INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

The JAFARDD Processor: A Java Architecture Based on a Folding Algorithm, with Reservation Stations, Dynamic Translation, and Dual Processing

by

Mohamed Watheq Ali Kamel El-Kharashi

B. Sc., Ain Shams University, 1992
M. Sc., Ain Shams University, 1996

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

We accept this dissertation as conforming to the required standard

---

Dr. F. Gebali, Supervisor (Department of Electrical and Computer Engineering)

Dr. K. F. Li, Supervisor (Department of Electrical and Computer Engineering)

Dr. N. J. Dimopoulos, Departmental Member (Department of Electrical and Computer Engineering)

Dr. D. M. Miller, Outside Member (Department of Computer Science)

Dr. H. Alnuweiri, External Examiner (Department of Electrical and Computer Engineering, University of British Columbia)

© Mohamed Watheq Ali Kamel El-Kharashi, 2002

University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part by photocopy or other means, without the permission of the author.

Supervisors: Dr. F. Gebali and Dr. K. F. Li

ABSTRACT

Java's cross-platform virtual machine arrangement and its special features that make it ideal for writing network applications, also have a tremendous negative impact on its operations. In spite of its relatively weak performance, Java's success has motivated the search for techniques to enhance its execution.

This work presents the JAFARDD (*a Java Architecture based on a Folding Algorithm, with Reservation stations, Dynamic translation, and Dual processing*) processor designed to accelerate Java processing. JAFARDD dynamically translates Java bytecodes to RISC instructions to facilitate the use of a typical general-purpose RISC core. This enables the exploitation of the instruction level parallelism among the translated instructions using well established techniques, and facilitates the migration to Java-enabled hardware.

Designing hardware for Java requires an extensive knowledge and understanding of its instruction set architecture which were acquired through a comprehensive behavioral analysis by benchmarking. Many aspects of the Java workload behavior were collected and the resulting statistics were analyzed. This helped identify performance-critical aspects that are candidates for hardware support. Our analysis surpasses other similar ones in terms of the number of aspects studied and the coverage of the recommendations made.

Next, a global analysis of the design space of Java processors was carried out. Different hardware design options and alternatives that are suitable for Java were explored and their trade-offs were examined. We especially focused on the design methodology, execution engine organization, parallelism exploitation, and support for high-level language features. This analysis helped identify innovative design ideas such as the use of a modified Tomasulo's algorithm. This, in turn, motivated the development of a bytecode folding algorithm that integrates with the reservation station concept in JAFARRD.

While examining the behavioral analysis and the design space exploration ideas, a list of global architectural design principles started to emerge. These principles ensure JAFARRD can execute Java efficiently and are taken into consideration while the various instruction pipeline modules were designed.

Results from the behavioral analysis also confirmed that Java's stack architecture creates virtual data dependencies that limit performance and prohibit instruction level par-

allelism. To overcome this drawback, stack operation folding has been suggested in the literature to enhance performance by grouping contiguous instructions that have true data dependencies into a compound instruction. We have developed a folding algorithm that, unlike existing ones, does not require the folded instructions to be consecutive. To the best of our knowledge, our folding algorithm is the only one that permits nested pattern folding, tolerates variations in folding groups, and detects and resolves folding hazards completely. By incorporating this algorithm into a Java processor, the need for, and therefore the limitations of, a stack are eliminated.

In addition to an efficient dual processing configuration (i.e., Java and RISC), JAFARDD is empowered with a number of innovative design features, including: an adaptive feedback fetch policy that copes with the variation in Java instruction size, a smart bytecode queue that compensates for the lack of a stack, an on-chip local variable file to facilitate operand access, an early tag assignment to dispatched instructions to reduce processing delay, and a specialized load/store unit that preprocesses object-oriented instructions.

The functionality of JAFARDD has been successfully demonstrated through VHDL modeling and simulation. Furthermore, benchmarking using SPECjvm98 showed that the introduced techniques indeed speed up Java execution. Our bytecode folding algorithm speeds up execution by an average of about 1.29, eliminating an average of 97% of the stack instructions and 50% of the overall instructions.

Compared to other proposals, JAFARDD combines Java bytecode folding with dynamic hardware translation, while maintaining the RISC nature of the processor, making this a much more flexible and general approach.

Table of Contents

Abstract ii

Table of Contents v

List of Tables xiii

List of Figures xvi

List of Abbreviations xx

Trademarks xxii

Acknowledgement xxiii

Dedication xxiv

1 Introduction 1

1.1 The Java Virtual Machine 2

1.1.1 Different Bytecode Manipulation Methods 2

1.1.2 Performance-Hindering Features 4

1.2 Motivations 5

1.3 Research Objectives 6

1.4 Design Challenges 7

1.5 Methodology and Road Map 8

2 Java Processor Architectural Requirements: A Quantitative Study 11

2.1 Introduction 11

2.2 Experiment Framework 12

2.2.1 Study Platform 12

2.2.2 Trace Generation 13
2.2.2.1 JDK Core Organization 14
2.2.2.2 Core Instrumentation 15
2.2.2.3 Execution Trace Components 15
2.2.3 Trace Processing 15
2.2.4 Benchmarking 16
2.3 JVM Instruction Set Architecture (ISA) 17
2.3.1 Class Files and Their Loader 18
2.3.2 Execution Engine 19
2.3.3 Runtime Data Areas 19
2.4 Access Patterns for Data Types 20
2.4.1 JVM Data Types 21
2.4.2 Single-Type Operations 21
2.4.3 Type Conversion Operations 22
2.5 Addressing Modes 22
2.5.1 JVM Addressing Modes 23
2.5.2 Runtime Resolution and Quick Instruction Processing 24
2.5.3 General Usage Patterns 25
2.5.4 Quick Execution 26
2.6 Instruction Set Utilization 26
2.6.1 JVM Instruction Set 27
2.6.2 Dynamic Opcode Distribution 33
2.6.3 Frequency of Null Usage 35
2.6.4 Branch Prediction 35
2.6.5 Effect on Stack Size 35
2.7 Instruction Encoding 36
2.7.1 JVM Instruction Encoding 36
2.7.2 Local Variable Indices 39
2.7.3 Constant Pool Indices 40
2.7.4 Immediates 40
2.7.5 Dynamic Instruction Length 41
2.7.6 Array Indices 44

2.7.7 Branching Distances 45

2.7.8 Encoding Requirements for Different Instruction Classes 45

2.8 Execution Time Requirements 46

2.8.1 Performance Critical Instructions 49

2.9 Method Invocation Behavior 49

2.9.1 Hierarchical Method Invocations 49

2.9.2 Local Variables 49

2.9.3 Operand Stack 53

2.9.4 Stack Primitive Operations 55

2.9.5 Native Invocations 57

2.10 Effects of Object Orientation 57

2.10.1 Frequency of Constructors Invocations 58

2.10.2 Heavily Used Classes 58

2.10.3 Multithreading 59

2.10.4 Memory Management Performance 61

2.11 Conclusions 61

3 Design Space Analysis of Java Processors **66**

3.1 Introduction 66

3.2 Design Space Trees 66

3.3 Design Methodology 67

3.3.1 Generality 67

3.3.1.1 Java-Specific Approaches 68

3.3.1.2 General-Purpose Approaches 69

3.3.2 Bytecode Processing Capacity 72

3.3.3 Bytecode Issuing Capacity 72

3.3.4 Complex Bytecode Processing 73

3.4 Execution Engine Organization 73

3.4.1 Complexity 73

3.4.1.1 JVM Compared to RISC Cores 74

3.4.1.2 CISC Features for JVM Implementation 75

3.4.1.3 Decoupling from RISC and CISC 76

3.4.2 Registers 76

3.4.3 Caches 76
3.4.4 Stack 77
3.4.4.1 Frame Organization 77
3.4.4.2 Realization 78
3.4.4.3 Suitable Size 80
3.4.4.4 Components 80
3.4.4.5 Spill/Refill Management 80
3.4.5 Execution Units 81
3.5 Parallelism Exploitation 81
3.5.1 Pipelining 81
3.5.2 Multiple-Issuing 83
3.5.2.1 VLIW Organization for Bytecodes 83
3.5.2.2 Superscalarity through Reservation Stations 84
3.5.3 Thread Level Parallelism 85
3.6 Support for High-level Language Features 85
3.6.1 General Features 86
3.6.2 Object Orientation 86
3.6.3 Exception Handling 86
3.6.4 Symbolic Resolution 87
3.6.5 Garbage Collection 87
3.7 Conclusions 88

4 Overview of the JAFARDD Microarchitecture 90
4.1 Introduction 90
4.2 Global Architectural Design Principles 91
4.3 Design Features 93
4.4 Processing Phases 93
4.5 Pipeline Stages 93
4.6 Overview of the JAFARDD Architecture 95
4.7 Dual Processing Architecture 97
4.8 Conclusions 98

5 An Operand Extraction (OPEX) Bytecode Folding Algorithm 99
5.1 Introduction . . . 99
5.2 Folding Java Stack Bytecodes . . . 100
5.3 The OPEX Bytecode Folding Algorithm . . . 105
5.3.1 Classification of JVM Instructions . . . 105
5.3.2 Anchor Instructions . . . 108
5.3.3 Basics of the Algorithm . . . 109
5.3.4 Tagging . . . 111
5.3.5 Model Details . . . 112
5.4 Algorithm Hazards . . . 114
5.4.1 Hazard Detection . . . 116
5.4.2 Resolution by Local Variable Renaming . . . 117
5.5 Operation of the FIG Unit . . . 118
5.5.1 State Diagram . . . 118
5.5.2 Algorithm . . . 118
5.6 Special Pattern Optimizations . . . 125
5.7 Conclusions . . . 125

6 Architecture Details 127
6.1 Introduction . . . 127
6.2 Front End . . . 128
6.2.1 BF Architectural Design Principles . . . 128
6.2.2 An Adaptive Feedback Fetch Policy . . . 128
6.2.3 Bytecode Processing within the BF . . . 129
6.2.4 BF Architectural Block . . . 129
6.2.5 BF Operation Steps . . . 129
6.3 Folding Administration . . . 131
6.3.1 FIG Architectural Design Principles . . . 131
6.3.2 Bytecode Processing within the FIG . . . 132
6.3.3 FIG Architectural Module . . . 132
6.3.4 FIG Operation Steps . . . 132
6.4 Queuing, Folding, and Dynamic Translation . . . 132
6.4.1 Bytecode Queue Manager (BQM) . . . 133

6.4.1.1 BQM Architectural Design Principles 133
6.4.1.2 BQM-Internal Operations 133
6.4.1.3 Bytecode Processing within the BQM 134
6.4.1.4 BQM Architectural Module 135
6.4.1.5 BF Minicontroller 139
6.4.1.6 BQM Operation Steps 139
6.4.1.7 BQM Operation Control, Priorities, and Sequencing . . . 142
6.4.2 Folding Translator (FT) 144
6.4.2.1 FT Architectural Design Principles 144
6.4.2.2 Produced Instruction Format 145
6.4.2.3 Bytecode Processing Within the FT 146
6.4.2.4 FT Architectural Module 147
6.4.2.5 Non-Translated Instructions 149
6.5 Dynamic Scheduling and Execution 149
6.5.1 Local Variable File (LVF) 149
6.5.1.1 LVF Architectural Design Principles 149
6.5.1.2 LVF-Internal Operations 150
6.5.1.3 Early Assignment of Instruction/LV Tags 151
6.5.1.4 LVF Architectural Module 153
6.5.1.5 LVF Minicontroller 155
6.5.1.6 LVF Operation Steps 156
6.5.1.7 LVF Operation Control, Priorities, and Sequencing . . . 156
6.5.2 Reservation Stations (RSs) 158
6.5.2.1 RS Architectural Design Principles 159
6.5.2.2 Tomasulo's Algorithm for Java Processors 159
6.5.2.3 LV Renaming 160
6.5.2.4 Instruction Shelving 161
6.5.2.5 Instruction Dispatching 162
6.5.2.6 RS Internal Operations 163
6.5.2.7 RS Unit Architectural Module 164
6.5.2.8 RS Unit Minicontroller 165
6.5.2.9 RS Operation Steps 166

6.5.2.10 RS Operation Control, Priorities, and Sequencing 166
6.5.3 Generic Execution Unit (EX) 169
6.5.3.1 EX Architectural Design Principles 169
6.5.3.2 Generic EX Architectural Module 169
6.5.4 Load/Store Execution Unit (LS) 170
6.5.4.1 LS Architectural Design Principles 170
6.5.4.2 Bytecode Processing within the LS 170
6.5.4.3 LS Architectural Module 171
6.5.4.4 LS Operation Steps 172
6.5.5 Common Data Bus (CDB) 174
6.6 Conclusions 176

7 A Processing Example and Performance Evaluation **177**
7.1 Introduction 177
7.2 A Comprehensive Illustrative Example 177
7.3 Speedup Formula 183
7.4 Experimental Framework 185
7.4.1 Study Platform 185
7.4.2 Trace Processing 185
7.4.3 Benchmarking 185
7.5 Generated Folding Patterns 186
7.6 Analysis of Folding Patterns 188
7.7 Performance Enhancements 190
7.8 Processor Modules at Work 191
7.9 Global Picture 193
7.10 Alternative Architectures 194
7.11 Conclusions 196

8 Conclusions **200**
8.1 Summary 200
8.2 Contributions 200
8.2.1 Java Workload Characterization 201
8.2.2 A Methodical Design Space Analysis 202

8.2.3 The Identification of Global Architectural Design Principles 202
8.2.4 Stack Dependency Resolution 203
8.2.5 An Operand Extraction Bytecode Folding Algorithm 203
8.2.6 An Adaptive Bytecode Fetch Policy 204
8.2.7 Dynamic Binary Translation 204
8.2.8 A RISC Core Driven by a Deep and Dynamic Pipeline 204
8.2.9 On-Chip Local Variable File 204
8.2.10 A Modified Tomasulo's Algorithm 205
8.2.11 Complex Instruction Handling via a Load/Store Unit 205
8.2.12 A Dual Processing Architecture 205
8.3 Directions for Future Research 206

Bibliography **208**

Appendix A Folding Notation **226**

Appendix B Sequencing Notation **228**

List of Tables

Table 2.1 A comparison between UltraSPARC and JVM architectures. 13
Table 2.2 JVM-supported data types. 21
Table 2.3 JVM addressing modes with examples. 24
Table 2.4 Prefix codes for JBCs. 29
Table 2.5 Classes of JVM instructions. 31
Table 2.6 An example of JVM code generation for a Java program. 32
Table 2.7 Dynamic frequencies of using different instruction classes. 34
Table 2.8 Total execution time for different instruction classes. 48
Table 2.9 The top ranked instructions in execution frequency, execution time per call, and total execution time. 50
Table 2.10 Method invocation statistics. 50
Table 2.11 Observations and recommendations for JVM instruction set design. . 63
Table 2.12 Observations and recommendations for JVM HLL support. 64

Table 3.1 Summary of the recommended design features for Java hardware. . . 89

Table 5.1 Foldable categories of JVM instructions. 106
Table 5.2 Unfoldable JVM instructions. 108
Table 5.3 Information related to each foldable JVM instruction category. 109
Table 5.4 Different folding pattern templates recognized by the folding information generation unit (FIG). 114
Table 5.5 Information generated by the state machine about recognized folding templates. 122
Table 5.6 Folding optimization by combining successive destroyer, duplicator, and/or swaper anchors. 126

Table 6.1 Mapping different anchor instructions to the folding operations performed by the bytecode queue manager (BQM). 135

Table 6.2 Summary of folding operations done in the framework of terminating BQM-internal operation. 136
Table 6.3 JVM opcodes terminated internally at the BQM. 136
Table 6.4 Examples on the folding operations done in the framework of terminating BQM-internal operations. 136
Table 6.5 Summary of folding operations done in the framework of non-terminating BQM-internal operations and corresponding BQM output. 137
Table 6.6 Examples on the folding operations done in the framework of non-terminating BQM-internal operations. 137
Table 6.7 Ministeps involved in performing BQM-internal operations. 141
Table 6.8 Sequencing description of the ministeps involved in performing each of the BQM-internal operations. 142
Table 6.9 Mapping different folding templates to the folding translator unit (FT) output. 147
Table 6.10 Examples on dynamic translation for different folding templates. . . . 148
Table 6.11 Correspondence between no-effect, producer, consumer, and increment JVM categories and the dynamically produced instruction fields. . . . 148
Table 6.12 Mapping different anchor instructions to the operations performed by the local variable file (LVF). 151
Table 6.13 Port usage for different LVF-internal operations. 154
Table 6.14 Mapping different folding templates to the local variable file (LVF) inputs. 155
Table 6.15 Steps required for each LVF-internal operation. 157
Table 6.16 Steps required for RS unit-internal operations. 167
Table 6.17 JVM load/store anchor operations internally implemented by the LS. 171
Table 6.18 Mapping of folding group inputs and destination tag onto the LS internal registers. 173
Table 6.19 Steps needed to perform LS operations. 175

Table 7.1 SPECjvm98 Java benchmark suite summary. 186
Table 7.2 Associating JVM instruction categories with their basic requirements and mapping them to relevant modules in JAFARDD. 198

Table 7.3 Comparison between the three approaches in supporting Java in hardware: direct stack execution, hardware interpretation, and hardware translation. 199

Table 8.1 How JAFARDD addresses the global architectural design principles. . 201

List of Figures

Figure 1.1 Alternative approaches for running Java programs. 3
Figure 1.2 Research methodology and dissertation road map. 9

Figure 2.1 Sample of collected JVM trace components. 16
Figure 2.2 JVM runtime system components. 18
Figure 2.3 Distribution of data accesses by type. 22
Figure 2.4 Usage of type conversion instructions. 23
Figure 2.5 Summary of addressing mode usage. 26
Figure 2.6 Summary of quick execution of instructions. 27
Figure 2.7 Summary of executing instructions that may change the program counter. 36
Figure 2.8 Statistics of conditional branches. 37
Figure 2.9 Distribution of the effect on stack size for each instruction class. . . 37
Figure 2.10 Instruction layout for JVM. 38
Figure 2.11 Distribution of number of bits representing a LV index. 40
Figure 2.12 Distribution of number of bits representing a CP index. 41
Figure 2.13 Distribution of number of bits representing immediates. 42
Figure 2.14 Distribution of dynamic bytecode count per instruction. 43
Figure 2.15 Distribution of number of operands per instruction. 43
Figure 2.16 Distribution of number of bits representing an array index. 44
Figure 2.17 Distribution of number of bits representing the offset and absolute jump-to address. 45
Figure 2.18 Instruction encoding requirements for each instruction class. 47
Figure 2.19 Distribution of levels of method invocations. 51
Figure 2.20 Distribution of number of LVs allocated by individual method invocations. 52

Figure 2.21 Distribution of accumulated LVs allocated through hierarchical invocations. 52
Figure 2.22 Overlapping and non-overlapping stack allocation schemes. 53
Figure 2.23 Summary of individual method invocation stack sizes. 54
Figure 2.24 Summary of stack sizes in hierarchical method invocations. 54
Figure 2.25 Summary of stack size changes due to instruction execution. 55
Figure 2.26 Summary of instruction effects on stack size. 56
Figure 2.27 Summary of stack sizes ignored per method invocation. 57
Figure 2.28 Distribution of invoking different constructors. 58
Figure 2.29 Total execution time per Object, Thread, and String classes. 59
Figure 2.30 Total execution time for object-specific instructions. 60
Figure 2.31 Time requirements for memory allocations. 62

Figure 3.1 Java processors design space. 67
Figure 3.2 Design tree elementary primitives. 67
Figure 3.3 Different hardware design approaches for Java processors. 68
Figure 3.4 Execution engine organization design space. 73
Figure 3.5 Design space of the stack. 77
Figure 3.6 Different ways of organizing classical S-caches. 79
Figure 3.7 Design space tree for parallelism exploitation. 82
Figure 3.8 Design space for supporting Java high-level language features. . . . 86

Figure 4.1 General JBC processing phases. 94
Figure 4.2 Pipeline stages for JBC processing. 94
Figure 4.3 Block diagram of the JAFARDD architecture. 95
Figure 4.4 JBC, RISC, and dual execution processor pipelines. 97
Figure 4.5 A dual processor architecture capable of running Java and other HLLs. 98

Figure 5.1 Steps in executing stack instructions. 101
Figure 5.2 JBC folding example. 102
Figure 5.3 Nested pattern folding example. 104
Figure 5.4 An example showing the OPEX bytecode folding algorithm at work. 111
Figure 5.5 Making incomplete folding groups complete by tagging. 113

Figure 5.6 Checking the OPEX bytecode folding algorithm for dependency violations. 116
Figure 5.7 WAR hazard detection and resolution. 117
Figure 5.8 State diagram for the operation of the FIG unit. 119
Figure 5.9 Parameters and pointers relevant to the bytecode queue and the auxiliary bytecode queue as maintained by the FIG unit. 120

Figure 6.1 Normal and overflow BQ zones. 129
Figure 6.2 Inputs and outputs of the BF and the I-cache. 130
Figure 6.3 Keeping the FIG out of the main pipeline flow path. 132
Figure 6.4 Inputs and outputs of the BQM. 138
Figure 6.5 Inputs and outputs of the BF minicontroller. 139
Figure 6.6 Arrangements made within the BQM to prepare for JBC folding and issuing. 140
Figure 6.7 BQM output assembled from queued JBCs. 142
Figure 6.8 How the BQ looks like after executing each BQM-internal operation. 143
Figure 6.9 Generating producers while executing BQM-internal operations that put JBCs on the bytecode queue manager (BQM) output. 143
Figure 6.10 Tag handling inside the bytecode queue manager (BQM). 144
Figure 6.11 Inputs and outputs of the FT and the instruction format at its output and the entrance of the dynamic scheduling and execution pipeline stage. . 145
Figure 6.12 JAFARDD native instruction format. 145
Figure 6.13 An example of translating a folding group into a JAFARDD instruction. 147
Figure 6.14 Inputs and outputs of the tag generation unit (TG). 152
Figure 6.15 Inputs and outputs of the local variable file (LVF). 154
Figure 6.16 Inputs and outputs of the LVF minicontroller. 156
Figure 6.17 A register entry and a reservation station entry in Tomasulo's algorithm. 160
Figure 6.18 Design space for LV renaming. 160
Figure 6.19 Design space for shelving. 161
Figure 6.20 Design space for instruction dispatch scheme. 162
Figure 6.21 Inputs and outputs of a reservation station (RS) unit. 165

Figure 6.22 Inputs and outputs of the immediate/LV selector. 165
Figure 6.23 Inputs and outputs of the RS unit minicontroller. 166
Figure 6.24 Inputs and outputs of a generic execution unit (EX). 170
Figure 6.25 Inputs and outputs to the LS unit. 172
Figure 6.26 Inputs and outputs to the CDB arbiter. 176

Figure 7.1 A JBC trace example. 179
Figure 7.2 Percentages of occurrence of different instruction categories. 187
Figure 7.3 Percentages of occurrence of different folding cases recognized by the folding information generation (FIG) unit. 188
Figure 7.4 Percentages of patterns that are recognized and folded by JAFARDD. 189
Figure 7.5 Percentages of occurrence of anchors with different numbers of producers (u) and consumers (v). 190
Figure 7.6 Percentages of tagged patterns among all folded patterns. 191
Figure 7.7 Percentages of nested patterns among all folded patterns. 192
Figure 7.8 Percentages of eliminated instructions relative to all instructions and relative to stack instructions (producers and non-anchor consumers) only. . 193
Figure 7.9 Speedup of folding. 194
Figure 7.10 Percentages of occurrence of different folding operations performed by the bytecode queue manager (BQM). 195
Figure 7.11 Percentages of occurrence of different folding patterns at the output of the folding translator unit (FT). 196
Figure 7.12 Percentages of occurrence of different operations performed by the local variable file (LVF). 197
Figure 7.13 Percentages of occurrence of different folding patterns processed by the load/store unit (LS). 198
Figure 7.14 Percentages of usage of different execution units (EXs). 199

List of Abbreviations

ALU	Arithmetic and Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BC	Bytecode Counter
BF	Bytecode Fetch Unit
BQ	Bytecode Queue
BQM	Bytecode Queue Manager
BR	Branch Unit
CDB	Common Data Bus
CFG	Complete Folding Group
CISC	Complex Instruction Set Computer
CN	Constant
CP	Constant Pool
CPI	Average Clock Cycles Per Instruction
CPU	Central Processing Unit
D-cache	Data Cache
EX	Execution Unit
FIG	Folding Information Generation Unit
FIQ	Folding Information Queue
FP	Floating Point Unit
FRAME	Stack Frame Base Register
FT	Folding Translator
FSM	Finite State Machine
GUI	Graphical User Interface
HLL	High-Level Language
I-cache	Instruction Cache
IFG	Incomplete Folding Group

ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
JAFARDD	A Java Architecture based on a Folding Algorithm, with Reservation Stations, Dynamic Translation, and Dual Processing
JBC	Java Bytecode
JDK	Java Development Kit
JIT	Just-in-Time
JPC	Java Instruction Per Clock Cycle
JVM	Java Virtual Machine
LS	Load/Store Unit
LV	Local Variable
LVF	Local Variable File
MI	Multi-Cycle Integer Unit
MMU	Memory Management Unit
OOO	Out-Of-Order
OPEX	Operand Extraction
OPTOP	Top of Operand Stack Register
OS	Operating System
PC	Program Counter
RAW	Read-After-Write
RISC	Reduced Instruction Set Computer
RS	Reservation Station
S-cache	Operand Stack Cache
SI	Single-Cycle Integer Unit
SIMD	Single Instruction Multiple Data
TLP	Thread Level Parallelism
TG	Tag Generation Unit
VARS	Local Variable Base Register
VLIW	Very Long Instruction Word
WAR	Write-After-Read
WAW	Write-After-Write

Trademarks

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Trademarks and registered trademarks used in this work, where the author is aware of, are listed below. All other trademarks are the property of their respective owners.

aJ-100	is a registered trademark of aJile Systems, Inc.
Bigfoot	is a registered trademark of Digital Communication Technologies (DCT) company
Cjip	is a registered trademark of Imsys company
GP1000	is a registered trademark of Imsys company
Java	is a registered trademark of Sun Microsystems, Inc.
JEM1	is a registered trademark of Rockwell Automation, Inc.
JEM2	is a registered trademark of Rockwell Automation, Inc.
JSCP	is a registered trademark of Negev Software Industries (NSI), Ltd.
Lighfoot	is a registered trademark of Digital Communication Technologies (DCT) company
MAJC	is a registered trademark of Sun Microsystems, Inc.
microJava	is a registered trademark of Sun Microsystems, Inc.
MIPS	is a registered trademark of MIPS technologies, Inc.
PA-RISC	is a registered trademark of HP company
PSC1000	is a registered trademark of Patriot Scientific (PTSC) company
picoJava-I	is a registered trademark of Sun Microsystems, Inc.
picoJava-II	is a registered trademark of Sun Microsystems, Inc.
RX000	is a registered trademark of MIPS technologies, Inc.
StrongARM	is a registered trademark of ARM company
VAX	is a registered trademark of HP company

Acknowledgement

All praise be to Allah the High, "who teacheth by the pen, teacheth man that which he knew not.", Quran[96:4, 96:5]. I say what Prophet Solomon said: "··· O my Lord! so order me that I may be grateful for Thy favours, which thou hast bestowed on me and on my parents, and that I may work the righteousness that will please Thee: And admit me, by Thy Grace, to the ranks of Thy righteous Servants.", Quran[27:19]

I would like to express my deepest appreciation to my dissertation supervisors, Dr. Fayez Gebali and Dr. Kin Li. I greatly valued the freedom and flexibility which they entrusted me, the generous financial support they have provided me, and the environment they made available for me for learning and quality research. I also thank them for helping me improve my technical writing skills. I have benefited substantially as a result of their many comments and reviews. They have always been patient and supportive. Perhaps most of all, they were as much friends as they were mentors. Their confidence in me and my abilities was an incentive for me to finish this work.

Next, I am very grateful to Professors Nikitas Dimopoulos and Michael Miller for serving on my supervisory committee, and Dr. Hussein Alnuweiri for agreeing to be the external examiner in my oral examination. Their time and effort are highly appreciated.

I will be forever indebted to: Dr. Dave Berry, Dr. Nigel Horspool, Dr. Micaela Serra, Dr. Ali Shoja, Dr. Issa Traore, and Dr. Geraldine Van Gyn, for seeing the potential in me, and giving me the opportunity to gain an enormous teaching experience and improve it.

This dissertation would never have been written without the generous help and support that I have received from numerous colleagues along the way, I would now like to take this opportunity to express my sincerest thanks to: Mohamed Abdallah, Ahmad Afsahi, Mostofa Akbar, Casey Best, Zeljko Blazek, Claudio Costi, John Dorocicz, Tim Ducharme, Hossam Fattah, Jeff Hornsberger, Ekram Hossain, Ken Kent, Erik Laxdal, Akif Nazar, Stephen Neville, Newaz Rafiq, Hamdi Sheibani, Caedmon Somers, and Michael Zastre.

At this point, I would like to thank: Lynne Barrett, Moneca Bracken, Steve Campbell, Isabel Campos, Nancy M. Chan, Kevin Jones, Vicky Smith, Mary-Anne Teo, Nanyan Wang, and Xiaofeng Wang, all of whom had helped me in some capacity along the way.

Lastly, my special thanks extend to all my friends: Yousry Abdel-Hamid, Saleh Arhim, Mohamed Fayed, Yousif Hamad, and Mohamed Yasein, who made me feel at home!

Dedication

To

all my students,

who always motivate me

to excel!

Chapter 1

Introduction

Java is a general-purpose object-oriented programming language with syntax similar to C++. The Java language and its virtual machine are excellent applications of dynamic object-oriented ideas and techniques to a C-based language.

Java was introduced to aid the development of software in heterogeneous environments, which requires a platform-independent and secure paradigm. Additionally, to enable efficient exchange over the Internet, Java programs need to be small, fast, secure, and portable. These needs led to a design that is rather different from the established practice. However, instead of serving as a testbed for new and experimental software technology, Java combines technologies that have already been tried and proven in other languages.

The authors of Java wrote an influential *white paper* explaining their design goals and accomplishments [1]. In summary, Java is most notable for its simplicity, robustness, safety, platform independency and portability, object-orientation, strict type checking, support of runtime code loading, forbidden direct memory access, automatic garbage collection, structured exception handling, distributed operation, and multithreading. Besides this, Java also rids itself of many extraneous language features thus offering safe execution [2, 3, 4, 5]. These characteristics, encapsulated in the *write once, run anywhere* promise, make Java an ideal tool and a current de facto standard for writing web applications that can run on any client CPU. Effectively, Java operates in a *server-based* mode: *zero-cost* client administration with applications downloaded to clients on demand.

This introduction chapter is organized as follows. Section 1.1 presents a brief introduction to the Java virtual machine. Research motivations and objectives are summarized in Sections 1.2 and 1.3. Section 1.4 discusses the challenges expected in designing hardware support for Java. Finally, we detail the adopted methodology and dissertation road map.

1.1 The Java Virtual Machine

Java's portability is attained by running Java on an intermediate virtual platform instead of being compiled to code specific for a particular processor [6, 7, 8]. The Java Virtual Machine (*JVM*) is the name coined by Sun Microsystems for the Java underlying hypothetical target architecture. The JVM is virtual because it is defined by an implementation-independent specification. Java programs are first compiled to an intermediate processor-independent bytecode representation. Java bytecodes (*JBCs*) are interpreted at runtime to native code for execution on any processor-operating system (*OS*) combination that has an implementation of the JVM specification either emulated in software or supported directly in the hardware. JVM provides a cross-platform package not only in the horizontal dimension, but also in the temporal one. Regardless of what new platforms appear only the JVM and the Java compiler need to be ported, not the applications [4, 6, 7, 8, 9, 10, 11].

