96	19.74
HiBench	*linear*	*G:H*	0.38	0.39	0.41	0.78	0.85	0.97	17.73	18.97	19.78
HiBench	*linear*	*PAUSE*	0.35	0.58	1.67	0.85	0.92	1.08	14.72	17.32	18.57
HiBench	*linear*	*THRU*	0.36	0.48	1.18	0.75	0.96	1.22	14.75	18.46	19.57
HiBench	*lr*	*B:BF*	0.76	0.97	1.04	1.64	1.71	1.78	20.43	21.01	21.51

Table D.4: (Continued)

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
HiBench	*lr*	*B:dBF*	0.86	1.00	1.03	1.42	1.68	1.93	20.53	21.01	22.06
HiBench	*lr*	*G:BF*	0.69	0.84	1.06	0.92	1.19	1.71	20.93	21.29	21.81
HiBench	*lr*	*G:dBF*	0.78	0.81	0.83	1.02	1.11	1.20	20.79	21.27	21.52
HiBench	*lr*	*G:H*	0.61	0.83	1.13	0.97	1.16	1.79	20.86	21.17	21.73
HiBench	*lr*	*PAUSE*	0.63	0.80	0.86	0.84	1.10	1.23	20.98	21.45	21.97
HiBench	*lr*	*THRU*	0.79	0.82	0.84	1.04	1.10	1.15	20.44	21.13	21.95
LoadStallSuite	*dm*	*B:BF*	0.02	0.02	0.02	0.71	0.73	0.75	25.03	25.26	25.52
LoadStallSuite	*dm*	*B:dBF*	0.02	0.02	0.02	0.61	0.73	0.82	25.21	25.58	28.12
LoadStallSuite	*dm*	*G:BF*	0.04	0.04	0.04	0.92	1.25	1.57	25.76	26.64	28.59
LoadStallSuite	*dm*	*G:dBF*	0.04	0.04	0.04	1.04	1.34	1.59	25.90	26.62	27.31
LoadStallSuite	*dm*	*G:H*	0.04	0.04	0.04	0.88	1.34	1.63	25.58	26.60	27.11
LoadStallSuite	*dm*	*PAUSE*	0.03	0.03	0.04	0.67	0.87	0.93	25.37	25.90	27.85
LoadStallSuite	*dm*	*THRU*	0.03	0.04	0.04	0.87	0.90	0.92	25.71	25.83	25.99
LoadStallSuite	*dwm*	*B:BF*	0.03	0.03	0.03	2.40	2.76	3.06	17.94	18.01	18.12
LoadStallSuite	*dwm*	*B:dBF*	0.03	0.03	0.03	2.77	2.99	3.19	17.90	17.99	18.10
LoadStallSuite	*dwm*	*G:BF*	0.06	0.09	0.13	1.99	2.84	3.41	17.95	18.18	18.37
LoadStallSuite	*dwm*	*G:dBF*	0.04	0.09	0.13	1.95	2.72	3.41	18.05	18.22	18.38
LoadStallSuite	*dwm*	*G:H*	0.07	0.10	0.14	2.03	2.65	3.07	17.96	18.13	18.31
LoadStallSuite	*dwm*	*PAUSE*	0.04	0.04	0.04	2.86	2.90	2.99	18.26	18.34	18.40
LoadStallSuite	*dwm*	*THRU*	0.04	0.04	0.04	2.82	2.93	3.02	18.26	18.32	18.41
LoadStallSuite	*ic*	*B:BF*	0.04	0.04	0.05	0.66	0.80	1.30	12.47	17.47	21.73
LoadStallSuite	*ic*	*B:dBF*	0.04	0.04	0.04	0.62	0.70	0.82	16.37	19.20	21.59
LoadStallSuite	*ic*	*G:BF*	0.08	0.08	0.08	1.96	2.05	2.15	19.02	20.86	23.25
LoadStallSuite	*ic*	*G:dBF*	0.08	0.08	0.08	1.99	2.10	2.45	17.97	19.37	22.76
LoadStallSuite	*ic*	*G:H*	0.08	0.08	0.08	1.98	2.09	2.24	16.88	19.81	22.84
LoadStallSuite	*ic*	*PAUSE*	0.08	0.08	0.08	2.01	2.19	2.35	12.55	13.33	14.88

Table D.4: (Continued)

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
LoadStallSuite	*ic*	*THRU*	0.08	0.08	0.09	2.01	2.07	2.20	13.30	15.37	17.07
LoadStallSuite	*ol*	*B:BF*	0.05	0.05	0.07	0.45	0.74	2.04	8.39	20.01	24.46
LoadStallSuite	*ol*	*B:dBF*	0.05	0.06	0.12	0.45	0.50	0.83	17.53	23.40	26.54
LoadStallSuite	*ol*	*G:BF*	0.11	0.15	0.19	2.43	2.59	2.73	30.63	31.50	32.34
LoadStallSuite	*ol*	*G:dBF*	0.11	0.15	0.18	2.46	2.59	2.69	31.06	31.62	32.48
LoadStallSuite	*ol*	*G:H*	0.13	0.16	0.18	2.33	2.42	2.59	29.07	30.63	32.18
LoadStallSuite	*ol*	*PAUSE*	0.19	0.52	0.85	2.53	2.64	2.72	22.78	24.50	26.17
LoadStallSuite	*ol*	*THRU*	0.11	0.17	0.42	2.43	2.49	2.61	24.62	25.63	26.27
LoadStallSuite	*sm*	*B:BF*	1.97	2.15	2.28	2.19	2.27	2.41	15.26	15.81	16.56
LoadStallSuite	*sm*	*B:dBF*	1.98	2.16	2.21	2.16	2.28	2.39	15.36	15.81	16.38
LoadStallSuite	*sm*	*G:BF*	2.02	2.12	2.21	2.21	2.29	2.38	15.33	15.97	16.58
LoadStallSuite	*sm*	*G:dBF*	2.06	2.17	2.33	2.25	2.32	2.45	15.28	15.63	16.51
LoadStallSuite	*sm*	*G:H*	2.04	2.12	2.21	2.26	2.32	2.41	15.51	15.88	16.27
LoadStallSuite	*sm*	*PAUSE*	2.07	2.20	2.30	2.21	2.31	2.47	14.98	15.65	16.14
LoadStallSuite	*sm*	*THRU*	2.02	2.17	2.30	2.18	2.28	2.32	14.57	15.75	16.81
LoadStallSuite	*ts*	*B:BF*	0.06	0.07	0.08	19.39	20.07	20.59	12.00	13.43	14.70
LoadStallSuite	*ts*	*B:dBF*	0.10	0.13	0.17	19.28	20.41	21.19	11.10	11.66	12.64
LoadStallSuite	*ts*	*G:BF*	0.05	0.06	0.08	14.01	14.31	14.71	12.87	14.23	15.04
LoadStallSuite	*ts*	*G:dBF*	0.05	0.06	0.07	14.15	14.40	14.62	13.37	14.55	15.58
LoadStallSuite	*ts*	*G:H*	0.06	0.07	0.09	5.00	5.71	6.20	23.47	24.92	26.61
LoadStallSuite	*ts*	*PAUSE*	0.05	0.06	0.07	1.14	1.18	1.22	27.34	28.29	30.09
LoadStallSuite	*ts*	*THRU*	0.07	0.09	0.10	1.28	1.31	1.33	30.88	31.45	32.61
Renaissance	*akka-uct*	*B:BF*	1.99	2.12	2.25	1.80	1.94	2.11	10.56	10.99	11.28
Renaissance	*akka-uct*	*B:dBF*	1.95	2.03	2.11	1.85	1.90	2.02	10.57	10.97	11.26
Renaissance	*akka-uct*	*G:BF*	2.32	2.48	2.64	2.10	2.17	2.22	13.44	14.03	14.34
Renaissance	*akka-uct*	*G:dBF*	2.25	2.45	2.76	2.01	2.10	2.19	13.25	13.86	14.62