1.1.1 Different Bytecode Manipulation Methods

The execution of a Java program could take one of many alternative routes that map the virtual machine to the native one [12]. Figure 1.1 contrasts these routes:

- **Interpretation** Java interpreters translate JBCs on the fly into the underlying CPU native code, emulating the virtual machine in software. Interpretation is simple, relatively easy to implement on any existing processor, and does not require much memory. However, this involves a time-consuming loop that fetches, decodes, and executes the bytecodes until the end of the program. This software emulation of the JVM results in executing more instructions than just the JBCs, which affects performance significantly. In addition, the software decoder, which is usually implemented as one big `case` statement, results in inefficient use of hardware resources. Moreover, interpretation decreases the locality of the I-cache and branch predication as it considers all JBCs as just data [13].
- **Just-in-time compilation (JIT)** JIT compilers translate JBCs at runtime into native code for future execution [14, 15, 16]. However, they do not have to translate the same code over and over again because they cache the translated native code. This can result in significant performance speedups over an interpreter. But a JIT compiler sometimes takes an unacceptable amount of time to do its job and expands code

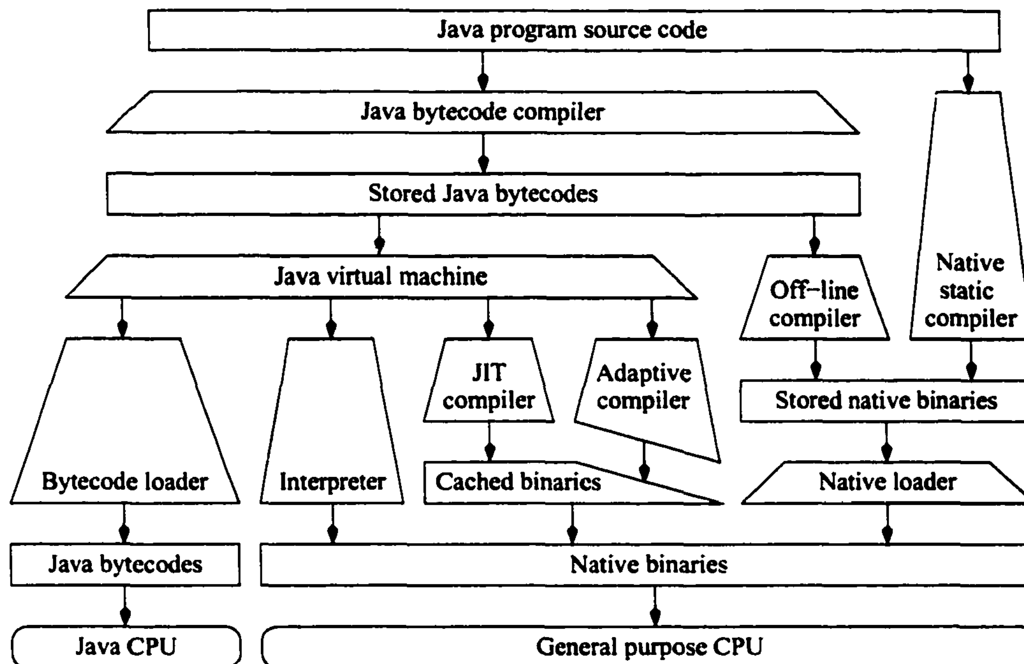


Figure 1.1. *Alternative approaches for running Java programs.*

size considerably. Generally a JIT compiler does not speed up Java applications significantly because it does not incorporate aggressive optimizations in order to avoid time and memory overheads. Additionally the generated code depends on the target processor which complicates the porting process [17, 18, 19, 20].

- **Adaptive compilation** An adaptive compiler behaves like a programmer who profiles a piece of code and optimizes only its time-critical portions (hot spots). Employing an adaptive complier, the virtual machine only compiles in a JIT fashion and optimizes the hot spots. Although a hot spot can move at runtime, only a small part of the program is compiled on the fly. Thus, the program memory footprint remains small and more time is available to perform optimizations [21].
- **Off-line compilation** These compilers convert JBCs to native machine code just before execution, and this requires all classes to be distributed and compiled prior to use. Since this process is performed off-line and the resultant code is saved on a disk, additional time may be devoted to optimizations [21].
- **Native compilation** Another way of getting Java programs executed is to compile them to the underlying machine native code. This makes Java lose its platform inde-

pendence. As an alternative of going directly to processor native code some compilers produce C code. This allows using all advanced C optimization techniques of the underlying C compiler [21].

- **Direct native execution** Dedicated processors that directly execute JBCs without the overhead of interpretation or JIT compilation could yield the best performance for more complex Java applications. They can deliver much better performance by providing special Java-centric optimizations that make efficient use of processor resources like the cache and the branch prediction unit [22]. Hardware support for Java is the topic of study in this dissertation.

1.1.2 Performance-Hindering Features

Java's unique features that make it efficient also contribute to its poor performance:

- **Productivity** The Java environment provides some features that increase programmer productivity at the cost of runtime efficiency, e.g., checking array bounds.
- **Abstraction** The JVM contains a software representation of a CPU, complete with its own instruction set. This hardware abstraction contributes a large factor to Java's slow performance.
- **Stack architecture** The JVM is based on a stack architecture. This architecture was chosen to facilitate the generation of compact code for reasons such as minimizing bandwidth requirements and download times over the Internet, and to be compatible on different platforms [23, 24]. However, stack referencing creates a bottleneck that adversely affects performance.
- **Runtime code translation** JBCs are translated either by an interpreter or a JIT compiler at runtime into native code. This extra step in execution slows down the runtime performance of Java programs.
- **Dynamic nature** Being a dynamically bound object-oriented language, Java suffers from the overhead of loading classes.
- **Security** Effective, but restrictive, security measures demand many runtime checks that affect performance.
- **Exceptions** Java employs a large number of runtime checks and exception generation, including array-bounds exception, null pointers, division-by-zero, illegal string-

to-number conversions, invalid type casting, etc. These consume a significant portion of the execution time.

- **Multithreading** Incorporating multithreading at the language construct level imposes synchronization overhead.
- **Garbage collection** Managing the memory system at runtime consumes CPU time and resources.

1.2 Motivations

Java's virtual arrangement and its special features that make it ideal for network software applications have a tremendous negative impact on its performance [25, 26]. The performance gap between JBCs and the optimized native binaries is very large. To execute Java codes, the normal instruction cycle (fetch, decode, etc.) is repeated twice: at the virtual machine and at the host hardware. Radhakrishnan *et al.* have shown that a Java interpreter requires, on the average, 35 SPARC instructions to emulate each JBC instruction [25]. Adding to that the overhead of fetching and decoding the bytecodes at the virtual machine level, we conclude that the main bottleneck in executing Java is the emulation in software. Compiling Java in advance can greatly improve its performance. However, this method leaves Java no more portable than other programming languages.

A number of approaches have been proposed to enhance Java performance, and among these hardware support has many distinct advantages [27, 28, 29, 30, 31, 32, 33]. Reducing the virtual machine thickness by moving some of its functionality into hardware will enhance Java's performance [12, 22, 34, 35, 36, 37, 38, 39].

Java is neither the first high-level language (*HLL*) to run on top of a virtual machine nor the first one to be implemented in hardware. The idea to convert virtual machines to real ones or to support a HLL in hardware has been tried before but with mixed results. Even in the early days of computing, computer designers were looking for ways to support HLLs. This led to three distinct periods. The stack architectures that were popular in 1960s represented the first good match for HLLs. However, they were withdrawn in the 1980s except for the Intel $x86$ floating point architecture that combines a stack and a register set. Java hardware, however, is now reviving these concepts again [40, 41]. The second period, which took place in 1970s, replaced some of the software functionality within hardware in

order to reduce the software development and execution costs. This provided high-level architectures that could simplify the task of software designers, like that in the DEC's VAX. In the 1980s, the third period, sophisticated compilers permitted the use of simple load-store RISC machines [42].

If we extrapolate to the next decade of computer design, we might expect new directions that serve the contemporary software features and anticipated computational workloads. Modern applications incorporate new paradigms (e.g., object-orientation), evolving functionality (e.g., internetworking and the web), vital modules (e.g., garbage collection), and necessary application requirements (e.g., security). In our opinion, these features will probably require processor designers to support them at the hardware level. This has triggered Java, multimedia, digital signal processing, and network extensions to general-purpose processor design which will lead to an overall system acceleration and better performance.

The transition from the traditional desktop computing paradigm to a secure, portable model opens up an unprecedented opportunity for Java processors [43, 44, 45, 46, 47]. Java processors will enrich the design of embedded multimedia devices and smart cards, making the on-demand delivery of a wide variety of services a reality [48, 49, 50, 51, 52]. Applications like e-commerce, remote banking, wireless hand-held devices, information appliances, and Internet peripherals are just a few examples of systems awaiting for and would benefit from research that provides hardware support for Java [53, 54, 55, 56, 57, 58, 59, 60].

1.3 Research Objectives

This dissertation explores the feasibility of accommodating Java execution in the framework of a modern processor. We present the *JAFARDD* processor, *a Java Architecture based on a Folding Algorithm, with Reservation Stations, Dynamic Translation, and Dual Processing*, to accelerate Java bytecode execution. Our research aims at designing processors that:

- **Provide better Java performance** Tailoring general-purpose processors to Java requirements will enable them to deliver much better Java performance than processors designed to run C or any other language.

- **Are general** By providing primitives required for generic HLL concepts, these processors will also be suitable for non-Java programming languages.
- **Implement the Java virtual machine logically** A logical hardware comprehension capability of the JBCs will narrow the semantic gap between the virtual machine and the native one. This allows Java's distinguished features to be efficiently utilized at the processor level while continuing to support other programming languages.
- **Utilize a RISC core** Instead of having a complete Java-specific processor, dynamically translating Java binaries to RISC instructions facilitates the use of a typical RISC core that executes simple instructions quickly. This approach also enables the exploration of instruction level parallelism (*ILP*) among the translated instructions using well established techniques and facilitates the migration to Java-enabled hardware.
- **Handle stack operations intelligently** Innovative ideas like bytecode folding and dynamic translation, will be incorporated to accommodate stack operations in a register-based, efficient RISC framework. Our objective is to provide a stackless architecture that is capable of executing stack-based JBCs. This will reduce the negative impact of the JVM's stack architecture.
- **Extract parallelism transparently** From the programmer's point of view the processor will appear as a JVM. However, parallelism will be extracted transparently through techniques like bytecode folding with no programming overhead.

1.4 Design Challenges

Designing processors that are optimized for Java brings up some new ideas in hardware design that have not been addressed effectively and completely before, and thus are challenging for any such an attempt [12, 22]. These hardware optimization issues include:

- **Maintaining processor generality** Although Java-enabled processors are required to support Java itself, performance degradation for any non-Java application might not be affordable. This necessity is a challenge for any microarchitecture design.
- **Handling JVM's variable-length instructions** JVM's variable instruction length makes instruction fetching and decoding difficult as it requires a large amount of pre-decoding and caching of previous execution properties.

- **Overcoming stack architecture deficiency** In a direct JVM stack realization, stack access consumes extra clock cycles. Furthermore stack referencing introduces virtual dependency between successive instructions that limit ILP. Matters become worse with processors that do not have an on-chip dedicated stack cache as the data cache has to be accessed during the processing of almost every instruction. This consumes more clock cycles especially if cache misses are encountered. Therefore, innovative design ideas are required to handle the stack bottleneck.
- **Processing complex JVM instructions** Although JVM's intermediate instruction set is not at the same complexity level as that of an HLL, it does contain some operations that are too complex than a regular CISC or RISC instruction (e.g., method invocations and object-oriented instructions). It is a challenge to achieve a high level of performance given the overhead of these instructions. Their execution consume many clock cycles and involve a number of memory accesses.
- **Coping with naturally sequential operations** Java uses a large number of inherently sequential and sophisticated operations, e.g., method invocations, constant pool resolution. These operations require many clock cycles and memory references for completion. Coping with them in an inexpensive way constitutes a major challenge for achieving a high processing throughput [61].
- **Managing excessive hardware requirements** Achieving a high level of performance by supporting Java in hardware requires more on-chip modules. This extra hardware may slow down the execution speed and result in a large core die size making it a challenge to compete with other RISC processors.

1.5 Methodology and Road Map

The work performed in this research is best presented and viewed as a seven-stage process, as shown in Figure 1.2, which also highlights the contributions of each stage, as well as the iterative feedback paths in the design process. Work done at each stage is organized and presented in a subsequent chapter.

Designing hardware for Java requires an extensive knowledge of the JVM internals. At the first stage, *benchmarking the Java virtual machine*, we conducted a comprehensive behavioral analysis of the Java instruction set architecture through benchmarking. Meaningful

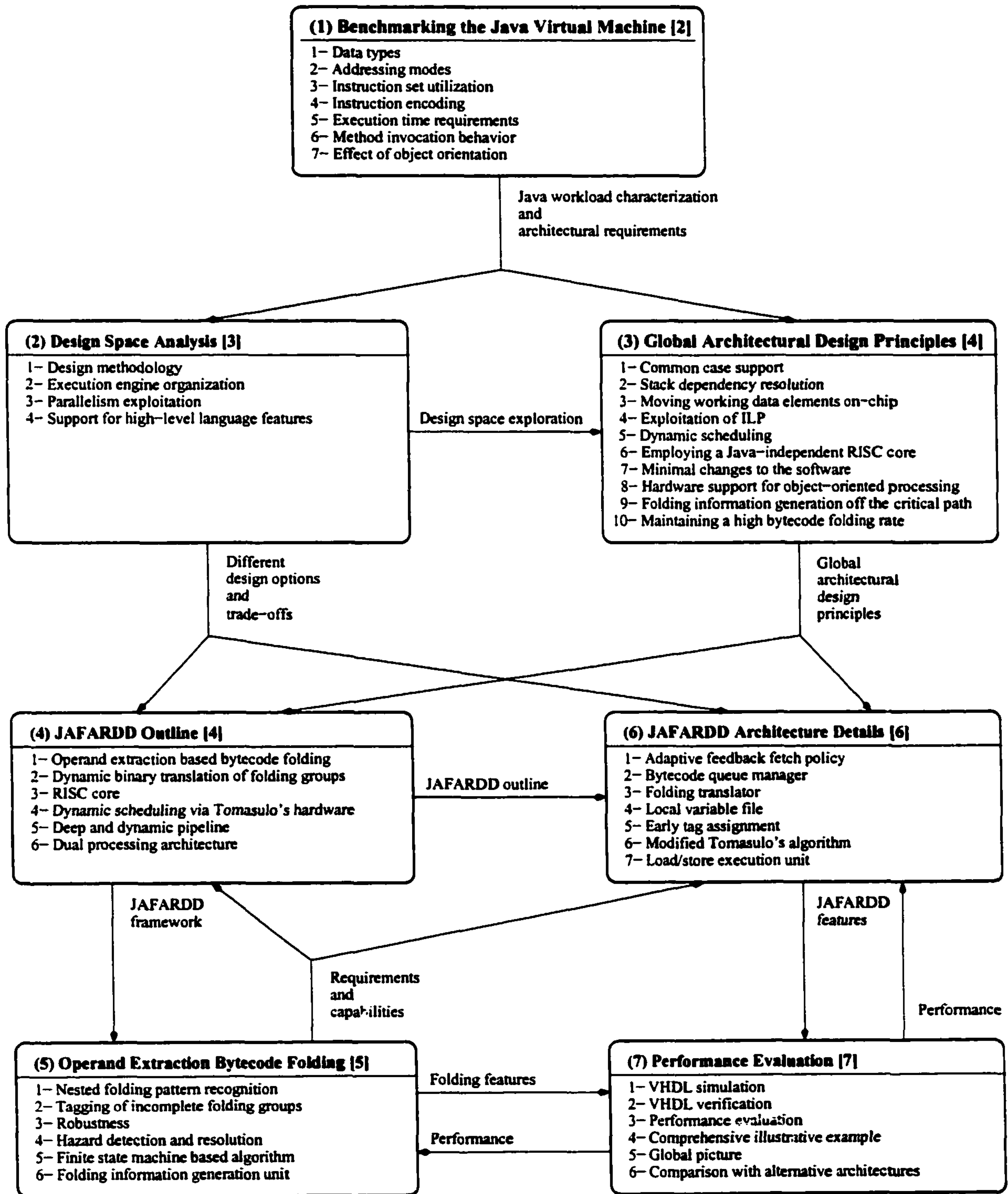


Figure 1.2. *Research methodology and dissertation road map.*

The number in ( ) indicates the stage, whereas the number in [ ] indicates the chapter in the dissertation.

information about access patterns for data types and addressing modes, instruction set utilization, instruction encoding, execution time requirements, method invocation behavior, and the effect of object orientation were collected and analyzed. This stage, presented in Chapter 2, resulted in a clearer understanding of the Java workload characterization and the architectural requirements of Java hardware.

The second stage, *design space analysis*, included a global analysis of the design space of hardware support for Java. At this stage, presented in Chapter 3, we explored different hardware design options that are suitable for Java by examining the design methodology, execution engine organization, parallelism exploitation, and support for HLL features. We weighed different design alternatives and highlighted their trade-offs.

Chapter 4 documents two stages: *global architectural design principles* and *JAFARDD outline*. We compiled the Java workload characterization obtained in the first stage with the design space exploration obtained in the second stage into a list of architectural design principles at the global level that are necessary to ensure JAFARDD can execute Java efficiently. Based on the outcome of these two stages, we proposed the JAFARDD processor.

Results gathered from benchmarking the JVM confirmed that the main bottleneck in executing Java is the underlying stack architecture. To overcome this deficiency, we have introduced the *Operand extraction bytecode folding* algorithm in the fifth stage. This folding algorithm permits nested pattern folding, tolerates variations in folding groups, and detects and resolves folding hazards completely. By incorporating this algorithm into a Java processor, the need for, and therefore the limitations of a stack are eliminated. Chapter 5 presents the operation details of this folding algorithm.

In the sixth stage, presented in Chapter 6, the *JAFARDD architecture details* are studied. We discussed the distinguishing features of JAFARDD in this chapter emphasizing the instruction pipeline modules.

Finally in the *performance evaluation* stage, the functionality of JAFARDD was successfully demonstrated through VHDL modeling and simulation. Benchmarking of our proposal using SPECjvm98 to assess performance gains was also carried out. Chapter 7 summarizes the findings.

The dissertation ends with conclusions and future work in Chapter 8.

Chapter 2

Java Processor Architectural Requirements: A Quantitative Study

2.1 Introduction

Designing hardware for Java requires an extensive working knowledge of its virtual machine organization and functionality. The JVM instruction set architecture (*ISA*) defines categories of operations that manipulate several data types, and uses a well defined set of addressing modes [6, 7, 62]. The JVM specification defines the instruction encoding mechanism required to package this information into a bytecode stream. It also includes details about the different modules needed to process these bytecodes. At runtime, the JVM implementation and the execution environment affect the instruction execution performance. This is manifested directly in the wall-clock time needed to perform a certain task and indirectly in the various overheads associated with executing the job [8].

While the JVM ISA shares many general aspects with traditional processors, it also has its own distinguished features, because the JVM is an intermediate layer for an HLL. For example, a generic branch prediction hardware mechanism affects the processing of all programming languages, including Java. On the other hand, method invocation handling could be specific to JVM's stack model.

The goal of this chapter is to conduct a comprehensive behavioral analysis of the JVM ISA and its support for Java as an HLL. Benchmarking the Java ISA reveals its execution characteristics. We will analyze access patterns for data types and addressing modes, as well as instruction encoding parameters. Additionally, the characteristics of executed instructions will be measured and the utilization of Java classes will be assessed. Recommendations for hardware improvements and encoding formats will be provided.

In order to carry out the analysis, a Java interpreter is instrumented to produce a benchmark trace. Meaningful data is collected and analyzed. General architectural requirements for a Java processor are then suggested. In doing this study, we followed the methodology used by Patterson and Hennessy in studying the instruction set design [42].

The study of the Java ISA is an important part in improving its performance. Our rationale for conducting such a study is based on the observation that modern programs spend 80-90% of their time accessing only 10-20% of the ISA [42]. To be most effective, optimization efforts should focus on just the 10-20% part that really matters to the execution speed of a typical program [63]. The results collected here may be used to devise a bytecode encoding scheme that is suitable for a broad range of Java-supporting CPUs. The results may also affect the internal datapath design of a Java architecture.

This chapter is organized as follows. The experimental framework is explained in Section 2.2. Section 2.3 is a brief introductory description of the JVM ISA. The analysis of the JVM instruction set design is discussed in Sections 2.4 to 2.7. Sections 2.6 to 2.10 examine HLL support at the processor level through the analysis of instruction set utilization, instruction execution time, method invocation behavior, and the effect of object orientation. Section 2.11 draws related conclusions.

2.2 Experiment Framework

Here, we explain how the code trace is generated and behavioral information is extracted.

2.2.1 Study Platform

The machine used in this study is an UltraSPARC $I/140$ that has a single UltraSPARC I processor running at 143 MHz with 64 Mbytes of memory. The OS is Solaris 2.6 Hardware $3/98$. We used the Java compiler and interpreter of Sun's Java Development Kit (*JDK*) version 1.1.3. In order to gain some insights into the benchmark platform, Table 2.1 examines the architectural features of UltraSPARC that map well to some JVM's characteristics, which might affect JBCs execution [64].

Table 2.1. *A comparison between UltraSPARC and JVM architectures.*

UltraSPARC features	Corresponding JVM characteristics
64-bit architecture	32-bit architecture
32-bit instruction length	Variable number of bytecodes
32-bit registers	Majority of the data types are 32 bits, which could be mapped easily on these registers; the rest are 64 bits
Supports 8−, 16−, 32−, and 64−bit integers and single and double precision floating points	Supports all these data types, plus characters, references, and return values
Provides signed (in two's complement) and unsigned operations	Does not provide unsigned operations. Signed operations are done in two's complement
Requires memory alignment	Does not enforce any alignment
Big endian architecture	Big endian virtual architecture
Instructions use triadic register addresses	This could be used in software folding [65]
Addressing modes: register-register and register-immediate	Local variables and stack entries could be mapped onto processor registers to use these two addressing modes
Stack is allocated into memory; no hardware stack is provided	A stack-based machine
Supports both single and double precision floating-point operations	Supports both single and double precision floating-point operations
Instruction classes: load/store, read/write, ALU, control flow, control register, floating point operations, and coprocessor operations	Instruction classes: scalar data transfer, ALU, stack, object manipulation, control flow, and other complex ones
Incorporates dynamic branch prediction	This could help executing conditional branches
Windowed register files	This could help in nested method invocation set-up
Has on-chip graphics support	JVM class libraries include a comprehensive set of graphics packages
Provides atomic read-then-set memory operation and spin locks for synchronization	Multithreading synchronization is done via monitors, which could benefit from these hardware primitives
Has an on-chip MMU	Requires extensive memory management at runtime
Solaris does not support garbage collection	Incorporates garbage collection

2.2.2 Trace Generation

A Java interpreter was instrumented by inserting probes to produce the required trace. This enabled information gathering when the interpreter fetched JBCs and started executing them. Inserting trace-collecting statements requires access to the source code of a JVM

implementation. For this purpose, we obtained a licensed source release for JDK version 1.1.3 from Sun [66].

2.2.2.1 JDK Core Organization

JDK is organized into a number of modules to implement the JVM. The module of interest is the core itself, which is responsible for executing Java methods' bytecodes. This core is implemented in a single file `executeJava.c`. The main method in it is `executeJava()`. The body of this function is a long infinite loop that emulates the work of a processor. The pseudocode in Algorithm 2.1 shows the different stages in this infinite loop. The execute stage may involve fetching other bytecodes to retrieve any required operands and/or writing results back. If the executed opcode involves branching to a new location or invoking another method, the program counter is updated. The loop also contains some other advanced stages to deal with exceptions and monitors.

Algorithm 2.1 *Pseudocode for the JVM execution engine.*

```
Initialize the program counter and the stack top pointer;
while (true) {
    /* Fetch: retrieves the bytecode that the PC points to*/
    opcode = memory [program counter];
    /* Decode: interprets the fetched opcode and picks the appropriate action */
    switch (opcode) {
        . . .
        case opcode_xxx:
            START_TIMER; /* statement added for trace generation*/
            /* Execute: issues native instructions that perform the opcode function */
            opcode execution statements;
            STOP_TIMER; /* statement added for trace generation*/
            TRACE_COLLECTING_STATEMENTS; /* statement added for trace collection*/
            /* Pointer update*/
            adjust program counter;
            adjust stack top pointer;
        break;
        . . .
```

```
    }
    advanced stages;
}
```

2.2.2.2 Core Instrumentation

The Sun source code uses a simple trace statement that produces limited information. We changed this to collect more data. This trace statement was placed inside the switch-case statement just after opcode execution and before pointers update, as shown in Algorithm 2.1. An accurate mechanism was needed to determine the dynamic execution time for each of the executed opcodes. To ensure minimal overhead, the timer started just before and stopped immediately after opcode execution, as shown in Algorithm 2.1. Another critical issue was the need to use a high resolution timer since most instructions execute in the order of microseconds. A high resolution timer in Sun Solaris was used. Accessed via the system call `gethrtime()`, the timer reports time in nanoseconds.

2.2.2.3 Execution Trace Components

Figure 2.1 shows an annotated trace sample, including examples of all the collected components. The trace shown is more than a collection of dynamically executed instructions; it contains information about the stack state, branch prediction, etc. The binary information is converted into an easy to understand symbolic form (e.g., each constant pool (*CP*) index was accompanied by the symbolic name of the referenced item). The figure illustrates the level of details in the collected information, which may help in making hardware decisions.

2.2.3 Trace Processing

A benchmark was run on the instrumented bytecode interpreter. This produced a collection of raw data stored in the form of lines, each documenting the trace of executing one JVM instruction. These data were then consolidated in an analyzable form for each component of interest. Each JVM instruction has certain properties, including the mnemonic, opcode, addressing mode, class and subclass, sub-subclass, data type, and data type size. Information was extracted from each trace line and stored in a JVM database. Queries were applied to the database system to obtain statistical information, such as data type utilization, etc. Statistical information was then converted into a graphical form for easier interpretation.

Current thread start address	Program counter	Construction execution statement	Class name
EE3000A8	0	execute_java_constructor new	java/lang/String

Method entering time	Required stack size	Required number of local variables	Method entering instruction	Method name
243690440025	8	4	Entering	String.length

Current thread start address: EE3000A8														
Program counter	Time consumed (nsec)	No. of JBCs	Stack change†	Opcode	Accessed local variable number‡	Immediate				Branch status⋆	CP index	Array index	Stack flow direction⊗
						Jump offset	Jump-to address	No. of arguments	immediate				
2BB14	1070	1	+1	aload_0	$r0								↑
45701	1776	4	-1	wide istore	$r256								↓
47900	818	1	·0	return									↔
48D5F	1385	3	0	goto		36	5503D			T			↔
2EB2D	7204	1	-3	aastore								?[0]⋄	↓
3514B	1665	2	1	bipush					16				↑
42A88	3651	3	-1	invokevirtual_quick_w				(1)			(#27)∘		↕
4166C	8491	3	-1	ifnonnull		19	54D9C			NT			↓

Method leaving time	Method leaving instruction	Method name
243690440954	Entering	String.length

- † Stack change is positive in case of pushing. The '·' at the start of the stack change means that this instruction returns from a method and the stack is ignored.
- ‡ Local variables are organized as a set of registers, with a '$' marker.
- ⋆ Branch status: T and NT mean *taken* and *not taken*, respectively.
- ∘ CP indices are identified by a '#' marker.
- ⋄ Array indices are identified by a '?' marker.
- ⊗ Effect on stack size: ↑, ↓, ↕, ↔ mean push, pop, push and pop, and no effect, respectively.

Figure 2.1. *Sample of collected JVM trace components.*

2.2.4 Benchmarking

Pendragon Software's CaffeineMark 3.0 was used as the benchmark since it is computationally interesting and exercises various aspects of the JVM, and was the most commonly used benchmark at the time of our study [67]. It is a synthetic benchmark suite that runs nine tests: *Sieve* (finds prime numbers using the classic Sieve of Eratosthenes), *Loop* (uses sorting and sequence generation to measure loop performance), *Logic* (executes decision-making instructions), *String* (manipulates strings to test symbolic processing capabilities), *Float* (simulates a $3D$ rotation of objects around a point), *Method* (executes methods recursively to see how well JVM handles method invocations), *Graphics* (draws random rectangles and lines), *Image* (draws a sequence of $3D$ graphics repeatedly), and *Dialog* (writes a set of values into labels and editboxes on a form). CaffeineMark involves extensive use of computations, in both integer and floating point formats. It also uses the graphic classes of the application programming interface (*API*). In addition, it employs a large amount of data

movement, as well as garbage collection in array manipulation. It also evaluates the performance of executing object-oriented instructions. An important advantage of CaffeineMark is that it does not require any user input, so it is ideal for measuring user-independent performance. It consists of 21 classes having a total of 32,094 static JBCs. The generated trace produced 7,522,280 trace lines, each representing an individual executed instruction.

2.3 JVM Instruction Set Architecture (ISA)

The JVM specification defines an ISA and some functional units to be implemented either virtually (using software emulation) or physically (as a part of the host processor). The ISA defines categories of operations that process several data types and a well defined set of addressing modes. The JVM specification also defines the instruction encoding mechanism required to package this information into the bytecode stream. Figure 2.2 gives a detailed block diagram of the software implementation of JVM. The difference between this organization and a hardware implementation is that in the software schemas, JVM modules are emulated, while in hardware-based Java systems, different modules can exist physically within the processor.

JVM is based on a stack architecture. As a zero-address machine with no accumulator, all operations on data utilize the stack. One of the most pervasive architectural impacts of the stack architecture is on instruction encoding and the number of instructions needed to perform a task. Instruction format tends to be simple, easy to encode and yields a small average instruction length. However, it requires more instruction count. But above all, the stack organization improves portability by making JVM implementation on a wide variety of host architectures possible [23]. It is worth mentioning that the instruction set's stack-centered approach was chosen over a register-centered approach since it lends itself to architectures with only a few or irregular registers, e.g., Intel $80x86$.

In this section, we highlight the architectural view of the JVM[1].

[1]Discussion of native method support, exception handler, threading manager, security manager, and garbage collector is out of the scope of this chapter. Here, we discuss only the modules in Figure 2.2 that are relevant to this dissertation.

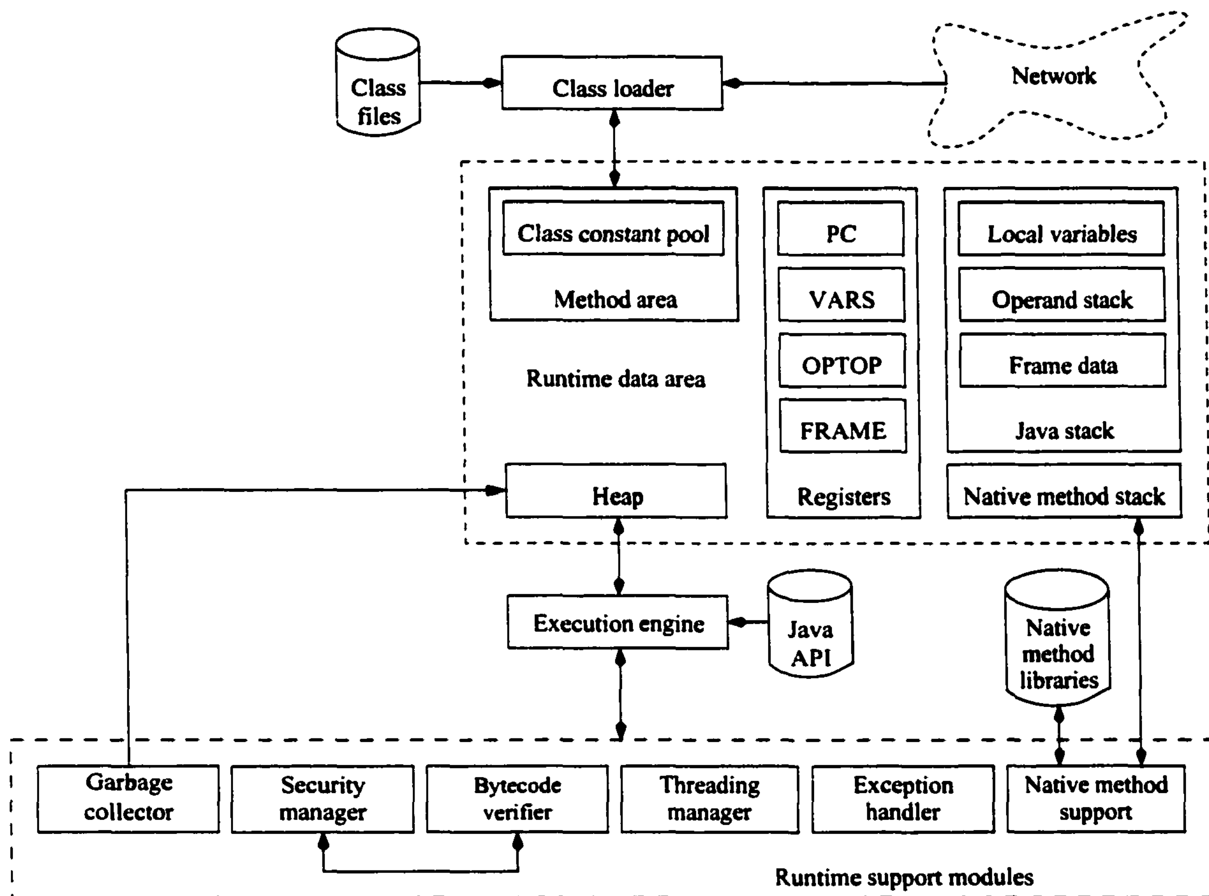


Figure 2.2. *JVM runtime system components.*

2.3.1 Class Files and Their Loader

Java programs are compiled into *Java class files*. This is the data format used to transmit Java programs locally or over the Internet. Class files contain several critical pieces of code and static data: class (or interface) method bytecodes, a symbolic reference to the class's superclass, a list of the fields defined by the class, a class CP, and any other data required by the runtime system [68].

JVM specification precisely defines the layout of the information stored in the class file. This is to guarantee transportability to all machines. Once loaded into the memory, however, this information can be stored in any format needed by a particular JVM implementation.

The class loader module is employed to load classes and interfaces into a runtime in-

stantiation of JVM. Some classes, such as built-in classes are loaded from the disk. Others may be loaded over the Internet. Specifically, class loaders are responsible for loading, linking (including verification, preparation, and resolution), and initialization of classes. These actions are usually carried out dynamically the first time a class is referenced.

2.3.2 Execution Engine

The Java execution engine acts as the machine's virtual processor. This piece of software or hardware receives a stream of bytecodes that make up a method, decodes them, and actually carries out the instructions contained in them.

The JBCs are executed one at a time. An execution engine fetches an opcode and its operands, if any. It executes the action specified by the opcode and then fetches the next opcode. Execution of bytecodes continues until a thread completes either by returning from its starting method or if there is no exception thrown.

2.3.3 Runtime Data Areas

As Figure 2.2 shows, JVM memory hierarchy is divided logically into the four Java runtime data areas as listed below (in addition to a native method stack). For software implementations of the JVM, all these data areas reside in the main memory. However, hardware implementations may elect to move some or all parts into hardware.

- **Method area** Information extracted from parsing loaded class files (such as JBCs, types, static data, etc.) is stored in this logical memory area. It is equivalent to the code segment in other architectures. As it is garbage collected, it is allowed to expand and shrink in size and needs not to be contiguous. The method area includes the class CP, which is an ordered set of constants (*CNs*) (referenced by indices) associated with each loaded type. This set includes literals (string, integer, and floating-point CNs) and symbolic references to types, fields, methods, and other CN objects that are referred to by the class structure or by the executing code. It is essentially a symbol table. The class CP plays a central role in the dynamic linking of Java programs.
- **Heap** At runtime, instantiated objects and arrays are allocated on a single heap.
- **Registers** JVM makes use of four special-purpose user-invisible registers: **(1)** *program counter (PC)* that is one word in size and can hold both a native pointer and

a return address; **(2)** ***local variable base (VARS)*** that contains a pointer to the base of the current method set of local variables (*LVs*); **(3)** ***top of operand stack (OP-TOP)*** which contains a pointer to the top of the current method's operand stack; and **(4)** ***stack frame base (FRAME)*** that points to the base of the current stack frame in memory.

- **Java stack** Each newly spawned thread gets its own stack that stores the state of the thread's non-native method invocations in discrete stack frames. A new frame is pushed onto the thread's stack whenever it invokes a method. This frame is popped and discarded upon returning from the invocation, and the previous frame becomes the current one. Method invocation state includes: **(1)** LVs that are organized as a zero-based array of words accessed through indices. Compilers place the invocation parameters into this array first, in their declaration order. (In case of an instance method, the hidden reference `this` is placed in location number zero.) In effect, the instruction set treats LVs as a set of registers that are referenced by indices; **(2)** *operand stack* that is organized as an array of 32-bit words, accessed only through pushing and popping values to hold intermediate expression calculations. It is not a global stack; each method invocation is given its own; **(3)** *frame data* that includes the next PC and pointers to the CP area, method JBCs, debugging data, monitor area, and calling frame. The LVs and the operand stack are stored directly on the Java stack, whereas the data frame includes pointers to the actual data. The sizes of of LV area and operand stack depend on the needs of each individual method. These sizes are determined at compile-time and included in the class file data for each method. The size of the frame data is implementation dependent.

2.4 Access Patterns for Data Types

This section studies the access patterns for JVM data types. Heavily used data types should be incorporated into Java hardware for performance improvement. Access pattern information will also prove useful when decisions are made about storage allocation.

2.4.1 JVM Data Types

JVM instructions have one slight oddity: there is somewhat more type information than is strictly necessary. JVM requires that the type of all operands be determined at compile time. This type information is usually encoded in the opcode or by referencing the CP, which includes entries for different data types. To help achieve this, JVM provides plenty of programming data types that includes advanced constructs like Java fields, objects, and threads. Table 2.2 shows JVM-supported data types and their sizes [8]. JVM data types can be divided into two sets: *primitive types* that are data themselves (do not refer to anything) and *reference types* that refer to dynamically created objects. Values candidate for reference data types can be a null value, a reference to a class instance (including arrays), or a reference to a class instance implementing an interface.

Table 2.2. *JVM-supported data types.*

Data type				Size (bytes)	Description
Primitive	Numeric	Integer	byte	1	signed 2's complement integer
			short	2	signed 2's complement integer
			int	4	signed 2's complement integer
			long	8	signed 2's complement integer
			char	2	16-bit unsigned Unicode character
		Float	float	4	IEEE 754 single precision floating point
			double	8	IEEE 754 double precision floating point
	ReturnAddress			4	used as addresses for jsr, ret, jsr_w
Reference				4	reference to a Java object, array, or null

2.4.2 Single-Type Operations

Figure 2.3 shows the distribution of data accesses by type. In theses figures *generic* refers to operations that could be applied to any data type. Operations that have no associated data type are grouped under *void*. As shown in these figures, *integer* and *reference* are the data types most used by the typed operations. Java architectures therefore need to support *integer* and *reference* data types in hardware, e.g., it could provide multiple functional units that process *integer* and *reference* types in a superscalar fashion. Also, since JVM is a 32-

bit architecture, it uses 32-bit data types more than any other type. The width of the register file and CPU datapaths must be designed accordingly.

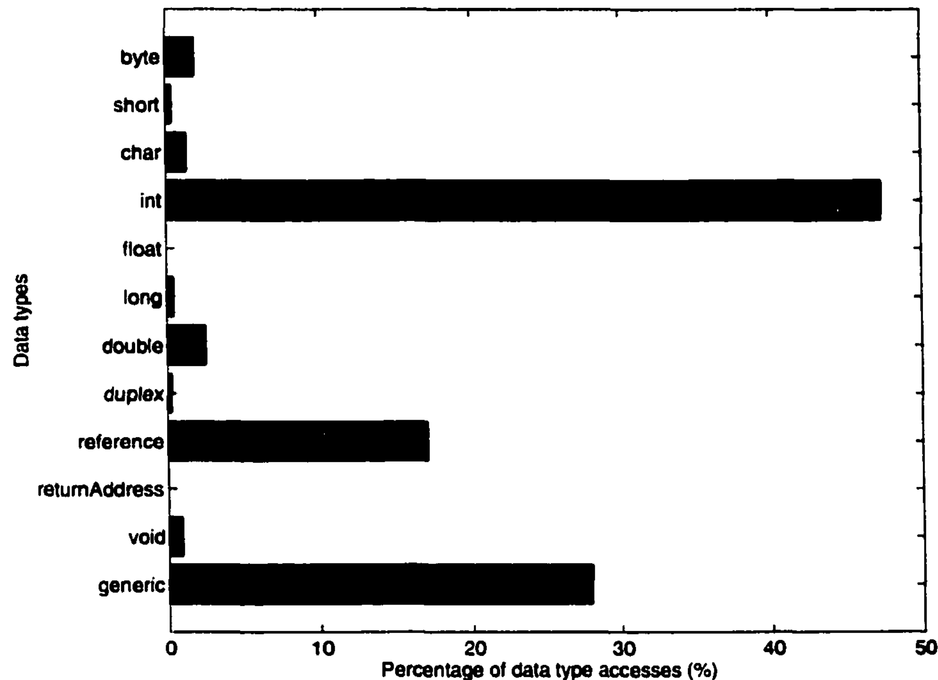


Figure 2.3. *Distribution of data accesses by type.*

2.4.3 Type Conversion Operations

JVM has a set of instructions that converts data from one type to another. This is necessary for a strongly typed programming language like Java. Figure 2.4 shows that the conversion from integer dominates all type conversion operations, especially to character. Only the data types that were involved in type conversion operations are shown in this figure. This information, combined with the results from Subsection 2.4.2, implies that integer conversion operations are the ones that are used the most. For better Java performance, an ALU design needs to perform this conversion in one clock cycle or less.

2.5 Addressing Modes

This section examines the usage of JVM addressing modes.

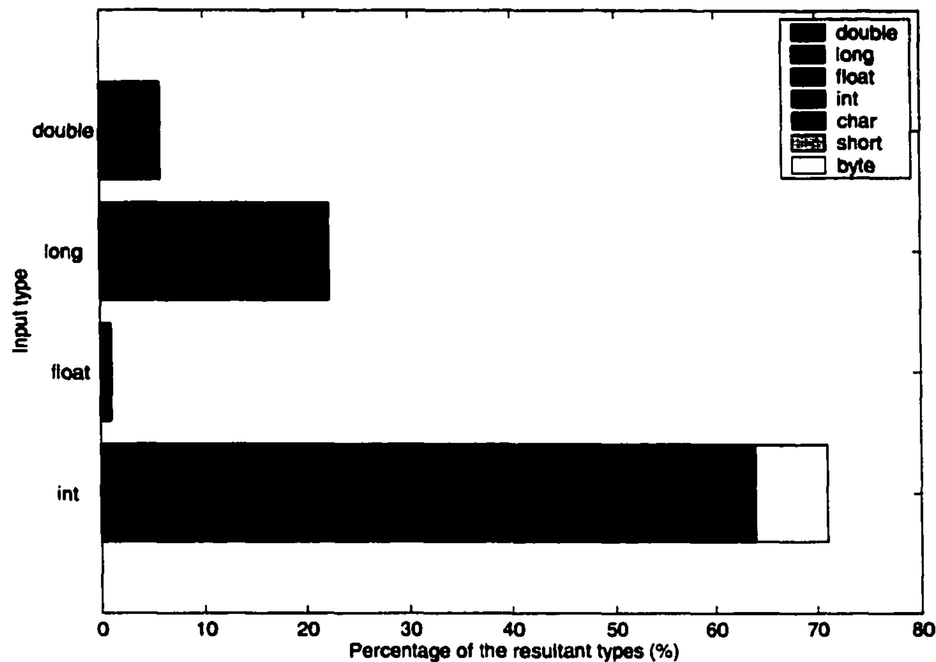


Figure 2.4. *Usage of type conversion instructions.*

2.5.1 JVM Addressing Modes

Memory addressing means how memory addresses are interpreted and how they are specified. This determines what object is accessed as a function of the address and its size. The traditional concept of addressing modes, as used in general-purpose processors, is not exactly applicable to the JVM, which uses a stack-based intermediate language. This stack architecture, together with the object orientation features of Java, are reflected in the combination of traditional (e.g., immediate) and non-traditional (e.g., LV and class CP indexed) addressing modes shown in Table 2.3. There are seven addressing modes in the JVM [8]:

- **Immediate** In this mode, instructions do not access memory. The operand is stored directly in the bytecode stream, either explicitly in the bytecode following the opcode, or hardcoded in the opcode itself. Immediates are used as CNs, branching addresses, or for table switching. It is also used for specifying the type of an array.
- **Local variable** These are the instructions that address LVs. The LV index is either mentioned explicitly in the next bytecode or hardcoded in the opcode. The `iinc` instruction is in this group. To form the actual address, JVM offsets the LV index

Table 2.3. *JVM addressing modes with examples.*

Addressing mode	Operand source(s)	Examples
Immediate	Bytecode stream	bipush 5,iconst_0, goto 13014
Local variable	VARS† [Bytecode stream]‡	iload_3, fstore 5, iinc 3 4
Stack	Stack top	pop, dup, fdiv, lrem
Constant pool indexed	CP[Bytecode stream]	ldc 220, getstatic 540, invokevirtual 591, new 2
Array indexed	Heap⋆[Stack top]	iastore, faload, arraylength
Object reference	Heap[Stack top]	athrow, monitorenter, monitorexit
Quick reference	Heap[Stack top]	ldc_quick, getfield_quick, new_quick

- † VARS is the base of the LV area.
- ‡ [] means indexing in the specified area.
- ⋆ Heap is the place for storing arrays and objects.

from the base of the current LV area (stored in register VARS).

- **Stack** These instructions access the stack top only. Though the stack is involved in all addressing modes, this category involves stack operands only. No other memory is further referenced. Its effective address is stored in the register OPTOP.
- **Constant pool indexed** These are the instructions that reference the CP. The index given here specifies an offset in the CP area.
- **Array indexed** Array referencing is done by providing the array reference and the index. Combining the array reference with the index gives a heap address for the array element to be accessed.
- **Object reference** These are the instructions that access objects without going through the CP. The object reference specifies the heap address of the object.
- **Quick reference** All the quick instructions use this mode, and may include some of the previous addressing modes, which are already resolved to address the heap.

2.5.2 Runtime Resolution and Quick Instruction Processing

The JVM uses the information from the CP to *resolve* symbolic references. Resolution is the process of finding the entity identified by the symbolic reference and replacing that reference with a direct reference. The symbolic reference is resolved into a direct reference

by searching through the method area until the referenced entity is found (possibly loading or creating it if it does not already exist). Once resolved to an address, the actual value can be accessed. Method invocations and field reference also require resolution. Note that entries in the CP are resolved only once since the runtime system caches the information it obtains after resolving the item. This requires maintaining pointers directly to the resolved item. Runtime resolution is the key to part of Java's appeal; it provides dynamic linking to support independent and incremental compilation of Java classes.

Sun's JVM implementation uses an optimization strategy for runtime resolution that involves self-modifying code. Quick variations of some opcodes exist to support efficient dynamic binding and optimize performance [22]. The first time a method or field is referenced, the original instruction resolves which method or field is correct for the current runtime environment. The original instruction also replaces itself with its quick variant. The next time the program encounters this particular instruction, the quick variation skips the time consuming resolution process, and simply performs the desired operation directly on the already resolved method or field. The quick opcodes have the same semantics as the defined, documented opcodes they replace. But, the quick versions assume that the CP resolution process has already been completed successfully. The JVM uses hidden opcodes not exposed as part of the defined architecture for this quick variant. These opcodes mostly hold the original name suffixed `_quick`.

2.5.3 General Usage Patterns

Addressing mode usage patterns are shown in Figure 2.5. The LV access dominates all other addressing modes. The immediate access and the quick reference also occurred frequently. Quick reference refers to operations that were replaced with a quick version for efficiency. Hardcoded addressing modes, in which the operand value is encoded in the instruction itself, occurred in more than one third of the total addressing modes used. The size requirements of immediates and LV index will be elaborated later in Section 2.7. The graph also shows a significant use of stack address, which is reasonable for a stack-based machine like the JVM. It can be concluded that any Java architecture needs to support at least LV, immediate, quick referencing, and stack addressing modes.

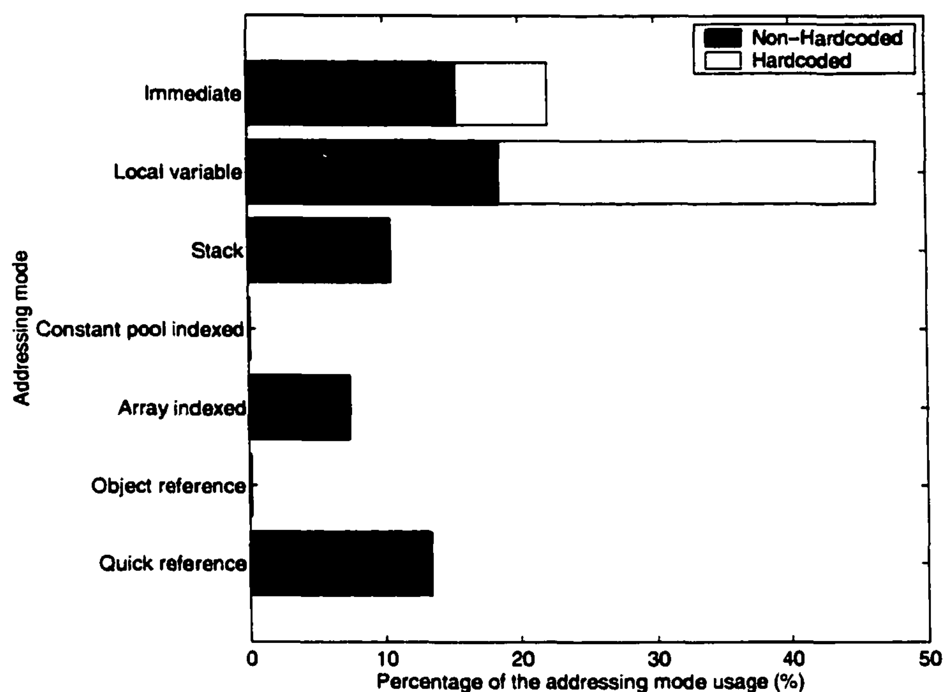


Figure 2.5. *Summary of addressing mode usage.*

2.5.4 Quick Execution

The symbolic representation of the Java class file slows down JVM execution [21, 26]. Therefore, hardware techniques for symbolic resolution and dynamic linking are required to speed up Java performance. Sun's JDK adopts a software technique for overcoming this issue, by *quick executing* JVM instructions that are previously resolved [4]. Furthermore, *quick wide* versions for some instructions are provided. As Figure 2.6 shows, the quick versions are executed much more than the non-quick ones for the same instruction. Of the instructions that have both versions, about 75.7% are executed in a quick mode, 23.7% in a quick wide mode, and only 0.6% run in a non-quick mode. Hardware mechanisms replacing cached non-quick instructions with their quick versions could speed up execution.

2.6 Instruction Set Utilization

This section studies the frequency of using the different instruction classes. Amdahl's rule for performance improvement in computer architecture suggests making the common case

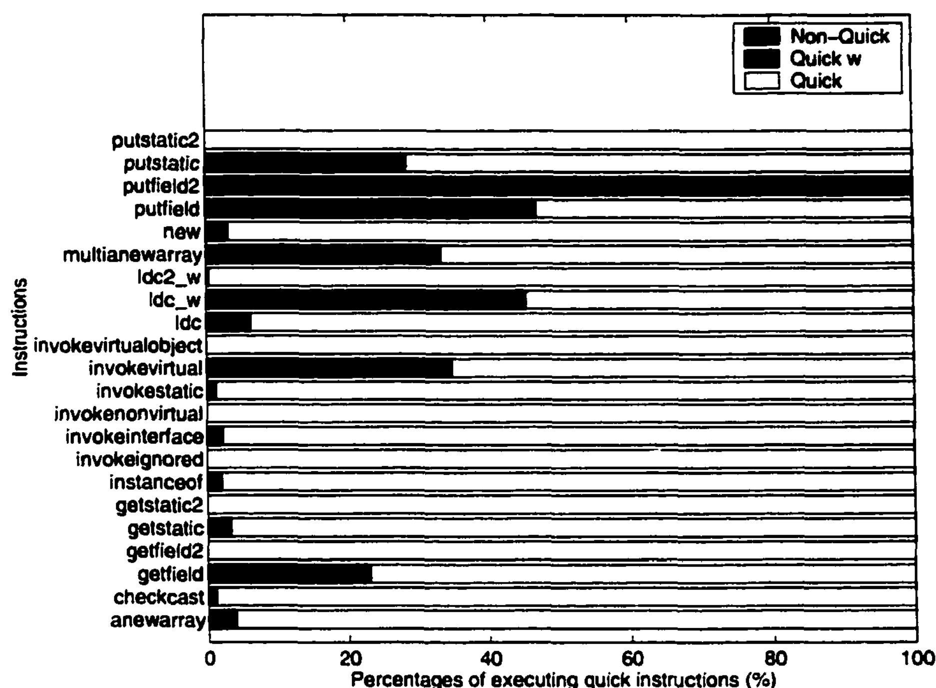


Figure 2.6. *Summary of quick execution of instructions.*

fast and the rare case correct [63, 69]. Therefore, the design of a Java processor should concentrate on the most frequently used instructions and support them efficiently [13, 15, 16, 22, 31, 39].

2.6.1 JVM Instruction Set

JVM supports a rich instruction set. This includes instructions for arithmetic, flow control, accessing elements in arrays, etc. Many of them are similar to those of a traditional processor, whereas others are specific for Java. Supported instructions range from high-level ones (like those supporting object orientation) to low-level operations (like those manipulating the stack). In this subsection we study the different instructions supported by the JVM [1, 5, 8, 12].

Although the general properties of the JVM instruction set are similar to that of many standard CPU's, the unique combination of the different advanced features made for Java, has resulted in some distinguishing twists [6, 7, 8]:

- **Relatively large group** The JVM architecture defines 201 standard opcodes that can

appear in valid `.class` files. In addition, Sun's implementation describes 26 *quick variations* and three reserved opcodes.

- **Incomplete set** Java instruction is not complete since it lacks some major instruction types. In JVM, there is no system call support. This itself is an obstacle in building a Java OS. Extensions to JVM are required for this. Also, JVM does not provide any support for graphics, string manipulations, decimal processing, etc. Currently, these functions are supported through the class library.
- **Self-contained opcodes** Many JVM instructions consist of only an opcode and take no operands. This irregularity complicates the hardware fetch and decode process as the opcode has to be decoded first to decide if an operand needs to be fetched. Additionally, to handle these one-byte opcodes, the JVM specification requires all instructions in the class file (except two that deals with table jumping) be aligned on byte boundaries. However, word alignment is more efficient for execution on some systems.
- **Typed operations** Most JVM opcodes indicate the type they operate on. Values on the operand stack must be used in a manner appropriate to their type. It is illegal to push a data of a certain type and pop it as another. This feature allows instruction verification all at once by a dataflow analyzer at loading time, rather than verifying each instruction as it is executed, which speed ups processing. The mnemonic for these instructions indicate their type by a single-character prefix that starts their mnemonic (like `fsub`). Instructions, such as `instanceof`/`arrayLength`, do not include a prefix as their type is obvious (object reference/array). Only a few instructions operate on all types (like `dup`). Some opcodes, such as `goto`, do not operate on typed values and so do not have a prefix. Table 2.4 lists prefix codes for JBCs.
- **Nonorthogonality** Two aspects of the ISA are said to be orthogonal if they are independent (primary components of an instruction set are operations, data types, and addressing modes). To enable each opcode to be represented by a single byte, not all operations are supported on all types. Most operations are not supported for type byte, short, and char. These types are converted to integer when moved from the heap or method area to the stack frame. They are operated on as integers and then converted back to byte, short, or char before being stored back into the heap or method area. The nonorthogonality can also be found in some instructions that are associ-

ated with a data type, like push (`bipush`), while others are not, like `dup`/`pop`. This asymmetry makes programming and implementing the JVM more complex. In addition, some operations are supported for objects but not for scalar, e.g., there is no `new` function for scalar data types.

- **Synonymity** As shown before, many of the instructions that normally operate on different data types have identical functions. For instance, `iload` and `fload` both copy a 32-bit value from a LV to the top of the stack. The only difference is that subsequent instructions treat the data moved by `iload` as an integer, but that moved by `fload` as a floating point value. Sometimes the type information is necessary to perform the operation correctly. Internal to an implementation of the JVM, these instructions can be treated as synonymous. Thus, the underlying hardware can implement a smaller subset of distinct operations.
- **Hybrid data types** To improve the performance of the commonly used scalar operands, JVM has instructions for both scalar operations and object-oriented programming. It supports the basic scalar types by having instructions to directly load, store, and operate on these built-in types. On the other hand, JVM is integrated with object-oriented features that reflect the semantics of the Java programming language. It has instructions to allocate new objects, access data members, and call methods within objects. In this duality, Java is unique. Most other languages' virtual machines are either all scalar or all object-oriented.

Table 2.4. *Prefix codes for JBCs.*

Type	Prefix codes	Example
byte	b	baload
short	s	saload
int	i	istore
long	l	ldiv
char	c	castore
float	f	fsub
double	d	ddiv
reference	a	aastore

In what follows we will classify the Java instruction set into classes. Inside each class

further subclasses will be provided. Classifying the instruction set is useful in designing an underlying hardware support. In the course of our analysis, the outline characteristics for each class will be mentioned. Table 2.5 divides JVM instructions into classes and gives the instruction count for each class [1, 5, 8]. Examples from each class are also included in the table. To have a flavor, Table 2.6 shows a piece of Java code converted to JVM code.

- **Scalar data transfer** These instructions transfer data between the stack and LVs or push CNs on the stack. The LV access subclass, which is the largest single group of JVM instructions, including scalar loads/stores from/to LVs. The CN push subclass includes several `const` instructions that push CNs on top of the stack, implementing the common cases efficiently in time and space. This subclass also contains the `bipush` and `sipush` instructions that push immediate integer CNs onto the stack. Also included in this class is the `nop` and `wide` opcodes.
- **ALU** These are the classic stack-architecture operations: one-byte opcodes that fetch operands from the top of the stack, operate on them, then push the result back onto the stack. Within this class, further divisions include: **(1)** *arithmetic*, which are the `add`, `sub`, `mul`, `div`, `rem`, and `neg` instructions applied to integer, long, float, and double data types; **(2)** *logical*, which are the bitwise and shift operations applied to integer and long data types; **(3)** *conversion*, which carry different type conversions; **(4)** *comparison*, which are provided for basic scalar types long, float, and double; and **(5)** *LV increment*, which is the `iinc` that adds an immediate value directly to a LV, speeding up this frequently occurred operation.
- **Stack** All instructions specific to the stack, e.g., `pop` and `dup`, are categorized here. These instructions do not access any other storage area.
- **Object manipulation** One of the Java most powerful aspects is its incorporation of the object orientation features at the JBC level and in the memory model. A number of opcodes support operations on objects (including array operations). The object-specific subclass further includes: **(1)** *object creation*, including the opcode `new`; **(2)** *field access*, which is the set of `put` and `get` instructions that take care of retrieval and storage of data in object data fields; **(3)** *type checking* that includes `checkcast` and `instanceof` to check the class type; and **(4)** *CP access*, which includes `ldc` instructions that resolve a CP item and push the resolved CN value onto the stack. The array-specific subclass is used for allocating arrays and for loading and storing

Table 2.5. *Classes of JVM instructions.*

Classes and number of instructions		Subclasses and number of instructions		Sub-subclasses number of instructions		Examples
Scalar data transfer	69	No effect	2			nop, wide
		LV access	50			iload_1,istore_0
		CN push on the stack	17			iconst_0, bipush
ALU	57	Arithmetic	24			iadd,lsub
		Logical	12			iand,lor
		Conversion	15			i2f,f2l,d2l
		Comparison	5			lcmp,fcmpg
		LV increment	1			iinc
Stack	9	Stack-generic				pop,dup2,swap
Object manipulation	46	Object-specific	25	Object creation	1	new
				Field access	14	getfield,putfield
				Type checking	4	instanceof
				CP access	6	ldc,ldc_w
		Single-dimensional array-specific	21			newarray,iaload
Control flow	21	Conditional branch	16	Equal/not equal	6	ifeq,if_icmpeq
				Null/not null	2	ifnull,ifnonnull
				Greater than or equal	4	ifge,ifgt
				Less than or equal	4	iflt,ifle
		Unconditional branch	2			goto,goto_w
		Subroutine jump	2			jsr,jsr_w
		Subroutine return	1			ret
Complex	28	Multidimensional array-specific	2			multianewarray
		Table jump	2			lookupswitch
		Method handling	18			invokespecial,return
		Exception throw	1			athrew
		Synchronization	2			monitorenter
		Reserved opcodes	3			breakpoint,impdep1

array elements (from/to LVs).

- **Control flow** JVM has an optimized conditional and unconditional branch architecture, that implements compare and branch constructs efficiently in time and space.

Table 2.6. *An example of JVM code generation for a Java program.*

Java	JVM	Comments
`Class Data {`		
`int data [];`		
`int sum () {`		
`int tmp [] = data;`	`aload_0`	Load `this`
	`getfield 9`	Load `data`
	`astore_1`	Store in `tmp`
`int total = 0;`	`iconst_0`	Push 0
	`istore_2`	0 in `total`
`for(int i = tmp.length;` $--i \geq 0;$`)`	`aload_1`	Load `tmp`
	`arraylength`	Get `length`
	`istore_3`	Store in `i`
	A `iinc 3 -1`	sub 1 from `i`
	`iload_3`	Load `i`
	`iflt B`	Exit on $i < 0$
`total` $+=$ `tmp [i];`	`iload_2`	Load `total`
	`aload_1`	Load `tmp`
	`iload_3`	Load `i`
	`iaload`	Load `tmp [i]`
	`iadd`	Add in `total`
	`istore_2`	Store in `total`
	`goto A`	Do it again
`return total;`	B `iload_2`	Load `total`
	`ireturn`	Return `total`
`}`		
`}`		

There is a set of instructions performing the same functions, but associated with different data types. Branching instructions are organized in a number of subclasses: conditional and unconditional branches, and subroutine jumps and returns.

- **Complex** Instructions in this class require complex processing. They are divided into a number of subclasses: multidimensional array-specific, table jump, method handling, exception throw, thread synchronization, and the reserved opcodes. Instructions in this class can be very long, both in execution time and code size.

2.6.2 Dynamic Opcode Distribution

Table 2.7 shows the dynamic frequency of using different instruction classes in CaffeineMark and the breakdown of each class. The most heavily used instruction classes include the scalar data transfer (specifically LV access) and object manipulation (specifically object-specific ones). The following observations are made from Table 2.7:

- The most commonly used instructions are loads/stores between the LVs and the top of the operand stack. About 43% (83.20% of 51.76%) of the executed instructions belong to this subclass. Providing hardware support for it will improve Java performance. One possible way involves allocating LVs to on-chip general-purpose registers.
- The arithmetic operations exceed all others in the ALU class. So, ALU designs should be optimized for arithmetic.
- Within the object manipulation class, the object-specific group is used the most. Instructions in this group are accessed mainly as the *non-quick* version, whereas instructions in the array-specific group are accessed mainly as the quick version. In the object-specific subclass, the field access group has the highest frequency of usage. This is reasonable as Java is an object-oriented language. It can be concluded that hardware support for object orientation is a vital way to enhance Java performance. This support should facilitate field access.
- CP access, which involves memory access, does not occur often.
- The conditional comparisons are the most used among the control flow instructions, while equal or not-equal to zero is the most frequently used in this subclass. (In the JVM, all comparisons are performed with respect to zero.) This shows that the RISC-style zero register ($R0$) will be helpful here.
- Of interest is to compare the two types of conditional instructions in JVM: compare stack and branch (the conditional branch subclass), and compare stack and push (the comparison subclass). The former accounts for 10.57% (84.04% of 12.58%) of the total instructions executed while the latter results in 0.003% (0.03% of 10.70%) of the total. This indicates that comparison is mainly done for the purpose of branching. Optimizing the comparison process to suit branching could offer a significant overall performance improvement.

Table 2.7. *Dynamic frequencies of using different instruction classes.*

Classes and frequencies (%)		Subclasses and frequencies (%)		Sub-subclasses frequencies (%)	
Scalar data transfer	51.76	No effect	0.00		
		LV access	83.20		
		CN push on the stack	16.80		
ALU	10.70	Arithmetic	57.88		
		Logical	9.23		
		Conversion	2.36		
		Comparison	0.03		
		LV increment	30.50		
Stack	1.41	Stack-generic			
Object manipulation	18.13	Object-specific	58.40	Object creation	0.04
				Field access	96.20
				Type checking	1.16
				CP access	2.60
		Single-dimensional array-specific	41.60		
Control flow	12.58	Conditional branch	84.04	Equal/not equal	56.97
				Null/not null	4.43
				Greater than or equal	14.55
				Less than or equal	24.05
		Unconditional branch	15.78		
		Subroutine jump	0.09		
		Subroutine return	0.09		
Complex	5.42	Multidimensional array-specific	0.00		
		Table jump	0.61		
		Method handling	96.97		
		Exception throw	0.00		
		Synchronization	2.42		
		Reserved opcodes	0.00		

- Within the complex group, the method handling instructions are used the most. This is expected due to the object-oriented nature of Java. As the benchmark does not throw any exceptions, the percentage the exception instructions was used is zero,

though this may not be the case in other applications.

2.6.3 Frequency of Null Usage

Another interesting observation from Table 2.7 is the considerable usage of the CN null. The three instructions that make use of this CN (`aconst_null`, `ifnonnull`, and `ifnull`) are executed 39,601 times which is equivalent to 0.55% of the total instructions executed. Taking into consideration that Java is a full object-oriented language, we suggest reserving another register for the null value. This will help the object orientation performance in general, and in particular the comparison with null.

2.6.4 Branch Prediction

Modern processors predict speculatively the result of conditional control flow instructions. (Usually, devised speculation methodologies depend on data collected through benchmarks.) Figure 2.7 summarizes the benchmarked frequency of instruction groups that may change the program counter. Forward conditional instructions are the majority. Total taken branches exceed the untaken's. This general observation remains unchanged if we split branches according to their direction. Based on that, it is suggested to consider the default speculation direction for branch prediction to be taken. Figure 2.8 divides the execution frequency of each of the conditional branch instructions between taken and untaken. Our studies show that 65% of the conditional branches are taken in the benchmarked Java applications. This is a bit less than the typical percentage (75 – 80%) found in other non-Java applications [42]. A sophisticated branch predication unit that takes into consideration the opcode being executed could be designed based on this information.

2.6.5 Effect on Stack Size

Figure 2.9 shows the effect of instruction execution on the stack size. Such information is helpful for designing hardware that maximizes ILP without increasing stack access contention. Instructions issued for parallel execution should not access the stack simultaneously, otherwise contention will occur and potential speed improvement will not be realized. Having a multiport stack might alleviate the contention problem. In addition, if instruction folding is used, issued instructions should have a desirable effect on stack size.