Table D.4: (Continued)

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Renaissance	*akka-uct*	*G:H*	2.26	2.38	2.65	2.03	2.07	2.13	13.17	13.42	13.66
Renaissance	*akka-uct*	*PAUSE*	4.09	4.23	4.54	3.07	3.18	3.25	10.40	10.72	11.10
Renaissance	*akka-uct*	*THRU*	3.11	3.40	3.59	2.83	2.94	3.00	11.23	11.50	11.83
Renaissance	*finagle-http*	*B:BF*	5.88	6.11	6.22	3.30	3.34	3.39	16.79	17.27	17.65
Renaissance	*finagle-http*	*B:dBF*	5.97	6.11	6.25	3.27	3.32	3.36	16.99	17.26	17.53
Renaissance	*finagle-http*	*G:BF*	6.77	6.84	6.90	3.74	3.80	3.84	19.56	19.95	20.32
Renaissance	*finagle-http*	*G:dBF*	6.63	6.75	6.87	3.71	3.76	3.80	19.60	20.04	20.51
Renaissance	*finagle-http*	*G:H*	6.63	6.82	6.97	3.75	3.78	3.80	19.60	19.98	20.30
Renaissance	*finagle-http*	*PAUSE*	5.64	6.03	6.22	3.54	3.62	3.68	19.19	19.74	20.45
Renaissance	*finagle-http*	*THRU*	5.76	5.96	6.06	3.63	3.65	3.68	19.72	20.06	20.90
Renaissance	*fj-kmeans*	*B:BF*	0.15	0.17	0.20	1.25	1.29	1.32	28.38	28.70	29.72
Renaissance	*fj-kmeans*	*B:dBF*	0.15	0.16	0.19	1.25	1.28	1.32	28.11	28.56	28.83
Renaissance	*fj-kmeans*	*G:BF*	0.14	0.17	0.18	1.41	1.43	1.47	30.23	30.71	31.01
Renaissance	*fj-kmeans*	*G:dBF*	0.14	0.16	0.18	1.40	1.42	1.44	30.09	30.51	30.80
Renaissance	*fj-kmeans*	*G:H*	0.15	0.16	0.17	1.40	1.43	1.45	28.84	29.23	29.51
Renaissance	*fj-kmeans*	*PAUSE*	0.12	0.13	0.14	1.23	1.26	1.30	25.42	25.66	26.32
Renaissance	*fj-kmeans*	*THRU*	0.10	0.11	0.12	1.22	1.24	1.30	25.00	25.15	25.31
Renaissance	*future-genetic*	*B:BF*	1.03	1.14	1.25	1.33	1.55	1.73	11.40	12.13	12.86
Renaissance	*future-genetic*	*B:dBF*	1.04	1.14	1.25	1.38	1.55	1.88	11.42	12.63	13.76
Renaissance	*future-genetic*	*G:BF*	1.24	1.40	1.57	1.76	1.99	2.34	14.46	15.05	16.30
Renaissance	*future-genetic*	*G:dBF*	1.21	1.35	1.54	1.62	1.83	2.26	14.24	14.71	15.32
Renaissance	*future-genetic*	*G:H*	1.25	1.33	1.43	1.66	1.78	1.85	14.57	14.94	15.38
Renaissance	*future-genetic*	*PAUSE*	1.37	1.43	1.58	1.79	1.94	2.28	15.84	16.42	16.75
Renaissance	*future-genetic*	*THRU*	1.20	1.28	1.42	1.74	1.94	2.26	15.68	16.30	16.70
Renaissance	*mnemonics*	*B:BF*	0.64	0.82	1.18	1.70	1.85	2.06	12.76	13.74	14.89
Renaissance	*mnemonics*	*B:dBF*	0.54	0.89	1.79	1.60	1.71	1.81	12.42	14.08	15.11

Table D.4: (Continued)

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Renaissance	*mnemonics*	*G:BF*	0.48	0.61	0.75	1.50	1.74	1.99	15.40	16.47	17.41
Renaissance	*mnemonics*	*G:dBF*	0.55	0.64	0.73	1.56	1.84	2.00	15.48	16.38	17.47
Renaissance	*mnemonics*	*G:H*	0.53	0.60	0.64	1.49	1.80	2.16	14.91	15.94	17.09
Renaissance	*mnemonics*	*PAUSE*	0.56	0.65	0.70	1.50	1.69	1.96	16.54	17.19	17.82
Renaissance	*mnemonics*	*THRU*	0.37	0.48	0.61	1.66	1.94	2.25	15.30	16.73	17.62
Renaissance	*rx-scrabble*	*B:BF*	3.45	3.57	3.69	2.03	2.14	2.30	7.98	8.20	8.52
Renaissance	*rx-scrabble*	*B:dBF*	3.33	3.60	3.80	2.04	2.12	2.21	8.07	8.49	9.14
Renaissance	*rx-scrabble*	*G:BF*	3.54	3.73	3.91	2.03	2.14	2.23	8.88	9.30	9.78
Renaissance	*rx-scrabble*	*G:dBF*	3.35	3.60	3.94	2.04	2.13	2.24	9.37	9.79	10.38
Renaissance	*rx-scrabble*	*G:H*	3.24	3.63	3.84	2.05	2.18	2.33	9.36	9.81	10.36
Renaissance	*rx-scrabble*	*PAUSE*	2.93	3.50	3.90	2.09	2.16	2.25	10.85	11.54	11.98
Renaissance	*rx-scrabble*	*THRU*	2.31	2.58	2.93	2.17	2.26	2.32	12.61	12.96	13.47
SPECjvm2008	*compress*	*B:BF*	0.13	0.22	0.39	4.10	4.17	4.32	13.94	14.22	14.59
SPECjvm2008	*compress*	*B:dBF*	0.08	0.20	0.28	4.18	4.30	4.50	13.89	14.16	14.45
SPECjvm2008	*compress*	*G:BF*	0.05	0.12	0.18	5.15	5.29	5.84	13.09	13.31	13.72
SPECjvm2008	*compress*	*G:dBF*	0.06	0.15	0.24	5.22	5.36	5.74	12.78	13.13	13.36
SPECjvm2008	*compress*	*G:H*	0.10	0.15	0.21	5.18	5.34	5.79	12.88	13.17	13.62
SPECjvm2008	*compress*	*PAUSE*	0.05	0.14	0.24	5.13	5.41	5.86	13.10	13.37	13.66
SPECjvm2008	*compress*	*THRU*	0.07	0.18	0.28	5.14	5.27	5.69	13.06	13.29	13.40
SPECjvm2008	*crypto.aes*	*B:BF*	0.06	0.11	0.20	0.26	0.45	1.09	4.27	8.72	11.45
SPECjvm2008	*crypto.aes*	*B:dBF*	0.06	0.10	0.16	0.25	0.54	1.89	2.43	8.57	11.73
SPECjvm2008	*crypto.aes*	*G:BF*	0.05	0.07	0.11	0.13	0.24	0.53	5.74	10.70	14.11
SPECjvm2008	*crypto.aes*	*G:dBF*	0.04	0.06	0.10	0.11	0.26	0.67	5.24	11.03	15.38
SPECjvm2008	*crypto.aes*	*G:H*	0.04	0.07	0.13	0.13	0.51	2.70	1.30	9.77	13.15
SPECjvm2008	*crypto.aes*	*PAUSE*	0.05	0.09	0.14	0.20	0.54	2.09	2.07	8.85	13.39
SPECjvm2008	*crypto.aes*	*THRU*	0.05	0.09	0.12	0.18	0.60	3.74	1.19	10.48	13.97