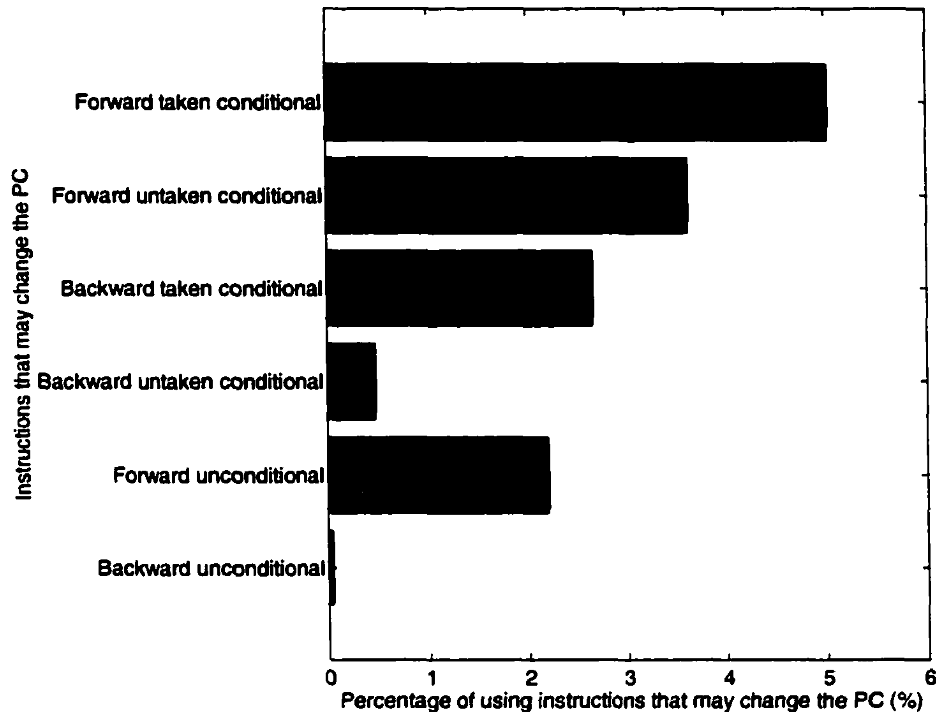


Figure 2.7. *Summary of executing instructions thc.t may change the program counter.*

2.7 Instruction Encoding

This section quantitatively analyzes the different fields in JVM instructions. The objective is to determine the optimum number of bits required for mapping JVM instruction fields onto proposed native instruction formats. The analysis presented here should not be considered contradicting the JVM specification that dictates the exact size of each instruction field. It is meant to provide guidelines for a Java architecture's native instruction format to include JVM instruction fields while attaining generality and efficiency.

2.7.1 JVM Instruction Encoding

The JVM specification requires JBCs to be provided as a stream of bytes grouped as variable-length instructions. However, it hardly mentions any encoding format for instructions [4, 9, 11]. In Figure 2.10, we show a possible layout of the instruction format that could accommodate different JVM instructions. Processing this irregular format in hardware might work against the generality of the architecture and could adversely affect the

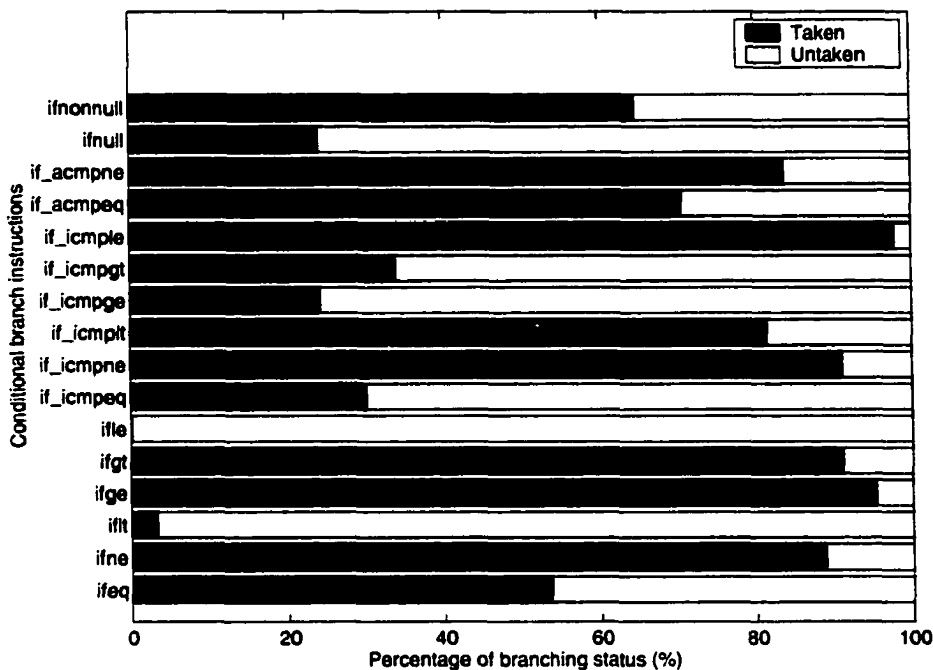


Figure 2.8. *Statistics of conditional branches.*

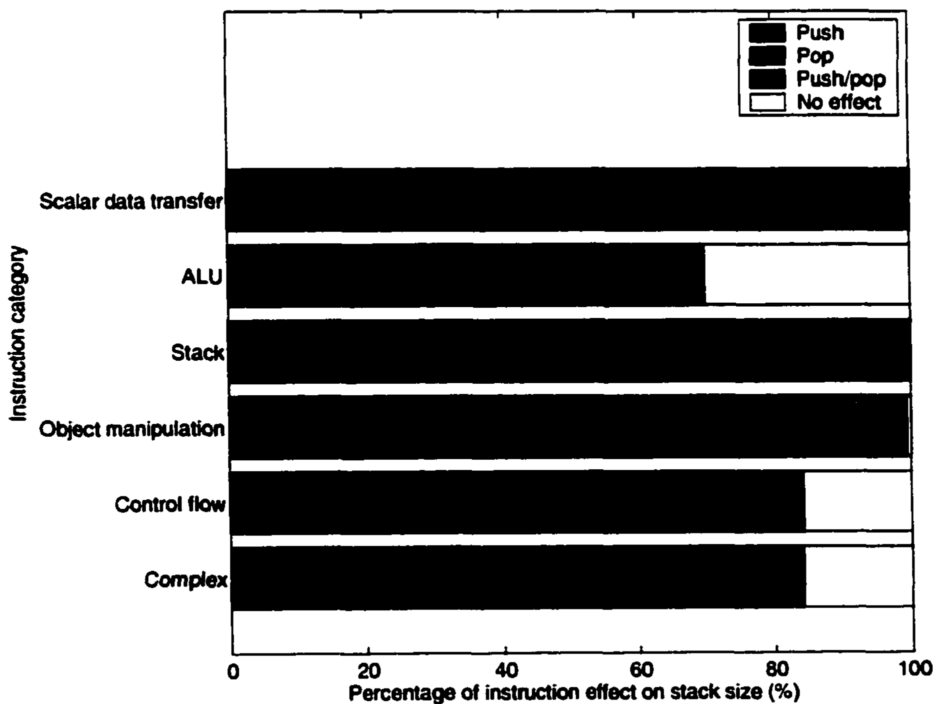


Figure 2.9. *Distribution of the effect on stack size for each instruction class.*

performance of instruction fetch and decode as well as the pipeline efficiency.

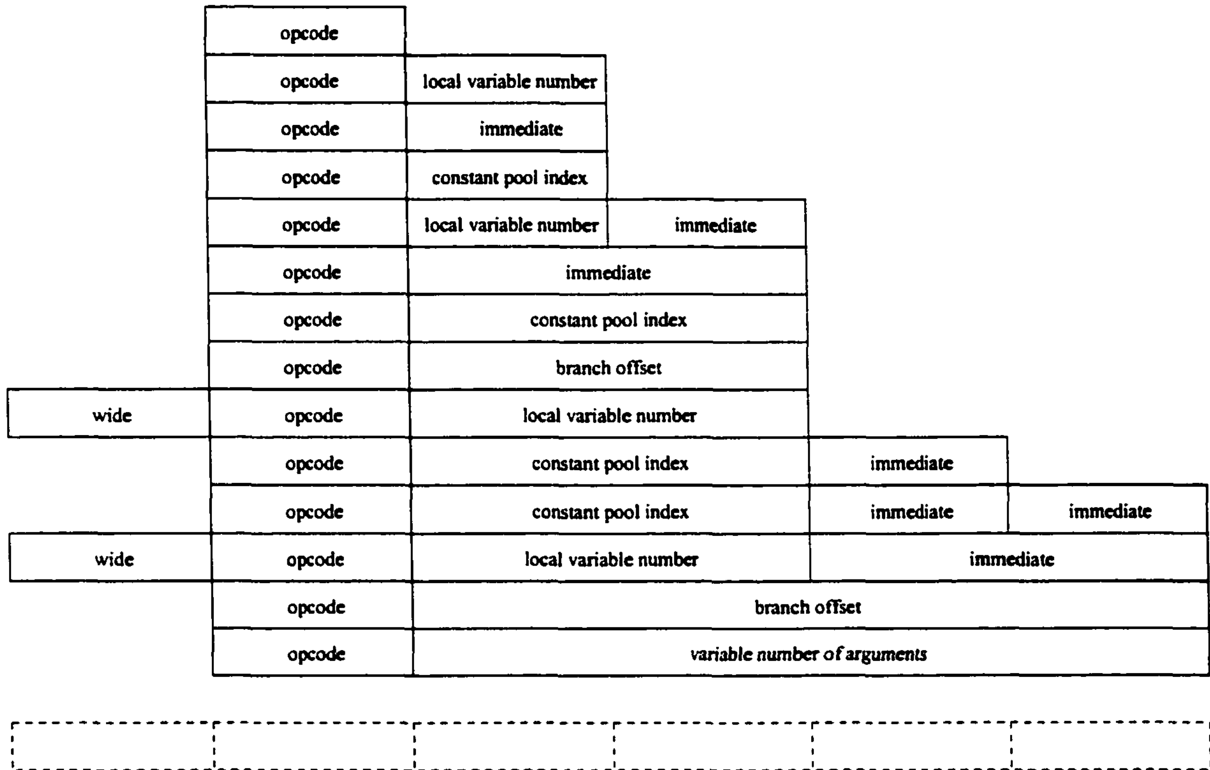


Figure 2.10. *Instruction layout for JVM.*
Dashed lines at the bottom of the figure show byte boundaries.

Each JVM instruction consists of a 1-byte opcode followed by zero or more operands. The opcode indicates the operation to be performed. Operands supply extra information needed by the JVM to perform the operation specified by the opcode. The opcode itself indicates whether it is followed by operands and the form the operands (if any) take (including the addressing mode information). Depending on the opcode, JVM may refer to data stored in other areas in addition to (or instead of) operands that trail the opcode. These data may be entries in the current class CP, any of the current frame's LVs, or a value sitting on top of the current frame's operand stack.

The following observations could be made about the way JBCs are encoded:

- **Hardcoded operands** JVM has a drawback in that many of its instructions have an implicit operand. These include instructions to compare with zero, store a CN (0, 1, 2, or 3) in a word, or access certain LV (0, 1, 2, or 3).

- **Small size instructions** JVM in its well defined architecture results in a very compact code; JBCs are relatively small. There are no register specifiers, LVs are accessed relative to a base pointer, and instructions implicitly operate on the stack.
- **Multiple operand sources** Although JVM instruction set's main focus is the operand stack, other sources of operands are also available, including the opcode itself, the bytecode stream, and the class CP.
- **Expandable operands** Some of the JVM instructions have a wide format that is formed by preceding the instruction with the wide prefix. This, in the grand tradition of the $x86$, expands the LVs or CP entries index for the general loads and stores from one byte to two.
- **Variable length format** JVM instructions are followed by a variable number of operands in zero or more bytes (up to 4, except in table switching instructions). The number of operands changes not only from one instruction to another but also for the same instruction. Some instructions, like table switching ones, support a variable number of operands. This results in a variable-length instruction set.
- **Various branch destinations** In JVM, the destination address of a control flow/method invocation instruction must always be specified. There are three ways of specifying the addressing target: **(1)** *direct jumps* in which control flow instructions specify the address explicitly in a PC-relative form; **(2)** *table switching*, which provides a number of addresses to select from based on a key value; **(3)** *virtual functions* that allow different routines to be called depending on the type of the data.

2.7.2 Local Variable Indices

Instead of specifying a set of general-purpose registers, JVM adopts the concept of LV referencing. As shown in Figure 2.11, Java methods typically require up to 16 LVs (with an average and standard deviation of 4 and 2.48 LVs, respectively), though the specification allows referencing up to 255, or $65,535$ in the case of wide instructions. The graph also shows almost no preference in accessing these variables. In designing hardware for Java, it is suggested that LVs be allocated to on-chip storage. A general-purpose register file can be configured to work as a storage area for LVs, allowing Java programs to run faster.

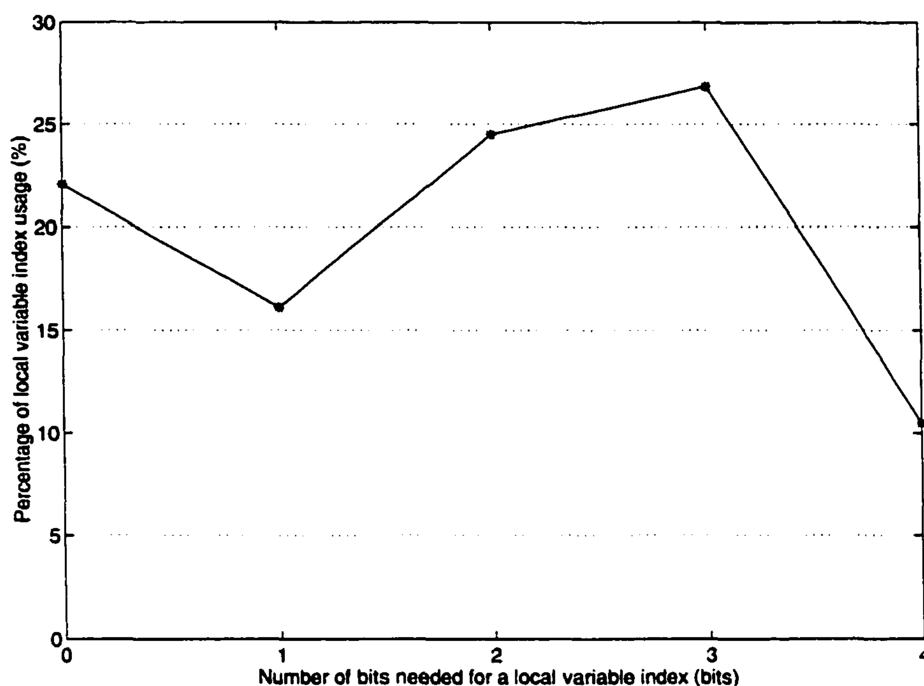


Figure 2.11. *Distribution of number of bits representing a LV index.*

2.7.3 Constant Pool Indices

The JVM's CP is a collection of all the symbolic data needed by a class to reference fields, classes, interfaces, and methods. A CP index size can either be up to 16 bits, or to 32 bits in a wide format. Figure 2.12 shows that 16 bits cover almost all CP accesses (the average is 8 bits and the standard deviation is 2.9 bits). Seven bits are enough to cover more than 50% of the total CP index usage and 10 bits can cover more than 80%. Based on the statistics shown in Figure 2.12, it is suggested that 13 bits be used to encode CP index values in JVM instructions, as this will cover 98% of the cases.

2.7.4 Immediates

As a stack machine, the JVM does not rely heavily on immediates for ALU operations. Immediates in the JVM are either pushed on the stack, or used as an offset in control flow or table switching instructions. Immediates could have a length of up to 32 bits. As shown in Figure 2.13, immediates used in CaffeineMark are up to 18 bits in length (the average is

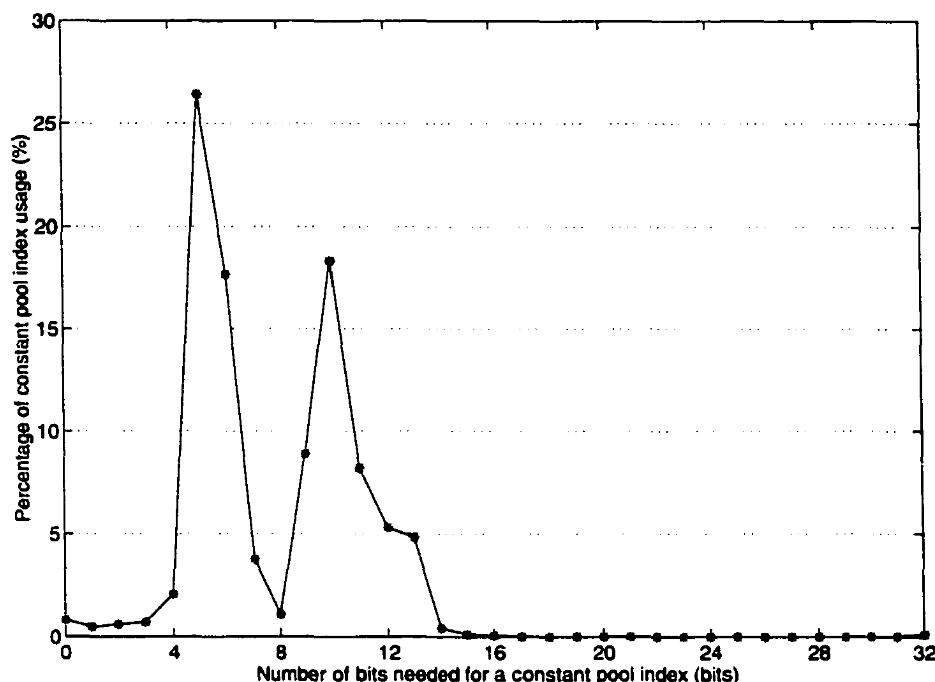


Figure 2.12. *Distribution of number of bits representing a CP index.*

4 bits and the standard deviation is 2.07 bits). The most frequently used immediates are 3 bits. Since hardcoded immediates are not counted in this study, the measurements show less frequent use of 0, 1, 2, and 3 bits than on architectures that do not use hardcoded operands. (When the immediate is part of the opcode encoding, we say it is hardcoded.) Three bits are enough to cover more than 50% of the total immediate usage and 6 bits can cover more than 80%. Based on the statistics in Figure 2.13, it is suggested that 8 bits be used to encode immediate values, as this will cover 98% of the cases. Situations that require more than 8 bits can be handled by the compiler using a special wide format. Another solution involves saving large immediates in the CP and addressing them through an index.

2.7.5 Dynamic Instruction Length

JVM instructions support a variable number of operands, resulting in a variable length instruction format. Figure 2.14 and 2.15 show the dynamic count of JBCs and operands, respectively. As shown in Figure 2.14, JBCs have an average of 1.77 bytes (with a standard deviation of 1.4). This is far less than the typical RISC instructions. However, in some

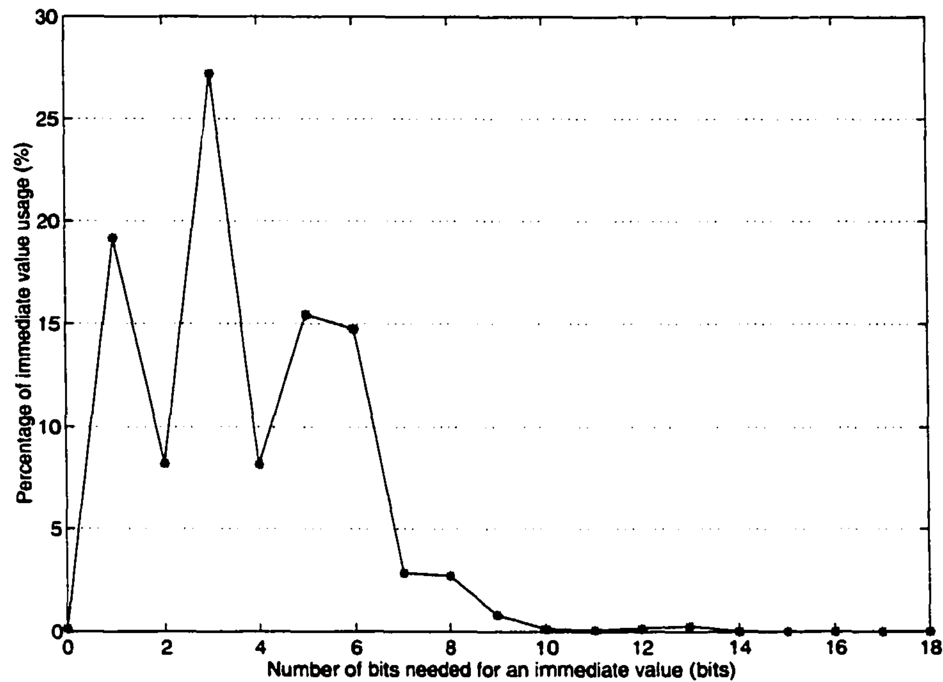


Figure 2.13. *Distribution of number of bits representing immediates.*

complex instructions like table switching, instructions can be up to 36 bytes in length. It is easy to see that padding JVM instructions so that each is 3 or 4 bytes would lead to a code expansion of 69.49 or 125.99%, respectively. On the other hand, Figure 2.15 shows that 99.97% of instructions have up to 2 operands. The average opcode has one operand and the standard deviation is 0.59. Fetched opcodes that are to be changed dynamically before execution are considered to have zero dynamic length. These include the ***non-quick*** instructions that are converted to a *quick* form, and the `invokespecial` that are converted to another invoke form. (The `executeJava` engine deals with these cases by the `withOpcodeChanged` macro.) For these instructions, length is measured as the actually executed opcode instead of the fetched one.

A number of conclusions for hardware support can be drawn based on these results. First, a very long instruction word (*VLIW*) machine can easily accommodate more than one JVM instruction per word. Second, the small size of the average JVM instruction lessens the size requirements of the instruction prefetch buffer and the instruction cache blocks. Finally, despite the network transfer overhead as a result of about 50% code expansion,

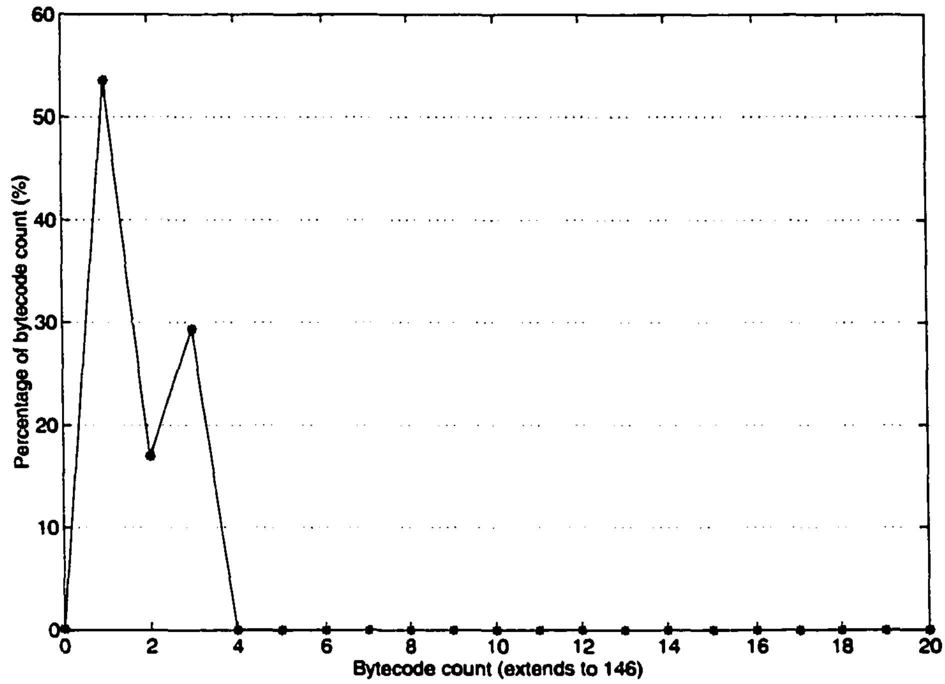


Figure 2.14. *Distribution of dynamic bytecode count per instruction.*

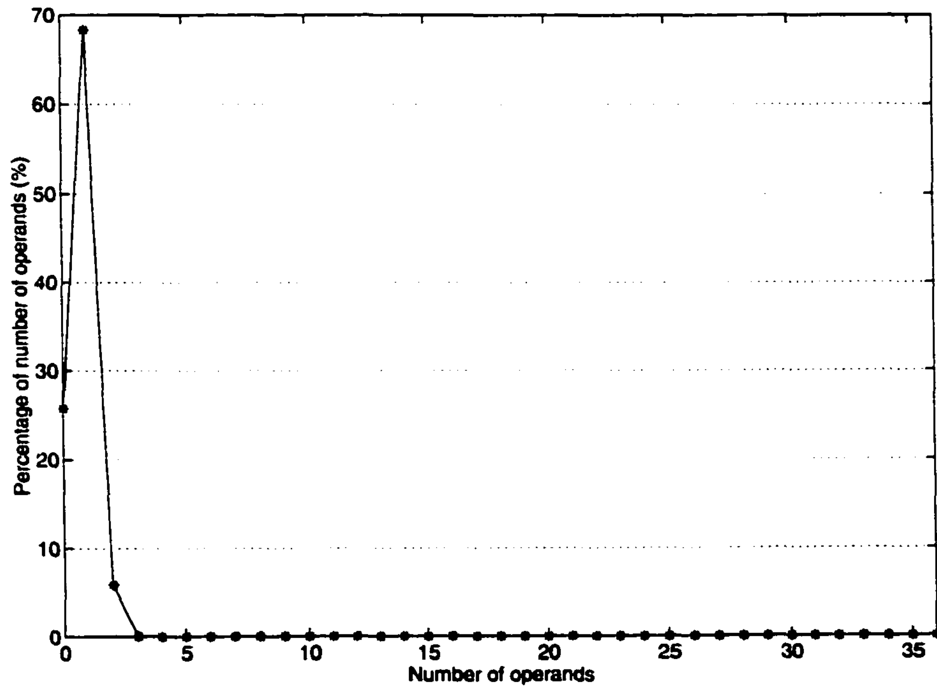


Figure 2.15. *Distribution of number of operands per instruction.*

a generalized fixed-length instruction format may lead to more efficient applet execution, e.g., faster instruction fetching, faster decoding, and better pipeline operations.

2.7.6 Array Indices

Java carries array information down to the JVM level. At runtime, the index required to access an array is popped from the operand stack. Although the index can be up to 32 bits, Figure 2.16 shows that CaffeineMark does not require more than 13 bits to access arrays (the average and standard deviation are 3 and 2.79 bits, respectively). For instruction encoding, a 3-bit index size covers more than 50% of the total array accesses and 5 bits cover more than 80%. The graph shows an interesting pattern: the maximum occurs at 0 then a decaying behavior follows with a sudden drop after 6. This indicates that about 90% of the accesses are to the first 64 elements of an array. This gives a useful guideline in data caching–the first few elements (up to the 64th) of an array should be cached. This could also have an impact on cache organization, such as the block size, replacing strategies, etc.

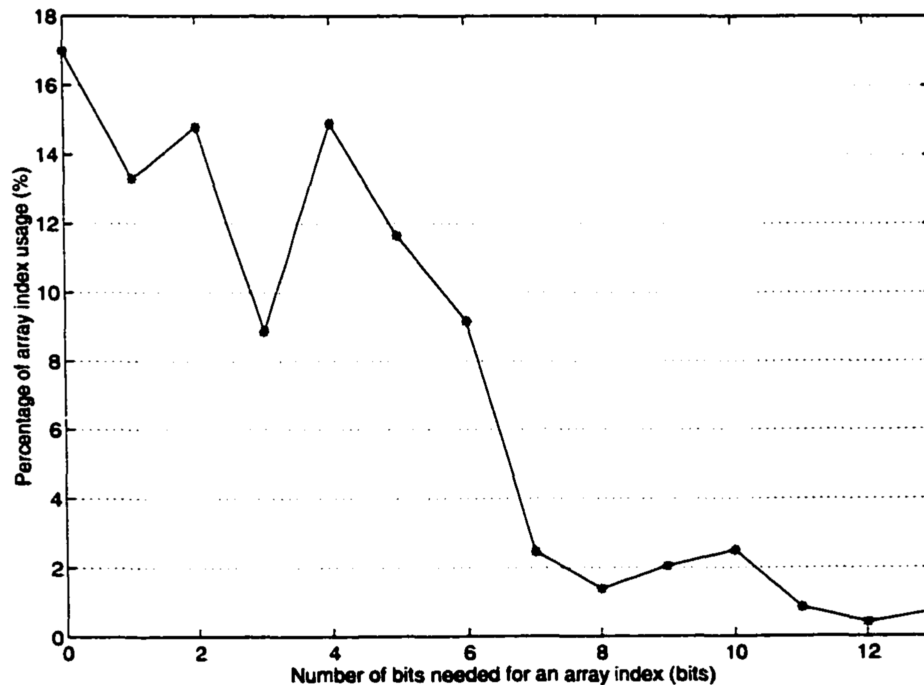


Figure 2.16. *Distribution of number of bits representing an array index.*

2.7.7 Branching Distances

JBCs include only the offset in branches and the runtime execution engine must convert them internally to the corresponding absolute addresses. As shown in Figure 2.17 (stars), CaffeineMark requires a target offset of less than 10 bits (the average is 4 bits and the standard deviation is 1.49 bits), though up to 16 bits are allowed. In the design of an instruction format, 4 bits cover more than 50% and 7 bits cover more than 80% of the cases. Eight bits appear enough for more than 98% of the offset distances. Furthermore, if a branch target buffer is used for branch speculation, a buffer size of 512 bytes (256 forward and 256 backward) is sufficient. Figure 2.17 also (circles) shows the statistics of absolute jump-to address.

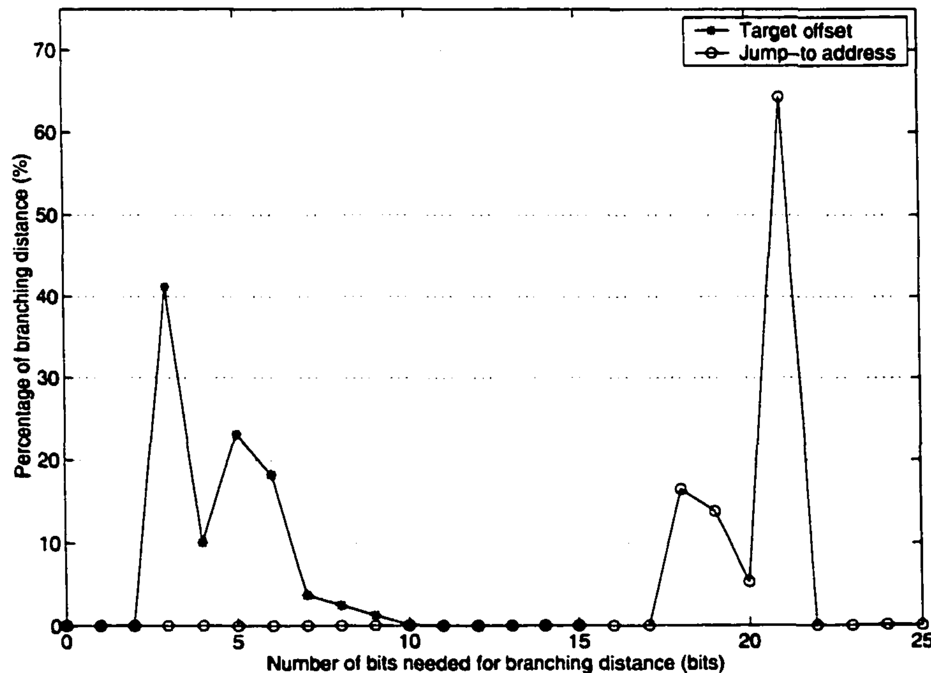


Figure 2.17. *Distribution of number of bits representing the offset and absolute jump-to address.*

2.7.8 Encoding Requirements for Different Instruction Classes

Figure 2.18 shows the instruction-encoding requirements for each instruction class: LV index size, CP index size, immediate size, number of bytecodes, and operand count. An

interesting observation from these figures is that each class has its own special requirements. We can see from Figure 2.18(a) that the scalar data transfer instructions require, on the average, the largest index size for LV access. The object manipulation and complex groups use the largest CP index size (Figure 2.18(b)). The control flow group needs the largest immediate size. (Figure 2.18(c)). The control flow class requires the longest instruction length (Figure 2.18). The control flow and complex classes require the largest operand count (Figure 2.18(e)). The ALU and the stack operations appear to need only moderate encoding. These results give hints for designing the suitable instruction format for each instruction class, e.g., control flow instruction format should accommodate large immediate sizes.

2.8 Execution Time Requirements

Execution time for JVM instructions depends on the execution environment. The results shown here simply give the relative execution time for the different instruction classes in our benchmarking environment (Sun JDK on UltraSPARC machines). Java processor designers can then identify features that are mostly used and most time-consuming. Table 2.8 summarizes the execution time requirements for each class and its subclasses. It illustrates clearly the significant difference in execution time between instruction classes:

- LV access is time consuming. Allocating LVs on-chip will save some of this time.
- CN push instructions are also time consuming. Having a way to generate and push a CN on the stack will help here.
- Arithmetic instructions exceed all other ALU operations in time requirement. So, the ALU design should be optimized for arithmetic.
- The object manipulation class of instructions takes a significant portion of the execution time of CaffeineMark. This reiterates the importance of improving object manipulation for overall Java performance. The object-oriented instructions' poor response is due to main memory access, CP access, and garbage collection.
- Object-specific instructions take significantly more time than the array ones, with field access being the most time consuming.
- Within the control flow group, the conditional branches' execution time, especially the equal/not equal instructions far exceed the others.