Table D.4: (Continued)

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
SPECjvm2008	*derby*	*B:BF*	2.82	3.14	3.43	2.08	2.17	2.25	10.06	10.77	11.44
SPECjvm2008	*derby*	*B:dBF*	2.82	3.32	4.30	2.06	2.13	2.23	10.03	10.96	11.90
SPECjvm2008	*derby*	*G:BF*	2.85	3.15	3.38	1.98	2.10	2.25	10.85	11.55	12.67
SPECjvm2008	*derby*	*G:dBF*	2.87	3.33	4.02	1.96	2.02	2.13	10.19	11.05	11.84
SPECjvm2008	*derby*	*G:H*	2.99	3.34	3.74	1.98	2.06	2.11	10.43	11.25	12.21
SPECjvm2008	*derby*	*PAUSE*	4.93	5.27	5.71	2.82	2.96	3.11	9.05	9.47	9.97
SPECjvm2008	*derby*	*THRU*	3.06	3.87	4.41	2.07	2.54	2.74	11.28	12.07	13.24

Appendix E

Medium Granularity Benchmark Results

The data obtained for the medium granularity observations is given in this appendix. Table E.1 shows the obtained correlation coefficient τ and the corresponding significance value ρ between the placements of GC algorithms per benchmark for coarse and medium granular data. Then the load stall performance data for JIT compiled code is presented in Table E.2.

Table E.1: Correlation Data

		Cycles		L1I Miss Ratio		L1D Miss Ratio		L2 Miss Ratio	
Suite	Benchmark	τ	ρ	τ	ρ	τ	ρ	τ	ρ
AcmeAir	–	0.333	0.1907	0.238	0.2810	0.238	0.2810	0.619	0.0345
Daytrader7	–	-0.047	0.6137	1.000	0.0002	1.000	0.0002	0.810	0.0054
HiBench	*als*	0.810	0.0054	0.619	0.0345	0.619	0.0345	0.714	0.0151
HiBench	*bayes*	0.810	0.0054	0.810	0.0054	0.810	0.0054	0.810	0.0054
HiBench	*gbt*	0.810	0.0054	0.333	0.1907	0.333	0.1907	0.810	0.0054
HiBench	*kmeans*	0.714	0.0151	0.714	0.0151	0.714	0.0151	0.619	0.0345
HiBench	*lda*	0.619	0.0345	0.810	0.0054	0.810	0.0054	0.619	0.0345
HiBench	*linear*	0.143	0.3863	0.238	0.2810	0.238	0.2810	0.238	0.2810
HiBench	*lr*	0.238	0.2810	0.333	0.1907	0.333	0.1907	0.429	0.1194
LoadStallSuite	*dm*	0.714	0.0151	0.810	0.0054	0.810	0.0054	0.810	0.0054
LoadStallSuite	*dwm*	0.905	0.0014	0.714	0.0151	0.714	0.0151	0.619	0.0345
LoadStallSuite	*ic*	0.619	0.0345	0.524	0.0681	0.524	0.0681	0.524	0.0681
LoadStallSuite	*ol*	0.810	0.0054	0.810	0.0054	0.810	0.0054	1.000	0.0002
LoadStallSuite	*sm*	0.714	0.0151	0.238	0.2810	0.238	0.2810	0.333	0.1907
LoadStallSuite	*ts*	0.238	0.2810	0.810	0.0054	0.810	0.0054	0.810	0.0054
Renaissance	*akka-uct*	1.000	0.0002	1.000	0.0002	1.000	0.0002	0.905	0.0014
Renaissance	*finagle-http*	0.810	0.0054	0.905	0.0014	0.905	0.0014	0.048	0.5000
Renaissance	*fj-kmeans*	0.905	0.0014	0.714	0.0151	0.714	0.0151	0.905	0.0014
Renaissance	*future-genetic*	0.714	0.0151	0.810	0.0054	0.810	0.0054	0.810	0.0054
Renaissance	*mnemonics*	0.810	0.0054	0.429	0.1194	0.429	0.1194	0.905	0.0014
Renaissance	*rx-scrabble*	0.810	0.0054	0.714	0.0151	0.714	0.0151	0.810	0.0054
SPECjvm2008	*compress*	0.619	0.0345	0.810	0.0054	0.810	0.0054	0.238	0.2810
SPECjvm2008	*crypto.aes*	0.619	0.0345	0.524	0.0681	0.524	0.0681	0.048	0.5000
SPECjvm2008	*derby*	-0.523	0.9655	0.905	0.0014	0.905	0.0014	0.905	0.0014

Table E.2: Just-In-Time Compiled Code Cache Miss Ratios

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
AcmeAir	–	*B:BF*	5.88	6.03	6.16	2.58	2.63	2.66	29.56	29.99	30.55
AcmeAir	–	*B:dBF*	5.93	6.02	6.17	2.51	2.56	2.61	29.69	30.07	30.47
AcmeAir	–	*G:BF*	5.78	6.01	6.21	2.56	2.65	2.71	30.09	30.54	30.98
AcmeAir	–	*G:dBF*	5.88	5.99	6.18	2.55	2.60	2.65	30.38	30.70	31.03
AcmeAir	–	*G:H*	5.88	6.01	6.12	2.59	2.65	2.72	29.98	30.34	31.11
AcmeAir	–	*PAUSE*	6.03	6.15	6.26	2.66	2.72	2.79	29.95	30.48	30.93
AcmeAir	–	*THRU*	5.43	5.67	5.78	2.61	2.69	2.87	30.27	30.57	31.11
Daytrader7	–	*B:BF*	7.79	7.97	8.08	3.96	4.00	4.05	20.31	20.65	20.95
Daytrader7	–	*B:dBF*	7.91	8.01	8.10	3.79	3.83	3.91	20.55	20.75	21.03
Daytrader7	–	*G:BF*	7.80	7.89	7.98	3.77	3.80	3.81	21.91	22.37	22.64
Daytrader7	–	*G:dBF*	7.84	7.94	8.09	3.66	3.74	3.79	21.79	22.10	22.46
Daytrader7	–	*G:H*	8.12	8.23	8.36	3.93	3.98	4.06	20.12	20.37	20.62
Daytrader7	–	*PAUSE*	8.03	8.14	8.30	3.84	3.89	3.92	20.72	21.09	21.38
Daytrader7	–	*THRU*	7.78	7.90	8.07	3.85	3.90	3.95	19.98	20.29	20.57
HiBench	*als*	*B:BF*	1.17	1.39	1.56	1.08	1.17	1.29	10.96	11.91	13.31
HiBench	*als*	*B:dBF*	1.29	1.37	1.59	1.07	1.16	1.26	11.08	11.99	12.80
HiBench	*als*	*G:BF*	0.92	1.05	1.18	1.13	1.18	1.26	14.04	14.76	15.41
HiBench	*als*	*G:dBF*	0.84	1.03	1.20	1.05	1.22	1.36	13.60	14.60	15.87
HiBench	*als*	*G:H*	0.91	1.01	1.13	1.06	1.17	1.28	13.99	14.73	15.93
HiBench	*als*	*PAUSE*	1.05	1.11	1.22	0.96	1.15	1.26	13.64	14.47	15.41
HiBench	*als*	*THRU*	0.86	1.02	1.27	0.92	1.18	1.35	13.86	14.73	15.47
HiBench	*bayes*	*B:BF*	0.76	0.81	0.88	1.59	1.68	1.80	17.96	18.72	19.32
HiBench	*bayes*	*B:dBF*	0.75	0.81	0.91	1.51	1.62	1.70	16.88	17.54	18.29
HiBench	*bayes*	*G:BF*	0.75	0.83	0.91	1.65	1.74	1.85	18.48	19.05	19.79
HiBench	*bayes*	*G:dBF*	0.75	0.80	0.84	1.65	1.77	1.98	16.58	18.22	19.85

Table E.2: (Continued)

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
HiBench	*bayes*	*G:H*	0.74	0.78	0.84	1.69	1.77	1.94	16.14	17.81	18.68
HiBench	*bayes*	*PAUSE*	1.06	1.13	1.22	1.76	1.80	1.87	17.08	18.16	19.11
HiBench	*bayes*	*THRU*	0.86	0.93	1.09	1.64	1.73	1.79	17.67	18.56	19.41
HiBench	*gbt*	*B:BF*	1.52	2.06	2.52	1.39	1.58	1.74	9.13	10.40	11.19
HiBench	*gbt*	*B:dBF*	1.28	1.99	2.70	1.35	1.57	1.90	9.14	10.85	12.76
HiBench	*gbt*	*G:BF*	0.76	0.92	1.23	1.09	1.21	1.31	14.63	16.01	17.22
HiBench	*gbt*	*G:dBF*	0.68	0.93	1.18	1.18	1.22	1.28	11.93	15.23	17.03
HiBench	*gbt*	*G:H*	0.83	1.18	2.32	1.14	1.24	1.33	13.01	15.18	16.48
HiBench	*gbt*	*PAUSE*	0.91	1.02	1.51	1.07	1.25	1.58	12.86	15.47	16.49
HiBench	*gbt*	*THRU*	0.80	0.93	1.39	1.14	1.25	1.66	12.45	15.51	16.79
HiBench	*kmeans*	*B:BF*	1.84	2.32	3.76	0.95	1.02	1.15	5.17	6.43	7.40
HiBench	*kmeans*	*B:dBF*	2.10	2.30	2.57	0.86	1.01	1.09	6.14	6.69	7.10
HiBench	*kmeans*	*G:BF*	1.76	2.13	2.62	0.96	1.07	1.18	6.08	6.68	7.15
HiBench	*kmeans*	*G:dBF*	1.70	2.13	2.87	0.88	0.98	1.11	6.49	7.09	7.71
HiBench	*kmeans*	*G:H*	1.63	1.97	2.65	0.95	1.02	1.13	6.25	6.99	7.83
HiBench	*kmeans*	*PAUSE*	2.92	3.32	4.12	1.30	1.39	1.48	4.01	4.24	4.68
HiBench	*kmeans*	*THRU*	2.08	2.57	2.92	1.13	1.23	1.33	4.41	5.01	5.38
HiBench	*lda*	*B:BF*	0.08	0.11	0.15	0.37	0.74	0.90	18.31	18.84	19.27
HiBench	*lda*	*B:dBF*	0.08	0.11	0.14	0.38	0.73	0.91	18.29	18.69	19.49
HiBench	*lda*	*G:BF*	0.25	0.27	0.32	1.45	1.49	1.52	19.40	19.55	19.69
HiBench	*lda*	*G:dBF*	0.21	0.24	0.27	1.46	1.49	1.51	19.28	19.44	19.87
HiBench	*lda*	*G:H*	0.22	0.24	0.31	1.45	1.51	1.53	19.34	19.49	19.98
HiBench	*lda*	*PAUSE*	0.26	0.29	0.33	1.46	1.49	1.51	19.34	19.53	19.80
HiBench	*lda*	*THRU*	0.25	0.28	0.32	1.47	1.51	1.54	19.29	19.46	19.70
HiBench	*linear*	*B:BF*	0.37	0.67	2.53	0.66	0.80	1.24	7.96	10.86	12.70
HiBench	*linear*	*B:dBF*	0.38	0.75	3.17	0.66	0.74	0.88	6.88	11.29	13.44

Table E.2: (Continued)

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
HiBench	*linear*	*G:BF*	0.38	0.43	0.50	0.62	0.79	1.00	10.13	11.18	12.04
HiBench	*linear*	*G:dBF*	0.37	0.48	0.79	0.62	0.72	0.88	8.76	11.48	12.96
HiBench	*linear*	*G:H*	0.41	0.68	1.86	0.65	0.86	1.37	7.79	10.64	12.10
HiBench	*linear*	*PAUSE*	0.40	0.59	1.31	0.67	0.79	0.88	9.92	11.33	12.44
HiBench	*linear*	*THRU*	0.37	0.47	0.57	0.66	0.84	1.09	8.80	11.29	12.74
HiBench	*lr*	*B:BF*	0.21	0.30	0.34	0.36	0.55	0.60	17.40	17.71	18.32
HiBench	*lr*	*B:dBF*	0.29	0.31	0.33	0.50	0.57	0.60	17.02	17.61	18.10
HiBench	*lr*	*G:BF*	0.20	0.22	0.30	0.34	0.39	0.56	17.62	18.02	18.44
HiBench	*lr*	*G:dBF*	0.19	0.21	0.33	0.32	0.38	0.58	17.72	18.07	18.33
HiBench	*lr*	*G:H*	0.20	0.21	0.22	0.33	0.36	0.40	17.90	18.12	18.51
HiBench	*lr*	*PAUSE*	0.20	0.23	0.39	0.32	0.38	0.69	18.04	18.27	18.61
HiBench	*lr*	*THRU*	0.19	0.20	0.22	0.30	0.35	0.40	17.86	18.24	18.73
LoadStallSuite	*dm*	*B:BF*	0.04	0.05	0.06	0.87	0.90	1.04	23.60	23.73	23.88
LoadStallSuite	*dm*	*B:dBF*	0.03	0.04	0.05	0.86	0.88	0.91	23.51	23.73	23.96
LoadStallSuite	*dm*	*G:BF*	0.05	0.06	0.07	1.13	1.37	1.55	26.43	27.18	27.89
LoadStallSuite	*dm*	*G:dBF*	0.05	0.06	0.08	1.12	1.49	1.70	24.74	25.95	26.42
LoadStallSuite	*dm*	*G:H*	0.05	0.06	0.09	1.46	1.60	1.74	25.42	25.84	26.19
LoadStallSuite	*dm*	*PAUSE*	0.05	0.06	0.09	1.08	1.12	1.18	24.65	24.84	25.08
LoadStallSuite	*dm*	*THRU*	0.04	0.05	0.07	1.10	1.11	1.14	24.89	25.08	25.41
LoadStallSuite	*dwm*	*B:BF*	0.05	0.06	0.08	2.99	3.18	3.38	17.11	17.31	17.42
LoadStallSuite	*dwm*	*B:dBF*	0.05	0.06	0.07	2.20	3.13	3.64	17.26	17.33	17.42
LoadStallSuite	*dwm*	*G:BF*	0.07	0.10	0.12	3.88	4.02	4.15	17.34	17.57	17.73
LoadStallSuite	*dwm*	*G:dBF*	0.06	0.08	0.11	3.82	4.05	4.26	17.38	17.52	17.82
LoadStallSuite	*dwm*	*G:H*	0.07	0.09	0.12	3.77	3.99	4.14	17.35	17.50	17.68
LoadStallSuite	*dwm*	*PAUSE*	0.06	0.09	0.12	2.76	3.09	3.41	17.63	17.78	17.85
LoadStallSuite	*dwm*	*THRU*	0.06	0.07	0.11	2.98	3.16	3.33	17.73	17.81	17.87