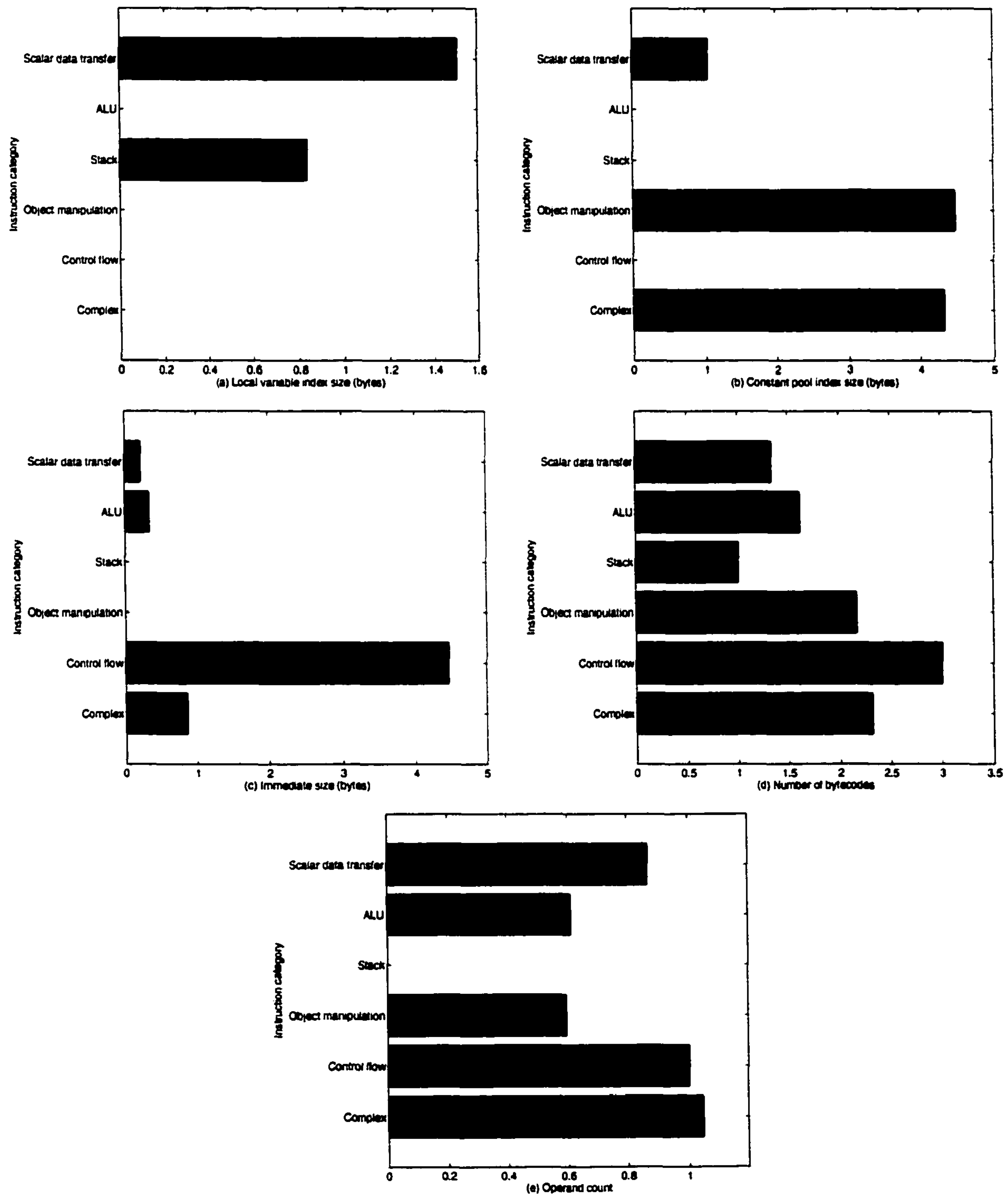


Figure 2.18. ***Instruction encoding requirements for each instruction class.***
Note that all numbers are averages and therefore not integers.

Table 2.8. *Total execution time for different instruction classes.*

Classes and percentage of average time (%)		Subclasses and percentage of average time (%)		Sub-subclasses percentage of average time (%)	
Scalar data transfer	6.50	No effect	0.00		
		LV access	84.50		
		CN push on the stack	15.50		
ALU	1.79	Arithmetic	50.77		
		Logical	9.68		
		Conversion	2.23		
		Comparison	0.10		
		Local variable increment	37.22		
Stack	0.17	Stack-generic			
Object manipulation	18.78	Object-specific	88.99	Object creation	0.00
				Field access	99.32
				Type checking	0.44
				CP access	0.24
		Single-dimensional array-specific	11.01		
Control flow	2.60	Conditional branch	85.99	Equal/not equal	54.80
				Null/not null	4.75
				Greater than or equal	14.30
				Less than or equal	26.15
		Unconditional branch	13.87		
		Subroutine jump	0.08		
		Subroutine return	0.06		
Complex	70.16	Multidimensional array-specific	0.00		
		Table jump	0.02		
		Method handling	99.94		
		Exception throw	0.00		
		Synchronization	0.04		
		Reserved opcodes	0.00		

- Total time requirement for method handling instructions is heavy. This shows that their execution is time intensive and requires some hardware support.

2.8.1 Performance Critical Instructions

Table 2.9, Panels (a)-(c) show the top 10 ranked instructions in execution frequency, the top 10 ranked instructions in execution time per invocation, and the top 20 ranked instructions in total execution time, respectively. From these panels it is worth noting that the top 17 ranked instructions in total execution time account for more than 90% of the total execution time. This agrees with the well known rule that 10% of the instructions (the benchmark executed 166 different opcodes) take about 90% of the execution time. Also, the top two most time consuming instructions are also the top two in total execution time.

2.9 Method Invocation Behavior

Object orientation encourages heavy use of small size method invocations. In the JVM, this method invocation requires a certain setup. Each method invocation has its own stack frame, which includes the LV area, the operand stack and other relevant information. The purpose of this section is to study the behavior of JVM method invocation in detail. We assume a common LV pool, that is, each method allocates LVs from this pool and keeps them (even if it invokes another method) till it returns. We also assume subsequent operand stack-method invocations allocate their operand stack on top of previous stacks. In this study, Java constructors are considered normal methods. Table 2.10 gives some method invocation statistics.

2.9.1 Hierarchical Method Invocations

As Figure 2.19 shows, JVM method invocations can be 120 levels deep with an average of 34 levels (standard deviation of 3.6). The most frequent number of nested invocations is 21. This result is useful for designing hardware to support method invocations. It gives some idea about the typical resource requirements, e.g., the operand stack, LV area, etc.

2.9.2 Local Variables

Each invoked JVM method has its own LV area for result storage and parameters passing. Previously, we saw the importance of handling LVs effectively. One suggestion is to cache this area directly on-chip. In this case, the critical question is how much on-chip area is

Table 2.9. *The top ranked instructions in execution frequency, execution time per call, and total execution time.*

Opcode	Frequency		Average execution time		Total execution time		
	Count	Rank	ns	Rank	μs	Rank	%
Panel (a): the top 10 ranked instructions in execution frequency							
aload_0	716,351	1	1068	126	765,264	5	
iload	691,703	2	985	136	681,672	6	
getfield_quick	481,668	3	1376	91	662,752	7	
iload_1	367,982	4	900	154	331,147	14	
istore	298,684	5	1034	129	308,861	15	
iload_2	270,067	6	909	152	245,560	17	
iconst_1	261,407	7	901	153	235,461	18	
iinc	233,643	8	1589	64	371,208	9	
ifeq	229,145	9	1563	68	358,254	11	
iadd	209,962	10	1197	111	251,420	16	
Panel (b): the top 10 ranked instructions in execution time per call							
invokestatic	208	139	90,434,706	1	1,881,0419	2	
invokestatic_quick	16,015	63	1,177,520	2	1,885,7980	1	
getstatic	562	124	590,195	3	331,689	13	
invokeinterface	62	151	442,620	4	27,442	69	
putstatic	118	147	360,347	5	42,521	53	
multianewarray_quick	6	163	307,368	6	1844	116	
invokevirtual	2481	94	147,128	7	365,025	10	
invokespecial	513	127	146,442	8	75,125	42	
putfield	541	126	119,158	9	64,464	47	
getfield	1245	111	105,336	10	131,144	31	
Panel (c): the top 20 ranked instructions in total execution time							
invokestatic_quick	16,015	63	1,177,520	2	18,857,980	1	33.76
invokestatic	208	139	90,434,706	1	18,810,419	2	33.67
putfield_quick	42,934	35	88,658	13	3,806,447	3	6.81
putfield_quick_w	37,237	36	95,306	11	3,548,922	4	6.35
aload_0	716,351	1	1068	126	765,264	5	1.37
iload	691,703	2	985	136	681,672	6	1.22
getfield_quick	481,668	3	1376	91	662,752	7	1.19
getfield_quick_w	142,632	13	2906	28	414,552	8	0.74
iinc	233,643	8	1589	64	371,208	9	0.66
invokevirtual	2481	94	147,128	7	365,025	10	0.65
ifeq	229,145	9	1563	68	358,254	11	0.64
invokevirtual_quick	89,455	23	3900	25	348,859	12	0.62
getstatic	562	124	590,195	3	331,689	13	0.59
iload_1	367,982	4	900	154	331,147	14	0.59
istore	298,684	5	1034	129	308,861	15	0.55
iadd	209,962	10	1197	111	251,420	16	0.45
iload_2	270,067	6	909	152	245,560	17	0.44
iconst_1	261,407	7	901	153	235,461	18	0.42
putfield2_quick	2614	92	89,302	12	233,435	19	0.42
ifne	139,647	15	1596	63	222,943	20	0.40

Table 2.10. *Method invocation statistics.*

Total number of executed methods	177,774
Average number of trace lines per method	40.263
Average number of bytecodes per method	71.396

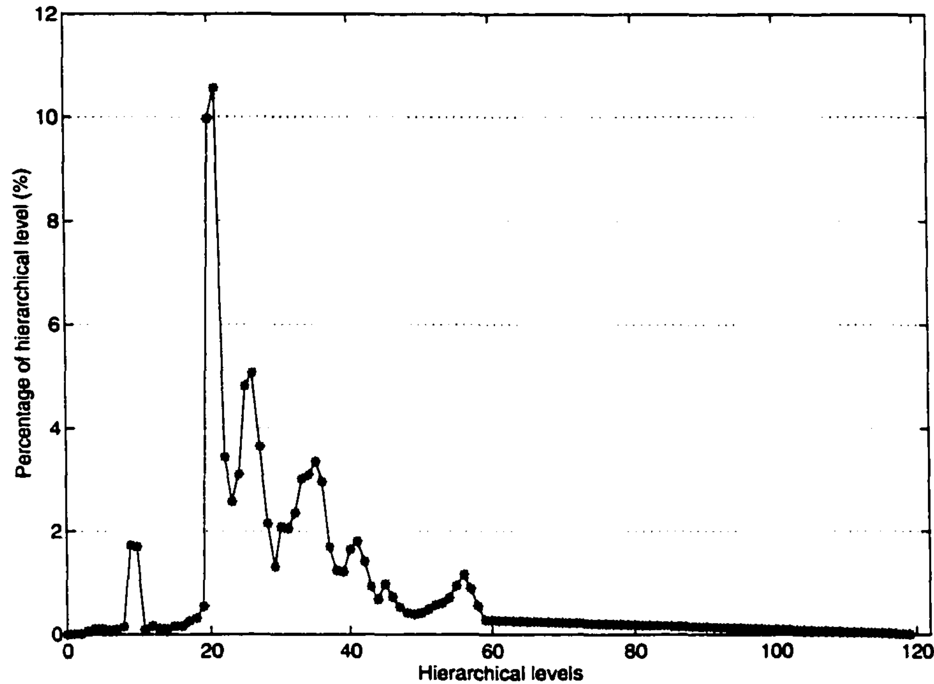


Figure 2.19. *Distribution of levels of method invocations.*

to be allocated for this purpose. This area is not only constrained by individual method invocation requirements but also by accumulated requirements through hierarchical invocations.

Figure 2.20 shows the number of LVs allocated by individual method invocations. Although up to 18 variables can be allocated, the average is 3 (with standard deviation of 2.08). Two LVs are enough for more than 50% of the total method invocations and four covers about 80% of the cases. This figure suggests that allocating eight variables in hardware for each method invocation will be sufficient. Eight variables cover more than 98% of total method invocations.

Figure 2.21 shows the number of accumulated LVs allocated through hierarchical method invocations. Accumulated means the total number used by all invoked methods through the hierarchy. Here, we assume a pool of LVs for all pending methods. The maximum number of variables required can be up to 256 with an average of 8 and a standard deviation of 2. With today's technology, having an on-chip area of 256 (or even 512) registers reserved for LVs can easily be accommodated. This will improve the performance of the JVM.

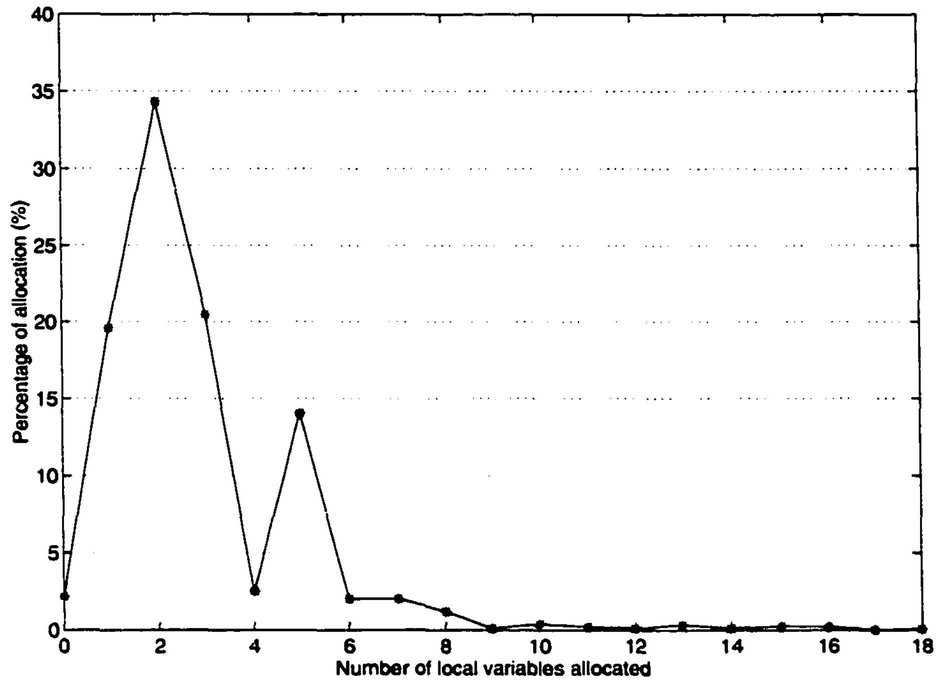


Figure 2.20. *Distribution of number of LVs allocated by individual method invocations.*

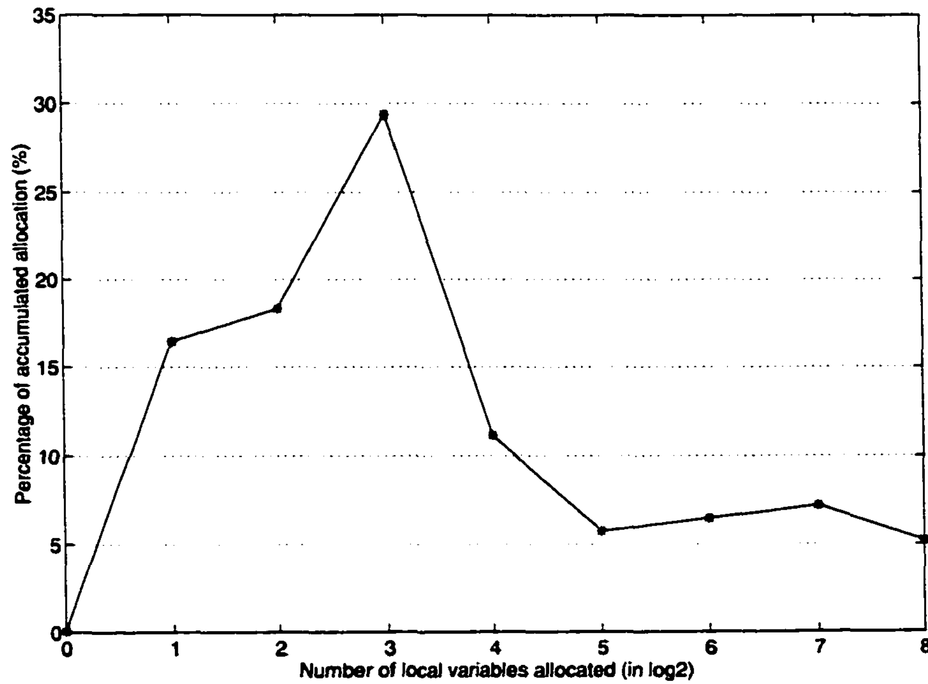


Figure 2.21. *Distribution of accumulated LVs allocated through hierarchical invocations.*

2.9.3 Operand Stack

Each JVM method has its own operand stack, which is used as a working area for intermediate results. Initial and final data are handled as LVs. The maximum size of the operand stack is specified by each method. This is required by the JVM specification so that the appropriate stack size is allocated before execution starts. Although the JVM allocates this size at method setup time, a method might not use all the allocated area. This means that the required stack size is always less than the preallocated one. The non-overlapping scheme in Figure 2.22(b) illustrates the difference between required and preallocated stack size.

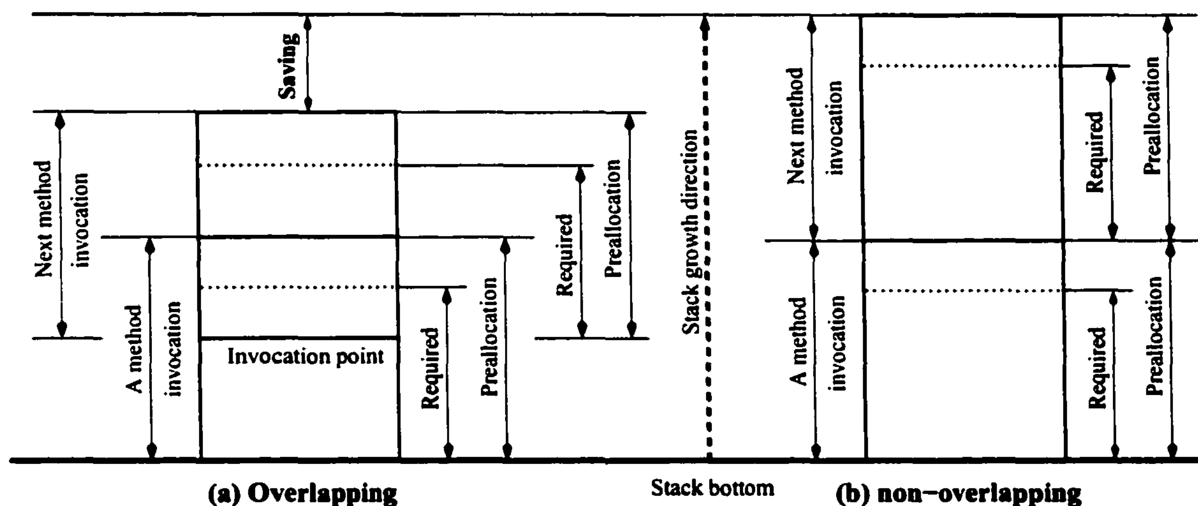


Figure 2.22. *Overlapping and non-overlapping stack allocation schemes.*

Figure 2.23 shows the percentage of the different stack sizes allocated and the percentage of the various sizes actually required by method invocations. A typical JVM method invocation is preallocated up to 12 words (with an average of 4 and a standard deviation of 1.7) and requires almost the same number (the average is 3 and the standard deviation is 1.74). Stack preallocations through hierarchical method invocations require up to 1024 locations, as shown in the plot in Figure 2.24, where we gathered operand stacks from successive invocations into one common stack. The average is 4 and the standard deviation is 2.18. Sixteen locations can support more than 75% of the hierarchical invocations.

Easy handling of the operand stack requires a specific hardware module. We suggest 1024 locations as the size of this stack. This should not be an obstacle with today's technology. We also suggest allocating a new stack frame on top of the caller frame, starting from

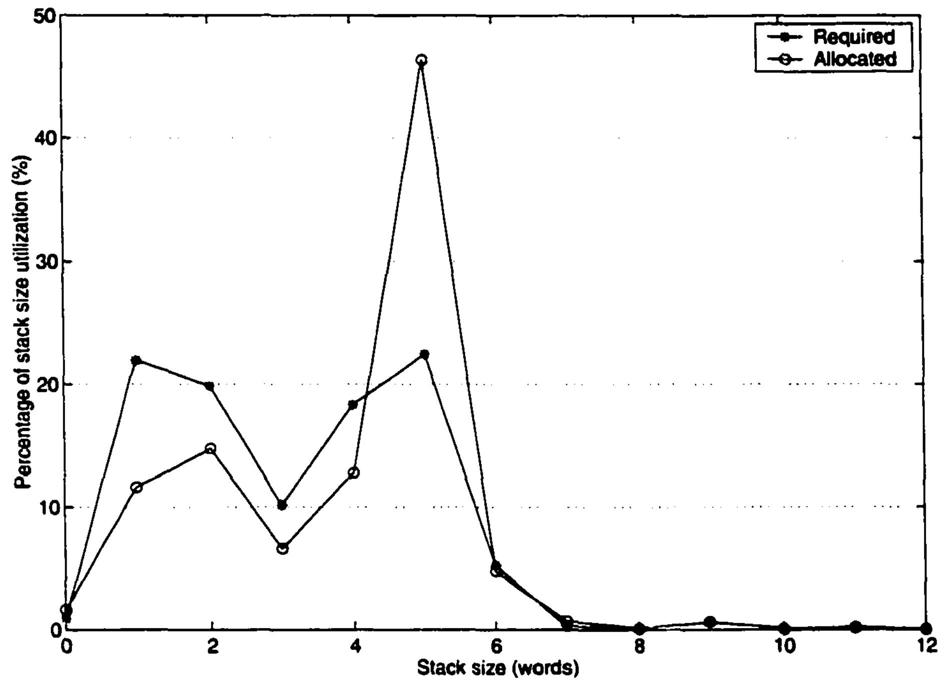


Figure 2.23. *Summary of individual method invocation stack sizes.*

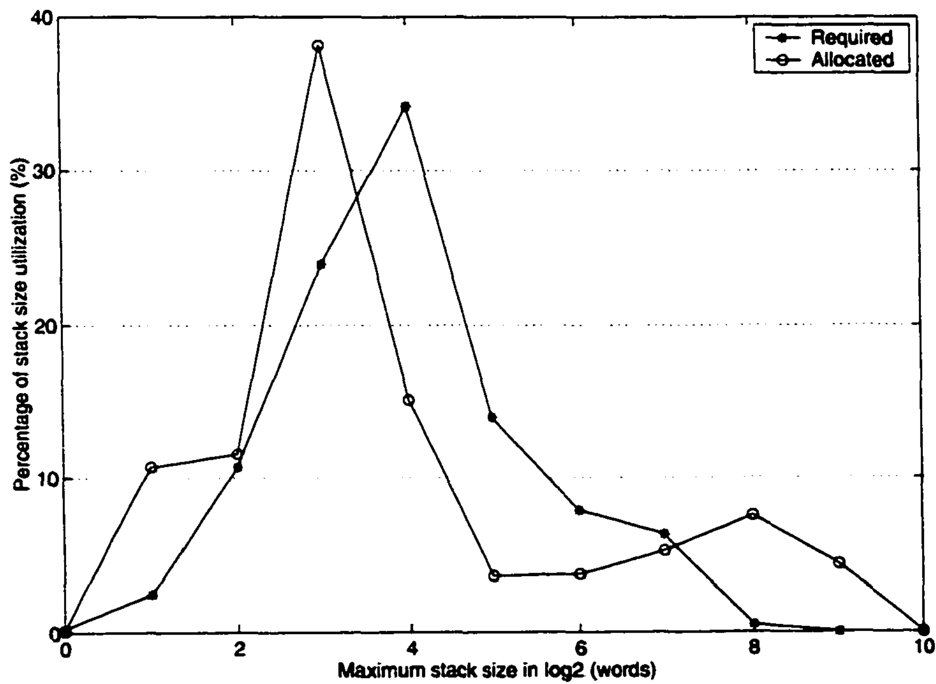


Figure 2.24. *Summary of stack sizes in hierarchical method invocations.*

its current location (not after the boundary of the allocated stack). This is motivated by the observation that the actual accumulated stack sizes are concentrated in lower stack portions. To guarantee that no overflow will occur, the active method preallocated stack size should be kept. This overlapping scheme will minimize the stack size required. Figure 2.22 illustrates the savings gained by this scheme.

2.9.4 Stack Primitive Operations

The basic effect of any JVM instruction on the stack is either pushing or popping. The amount of pushing and popping is of special importance in designing architecture to support the Java operand stack. Figure 2.25 shows the change in stack size during execution. This figure shows that the stack normally handles very few words, with pushing or popping of two words covering more than 98% of the overall stack size. The average is 0 words being pushed or popped with a standard deviation of 1.15. From Figure 2.25 we see that all JVM instructions effectively span from pushing 2 to popping 4 words. Method invocations pop their parameters from the stack.

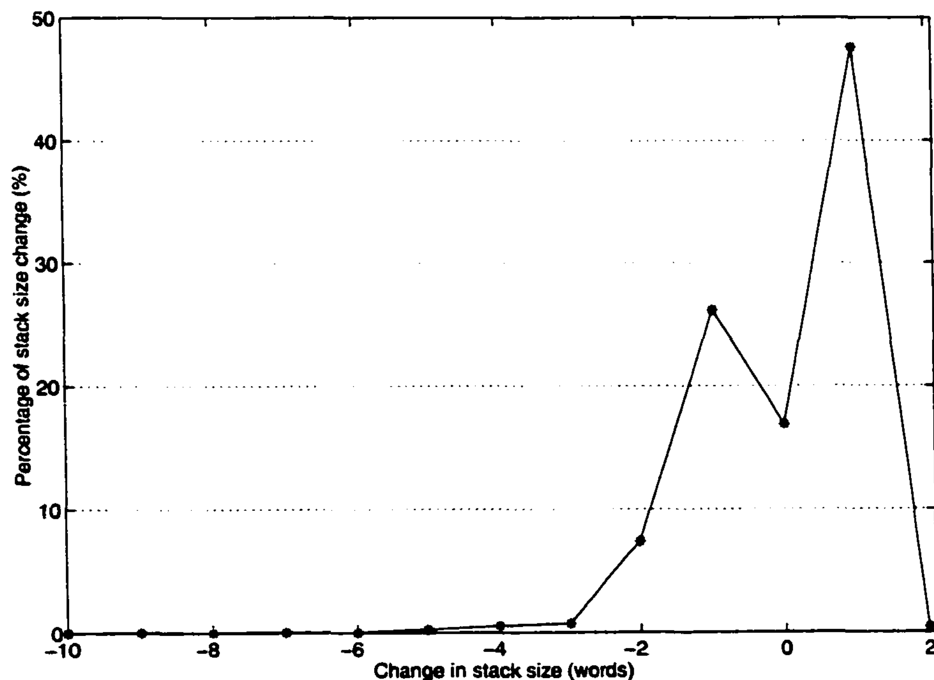


Figure 2.25. *Summary of stack size changes due to instruction execution.*

As Figure 2.25 shows, a typical JVM method pops up to 10 parameters. Figure 2.26 partitions all JVM instructions according to the effect on stack size. An interesting pattern can be observed: pushing on the stack is more common than popping from it. The explanation for this is that upon returning, a JVM method can ignore some data on the stack, which can be considered as popped already. Figure 2.27 shows the distribution of this ignored stack size. A hardware stack structure that is built using a register file may need a mechanism to handle flushing the ignored stack data. As the figure shows, although neglected stack data can be up to seven words, the ignored stack size is mostly one or two words.

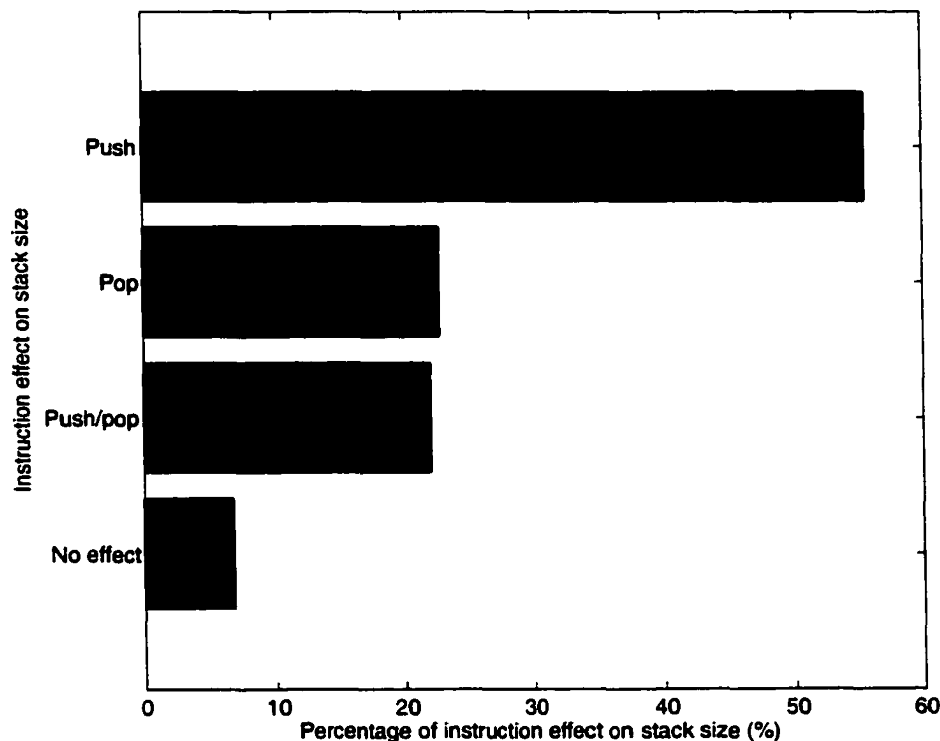


Figure 2.26. *Summary of instruction effects on stack size.*

We conclude from these results that a Java processor should provide hardware support for its stack. This stack module can either be a specific unit or a reconfigurable register file. In either case, basic stack operations need to be more efficient, ideally complete in less than one clock cycle. Special attention should be paid to method invocations. A hardware stack could accelerate stack setup and stack data flushing by creating new stack frames and destroying existing ones as quickly as possible. Support for parallelism is also desirable,

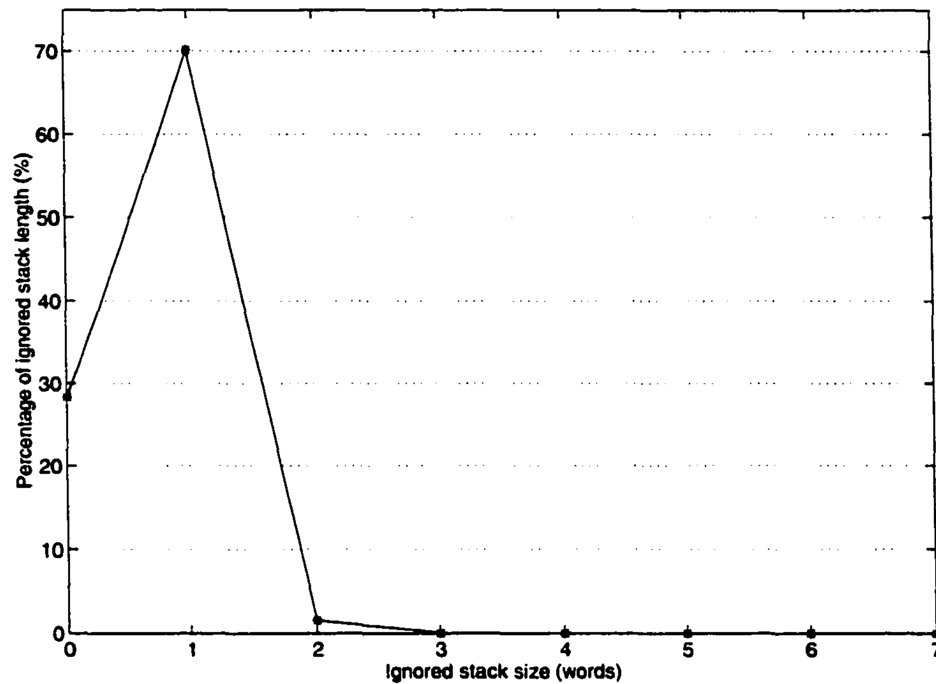


Figure 2.27. *Summary of stack sizes ignored per method invocation.*

which could be achieved by permitting random access or multiport access to the stack.

2.9.5 Native Invocations

In evaluating the native instruction execution, we counted the occurrence of the word *native* in the trace. We found it in 215 trace lines which accounts for 0.0029% of the total number of trace lines. This verifies that Java is highly portable.

2.10 Effects of Object Orientation

Results in Sections 2.6 and 2.8 show that the object manipulation group takes a major share of the instruction utilization and execution time. In this section we concentrate on some aspects of object orientation.

2.10.1 Frequency of Constructors Invocations

Constructors are vital for object orientation. They are invoked once for each instantiation of a class. Figure 2.28 shows the most invoked constructors. The *String* class is instantiated far more than the others. This hints at hardware support for string manipulation.