Table E.2: (Continued)

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
LoadStallSuite	*ic*	*B:BF*	0.26	0.32	0.39	1.54	1.89	3.43	13.11	17.57	23.15
LoadStallSuite	*ic*	*B:dBF*	0.26	0.30	0.38	1.60	1.83	2.17	14.52	18.26	21.46
LoadStallSuite	*ic*	*G:BF*	0.23	0.29	0.34	1.95	2.09	2.18	16.84	19.61	21.76
LoadStallSuite	*ic*	*G:dBF*	0.26	0.30	0.34	1.93	2.08	2.24	16.65	18.70	20.11
LoadStallSuite	*ic*	*G:H*	0.21	0.28	0.32	1.97	2.11	2.35	18.99	20.10	22.65
LoadStallSuite	*ic*	*PAUSE*	0.21	0.26	0.30	1.95	2.21	2.42	12.76	15.02	19.94
LoadStallSuite	*ic*	*THRU*	0.25	0.30	0.34	1.96	2.06	2.13	13.97	15.41	16.35
LoadStallSuite	*ol*	*B:BF*	0.65	0.95	1.61	1.14	1.31	1.41	15.05	20.20	21.53
LoadStallSuite	*ol*	*B:dBF*	0.68	0.85	1.01	1.25	1.45	2.12	16.53	20.82	22.04
LoadStallSuite	*ol*	*G:BF*	0.43	0.52	0.66	2.18	2.20	2.22	25.98	26.77	27.24
LoadStallSuite	*ol*	*G:dBF*	0.43	0.48	0.58	2.16	2.19	2.24	26.56	27.18	27.82
LoadStallSuite	*ol*	*G:H*	0.20	0.49	0.62	1.98	2.15	2.20	26.40	27.71	28.66
LoadStallSuite	*ol*	*PAUSE*	0.67	0.77	0.89	2.33	2.38	2.41	18.69	19.62	20.69
LoadStallSuite	*ol*	*THRU*	0.26	0.58	0.68	2.15	2.31	2.35	18.85	19.73	21.22
LoadStallSuite	*sm*	*B:BF*	0.52	1.79	3.45	0.99	2.22	6.53	4.18	13.38	41.65
LoadStallSuite	*sm*	*B:dBF*	0.91	3.70	10.93	1.11	1.95	2.62	0.56	12.25	22.45
LoadStallSuite	*sm*	*G:BF*	1.21	6.39	29.02	0.81	1.39	1.89	7.34	13.20	29.94
LoadStallSuite	*sm*	*G:dBF*	0.52	1.86	4.68	1.03	1.89	2.73	2.17	13.21	34.56
LoadStallSuite	*sm*	*G:H*	0.46	3.88	9.19	0.84	1.66	2.76	3.85	9.94	18.02
LoadStallSuite	*sm*	*PAUSE*	1.59	3.82	5.76	0.90	1.79	3.33	2.18	9.70	16.52
LoadStallSuite	*sm*	*THRU*	1.03	3.12	6.49	0.92	1.83	5.78	3.57	16.05	65.68
LoadStallSuite	*ts*	*B:BF*	0.17	0.24	0.35	14.84	15.52	16.36	16.82	17.95	19.16
LoadStallSuite	*ts*	*B:dBF*	0.26	0.33	0.42	13.87	14.18	14.68	16.59	18.11	19.20
LoadStallSuite	*ts*	*G:BF*	0.15	0.20	0.31	11.53	11.69	11.79	18.96	20.09	20.99
LoadStallSuite	*ts*	*G:dBF*	0.14	0.21	0.32	10.88	11.61	12.05	19.31	20.06	20.93
LoadStallSuite	*ts*	*G:H*	0.14	0.21	0.32	4.44	4.77	5.36	25.52	26.70	27.87

Table E.2: (Continued)