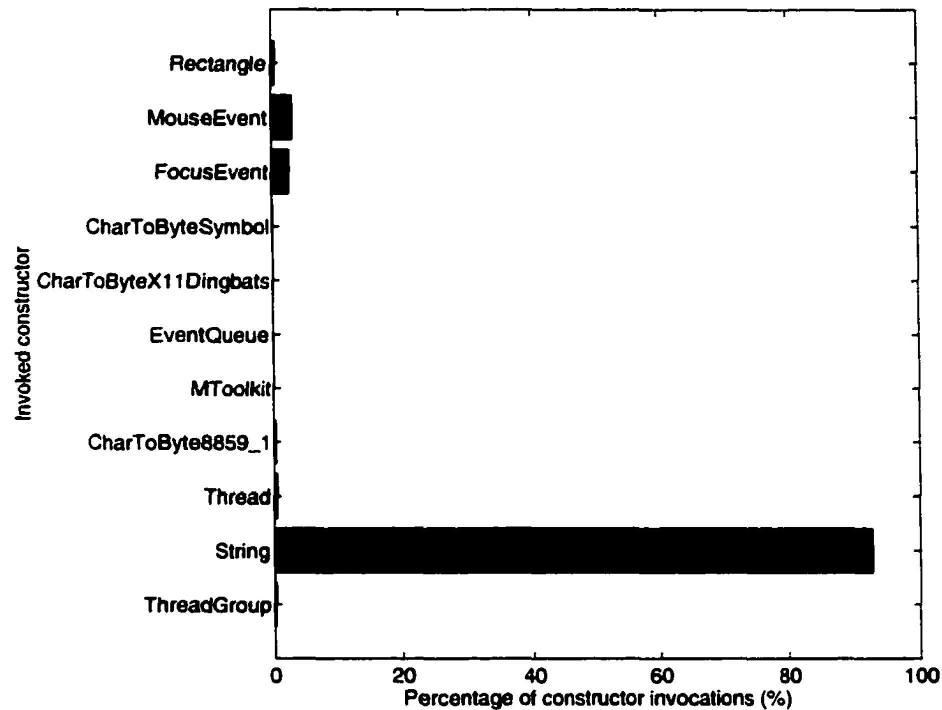


Figure 2.28. *Distribution of invoking different constructors.*

2.10.2 Heavily Used Classes

It is important to know the most invoked classes so that they can be optimized. In CaffeineMark, the most heavily used classes are the *Object*, *Thread*, and *String* classes, with their variants like *ThreadGroup*. (Only API classes are considered. We excluded the user defined and mouse-movement classes, as they are different from case to case.) This result should not be surprising. As a full object-oriented language, Java makes heavy use of the *Object* class. In Java, all classes inherit the *Object* class. Multithreading services are used by web applets to run concurrently with the browser: all applets must use the *Thread* class and inherit its characteristics. The *String* class is used extensively for symbolic referencing.

Further analysis of the performance of instructions that are heavily used inside these classes will help in better identifying performance critical instructions. Figure 2.29 shows the execution time in manipulating the three heavily used classes per instruction class. The object manipulation and complex groups dominate the others. As a generalized profile for the object manipulation instructions, Figure 2.30 shows the execution time in manipulating the three classes, for individual *non-quick* and *quick* object-specific opcodes, respectively.

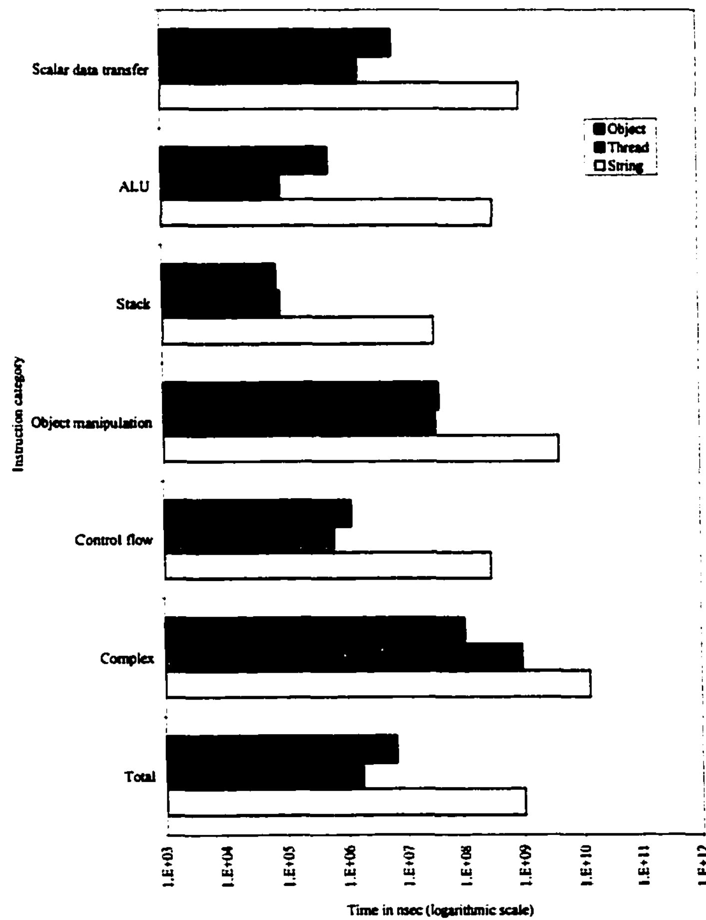


Figure 2.29. *Total execution time per Object, Thread, and String classes.*

2.10.3 Multithreading

CaffeineMark does not make heavy use of multithreading. It executes a total of 9 threads. To have a rough idea about multithreading utilization, we counted the occurrence of the

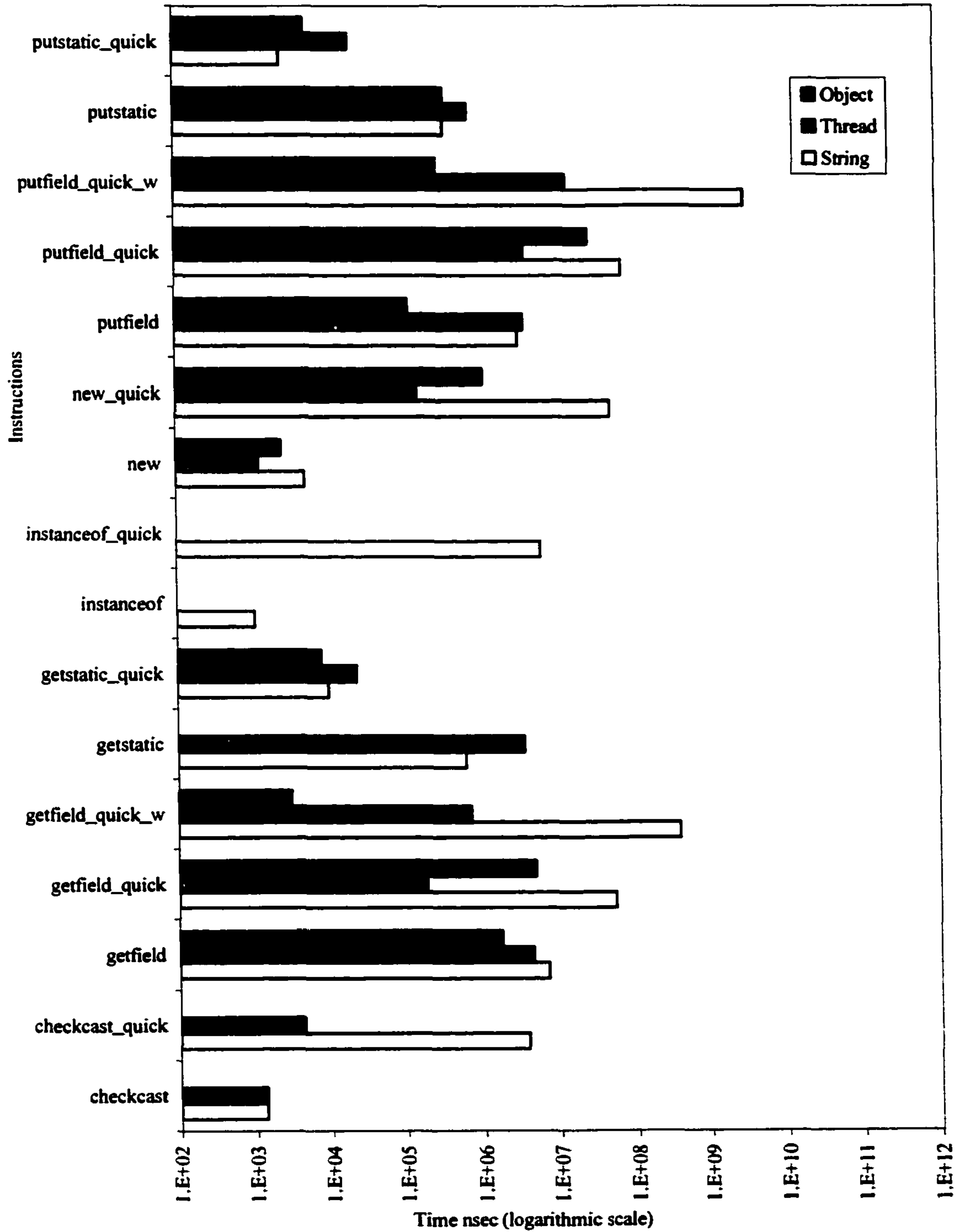


Figure 2.30. *Total execution time for object-specific instructions.*

word thread in the trace. We found it in 1763 lines which accounts for 0.0234% of the total number of lines. As our benchmark is not a multithreaded application, it shows that even though there is no multithreading at the programmer level, multiple threads are still used. Generally, threads are used behind the scene for graphical user interface (*GUI*), networking, etc. This indicates the performance gain that could be obtained by providing support for multithreading at the hardware level.

2.10.4 Memory Management Performance

An effect of Java's object-oriented programming style is that new objects are allocated for temporary values during program execution. This means that new objects are required for each interrupt or exception. This adds much overhead especially to the *Thread* class, which relies heavily on interrupts and exceptions for context switching and error handling. Java also depends heavily on garbage collection to free any unused memory fragments. Thus, the garbage collector efficiency greatly affects the performance of Java. The garbage collector is normally assigned a low priority thread. This means that it starts only when no other task is being executed. When the JVM faces a situation in which it needs to allocate memory and there is hardly enough, the garbage collector is rushed to free some of the unused fragments. Figure 2.31 shows the time spent by every memory allocation instruction over the life time of CaffeineMark (Sun's JDK, that was benchmarked, uses the mark-and-sweep algorithm for garbage collection.) The figure shows a steady time response with rising and falling spikes. The rising spikes indicate instances in which the garbage collector was called to free some space. As a matter of fact, the allocation instruction itself was blocked waiting for the garbage collector to finish. The falling edges show that there was enough memory for fast allocation. We can also see from the figure that the garbage collector invocations are spread out at regular intervals. This suggests the benefit of having a module working in the background observing memory references and deleting unused objects regularly.

2.11 Conclusions

In this chapter we conducted a behavioral analysis of the JVM ISA. The major contribution of this work is that it collects many aspects of Java workload behavior on the tested JVM

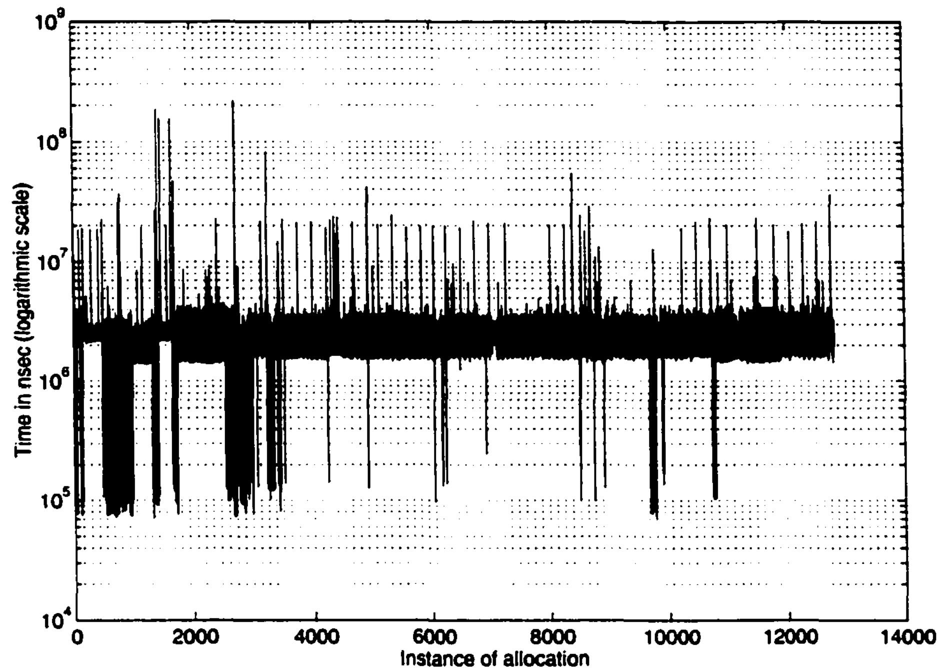


Figure 2.31. *Time requirements for memory allocations.*

(Sun's JDK). To the best of our knowledge, this is the only work that covers all the aspects discussed in a single study. (Other studies had focused on smaller subsets of the aspects considered here [70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85].)

Recommendations are made for JVM architectural requirements by analyzing collected statistics in the benchmark execution. Tables 2.11 and 2.12 summarize the observations and recommendations for a Java processor's architectural requirements.

Our study clearly shows that the advanced features of Java are its weakest links in terms of performance. It is exactly the aspects of Java relating to modularity, object orientation, and platform independence that are the major performance bottlenecks. These features come at the cost of inefficient memory management and method invocation overhead. Hardware support is required to increase the efficiency of object instructions and the object manipulation management system. Among possible ideas for this support are the following:

- **Allocate global CP** A heap for temporary objects (which have a very short life) and other high priority objects (such as exceptions) can be allocated efficiently using this global pool. Garbage collection can be optimized for this pool to make allocation of

Table 2.11. *Observations and recommendations for JVM instruction set design.*

Aspect	feature	Observations (O)/Recommendations (R)
Access patterns for data types	Single-type operations	**O:** Integer instructions dominate typed operations
		R: Support integer and reference data types
		R: Adopt 32-bit datapaths and register files
		R: Provide multiple integer functional units
		R: Process integer operations efficiently
	Type conversion operations	**O:** Conversion from integer dominates
		O: Conversion from integer to character dominates
		R: ALU should process conversion operations quickly
Addressing modes	General	**O:** Traditional addressing modes are not applicable
		O: LV, immediate, and stack accesses dominate
		R: Support LV access
		R: Support stack addressing
		R: Support immediate addressing
	Hardcoded operands	**R:** Encourage hardcoding code generation
	Quick opcodes	**O:** The quick versions are executed more frequently
		R: Hardware needs to support dynamic linking
		R: Hardware for symbolic resolution is required
		R: Replace cached non-quick operations with the quick version
Instruction encoding	General	**R:** Could select a format different from the specification
	Immediates	**R:** 8-bit field in the instruction is sufficient
	Array access	**R:** 6-bit index is suitable
		R: Cache up to the first 64 array elements
	CP indexing	**R:** 16-bit field is required
	LV indexing	**R:** 5-bit index field is required
		R: Allocate LVs on-chip
		R: Use the register file as a LV reservoir
		R: Allocate at least 32 registers for LVs
	Branching distance	**R:** 8-bit offset is the general case
		R: Provide a 512-instruction branch target buffer
	Instruction length	**O:** JVM instructions are variable in length
		O: Average is 1.77 bytes
		O: Two-operand operations are the common case
		R: VLIWs can accommodate multiple JVM instructions
		R: Relax the size requirements for instruction prefetch buffer
		R: Relax the size requirements for cache blocks

Table 2.12. *Observations and recommendations for JVM HLL support.*

Aspect	feature	Observations (O)/Recommendations (R)
Instruction utilization	Most frequently used instruction subclasses	**O:** Load/store from/to LVs **O:** Object-specific instructions **O:** Conditional branches
	Comparisons	**R:** Have a zero register in hardware **R:** Have a null register in hardware
	Branch prediction	**O:** Forward conditional branching is common **O:** Compare stack and branch is used more than compare stack and push **O:** Both forward and backward are usually taken
	Effect on stack size	**R:** Avoid parallelizing instructions that access the stack simultaneously
	Impact on instruction encoding	**O:** The control flow class requires the longest instruction length **O:** The object manipulation set uses the largest CP size **O:** The control flow instructions need the largest immediate size **O:** The scalar data transfer subclass accesses LVs with larger index size
Time requirement	General	**O:** Method handling instructions are the most time consuming **O:** Programs spend 90% of execution times with 10% opcodes
Method invocation	Hierarchical calls	**O:** Invocations can be up to 120 levels
	Required LVs	**R:** Eight LVs per method is a typical requirement **R:** A total of 256 LVs are sufficient most of the time
	Operand stack	**O:** Stack size/method averages 12 words **O:** Pushing on stack exceeds popping from it **R:** A total of 1024 locations are sufficient most of the time **R:** Overlapping stack allocations is recommended
	Stack primitive operations	**R:** Provide a hardware stack **R:** Accelerate stack setup and flushing **R:** Provide a stack with multiple ports **R:** Construct a stack that allows random access
	Native invocations	**O:** Java is highly portable
Object orientation	Constructor use	**O:** String constructors dominate
	Heavily used classes	**O:** String manipulation classes are heavily used **R:** Support string operations
	Multithreading	**O:** Multithreading is at the heart of the JVM
	Memory management	**R:** Support efficient garbage collectors

temporary objects more efficient.

- **Global LV area** Our study shows that manipulating LVs is one of the critical performance issues. This suggests a global pool of registers for LV allocation.
- **Stack cache** Since the JVM is a stack machine, hardware support for the operand stack will improve the performance. Caching the stack in hardware will reduce memory latency. This will not only affect positively the stack instructions, which occur frequently, but nearly all other instructions that implicitly access the stack.

We note that benchmarking results depend on the OS and hardware used in making the measurements. In addition, a different Java compiler may yield slightly different results. We do feel that our measurements obtained using CaffieneMark, and the analysis performed, are highly indicative of the behavior of a large class of typical Java applications. In general, the results collected here are tied to the JVM specification. However, the bytecode interpreter and the Java compiler used have a certain impact on some of the results collected. For example, the optimization techniques adopted by the compiler might affect the various percentages of instruction utilization. Similar performance studies targeting the interpreters and compilers would be needed to decouple the effect of these software components on the statistics gathered.

The work in this chapter has been published in [86, 87, 88, 89, 90].

The next chapter analyzes the design space of Java processors by exploring the available design options and their trade-offs.

Chapter 3

Design Space Analysis of Java Processors

3.1 Introduction

The best way to describe and discuss design methodologies for Java processors is to present them within their design space framework [91]. In this chapter, we explore the different options available to the architectural designer. Our study aims at highlighting these options and their trade-offs by traversing the design space tree top down. We will concentrate on the innovative features that are special to Java execution. Some early design decisions for our proposed architecture will be made and presented here. We will use picoJava-II, an already existing Java processor, as an example to examine what it offers and what desirable features are missing in its design [92].

Figure 3.1 shows the most important design aspects of Java processors emphasized in this chapter: design methodology (Section 3.3), execution engine organization (Section 3.4), parallelism exploitation (Section 3.5), and support for HLL features (Section 3.6). The design space notation is introduced in Section 3.2. Section 3.7 presents related conclusions.

3.2 Design Space Trees

The study presented in this chapter follows the same approach used in [91] of using the design space and its trees. This approach allows exploring different design options and aspects in a convenient graphical way. Figure 3.2 shows the three primitives used in building

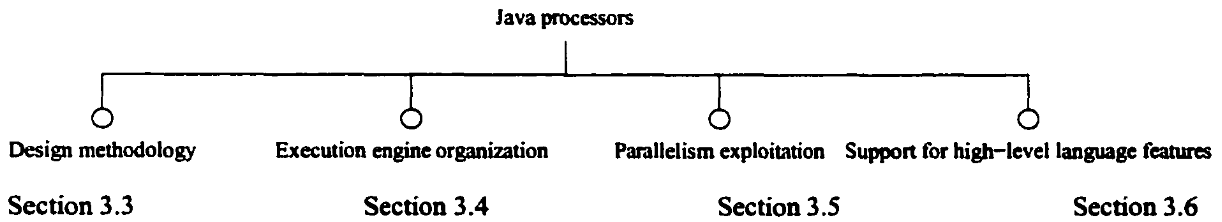


Figure 3.1. *Java processors design space.*

and expressing the meaning of the branches of the tree. In the course of our study, options selected for use in our architecture, if any, are shown in italic.

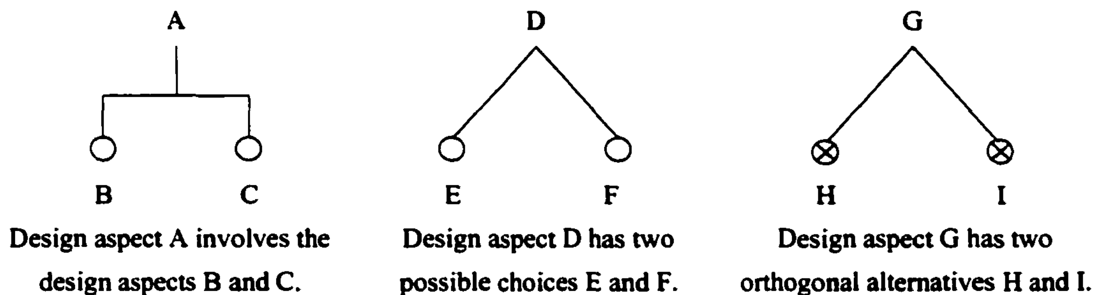


Figure 3.2. *Design tree elementary primitives.*

3.3 Design Methodology

Java processors can be implemented using a number of approaches. Each approach has its own pros and cons. Here we discuss the different methodologies to design hardware support for Java. Figure 3.3 explores the design space of Java processor hardware design methodology.

3.3.1 Generality

Java processors could either be Java-specific, a core that only recognizes Java, or *general-purpose*. General-purpose processors execute Java, as well as other languages, on a core that may have some Java-related features that convert JBCs to the processor's native code.

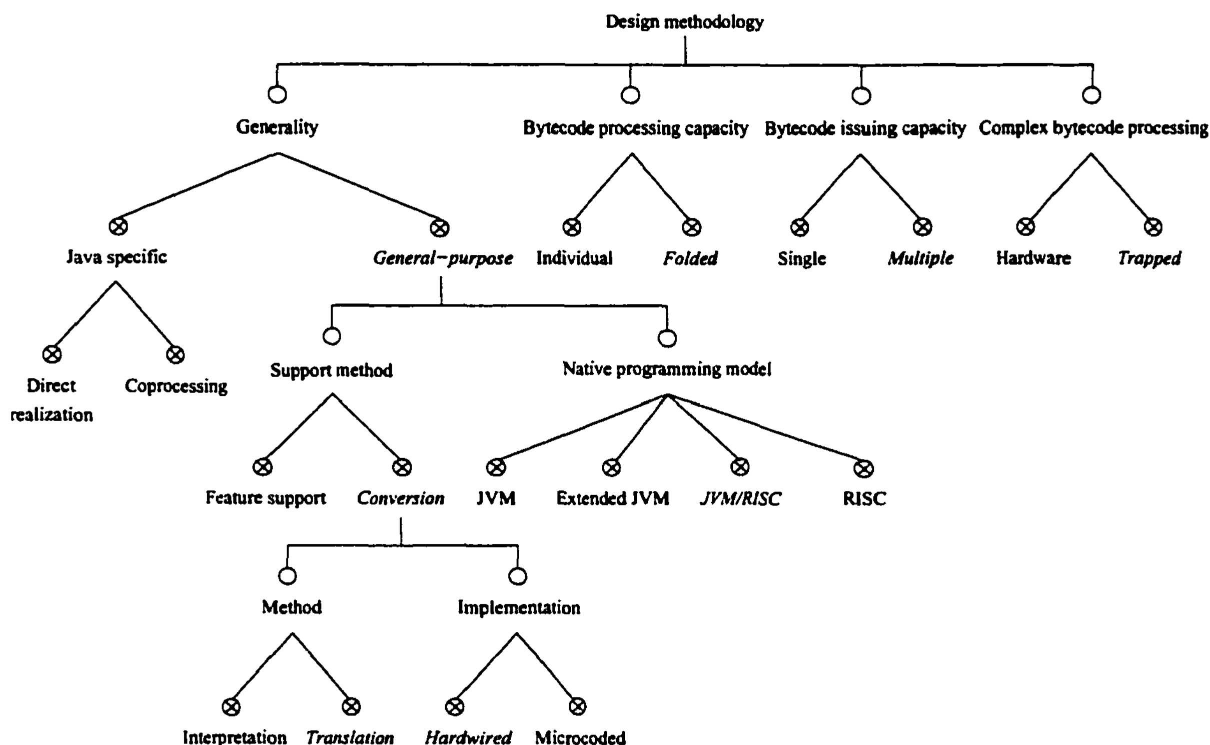


Figure 3.3. *Different hardware design approaches for Java processors.*

3.3.1.1 Java-Specific Approaches

Java-specific processors are designed to execute JBCs as their only native instruction set. Two schemes can be classified under this approach:

- **Direct realization** This approach incorporates a direct implementation of the JVM model in hardware, producing a stack machine executing most of the simple JVM opcodes. (In Subsection 3.3.4 we will discuss the different approaches for handling complex JVM opcodes, including method invocations, exception handling, etc.) Sun Microsystems' Java chips picoJava-I, picoJava-II, and microJava [93, 94, 95, 96, 97], DCT's Lightfoot and Bigfoot processors [98], and Patriot Scientific's PSC1000 [21, 99, 100] fall under this category. Applications written in languages other than Java can still run on these Java-specific processors. However, this will require compilers to produce JBCs. Sun Microsystems, for example, introduced C/C++ compilers that generate JBCs [21]. Some academic researchers have shown the complete process in designing Java-Specific cores using VLSI and FPGA technologies [101, 102, 103,

104, 105, 106].

- **Coprocessing** JBC coprocessors are attached to a general-purpose CPU core to translate JBCs into native instructions on the fly. This approach requires no modifications to the host CPU core, OS, or application software [107, 108, 109, 110]. Whenever the CPU encounters a JVM bytecode stream, it passes it to this coprocessor. This unit can be interfaced to the CPU through a plug-in card or it can reside on the same chip (in a two-module ASIC configuration). This approach facilitates migrating existing applications to Java-based ones and still maintains the generality of the system. Sun Microsystems introduced its Java blaster, an ISA card that turns PC's into Java based computers [26]. NSI's Java software coprocessor JSCP includes its own memory manager and thread scheduler. This makes it portable to real time OS that may have only part of the functionality necessary to implement Java [59].

3.3.1.2 General-Purpose Approaches

General-purpose architectures for executing Java have two logical views: a JVM ISA and a RISC-based one. In the JVM view, programmers are provided with an architecture that is optimized for JBC execution. In the other view a normal RISC architecture is available. At the heart of the architecture a RISC core is utilized for both views. A combination of hardware and software techniques are used to transform JBCs to the processor's native instructions. The statically determinable type state property of JBCs enables an easy realization of such transformation [1]. The native ISA of such processors could either be a common one, e.g., MIPS, or a special instruction set designed to closely match the JVM instruction set. Processors using this approach have the advantage of being general-purpose, which allows executing codes produced by non-Java applications. However, such approaches are usually more complex to construct than Java-specific ones.

The design space for general-purpose approaches consists of two items: support method and native programming model.

General-purpose processors follow one of the two methods to support Java in hardware: feature support or *conversion*.

- **Feature support** Processors can support Java by providing features that match the JVM architecture. Sun Microsystems' MAJC architecture is an example of such processors [111, 112, 113, 114, 115]. To support Java multithreading, MAJC exploits

scalable parallelism at the instruction, data, and thread levels. At the highest level of parallelism, support is provided as multiple processors on a chip. Next, hardware support for threading is included. Then comes an improved VLIW architecture. At the lowest level of parallelism is the single instruction multiple data (*SIMD*) structure [116, 117, 118, 119, 120, 121, 122, 123, 124]. MIPS is working on improvements to its RX000 architecture aiming at speeding up the execution of Java code and saving memory and bandwidth [26]. ARM tuned its StrongARM architecture for stack-based language, such as Java. Their processor can move a stack frame in and out of the register file with a single instruction [97, 125].

- **Conversion** Processors could include additional hardware to convert JBCs to native code. Converted instructions are fed to the processor core for execution. Conversion can be done through on-chip modules or off-chip coprocessors.

There are two common JBC conversion methods: *hardware interpretation* and hardware translation:

- **Hardware interpretation** Hardware interpretation techniques incorporate additional modules that perform the functions of a software interpreter. In this approach, each JBC is mapped to several native instructions. Radhakrishnan *et al.* proposes a hardware interpretation unit that utilizes a microcode ROM. Their unit could be added to general-purpose processors to enhance Java execution [126, 127, 128].
- **Hardware translation** Processors using this approach dynamically translate JBCs to native instructions. Different from the hardware interpretation technique, hardware translation maps many JBCs to one native instruction. Related JBCs are grouped (via folding, for example) first and then translated to a single native instruction. Glossner *et al.* proposed translating JVM instructions dynamically to *Delft-Java* instructions. Using a form of hardware register allocation, they transform stack bottlenecks into pipeline dependencies that are dealt with by well established techniques (no folding is involved) [129, 130, 131, 132, 133]. In this dissertation, we propose to dynamically translate JBC folding groups into MIPS instruction format. Our approach is more economical and more general than Glossner's as it allows the use of off-the-shelf RISC cores and produces a more general architecture.

Bytecode converter circuits can be implemented in two ways: *hardwired* or microcoded.

- **Hardwired** Hardwired Java processors implement all bytecode conversion modules using random logic circuits.
- **Microcoded** Microcoded Java chips implement the JVM instruction set as a microcode library that contains interpretation code for every supported bytecode. Imsys's rewritable microcode chips GP1000 and Cjip [134, 135, 136] have instruction sets for Java, Forth, and C/C++ using a JVM-like stack architecture. aJile Systems' aJ-100 Java processor also supports the critical parts of JVM's instruction set in microcode on stack-based embedded processors [137, 138]. Rockwell has modified an in-house processor to execute JBCs directly as its native instruction set by replacing the instruction set with new microcode for bytecode interpretation, producing JEM1 and JEM2 [21, 51, 139]. As mentioned before, Radhakrishnan *et al.* proposed a microcoded Java chip that interprets JBCs dynamically in hardware [127, 140].

General-purpose processors with Java support have either a JVM, an extended JVM, a *JVM/ RISC*, or a RISC native programming model:

- **JVM** This type of processor has only one native programming model, the execution of JVM code. However, as JVM's current instruction set is not sufficient to do system level operations, this approach has very limited applications.
- **Extended JVM** These Java processors extend beyond the defined JBCs with some system level operations that increase the generality of the processor in executing non-Java code. A careful inspection of the JVM ISA reveals a lack of opcodes executing the following operations: direct memory access, interrupt handling, multithreading assistance, multimedia extensions, cache administration, and status monitoring. JVMs typically rely on library calls to the underlying OS to perform these functions. picoJava-II extends the JVM ISA with 115 new instructions [92]. Added instruction types are diagnostic, register reads and writes, arbitrary loads and stores, other language support, and system software support. Extending the JVM would enable programming languages other than Java to run on such processors.
- **JVM/RISC** Processors of this type have two integrated programming models: one based on the JVM ISA and the other based on a general RISC machine. Siemens has developed a smart Java card based on Siemens' Triple-E embedded processors. The processor's instruction set is a superset of the original architecture instruction set. The resulting CPU is bilingual, executing either Triple-E code or JBCs [141]. Strøm

et al. have implemented a processor core in which two virtual processors operating independently share the same datapath [142, 143, 144]. However, in their proposal the two processors do not run in parallel. Strøm's proposal is based on the observation that approximately 50% of the RISC instruction set is common with the JVM architecture. By extending the RISC instruction set to include the JVM-specific instructions, both JVM and RISC codes can run on their processor. In this dissertation, we propose a dual-input pipeline architecture, where JBCs and RISC binaries enter the pipeline separately at the front end. JBCs are folded and dynamically translated to RISC binaries, which are then processed by the back end modules of the pipeline.

- **RISC** These are RISC processors that provide feature support for Java without executing any of the JBCs natively.

3.3.2 Bytecode Processing Capacity

JBCs could be processed/translated individually or *folded*. Bytecode folding based processors could execute stack push and pop in zero time by folding them with the main ALU operations. If the stack and LVs are allocated to registers, data dependency resolution techniques such as bypassing and data forwarding can save the time spent in manipulating the stack. Sun Microsystems' Java chips picoJava-I, picoJava-II, and microJava execute the JBCs folded [66, 145, 146, 147, 148, 149, 150]. Translation-based processors could also fold JBCs before translating them as we propose in this dissertation.

3.3.3 Bytecode Issuing Capacity

Converted JBCs can be issued singly or in *multiples*.

- **Single-issuing** In this option conversion code for a single JBC is issued one native instruction at a time to a regular RISC core [140].
- **Multiple-issuing** Three scenarios could take place here: **(1)** multiple JBCs are converted to one native instruction that is issued to a single-issuing core (as we propose in this dissertation); **(2)** a single JBC is converted to multiple instructions that are issued to a multiple-issuing core; or **(3)** multiple JBCs are converted to multiple instructions that are issued to a multiple-issuing core.

3.3.4 Complex Bytecode Processing

Executing all JVM instructions in hardware might not be feasible as the JBCs are highly different in their complexity. For instance, while JVM has an instruction like `iconst_0` that pushes a zero on the stack, it also has `amultinewarray` that creates a multidimensional array. Implementing such a complex instruction in hardware may complicate the design and make it unnecessarily slow. A more aggressive approach might select to implement in hardware only those instructions that directly improve Java execution, including the simplest ones. Complex, less commonly used instructions (e.g., array/object management, synchronization, method invocation, etc.) should be *trapped* and emulated in software. Overall, 23 JBCs are trapped and emulated in software in picoJava-II [92, 151].

3.4 Execution Engine Organization

In this section, we examine the main processing engine that is suitable for both Java and other languages. We will discuss, as shown in the design tree in Figure 3.4, themes related to complexity, registers, caches, stack, and execution units (*EXs*).

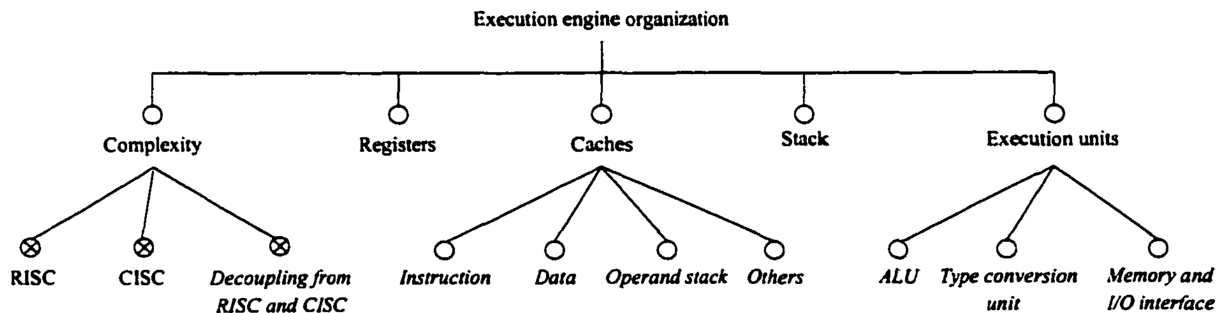


Figure 3.4. *Execution engine organization design space.*

3.4.1 Complexity

ISAs range in complexity between the CISC and RISC styles. As for a processor supporting an HLL, the CISC framework might be appealing initially. This style may be more suitable for executing the complex operations incorporated. However, this paradigm may not be able to use the efficiencies of all modern RISC technologies. In this subsection, we compare

the JVM against RISC, examine which CISC features would help in executing Java, and discuss how an architecture decoupled from both styles could be more desirable.

3.4.1.1 JVM Compared to RISC Cores

Here we compare both RISC and JVM execution in order to identify their differences and assess the suitability of the RISC style for Java. Based on this study we can highlight architectural features that need to be modified to narrow the semantic gap between the two.

- **Register centricity** In contrast to RISC machines that depend on a large set of registers, JVM addressing modes involve stack manipulation, making a JVM stack the dual of the RISC registers. This duality has a number of aspects: **(1)** being local to the CPU, registers are faster than the JVM stacks that are implemented in the memory; **(2)** registers are more efficient than the stack as they allow out-of-order (*OOO*) evaluation of expressions, whereas the stack forces only in-order evaluation of expressions; and **(3)** RISC code would simply reuse the value stored in the register file without any need for reloading or duplication, whereas popping a value from the operand stack deletes it. If this value is needed in subsequent operations, a load instruction would be called or the special instruction `dup` could be used to duplicate operands on stack to reduce unnecessary loads. In summary, allocating the operand stack and LVs in hardware in a reconfigurable register file would narrow the JVM/RISC gap, reduce operand access time, and reduce memory traffic.
- **Memory access** LVs are accessed in JVM only for loading and storing. All operations are done on the stack except in the case of `iinc` instruction that increments a LV directly. Allocating LVs and the stack in hardware diminishes the frequency of memory access and makes the JVM a load-store architecture.
- **ISA simplicity** JVM is similar to RISC in that it has a simple instruction set. This is efficient for pipelining and compiler optimization. However, JVM has a group of object-oriented instructions, which are somewhat complex. Using a RISC core requires emulating these instructions in software.
- **Instruction length** RISC always encodes all instructions in fixed length instructions (usually 32-bit). However, JVM operations are variable in length. The most frequently used lengths are from 1 to 4. As shown in Chapter 2, the JVM dynamic average instruction length is 1.7732 and the standard deviation is 1.4. So compared

with the RISC code, JVM has a smaller code size. This is an advantage for JVM since a number of its JBCs can be folded in a single RISC instruction.

- **Data types** RISC compilers rarely know much about the data types they manipulate. JVM, by contrast, employs a type from a strictly defined hierarchy. Type information should be filtered out before passing the instruction to a RISC-based processor.
- **Addressing modes** Like other stack languages, JVM works only in one of the two addressing modes; stack or memory. Operands of an instruction come from the same source (the stack in case of ALU). These homogeneous and simple addressing modes are similar to those used in RISC instructions.
- **Parallelism** JVM stack code requires a lot of effort (e.g., doing swaps of stack entries, instruction folding, reservation stations, etc.) to parallelize. However, extracting ILP from RISC code is easier.

3.4.1.2 CISC Features for JVM Implementation

Interestingly enough, the CISC style provides some merits that suit Java's stack structure:

- **Stack manipulation** CISC processors are rich in such instructions (the $x86$ uses the stack for floating point operations). CISC's `push` and `pop` instructions are very handy for JVM, but they were replaced in RISC cores with two instructions: a register increment or decrement followed by a load or a store. So, CISC engines might handle some Java stack operations more efficiently than some RISC processors.
- **Register count** Although traditional CISC cores appear in a weak position compared to RISC in the number of available physical registers, Java's stack organization compensates for this by using LVs and the operand stack.
- **Instruction length** JVM is like CISC machines in supporting variable length encoding. Experimental results in Chapter 2 show that JVM instructions range from 1 to over 100 bytes in length. Variable length encoding promotes small code size but complicates the decoding process.
- **Wide formats** In contrast to the RISC processors, JVM borrows the wide format from some of the CISC processors (like the $x86$). The JVM's `wide` modifier extends the next instruction to accommodate longer indices. RISC has eliminated such complications and has cast all instructions in the same format.

3.4.1.3 Decoupling from RISC and CISC

An efficient JVM implementation should combine the advantages of both CISC and RISC designs in one microchip. This approach assumes neither a pure RISC nor a classic CISC, but a hybrid style. Such an organization provides a front end CISC section in the CPU chip to translate the JVM code into RISC-like instructions for a back end RISC core that performs superscalar and/or superpipelined execution [42]. The methodology adopted in this dissertation follows this line of development.

3.4.2 Registers

JVM provides no directly-accessible general-purpose registers; it only uses 4 user-invisible special purpose registers to keep state information necessary for code execution [9].

Although JVM does not require accessing general-purpose registers, a good JVM hardware realization is likely to try to use hardware registers to hold at least the first few LVs. In this case getting the compiler to keep more frequently used values in lower-numbered LVs may improve performance. In addition a general-purpose processor that runs both Java and non-Java code will definitely demand having a register set for non Java code.

picoJava-II has introduced additional registers to serve as CP pointers, keep track of filling the S-cache, manage threads, function as a program status word register, represent traps and breakpoints, handle C/C++ procedural calls, define allowable memory regions, and contain version number and hardware configurations [9].

3.4.3 Caches

Java processors need enormous caching support in various forms to bring data and instructions closer to the core for fast processing [152, 153]. In addition to the usual instruction and data caching, stack and LV cachings are required (Figure 3.4):

- **Instruction cache (I-cache)** JBCs need to be brought to an on-chip I-cache. This cache should be smaller in size compared to those used by RISC processors as JVM instructions are shorter in length on average. For example, picoJava-II incorporates a direct mapped, write through I-cache that can range from 0 to 16 Kbytes in size and has a line size of 16 bytes with a 32-bit wide path to the memory bus and the pipeline [154].

- **Data cache (D-cache)** D-caches are required to cache the object heap, CPs, stack overflow, etc. picoJava-II includes a D-cache which ranges in size from 0 to 16 Kbytes and has a datapath and line size like the I-cache. To improve cache efficiency, this cache is a two-way set associative, write back and write allocate one [154].
- **Operand stack cache (S-cache)** From the programmer's perspective, the JVM stack resides in memory. For performance enhancement part of that stack needs to be kept on-chip in an S-cache, as shown before. For instance, picoJava-II core caches the operand stack's top entries in its 64-entry on-chip S-cache [154].
- **Others** Other caches can be used to facilitate branching (branch target buffer and branch prediction cache) and object addressing. As the LV access subcategory is the most utilized instruction group (Chapter 2), LV caching also would speed up Java.

3.4.4 Stack

As the JVM is a stack machine, implementing an advanced hardware stack can substantially improve its performance. Figure 3.5 depicts the design space of the stack.

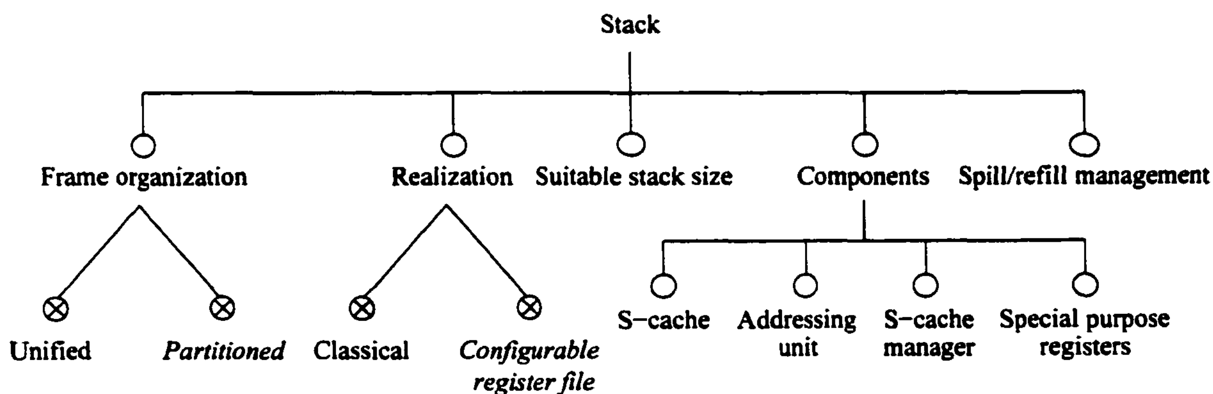


Figure 3.5. *Design space of the stack.*

3.4.4.1 Frame Organization

Java stack frames consist of three parts: LVs, operand stack, and frame data. The JVM specification does not require the three to be physically located in a single place, giving some freedom to the JVM implementer to select a convenient way to organize the Java

frame [155]. However, the specification requires accommodating both integer and floating point data, together with all other JVM supported data types, into one stack structure. The frame organization design space (Figure 3.5) can be split into:

- **Unified** In this organization, all stack parts are collected together in one shared area which is usually arranged as a classical stack with some smart features. Subsequent frames are allocated on top of their predecessors. Mapping Java frames on such an organization might be easier, but it would require full routing from each element to the different EXs to facilitate folding, etc. On the other hand, having only one hardware module for the stack shared among different threads requires certain arrangements to reach different stack parts on each context switch. picoJava-II stack uses this structure to optimize parameter passing from a caller to a method. Method calls are designed to allow stack parts to overlap between methods, enabling direct parameter passing without copying them. The machine pushes parameter values onto the top of the operand stack, where they become part of the called method LV area. Doing so saves some space which then reduces required stack spilling and filling operations [92].
- **Partitioned** This approach allocates a different hardware entity for each frame part, allowing the use of different internal organizations for each one. For example, it might be more appropriate to organize the LV area as a register file which matches very well the indexing mechanism utilized by the JVM. But the operand stack needs to be arranged as a last-in first-out structure or as a scratchpad small memory, as will be explained soon. On the other hand, as the frame data is just a list of pointers, a set of special purpose registers might be suitable for storing them. Partitioning the frame also allows a pool of resources allocated for each part and shared among different threads. This facilitates dynamic context switching. When it comes to routing the processor datapath, this approach is more efficient. It facilitates connecting each part to different EXs, which is an advantage for bytecode folding. Due to its suitability for parallelism exploitation, this approach is incorporated in this dissertation.

3.4.4.2 Realization

Although JBCs require a specific stack logical organization, it can be realized physically using different techniques, as shown in Figure 3.5:

- **Classical** Stacks can be realized using an on-chip memory organized as a classical last-in first-out structure. Although this is easier to control, it serves as a dedicated unit for Java only. Also, thread accesses to stack have to be serialized, which together with the serial nature of stack code diminish achieving any form of ILP. Classical stacks can be split into two types according to the methodology of pushing/popping (Figure 3.6): **(1)** a pointer-based structure that incorporates a sequentially addressed memory with a pointer to the top entry. This structure is simple, but has address calculation latency. Additionally, direct routing from the stack to different EXs requires access to all stack entries; and **(2)** a fixed-top stack where the top of the stack always resides at a fixed location. All existing elements are pushed down upon pushing an element and they are pushed up in case of popping an element. Although this structure might be a bit more complicated than the first type, direct routing can be achieved by connecting a few top elements of the stack to the datapath.
- **Configurable register file** A RISC-style register file can be configured to work as a stack or a register file. This will be more flexible than a dedicated stack in allowing both Java and non-Java code to be executed on the same architecture. JBC execution can benefit from the random access capabilities of register files in accessing hidden LVs and directly passing their values to EXs, which aids in the case of folding JBCs. Such an organization has another advantage: it facilitates hardware translation of JBCs by creating a circular address space that implements a buffer or a stack.

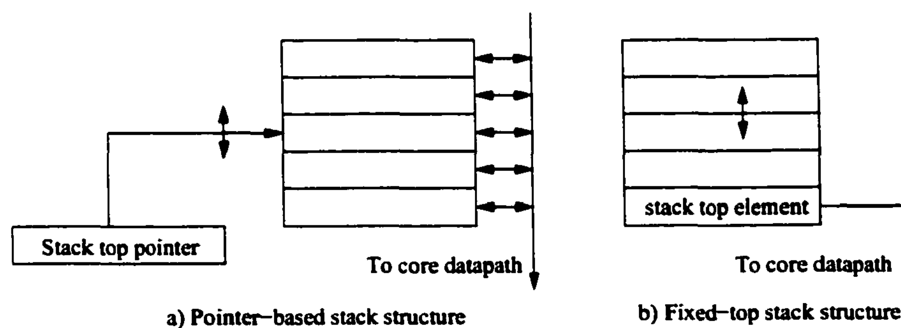


Figure 3.6. *Different ways of organizing classical S-caches.*

picoJava-II implements its stack as a single pointer based classical structure containing 64 32-bit entries. This organization aims at providing quick random access to LVs. But it can not be configured to work as a regular register file [92].

3.4.4.3 Suitable Size

The JVM assumes an unlimited stack. However, when it comes to a hardware realization of an S-cache, a question on its appropriate size is raised as a trade-off between access time and hardware resources. The larger the stack, the less the need to go back to the D-cache to spill/fill information. Whereas increasing the S-cache size implies allocating more on-chip resources. picoJava-II includes a 64 32-bit entry stack for the entire stack [92].

3.4.4.4 Components

An on-chip stack module needs, at least, the following units:

- **Stack cache** This unit caches the top most stack frames corresponding to the currently active execution threads.
- **Stack addressing unit** This unit generates addresses required to access target stack locations. The main task here is to make stack accessing independent from the organization technique (stack or register file).
- **Stack manager** This prevents S-cache underflows and overflows by moving data to and from the D-cache in advance. We will explore the design of this unit shortly.
- **Special purpose entries** These entries store the current frame pointer, the LV base, and the stack pointer. They are used by the stack addressing unit to generate the addresses required to access the data in the S-cache.

3.4.4.5 Spill/Refill Management

The finite size of the on-chip S-cache implies that it is actually holding only the top few entries of the main stack. But, an application could call several methods in a row that overflows the stack (the results in Chapter 2 show that hierarchical method invocations can go to 120 levels deep!) Conceptually, pushing the next item on the stack would cause the bottom entry to be flushed out to memory or more likely to the on-chip D-cache. Likewise, when the stack is running out of elements, popping the last element would cause the S-cache to refill from the D-cache. This means that any hardware implementation of a Java stack should be accomplished with an intelligent spill/refill module.

picoJava-II predicts and prepares for the S-cache growth in the background by a process called the dribbler. In the event that an S-cache overflow is predicted, values are saved into

the D-cache in the background. When there is room for data in the S-cache, values are restored. In practice, picoJava-II applies a hysteresis technique to this fill and flush logic. When the S-cache is nearly full, the dribbler begins moving items from the bottom of the S-cache to memory. Likewise, when the S-cache is nearly empty, the chip starts loading from memory so that the S-cache will not run dry. Programmers can set the S-cache's *high/low water mark* through a 5-bit field in a control register [92].

3.4.5 Execution Units

Java processors need to include a number of ALUs to achieve a higher degree of parallelism. Moreover, executing Java in hardware demands some more functional requirements. For example, for a strictly typed language like Java, type conversion is very important. A simple *type conversion unit* implemented as a combinational logic circuit can do this task. Additionally, a memory and I/O interface unit is needed to serve as the link between the processor core and any other interfaces that could reside on or outside the same die. picoJava-II's system bus is 32-bit wide, suitable for 32-bit data and I/O transfers [156].

3.5 Parallelism Exploitation

Higher processor performance can be achieved by increasing the degree of parallelism in executing the operations. This parallelism can be achieved at different levels of instruction issuing and execution. In this section we explore the opportunities of parallelism in Java processors. Mainly, two disciplines will be discussed: instruction level parallelism (*ILP*) [157, 158, 159, 160, 161] and thread level parallelism (*TLP*) (Figure 3.7).

3.5.1 Pipelining

In addition to aspects considered while designing regular processor pipelines, JVM brings other new issues [69]. These issues are related to stack processing, object-oriented manipulation, multithreading, etc. Figure 3.7 shows the design space for Java processor pipelines.

The number of pipeline stages is highly dependent on the processor internal organization and the bytecode execution methodology. However, deeper pipelines have a special benefit for multithreaded processors. They allow threading from more than one thread to

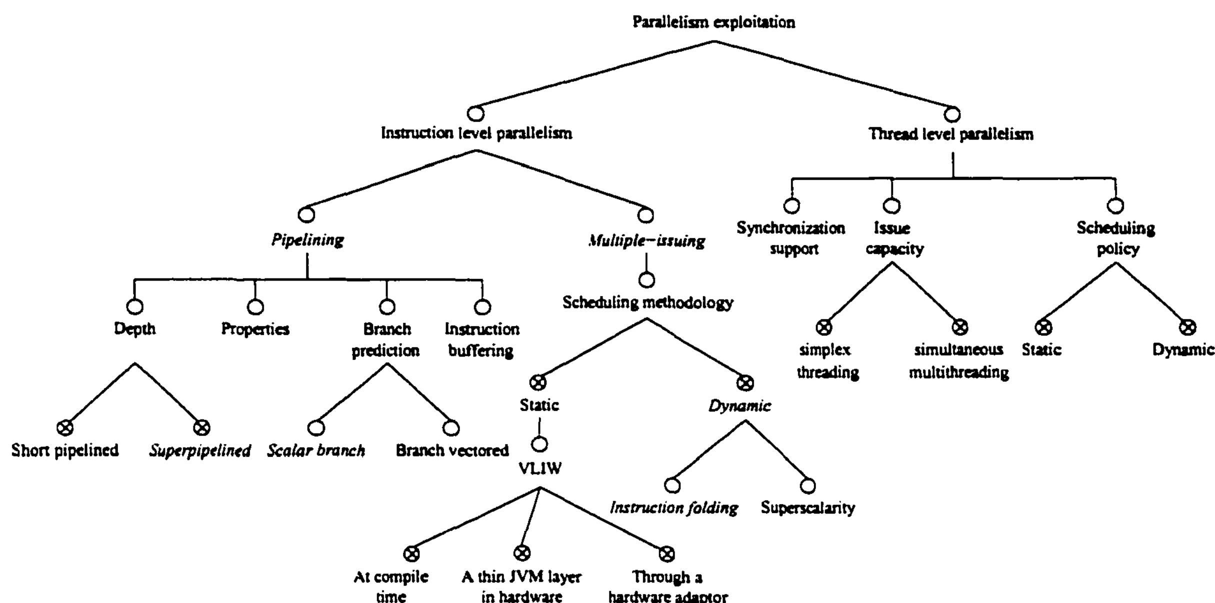


Figure 3.7. *Design space tree for parallelism exploitation.*

take place at the same time through the pipeline. picoJava-I uses only four stages, whereas picoJava-II splits two of these stages into two parts resulting in a 6-stage pipeline [92].

The stack nature of JBCs imposes some restrictions and exhibits certain characteristics on the instruction pipeline in the direct hardware execution of JBCs (an execution approach that transforms JBCs to another form may not encounter some of these issues):

- In processors directly executing JBCs, an instruction almost always depends on the result of the previous one, so only one operation at a time is allowed to occupy the execute stage. Therefore OOO superscalar execution will not be possible.
- In all likelihood a D-cache access is required as the input for the next stack instruction. Processors executing JBCs directly should not overlap access to the D-cache with the execution of the next instruction as is the case in picoJava-II.
- Accesses to the S-cache should take only one clock cycle, as is the case for accessing the register file in most processors.
- If the loaded data is required by the next instruction, an additional one-cycle load-use penalty will be incurred.
- Since compute instructions operate on stack data only and never on memory data, if the stack resides on chip, both compute and memory access instructions can be

processed simultaneously in the pipeline, as in Harvard architectures.

As in most advanced processors, an advanced branch prediction methodology should be used [162, 163, 164, 165, 166, 167]. This is especially critical for the performance of Java applications on deeply pipelined processors. picoJava-II does not have branch prediction logic. It simply assumes every branch as not taken, and there is a penalty of 3 cycles when a branch is taken [92]. Generally, JVM needs two forms of branch prediction (Figure 3.7):

1. **Scalar branching** Here the instruction has one jump-to address. All traditional techniques used for this purpose can be applied here.
2. **Branch vectored** This is useful in dealing with switch-case/table-lookup statements at the hardware level. JVM incorporates two table lookup instructions that involve multiple target addresses. Direct hardware support for these instructions will face a more complex branch prediction problem requiring the prediction of one of the multiple jump-to locations. The trivial solution involves treating switches as likely to re-use the last choice. HP's PA-RISC provides such an instruction [168].

An instruction buffer is needed to provide storage for prefetched instructions, thus keeping the subsequent pipeline modules busy all the time. A well designed instruction buffer can also help in branch prediction. picoJava-II incorporates a 16-byte instruction buffer.

3.5.2 Multiple-Issuing

Instruction issue and execution are closely related; the more parallel the instruction execution, the higher should be the number of instructions issued. In this subsection, we study how Java execution in hardware could be accelerated through multiple-issue cores. As Figure 3.7 shows, the scheduling methodology offers two options: static through VLIW and dynamic through superscalarity. Dynamic scheduling required for higher issuing rates can be achieved through a number of ways like instruction folding and reservation stations, etc.

3.5.2.1 VLIW Organization for Bytecodes

VLIW-based processors execute on pre-scheduled instructions [169, 170]. This is done at compilation time rather than at execution time. Although this static nature might prevent achieving higher issuing rates, it has the appeal of resulting in simpler hardware than the

dynamic one. It is easy to recognize a group of several combinable JVM instructions (like load, add, and store) and translate them into a single three-address operation. A high performance implementation will cache the current stack frame (or a few frames) in the processor registers; thus if LVs are used, this sequence would be implemented as a single register-to-register operation. Executing JBCs in their current formats on a VLIW engine demands reorganizing them. This can be done using one of three methods (Figure 3.7):

- **At compile time** Java compilers need to produce the required VLIW format. This requires modifying the class file format and the JVM specification in general. Compatibility issues make this approach difficult.
- **A thin JVM layer in hardware** A very thin JVM layer running on a Java processor could be designed to produce the VLIW formats from JBC streams.
- **Through a hardware adaptor** A recommended way is to organize a hardware adaptor to form the required VLIW format. While translating JBCs to the processor native ISA, it can reorganize them as VLIWs. A reasonable size cache can then keep these words for later referencing resulting in faster translation.

3.5.2.2 Superscalarity through Reservation Stations

From previous discussions, it is obvious that JVM stack code is very difficult, if not impossible, to be executed in a superscalar form. This is mainly due to the virtual dependency between successive instructions, created by using the stack. The instructions themselves may not be related at all. But, the stack forces the instructions to be issued, executed, and retired in order. As a matter of fact, any processor that directly implements the JVM architecture might not achieve superscalarity. For example, picoJava-II is not a superscalar architecture, operations are executed and retired in order. If an operand has to be fetched from the main memory during a cache miss, the CPU must stall to wait for it.

Structures that incorporate superscalar JVM architectures executing stack operations in parallel with adjacent instructions need first to find an efficient way to handle the above dilemma. The use of reservation stations is one possible approach. Reservation stations allow OOO execution, register renaming, and speculative execution. Such a structure demands allocating both the stack and LVs in registers. It will be the responsibility of a specialized unit (e.g., the folding unit) to interface the stack-based instruction stream to the register file and handle stack sharing among different threads.

3.5.3 Thread Level Parallelism

Multithreaded architecture represents a promising methodology in processor design to achieve a higher degree of parallelism and to hide input/output latencies [90]. These designs support building scalable structures to exploit TLP [171, 172, 173, 174, 175, 176].

Although thread primitives are an integral part of Java and the JVM supports multiple concurrent threads, not much information about threads is provided in the JVM specification. In current implementations threads are supported by runtime libraries and routines.

The design space for ILP includes each of the following [177]:

- **Synchronization support** JVM relies on monitors to synchronize multithreading access to different objects. picoJava-II achieves monitor entry and exit operations by the use of object header bits and entry cache [92].
- **Issue capacity** In simplex threading, one instruction from a thread is issued at a time. Although simple in structure, it lacks the ability to explore ILP. In simultaneous multithreading, one or more instructions from one or more threads are issued per cycle, thus achieving both ILP and TLP [178, 179, 180, 181]. As a fully multithreaded language, Java offers a golden opportunity for simultaneous multithreading design.
- **Scheduling policy** There exist two scheduling policy types: static where the hardware supports a fixed number of threads, and dynamic where a pool of resources are shared among a large number of threads as long as there is enough resources. Java's multithreading feature dictates the use of a dynamic scheduling policy.

The multithreading issues will not be explored further in this dissertation.

3.6 Support for High-level Language Features

Java execution can also be enhanced by supporting its HLL features. This can be achieved directly by incorporating high level operations in the native processor ISA [182, 183, 184, 185]. Another indirect way is to include some hardware modules that support high-level features like garbage collection or object addressing. In this section we discuss a number of methods for accelerating Java through HLL support (Figure 3.8). However, HLL support will not be addressed further in this dissertation.

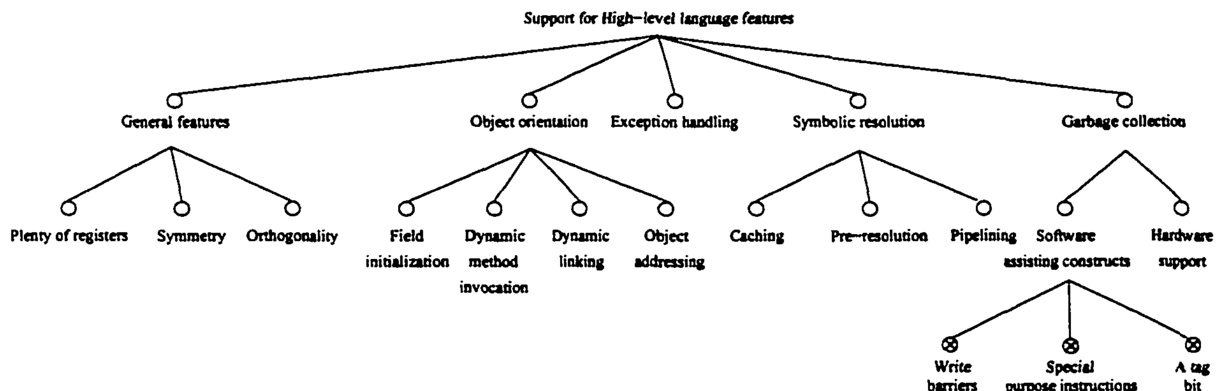


Figure 3.8. *Design space for supporting Java high-level language features.*

3.6.1 General Features

Any ISA can provide some features to assist executing Java without sacrificing the generality of the architecture [186, 187] (Figure 3.8):

- **Plenty of registers** This reduces the number of memory accesses, provides more on-chip resources for context switching, and assists compiler optimization.
- **Symmetry** This implies having an ISA that associates the same addressing modes to accessing operands in all instruction formats.
- **Orthogonality** This means that every data type is supported by the full opcode set.

3.6.2 Object Orientation

As a fully object-oriented language, Java requires some support for object orientation [152, 153, 188]. Objects mainly consist of three components: data, methods, and some symbol tables (to help accessing the first two components). Hardware could support object orientation in field initialization, dynamic method invocation, dynamic linking, and object addressing (Figure 3.8) as found in [131, 152, 153, 188, 189].

3.6.3 Exception Handling

Java is well known for its runtime support of error checking. It reacts to errors by throwing exceptions. One example is for array bounds check. Nothing is said about exceptions

handling in the specification of picoJava-II. But, throwing an exception may dramatically affect the overall performance. Most probably, raising an exception results in an I-cache miss. This requires going back to the main memory to transfer a block of instructions into the I-cache, which increases the processing time. In addition, directing the execution engine towards the exception handler flushes the pipeline, thus adding to the overall exception penalty. So, Java processors are in a desperate need for advanced techniques for processing exceptions [190, 191]. Methods used for achieving precise interrupts can be useful here. In addition, handlers for the frequently raised exceptions (like array bound checking) can be cached to shorten the processing latency.

3.6.4 Symbolic Resolution

The Java system requires the virtual machine to resolve symbolic references at runtime. For a traditional processor, the compiler and linker resolve all symbolic references and generate a simple addressing mode that requires only one memory reference to load a value from memory. Ideas to help this process include (Figure 3.8):

- **Caching** This requires having an on-chip global repository for dynamically resolved symbols. This container will store symbols with their resolved values.
- **Pre-resolution** JVM specification requires symbols to be resolved upon request. But, the hardware can speculatively resolve symbols and cache them for future use.
- **Pipelining** A special pipeline stage can be included for the resolution process. Overlapping the resolution with other useful tasks alleviates its latency effect.

3.6.5 Garbage Collection

Modern programming languages lean towards supporting garbage collectors to accelerate the software development cycle and reduce runtime memory leaks [192]. However, garbage collection imposes challenges on system design. For instance, it can ruin the caching mechanism. Its exhaustive search can destroy all the work that the cache and the branch predictor controller have done to keep the most current and important data close to the CPU. Hardware support for garbage collection can overcome these challenges (Figure 3.8). This support ranges between the two extremes: hardware assistance constructors for software implementations and full hardware support [54, 193, 194, 195]. Hardware

assistance constructors used by picoJava-II include write barriers, special purpose instructions, and tagging bits [92].

3.7 Conclusions

Virtual machine implementations of Java include either a processor realization or system software design with clever software support. In this study, we addressed the hardware solution, which appears to combine the best performance with the best memory footprint.

We carried out our study through the design space approach. Design space and its trees allow exploring different design options in a convenient graphical form. From this study it is quite clear that Java processors provide their designers with plenty of choices from which to select. In addition, the nature of Java and its virtual machine bring some new design issues to consideration. The depth and diversity of the studied design trees indicate that there are various opportunities for accelerating Java through hardware support.

In the course of our analysis, we made our best effort to examine every relevant aspect. We started at the top level and recursively visited each of the leaves. Investigated design options included the design methodology, the execution engine organization, parallelism exploitation, and support for HLL features. For each of these items further divisions were made to examine various options.

Our exploration was guided by two major issues: (1) supporting modern processor features while (2) maintaining the generality of the processor. Modern processors lean towards pipelining, multiple instruction issuing, multithreading, caching, etc. Each of these aspects was discussed in detail in the appropriate context.

We believe that the success of a Java processor is correlated with its generality represented in its ability to execute non-Java applications.

While traversing the design trees, some recommendations for Java architectures were made. Table 3.1 summarizes our study results highlighting the recommended technique(s) for each examined aspect. We conclude here that, by applying new hardware principles to the creation of Java chips, the architecture may deliver fast, native execution of Java.

Here, we want to stress the importance of handling JVM stack operations efficiently. As this study makes it clear, the stack is involved in executing almost every JBC. With the stack being a main bottleneck in the architecture, we conclude that providing innovative

solutions for this problem would enhance the overall performance of Java programs.

In the following chapters, we will show our design methodology for a Java processor and justify our selections. Based on the design space exploration examined in this chapter, decisions made on most of the execution engine organization and features will be presented.

The work in this chapter has been published in [196].

The next chapter introduces the JAFARDD processor and discusses its global architectural design principles.