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
LoadStallSuite	*ts*	*PAUSE*	0.14	0.19	0.31	2.03	2.05	2.09	28.81	29.08	29.71
LoadStallSuite	*ts*	*THRU*	0.19	0.27	0.33	2.05	2.16	2.22	28.66	29.26	29.59
Renaissance	*akka-uct*	*B:BF*	4.78	5.18	5.53	3.17	3.29	3.40	9.86	10.28	10.81
Renaissance	*akka-uct*	*B:dBF*	4.85	5.19	5.62	2.94	3.00	3.12	9.34	9.59	10.36
Renaissance	*akka-uct*	*G:BF*	4.17	4.48	4.71	3.15	3.38	3.47	11.93	12.44	12.90
Renaissance	*akka-uct*	*G:dBF*	4.13	4.42	4.56	3.02	3.15	3.23	10.88	11.15	11.49
Renaissance	*akka-uct*	*G:H*	4.15	4.44	4.67	3.25	3.33	3.41	11.78	12.19	12.54
Renaissance	*akka-uct*	*PAUSE*	5.07	5.50	5.83	3.37	3.49	3.57	9.63	9.98	10.30
Renaissance	*akka-uct*	*THRU*	4.13	4.47	4.78	3.11	3.33	3.42	10.10	10.54	10.83
Renaissance	*finagle-http*	*B:BF*	9.14	9.22	9.31	3.86	3.95	4.04	16.43	16.92	17.28
Renaissance	*finagle-http*	*B:dBF*	9.04	9.18	9.33	3.78	3.85	3.91	16.90	17.33	18.15
Renaissance	*finagle-http*	*G:BF*	8.97	9.18	9.31	3.85	4.05	4.12	18.05	18.47	18.95
Renaissance	*finagle-http*	*G:dBF*	9.00	9.17	9.29	3.87	3.99	4.08	17.88	18.22	18.63
Renaissance	*finagle-http*	*G:H*	8.99	9.19	9.33	3.91	4.05	4.11	18.01	18.40	18.68
Renaissance	*finagle-http*	*PAUSE*	8.84	9.02	9.21	3.90	3.98	4.04	17.84	18.45	18.99
Renaissance	*finagle-http*	*THRU*	8.70	8.84	8.95	3.96	4.03	4.09	17.75	18.10	18.39
Renaissance	*fj-kmeans*	*B:BF*	0.20	0.25	0.30	1.12	1.21	1.25	29.18	30.04	30.54
Renaissance	*fj-kmeans*	*B:dBF*	0.20	0.25	0.31	0.95	1.18	1.28	29.25	30.10	30.69
Renaissance	*fj-kmeans*	*G:BF*	0.20	0.24	0.28	1.40	1.44	1.47	31.29	31.56	31.83
Renaissance	*fj-kmeans*	*G:dBF*	0.18	0.24	0.26	0.83	1.37	1.46	30.30	31.35	31.92
Renaissance	*fj-kmeans*	*G:H*	0.18	0.22	0.26	1.37	1.39	1.43	29.11	29.66	30.20
Renaissance	*fj-kmeans*	*PAUSE*	0.19	0.23	0.26	1.37	1.39	1.44	27.03	27.77	28.25
Renaissance	*fj-kmeans*	*THRU*	0.19	0.21	0.27	1.34	1.38	1.44	27.23	27.57	28.09
Renaissance	*future-genetic*	*B:BF*	1.20	1.37	1.54	1.50	1.79	2.10	12.42	13.97	15.24
Renaissance	*future-genetic*	*B:dBF*	1.20	1.40	1.53	1.58	1.78	2.09	12.48	14.10	15.27
Renaissance	*future-genetic*	*G:BF*	1.04	1.25	1.40	1.60	1.84	2.17	14.32	14.87	15.31

Table E.2: (Continued)

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Renaissance	*future-genetic*	*G:dBF*	1.14	1.30	1.55	1.54	1.90	2.17	14.24	14.98	15.42
Renaissance	*future-genetic*	*G:H*	1.05	1.25	1.43	1.58	1.85	2.22	14.62	14.89	15.28
Renaissance	*future-genetic*	*PAUSE*	1.38	1.62	1.78	1.72	2.09	2.37	15.14	15.82	16.24
Renaissance	*future-genetic*	*THRU*	1.16	1.26	1.41	1.72	1.86	2.20	14.97	15.49	15.69
Renaissance	*mnemonics*	*B:BF*	1.41	1.76	2.67	1.66	1.85	2.23	8.11	9.34	10.33
Renaissance	*mnemonics*	*B:dBF*	1.53	1.72	1.85	1.59	1.80	2.30	8.52	9.55	10.01
Renaissance	*mnemonics*	*G:BF*	1.15	1.36	1.61	1.53	1.75	2.09	12.94	13.88	14.72
Renaissance	*mnemonics*	*G:dBF*	1.24	1.37	1.62	1.53	1.80	1.99	12.87	14.00	14.45
Renaissance	*mnemonics*	*G:H*	1.22	1.37	1.64	1.54	1.78	2.07	12.65	14.17	14.96
Renaissance	*mnemonics*	*PAUSE*	1.42	1.60	1.67	1.46	1.76	2.05	12.27	13.28	14.16
Renaissance	*mnemonics*	*THRU*	1.00	1.17	1.28	1.49	1.94	2.81	10.79	14.09	15.13
Renaissance	*rx-scrabble*	*B:BF*	6.84	7.17	7.64	2.17	2.27	2.41	4.07	4.19	4.31
Renaissance	*rx-scrabble*	*B:dBF*	6.86	7.23	7.92	2.04	2.18	2.45	3.95	4.20	4.40
Renaissance	*rx-scrabble*	*G:BF*	5.89	6.28	6.91	2.06	2.17	2.28	4.90	5.09	5.32
Renaissance	*rx-scrabble*	*G:dBF*	6.04	6.43	6.74	2.00	2.12	2.20	4.91	5.08	5.21
Renaissance	*rx-scrabble*	*G:H*	6.18	6.47	7.18	2.06	2.17	2.32	4.95	5.07	5.23
Renaissance	*rx-scrabble*	*PAUSE*	6.53	6.93	7.59	1.99	2.09	2.23	4.69	4.96	5.25
Renaissance	*rx-scrabble*	*THRU*	5.35	5.62	6.03	2.05	2.10	2.18	5.42	5.59	5.70
SPECjvm2008	*compress*	*B:BF*	0.11	0.13	0.16	4.69	4.89	5.23	12.72	13.47	13.90
SPECjvm2008	*compress*	*B:dBF*	0.11	0.14	0.16	4.65	4.89	5.26	13.27	13.49	13.95
SPECjvm2008	*compress*	*G:BF*	0.10	0.14	0.19	5.69	6.12	6.55	13.77	14.45	15.43
SPECjvm2008	*compress*	*G:dBF*	0.12	0.15	0.20	5.79	6.21	6.72	13.71	13.96	14.26
SPECjvm2008	*compress*	*G:H*	0.11	0.15	0.19	5.93	6.14	6.65	13.48	14.00	14.39
SPECjvm2008	*compress*	*PAUSE*	0.10	0.14	0.19	5.91	6.01	6.28	13.63	14.01	14.39
SPECjvm2008	*compress*	*THRU*	0.11	0.14	0.17	5.91	5.97	6.32	13.48	13.84	14.34
SPECjvm2008	*crypto.aes*	*B:BF*	0.06	0.08	0.12	0.19	0.33	0.52	1.64	2.82	4.51

Table E.2: (Continued)

			$L1i_{miss}$ (%)			$L1d_{miss}$ (%)			$L2_{miss}$ (%)		
Suite	Benchmark	Configuration	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
SPECjvm2008	*crypto.aes*	*B:dBF*	0.06	0.09	0.11	0.17	0.38	1.12	0.84	2.88	4.23
SPECjvm2008	*crypto.aes*	*G:BF*	0.06	0.08	0.11	0.12	0.31	0.59	1.76	3.36	5.72
SPECjvm2008	*crypto.aes*	*G:dBF*	0.06	0.08	0.11	0.11	0.49	2.54	0.39	3.55	6.13
SPECjvm2008	*crypto.aes*	*G:H*	0.06	0.07	0.11	0.11	0.54	3.06	0.36	3.70	6.55
SPECjvm2008	*crypto.aes*	*PAUSE*	0.05	0.08	0.11	0.24	0.79	1.97	0.51	1.94	3.59
SPECjvm2008	*crypto.aes*	*THRU*	0.05	0.07	0.11	0.17	0.54	1.82	0.56	2.57	4.73
SPECjvm2008	*derby*	*B:BF*	4.85	5.23	5.67	2.26	2.35	2.47	7.46	7.90	8.33
SPECjvm2008	*derby*	*B:dBF*	4.53	5.50	6.35	2.26	2.33	2.42	6.97	7.81	8.32
SPECjvm2008	*derby*	*G:BF*	4.99	5.35	5.83	2.30	2.37	2.50	7.70	8.35	9.02
SPECjvm2008	*derby*	*G:dBF*	5.17	5.48	6.12	2.24	2.32	2.40	7.83	8.18	8.72
SPECjvm2008	*derby*	*G:H*	5.07	5.41	6.00	2.20	2.28	2.40	7.76	8.37	8.75
SPECjvm2008	*derby*	*PAUSE*	5.37	6.60	7.73	2.21	2.52	2.74	6.86	7.98	8.68
SPECjvm2008	*derby*	*THRU*	5.43	6.07	7.08	2.38	2.46	2.60	7.70	8.41	8.86

Appendix F

Fine Granularity Load Stall Data

In this chapter the collected fine granularity data is presented. Table F.1 shows the stalling instruction information and Table F.2 the collected JBC data.

Table F.1: Stalling Instructions

Algorithm	Set	Mnemonic	Backend	Frontend/Backend	Other
B:BF	B_c	vldrimmd	7	0	0
B:BF	B_c	ldrimmw	5	1	0
B:BF	B_c	dmb	0	0	5
B:BF	B_o	vldrimmd	11	0	0
B:BF	B_o	ldrimmw	10	0	0
B:BF	B_o	ldrimmx	3	0	0
B:BF	B_o	ldrsbimmx	3	0	0
B:BF	B_o	ldrshimmx	3	0	0
B:dBF	B_c	vldrimmd	6	0	0
B:dBF	B_c	ldrimmw	2	0	0
B:dBF	B_c	dmb	0	0	1
B:dBF	B_c	prfmimm	0	0	1
B:dBF	B_o	ldrimmw	11	1	0
B:dBF	B_o	vldrimmd	8	0	0
B:dBF	B_o	ldrsbimmx	3	0	0
B:dBF	B_o	ldrshimmx	3	0	0
B:dBF	B_o	ldrimmx	1	1	0
B:dBF	B_o	dmb	0	0	1
B:dBF	B_o	strimmw	1	0	0
G:BF	B_c	dmb	0	0	13
G:BF	B_c	vldroffd	8	0	0

Table F.1: (Continued)

Algorithm	Set	Mnemonic	Backend	Frontend/Backend	Other
G:BF	B_c	ldrimmw	7	0	0
G:BF	B_c	vldrimmd	5	1	0
G:BF	B_c	ldrimmx	1	0	0
G:BF	B_c	vldrimms	1	0	0
G:BF	B_o	ldrimmw	23	3	0
G:BF	B_o	dmb	0	0	4
G:BF	B_o	ldrimmx	3	0	0
G:BF	B_o	ldrhimm	2	0	0
G:BF	B_o	ldrsbimmx	2	0	0
G:BF	B_o	ldrsboffx	2	0	0
G:BF	B_o	vldrimmd	2	0	0
G:BF	B_o	ldrhoff	1	0	0
G:BF	B_o	lsrw	0	0	1
G:dBF	B_c	dmb	0	0	11
G:dBF	B_c	ldrimmw	7	1	0
G:dBF	B_c	vldrimmd	7	1	0
G:dBF	B_c	vldroffd	8	0	0
G:dBF	B_c	ldrhimm	1	0	0
G:dBF	B_c	ldrimmx	1	0	0
G:dBF	B_c	strimmx	1	0	0
G:dBF	B_c	vldrimms	1	0	0
G:dBF	B_o	ldrimmw	23	3	0
G:dBF	B_o	dmb	0	0	7
G:dBF	B_o	ldrimmx	4	0	0
G:dBF	B_o	vldrimmd	4	0	0
G:dBF	B_o	ldrhimm	2	0	0
G:dBF	B_o	ldrsbimmx	2	0	0
G:dBF	B_o	ldrsboffx	2	0	0
G:dBF	B_o	strimmx	2	0	0
G:dBF	B_o	ldrhoff	1	0	0
G:dBF	B_o	ldroffw	1	0	0
G:dBF	B_o	lsrw	0	0	1
G:H	B_c	dmb	0	0	14
G:H	B_c	ldrimmw	7	1	0
G:H	B_c	vldroffd	8	0	0
G:H	B_c	vldrimmd	5	1	0
G:H	B_c	ldrhimm	2	0	0
G:H	B_c	ldrimmx	1	1	0
G:H	B_c	vldrimms	1	0	0
G:H	B_o	ldrimmw	20	3	0
G:H	B_o	vldrimmd	5	0	0
G:H	B_o	dmb	0	0	4

Table F.1: (Continued)

Algorithm	Set	Mnemonic	Backend	Frontend/Backend	Other
G:H	B_o	ldrimmx	3	0	0
G:H	B_o	ldrhimm	2	0	0
G:H	B_o	ldrsbimmx	2	0	0
G:H	B_o	ldrsboffx	2	0	0
G:H	B_o	ldrhoff	1	0	0
G:H	B_o	ldroffw	1	0	0
PAUSE	B_c	dmb	0	0	8
PAUSE	B_c	vldroffd	8	0	0
PAUSE	B_c	ldrimmw	5	1	0
PAUSE	B_c	vldrimmd	4	1	0
PAUSE	B_c	ldrimmx	1	0	0
PAUSE	B_c	vldrimms	1	0	0
PAUSE	B_o	ldrimmw	18	2	0
PAUSE	B_o	dmb	0	0	5
PAUSE	B_o	ldrimmx	4	0	0
PAUSE	B_o	vldrimmd	3	0	0
PAUSE	B_o	ldrhimm	2	0	0
PAUSE	B_o	ldrsbimmx	2	0	0
PAUSE	B_o	ldrsboffx	2	0	0
PAUSE	B_o	ldrhoff	1	0	0
PAUSE	B_o	strimmw	1	0	0
THRU	B_c	ldrimmw	9	1	0
THRU	B_c	dmb	0	0	9
THRU	B_c	vldrimmd	6	1	0
THRU	B_c	ldxrw	1	0	0
THRU	B_c	vldrimms	1	0	0
THRU	B_o	ldrimmw	16	2	0
THRU	B_o	dmb	0	0	6
THRU	B_o	vldrimmd	4	0	0
THRU	B_o	ldrhimm	2	0	0
THRU	B_o	ldrsbimmx	2	0	0
THRU	B_o	ldrhoff	1	0	0
THRU	B_o	ldrimmx	1	0	0
THRU	B_o	ldroffw	1	0	0
THRU	B_o	ldrsboffx	1	0	0

Table F.2: Stalling Bytecode

Algorithm	Set	JBC	Backend	Frontend/Backend	Other
B:BF	B_c	getfield	3	0	2
B:BF	B_c	daload	2	1	0
B:BF	B_c	dload	3	0	0
B:BF	B_c	monitorenter	0	0	2
B:BF	B_c	aload3	1	0	0
B:BF	B_c	checkcast	1	0	0
B:BF	B_c	goto	1	0	0
B:BF	B_c	ifge	1	0	0
B:BF	B_c	putfield	0	0	1
B:BF	B_o	iaload	8	0	0
B:BF	B_o	daload	7	0	0
B:BF	B_o	getfield	5	0	0
B:BF	B_o	baload	3	0	0
B:BF	B_o	saload	3	0	0
B:BF	B_o	aaload	1	0	0
B:BF	B_o	aload	1	0	0
B:BF	B_o	getstatic	1	0	0
B:BF	B_o	lload1	1	0	0
B:dBF	B_c	dload	4	0	0
B:dBF	B_c	getfield	3	0	1
B:dBF	B_c	checkcast	1	0	0
B:dBF	B_c	newarray	0	0	1
B:dBF	B_o	iaload	10	0	0
B:dBF	B_o	daload	5	0	0
B:dBF	B_o	baload	4	0	0
B:dBF	B_o	getfield	3	1	0
B:dBF	B_o	saload	3	0	0
B:dBF	B_o	newdup	1	0	1
B:dBF	B_o	invokeinterface	0	1	0
B:dBF	B_o	putfield	1	0	0
G:BF	B_c	daload	9	1	0
G:BF	B_c	getfield	7	0	2
G:BF	B_c	invokevirtual	1	0	5
G:BF	B_c	monitorenter	0	0	3
G:BF	B_c	putfield	0	0	3
G:BF	B_c	checkcast	2	0	0
G:BF	B_c	dload	1	0	0
G:BF	B_c	faload	1	0	0
G:BF	B_c	goto	1	0	0
G:BF	B_o	checkcast	9	2	0
G:BF	B_o	iaload	7	0	0
G:BF	B_o	getfield	5	1	0

Table F.2: (Continued)

Algorithm	Set	JBC	Backend	Frontend/Backend	Other
G:BF	B_o	invokevirtual	2	0	4
G:BF	B_o	aaload	4	0	0
G:BF	B_o	baload	4	0	0
G:BF	B_o	saload	3	0	0
G:BF	B_o	iand	0	0	1
G:BF	B_o	return1	1	0	0
G:dBF	B_c	getfield	8	1	3
G:dBF	B_c	daload	9	1	0
G:dBF	B_c	dload	3	0	0
G:dBF	B_c	invokevirtual	0	0	3
G:dBF	B_c	monitorenter	0	0	3
G:dBF	B_c	checkcast	2	0	0
G:dBF	B_c	putfield	0	0	2
G:dBF	B_c	caload	1	0	0
G:dBF	B_c	faload	1	0	0
G:dBF	B_c	goto	1	0	0
G:dBF	B_c	iconst1	1	0	0
G:dBF	B_o	checkcast	9	3	0
G:dBF	B_o	getfield	7	0	1
G:dBF	B_o	iaload	7	0	0
G:dBF	B_o	invokevirtual	2	0	4
G:dBF	B_o	baload	5	0	0
G:dBF	B_o	aaload	4	0	0
G:dBF	B_o	saload	3	0	0
G:dBF	B_o	aload0getfield	0	0	2
G:dBF	B_o	newdup	2	0	0
G:dBF	B_o	aload0	1	0	0
G:dBF	B_o	iand	0	0	1
G:dBF	B_o	invokeinterface	1	0	0
G:H	B_c	getfield	7	1	4
G:H	B_c	daload	9	1	0
G:H	B_c	invokevirtual	1	0	4
G:H	B_c	monitorenter	0	0	3
G:H	B_c	putfield	0	0	3
G:H	B_c	caload	2	0	0
G:H	B_c	checkcast	2	0	0
G:H	B_c	faload	1	0	0
G:H	B_c	goto	1	0	0
G:H	B_c	ifgt	0	1	0
G:H	B_c	istore	1	0	0
G:H	B_o	checkcast	9	2	0
G:H	B_o	getfield	7	1	0

Table F.2: (Continued)

Algorithm	Set	JBC	Backend	Frontend/Backend	Other
G:H	B_o	iaload	6	0	0
G:H	B_o	invokevirtual	1	0	4
G:H	B_o	aaload	4	0	0
G:H	B_o	baload	4	0	0
G:H	B_o	saload	3	0	0
G:H	B_o	aload2	1	0	0
G:H	B_o	invokeinterface	1	0	0
PAUSE	B_c	daload	8	1	0
PAUSE	B_c	getfield	5	1	3
PAUSE	B_c	monitorenter	0	0	3
PAUSE	B_c	checkcast	2	0	0
PAUSE	B_c	dload	2	0	0
PAUSE	B_c	aload0getfield	1	0	0
PAUSE	B_c	faload	1	0	0
PAUSE	B_c	getstatic	0	0	1
PAUSE	B_c	putfield	0	0	1
PAUSE	B_o	checkcast	6	2	0
PAUSE	B_o	iaload	7	0	0
PAUSE	B_o	invokevirtual	1	0	4
PAUSE	B_o	aaload	4	0	0
PAUSE	B_o	baload	4	0	0
PAUSE	B_o	getfield	3	0	0
PAUSE	B_o	saload	3	0	0
PAUSE	B_o	invokeinterface	2	0	0
PAUSE	B_o	newdup	1	0	1
PAUSE	B_o	lload1	1	0	0
PAUSE	B_o	putfield	1	0	0
THRU	B_c	monitorenter	1	0	6
THRU	B_c	getfield	4	1	1
THRU	B_c	checkcast	3	0	0
THRU	B_c	dload	2	0	0
THRU	B_c	invokevirtual	2	0	0
THRU	B_c	putfield	0	0	2
THRU	B_c	aaload	1	0	0
THRU	B_c	aload3	1	0	0
THRU	B_c	daload	0	1	0
THRU	B_c	faload	1	0	0
THRU	B_c	goto	1	0	0
THRU	B_c	iconst1	1	0	0
THRU	B_o	iaload	8	0	0
THRU	B_o	invokevirtual	2	0	4
THRU	B_o	checkcast	3	2	0

Table F.2: (Continued)

Algorithm	Set	JBC	Backend	Frontend/Backend	Other
THRU	B_o	aaload	4	0	0
THRU	B_o	getfield	3	0	1
THRU	B_o	baload	3	0	0
THRU	B_o	saload	3	0	0
THRU	B_o	aload	1	0	0
THRU	B_o	aload0getfield	0	0	1
THRU	B_o	invokeinterface	1	0	0

Vita

Candidate's full name: Jonas Rouven Schönauer

University attended:

Bachelor of Science (Applied Mathematics and Computer Science),
FH-Aachen, Germany, 2021

Publications:

-

Invited Talks:

J.R. Schönauer, H.S.A. Arafat, D.D. Bremner, K.B. Kent, J. Wang, "Overcoming Load Stalls: Enhancing Performance with the Eclipse OpenJ9 JVM Project," *The 33rd Annual International Conference on Computer Science and Software Engineering, Las Vegas, US, September 11–14, 2023.*

Non-refereed Posters:

J.R. Schönauer, D.D. Bremner, K.B. Kent and J. Wang, "Impact of Garbage Collection Policies on Load Stalls on AArch64 in OpenJ9," *Poster, 18th Annual Research Exposition of the UNB Faculty of Computer Science, Fredericton, Canada, April 12, 2024.*

J.R. Schönauer, D.D. Bremner, K.B. Kent and J. Wang, "Detection and Mitigation of Load Stalls on AArch64," *Poster, 17th Annual Research Exposition of the UNB Faculty of Computer Science, Fredericton, Canada, April 14, 2023.*