Table 3.1. *Summary of the recommended design features for Java hardware.*

Design aspect			Recommendation
Design methodology	Generality	Support method	Conversion
		Conversion method	Translation
		Converter implementation	Hardwired
		Native programming model	JVM/RISC
	Bytecode processing capacity		Folded
	Bytecode issuing capacity		Multiple
	Complex bytecode processing		Trapped
Execution engine organization	Complexity		Decoupling from RISC and CISC
	Registers		Incorporate a register-based architecture
	Caches		Include instruction, data, operand stack, and other caches
	Stack	Frame organization	Partition the frame
		Realization	Configurable register file
		Suitable stack size	Refer to Chapter 2
		Components	S-cache, addressing unit, S-cache manager, and special purpose registers
		Spill/Refill management	Use picoJava-II dribbler mechanism
	Execution units		Have a type conversion unit Design ALUs suitable for multiple-issuing cores
Parallelism exploitation	Pipelining	Depth	Superpipelined
		Branch prediction	Support scalar and vectored branching
		Instruction buffering	Include an instruction buffer
	Multiple-issuing	VLIW organization	Use a hardware adaptor
		Superscalarity	Use reservation stations
	Multithreading	Synchronization support	Use object header bits and cache monitor entries
		Issue capacity	Design as simultaneous multithreading
		Scheduling policy	Dynamic scheduling is more efficient
Support for high-level language features	General features		Consider having plenty of registers, symmetry, and orthogonality
	Object orientation		Boost field initialization, method invocation, dynamic linking, and object addressing in hardware
	Exception handling		Cache most frequently raised exceptions
	Resolution resolution		Employ caching, pre-resolution, and pipelining
	Garbage collection		Rely on hardware support

Chapter 4

Overview of the JAFARDD Microarchitecture

4.1 Introduction

This chapter describes the organization of JAFARDD (*a Java Architecture based on a Folding Algorithm, with Reservation Stations, Dynamic Translation, and Dual Processing*), a processor designed to enhance JBC execution. JAFARDD is a novel stackless architecture for executing JBCs directly in hardware. Our *Operand Extraction (OPEX) bytecode folding* algorithm, described in detail in Chapter 5, and the well known Tomasulo algorithm [197] work in concert to deliver higher performance. JAFARDD dynamically translates JBCs to RISC-style instructions. This facilitates the use of a typical RISC core to execute simple instructions quickly, and hence, a faster clock can be used. Many of our performance enhancement techniques improve on the key language features in Java.

In this chapter, a global view of JAFARDD's architectural design principles is presented with an overview of the proposed architecture. In Chapter 6, details of each pipeline stage and on-chip unit will be thoroughly discussed. We will reflect on the global design principles and examine how each pipeline module is designed to address these concerns.

For the clarity of presentation, the following assumptions and simplifications are made, without affecting the generality of our discussion:

1. There is no `nop` instruction processed as it is the case for most Java programs [87].
2. `wide` modifiers are ignored. Benchmark results show very low occurrence of this modifier in real Java code [86, 87].
3. The cache reads only complete JVM instructions per access. That is, instructions of

length greater than one are not split between two successive cache reads.

4. Only **integer and objectref** data types are processed. Extensions to other data types follow a similar approach, but might unnecessarily complicate our discussion. Therefore, we choose to describe the integer portion of the datapath and the object processing only.

This chapter is organized as follows. Section 4.2 discusses the global architectural design principles of JAFARDD. The design features of the proposed microarchitecture are presented in Section 4.3. Section 4.4 outlines the processing phases of JBCs, whereas Section 4.5 summarizes the instruction pipeline stages. A detailed block diagram of the JAFARDD architecture is given in Section 4.6. In Section 4.7, we give a brief idea about the dual processing capability of JAFARDD. Conclusions are drawn in Section 4.8.

4.2 Global Architectural Design Principles

After a close examination of our bytecode folding algorithm (Chapter 5), we realized that dynamic translation of JBCs to native code is an integral part of performance improvement. We have also studied the well established Tomasulo's technique that facilitates OOO execution so that processing speed can further be enhanced.

We have identified several architectural design principles at the global level that are necessary to ensure that JAFARDD can execute Java efficiently.

Architectural Design Principle 1 ***Common case support***
A major guideline for designing JAFARDD is to fully comply with the well known Amdahl law that recommends making the common case fast and the rare case correct [42, 63].

Architectural Design Principle 2 ***Stack dependency resolution***
Stack access is a major bottleneck in JVM implementations. To achieve any ILP, JAFARDD has to be stack-independent. This implies that some of the stack functionality has to be compensated for by other modules.

Architectural Design Principle 3 ***Working data elements storage on-chip***
In software implementations of the JVM, LVs, stack, and other data elements reside in the main memory (or the data cache). To improve performance, frequently accessed data elements should be moved closer to the processor.

Architectural Design Principle 4 ***Exploitation of ILP***
ILP exploitation among JBCs and the translated RISC instructions is required to speed up processing.

Architectural Design Principle 5 ***Dynamic scheduling***
Like most modern processors, JAFARDD should incorporate dynamic scheduling and OOO execution of JBCs. Pipeline modules need to be organized in such a way to facilitate these mechanisms and at the same time handle hazards.

Architectural Design Principle 6 ***Utilization of a Java-independent RISC core***
Instead of a standalone Java-specific core, building a Java processor based on a typical RISC core would allow performance enhancement using well established techniques. Therefore, JAFARDD should try to match the Java requirements with constructs that are general enough to be used in other languages without incorporating Java-specific language features themselves in the instruction set.

Architectural Design Principle 7 ***Making minimal changes to the software systems***
To facilitate practical use, the architecture should be organized in a way that requires no or minimal changes in the software.

Architectural Design Principle 8 ***Hardware support for object-oriented processing***
As Java is a full object-oriented language, efficient hardware support for object processing would be desirable.

Architectural Design Principle 9 ***Folding information generation off the critical path***
Using critical path analysis, Radhakrishnan et al. showed that the fold and decode is the longest pipeline stage in the picoJava-II processor [127, 146, 159]. To reduce delay, they proposed to remove bytecode folding from the main pipeline stream using a fill unit and a decoded bytecode cache. JAFARDD needs to employ these findings by generating folding information off the main pipeline.

Architectural Design Principle 10 ***Maintaining a high bytecode folding rate***
Folding information should be generated at a rate higher than the actual folding rate to keep pipeline modules utilized at all times, thus addressing Architectural Design Principle 9.

4.3 Design Features

Design features that characterize JAFARDD and address the global architectural design principles are:

(1) OPEX bytecode folding to provide a stackless venue for executing stack code; (2) dynamic binary translation of folding groups to permit JBC processing using a RISC core; (3) dynamic Tomasulo's hardware to facilitate runtime scheduling; (4) deep and dynamic pipeline that provides high throughput and efficient ILP exploitation; and (5) load/store units to manipulate object-oriented instructions.

No software interpretation is done in JAFARDD. JBCs are folded by hardware into groups and then dynamically translated to a native RISC format using a hardwired core. This approach: (1) achieves a high JBC processing rate; (2) facilitates the use of a typical RISC core to execute simple instructions quickly; (3) enables ILP exploitation among translated instructions using well established RISC techniques; (4) provides backward compatibility to execute non-Java software and facilitates the migration to Java-enabled hardware; (5) is able to generate highly optimized native code; and (6) narrows the semantic gap between the JVM and the processor, without adding extra semantic contents to the native instruction.

4.4 Processing Phases

From the JBCs perspective, they are processed in the following four phases (Figure 4.1): (1) *Front end* that feeds raw JBCs into the instruction pipeline by controlling the instruction cache; (2) *Folding administration* that snoops on the instruction cache output bus to generate information about folding groups based on read bytecodes. This phase runs in parallel with the main processing stream; (3) *Queuing, folding, and dynamic translation* that folds and translates JBCs on the fly to a decoded RISC instruction format; and (4) *Dynamic scheduling and execution* that hosts the modified Tomasulo algorithm for JBC execution.

4.5 Pipeline Stages

In the JAFARDD instruction pipeline, the four processing phases are realized in the following seven stages (Figure 4.2): (1) *Bytecode fetch* that fetches JBCs from the instruction

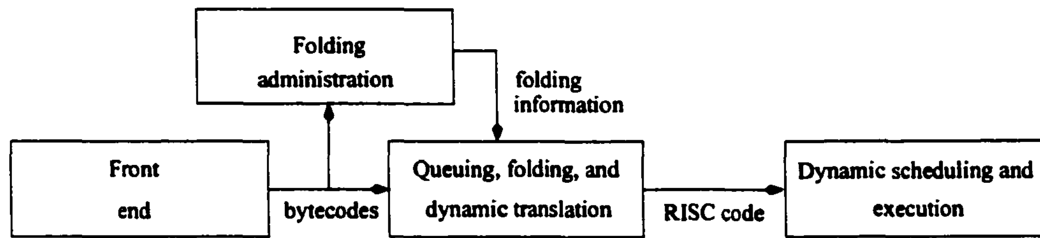


Figure 4.1. *General JBC processing phases.*

cache and sends them to the bytecode queue. Outside the main pipeline flow and parallel to it lies the folding administration off-pipeline stage. This stage captures fetched JBCs, analyzes them, and generates folding information; (2) *Bytecode queue and fold* that is a dual task stage; it first queues the bytecodes fetched from the instruction cache, then at issue time, it dequeues and folds those bytecodes using the folding information generated in the folding administration stage; (3) *Bytecode translation* that dynamically translates folded JBCs to fields of a decoded RISC instruction format; (4) *Local variable access* that accesses LVs as required; (5) *instruction shelve* that hosts reservation stations at the entry of various execution units; (6) *Execute* that is the execution stage; and (7) *Write back* that writes back the result into LVs.

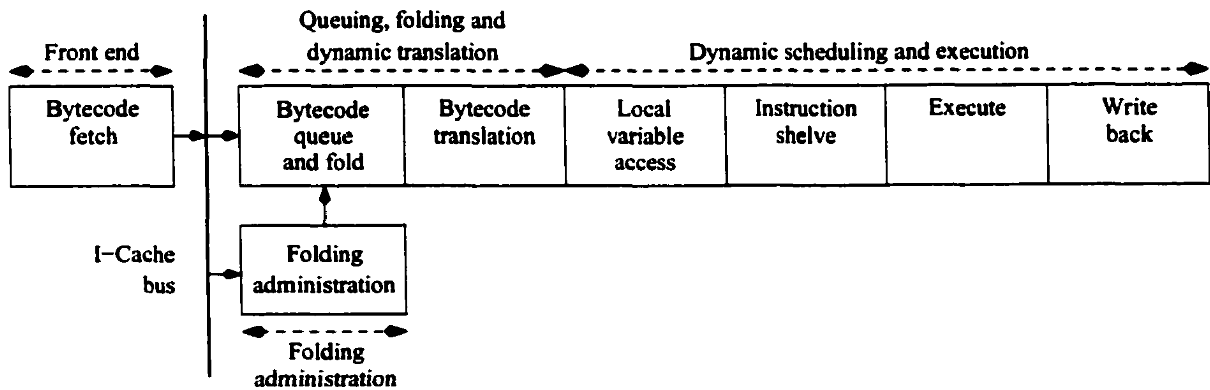


Figure 4.2. *Pipeline stages for JBC processing.*

4.6 Overview of the JAFARDD Architecture

Figure 4.3 shows the layout of JAFARDD architecture. The four processing phases are partitioned as the dashed regions in the figure.

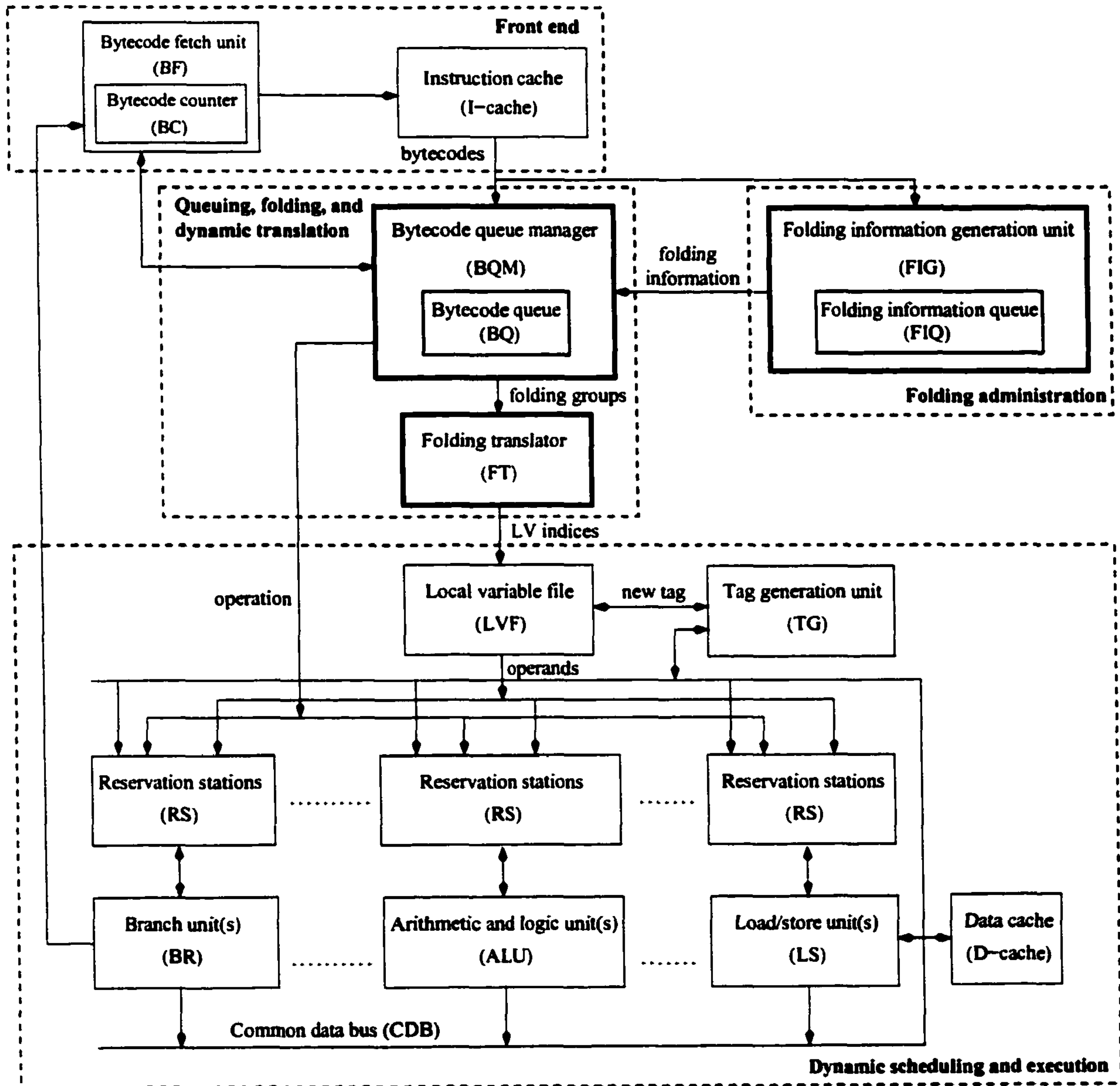


Figure 4.3. *Block diagram of the JAFARDD architecture.*

Detailed signal information are omitted for simplification. Modules needed for folding have thick borders. Processing phases are partitioned in dashed regions.

At runtime, JBCs are brought to the instruction cache (*I-cache*). The bytecode fetch unit (*BF*) moves several JBCs, as indexed by the bytecode counter (*BC*), from the I-cache into the bytecode queue (*BQ*). At the same time, parallel to the main pipeline, the folding information generation unit (*FIG*) snoops on the I-cache bus to generate folding information on read JBCs and saves it in the folding information queue (*FIQ*). At issue time, the bytecode queue manager (*BQM*) folds JBCs based on FIQ information. Each folding group is translated into a single, RISC-style instruction in the folding translator (*FT*).

In many software implementations of the JVM, LVs reside in the main memory or the data cache (*D-cache*). In JAFARDD, a register file holding JVM-visible LVs is used to improve data access. This implies that referencing an LV has the same simplicity and flexibility as handling a register in a RISC processor, with no memory access. In addition, some other LVs used internally, such as by the Tomasulo algorithm, are also included in this local variable file (*LVF*). It is assumed that all LVs are available in the three-port LVF. Method setup, which is responsible for fitting all the LVs into the LVF, will not be discussed here. Interested readers should refer to a JVM specification reference [6, 7, 8].

There are three types of execution units (*EXs*). The load/store units (*LSs*) are capable of accessing the main memory or D-cache to retrieve/store object fields and CP items, as well as initiating memory-related instructions by trapping to emulation routines in the OS. Also included are a set of EXs that perform regular ALU-type operations. ALU classes are single-cycle integer units (*SIs*), multi-cycle integer units (*MIs*), and floating point units (*FPs*). Branch units (*BRs*) specialized in processing branch instructions, possibly coupled with prediction and recovery mechanisms, are the third type of execution units.

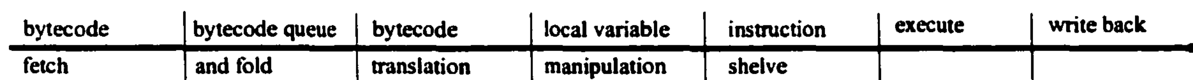
Associated with each EX is a set of reservation stations (*RSs*) to ensure the availability of all required operands before allowing the corresponding instructions to proceed to the EXs. These RSs are used for efficient runtime scheduling according to the Tomasulo algorithm. In the same pipeline stage as the LVF is the tag generation unit (*TG*), which generates destination tags for the RSs, as will be explained in Chapter 6. Results are broadcasted on the common data bus (*CDB*) to be captured by other modules.

Details of structure and operation of each module in Figure 4.3 will be clarified in Chapter 6. Observe that the proposed processor architecture does not include any major control unit. Control is inherent in the pipelined architecture of the processor and is supported by a few, distributed minicontrollers that are constructed as simple combinational circuits.

4.7 Dual Processing Architecture

Using the instruction pipeline convention presented, the JBCs are processed in the stages shown in Figure 4.4(a). Examining a typical RISC pipeline (Figure 4.4(b)), we draw the conclusion that the two pipelines can co-exist in a seamless implementation (Figure 4.4(c)).

(a) JBC processor pipeline



(b) RISC core processor pipeline

(c) Dual execution Java processor pipeline

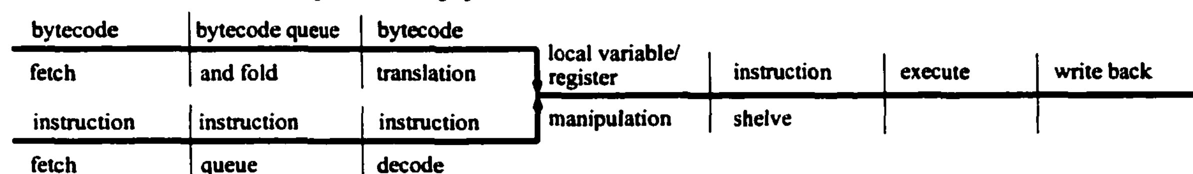


Figure 4.4. *JBC, RISC, and dual execution processor pipelines.*

The first three stages of the dual architecture pipeline (Figure 4.4(c)) call for separate modules for the JBC and the native code streams. Once the folded JBCs are translated into native code, the remaining pipeline stages are unique. Thus, the dynamic JBC translation should be designed to ensure complete decoupling of the two halves of the pipeline.

Our dual architecture pipeline allows the execution of both Java and non-Java code on the same core. Figure 4.5 shows a high level view of this arrangement. Although the processor structure has only one microarchitecture, the thin hardware translation module introduces a duality to its ISA. From a programmer's perspective, there are two logical views: (1) a *JVM ISA* which is a stack-based machine, supporting all Java standard features, e.g., object orientation; (2) a *general RISC machine* capable of exploiting ILP and contains features in addition to those required by the JVM. It is worth mentioning that we are not proposing a complete elimination of the JVM, but just to keep a thin, fast layer that drives the underlying hardware by providing dynamic linking, bytecode verification, memory management, etc.

For the rest of this dissertation, we will focus on the JBC processing path of the pipeline.

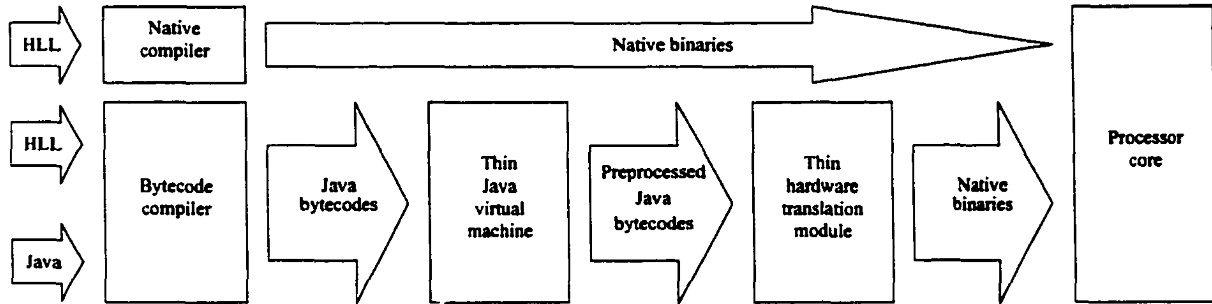


Figure 4.5. *A dual processor architecture capable of running Java and other HLLs.*

4.8 Conclusions

In this chapter, JAFARDD, a novel processor architecture aimed at enhancing Java performance, was introduced. Our processor adopts the OPEX bytecode algorithm for folding JBCs. In addition to its flexibility, this algorithm allows recognition of nested patterns. These two characteristics allow folding as many patterns as possible, which diminishes the need for a stack. Reservation stations are employed for dynamic scheduling. Incorporating RSs together with the BQM and LVF, is our approach for stack dependency resolution.

The work in this chapter has been published in [198, 199, 200, 201].

In Chapter 5, we present and discuss in detail the OPEX bytecode folding algorithm. In Chapter 6, the operational details of JAFARDD pipeline are examined. The performance of our processor is evaluated in Chapter 7.

Chapter 5

An Operand Extraction (OPEX) Bytecode Folding Algorithm

5.1 Introduction

As discussed before, the JVM is based on a stack architecture. This stack architecture was chosen to facilitate compact code generation for reasons such as minimizing bandwidth requirements and download times over the Internet, and to be compatible on different platforms [7, 23, 24, 202]. Direct hardware implementation of this architecture inherits all the weaknesses of the stack paradigm, in particular, the introduction of virtual data dependencies between successive instructions. Almost all instructions need to access the top of the operand stack (or *stack* for short) as a source and/or destination, therefore severely limiting performance and prohibiting any form of ILP, such as superscalarity.

To compensate for these stack implementation limitations, Sun Microsystems first realized a smart stack architecture and a number of enhancements have been recently suggested [92, 95, 97, 203, 204, 205, 206, 207, 208]. A common trait in these proposals is to employ some form of stack operations folding.

Existing folding approaches check each JBC instruction with the following one to see if they could be folded together or not (as will be explained later). If they happen to be foldable, the folded instruction will become a new instruction that will be checked for foldability with the succeeding instructions and so on [92, 95, 97, 203, 204, 205, 206, 207, 208]. Although these suggested folding techniques help improve Java's execution time, they fail to completely overcome the serial nature of the stack operations.

In this chapter, we propose a different stack operations folding algorithm specifically designed to overcome the limitations of existing algorithms. This technique is referred to as

operand extraction (OPEX) bytecode folding algorithm. The motivation of the introduced technique is to allow simultaneous multiple execution of JBCs and nested pattern folding.

The main concept of this technique is to identify principal operations and subsequently pick their operands out of the Java BQ. Once a group of instructions is identified for folding, instruction issuing related information is stored. A processor core could then make use of this information to issue multiple instructions per clock cycle. Results from a trace-driven benchmark evaluation clearly showed performance gains. Furthermore, this algorithm removes the virtual dependencies imposed by the stack paradigm, thus permitting ILP.

This chapter is organized as follows. Bytecode folding is described and existing techniques are briefly reviewed in Section 5.2. The proposed algorithm is presented in detail in Section 5.3. Potential data hazards along with techniques to detect and resolve them are discussed in Section 5.4. The operation of the folding unit is included in Section 5.5. Methods for providing further optimizations are highlighted in Section 5.6. Section 5.7 presents related conclusions.

5.2 Folding Java Stack Bytecodes

In a direct JVM stack implementation, most instructions reference an actual data stack [7, 23, 24]. This leads to the following typical scenario as depicted in Figure 5.1(a): certain instructions (producers) push CNs or data from LVs on top of the stack (steps 1 and 2 in the figure). Then, another instruction (e.g., operator) gets one or more items from the stack top (steps 3 and 4), operates on them, and writes the result back to the stack (step 5). Finally, an instruction (consumer) moves the data from the stack top to a LV (step 6). Executing JBCs consumes extra clock cycles to access the stack. Moreover, stack reference creates a bottleneck that adversely affects performance and introduces a form of virtual dependency between successive instructions. Furthermore, processors that do not include an on-chip dedicated S-cache have to access the D-cache (or directly the main memory) for almost every instruction. This consumes more clock cycles especially if cache misses are encountered. In contrast to that, RISC-style processors do not have to pay that price to read from or write to registers as they are always located on-chip.

Example 5.1 *The simple arithmetic expressions* $LV_1 = LV_2 - LV_3$, $LV_4 = LV_5 + LV_6$, *where* LV_l *is LV number* l, *translate into the JBC sequence shown in the middle column in*

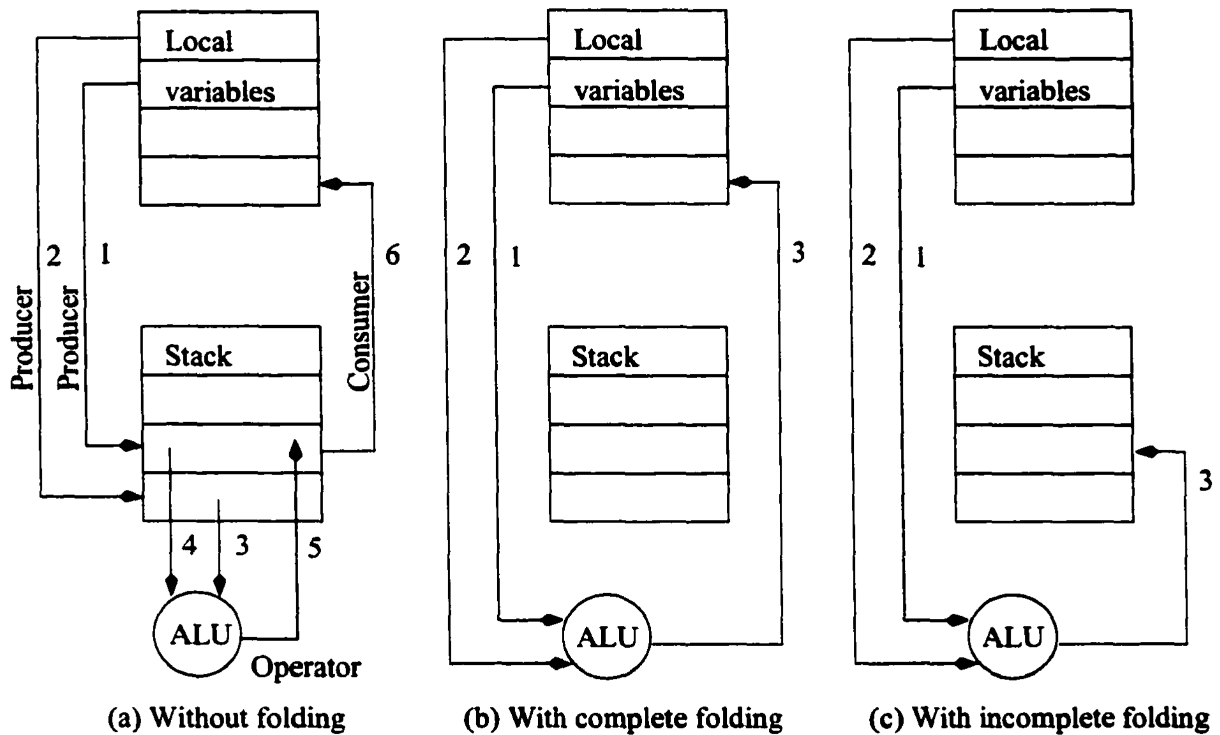


Figure 5.1. *Steps in executing stack instructions.*

Numbers represent sequence of events.

Figure 5.2. Although there is no true data dependency between the above two statements, the second expression cannot be evaluated concurrent with or preceding the first one as both expressions are using the stack as an intermediate target. Operations have to be issued and executed in the sequence shown in the figure, resulting in 8 clock cycles to execute these two expressions. ■

To improve Java performance, Sun Microsystems introduced the notion of bytecode folding [21, 26, 92, 94, 95, 97, 139, 151, 154, 156]. Contiguous instructions that have a true data dependency are grouped together in a compound instruction, called a folding group. A folding group could be executed in one clock cycle using the following scenario depicted in Figure 5.1(b): an operator retrieves operands directly from the source, instead of the stack (steps 1 and 2 in the figure), operates on them, and finally writes the execution result back to the destination (step 3) without referencing the stack. (The difference between complete and incomplete folding will be clarified later.)

Issuing sequence	Without folding	With folding
1	iload_2	iload_2, iload_3, isub, istore_1
2	iload_3	iload 5, iload 6, iadd, istore 4
3	isub	
4	istore_1	
5	iload 5	
6	iload 6	
7	iadd	
8	istore 4	

Figure 5.2. *JBC folding example.*

JBCs (or folding groups) are issued in sequence starting with the topmost line. Assume both unfolded and folded instructions take 1 clock cycle to execute. The middle column shows JBCs issued one at a time (without folding) consuming a total of 8 clock cycles. The last column shows JBCs issued in groups (folded) consuming a total of 2 cycles.

Example 5.2 *The third column in Figure 5.2 shows how folding could reduce the number of clock cycles needed to execute the two expressions in Example 5.1. With this optimization, the two expressions take only two clock cycles provided that sufficient resources (i.e., load/store units; data paths; etc.) are available.* ■

Existing folding mechanisms can be classified into two categories: one based on pattern matching and the other based on recursive grouping.

Pattern matching folding units are designed to recognize certain patterns (sequences of instructions) [9, 21, 22, 26, 97, 153, 159]. They try to match a certain number of instructions at the front of the BQ with the deepest predefined pattern that they are designed to detect. (The depth, d, of a folding pattern is defined as the number of folded instructions in the pattern.) Sequences like (producer, producer, operator, consumer), (producer, operator, consumer), (producer, producer, operator), and (operator, consumer) are commonly used patterns. Sun Microsystems' folding algorithm is based on the pattern matching technique [32, 92, 94, 95, 139, 151, 154, 156].

A recursive grouping folding algorithm starts at the BQ front and recursively checks for foldability every two consecutive instructions [203, 204, 205, 206, 207, 208]. If the

two checked instructions are foldable, they will be marked, in the BQ, as a single combined instruction. (Two consecutive instructions are said to be foldable if they can be grouped together and issued as one compound instruction.) This compound instruction is then checked with the following unfolded one and so on recursively, until no folding is possible. For implementation feasibility, folding cannot continue indefinitely; such a technique therefore specifies an upper limit for foldability (n): the maximum number of bytecodes that could be folded at once. (A folding algorithm is described as n-foldable if it is capable of folding up to n stack instructions. Thus, if d is the depth of a pattern generated by a certain folding algorithm, $(\forall d)(d \leq n)$.) Reaching this limit necessitates terminating the foldability check. Chang *et al.* have proposed a 4-foldable strategy that is based on this technique [203, 204, 205, 206]. Kim *et al.* have proposed modifications to Chang's technique [207, 208].

Both the pattern matching and recursive grouping approaches result in folding groups of length greater than or equal to one bytecode. (A single Java instruction is issued if it can not be folded with the following instruction). After executing these folded instructions, the next instruction is then tested for foldability with the one that follows it and so on.

Folding the JBCs using any of these approaches enhances the performance over the direct execution approach by bypassing the stack and routing the operands directly to/from ALU from/to referenced LVs. This saves the clock cycles that would have been taken by the producer and consumer stack instructions that are folded. (Note that these existing approaches do not fold all stack instructions.) However, the above two folding approaches suffer from the following shortcomings that result from the use of simple linear search and/or pattern matching procedures: **(1)** they do not recognize nested patterns; **(2)** although the per issue capacity is enhanced by folding, folding groups have to be issued in order and one group at a time, as the above folding approaches fail to completely resolve the virtual dependency between instructions; and **(3)** these approaches are not robust enough. Small changes in a foldable pattern, though not affecting the semantics of the instructions, cannot be handled. For example, single or multiple `nop's` or a `wide` modifier inserted in a pattern either disrupts the folding unit or requires a complex architecture to handle it.

Example 5.3 *The two arithmetic statements in Example 5.1 above are rearranged and nested in the middle column in Figure 5.3 with a* `nop` *added. Without folding, executing the JBC sequence takes* 9 *clock cycles.*

Issuing sequence	Without folding	With folding
1	iload_2	iload_2
2	iload_3	iload_3
3	iload 5	iload 5, iload 6, iadd, istore 4
4	iload 6	isub
5	iadd	nop
6	istore 4	istore_1
7	isub	
8	nop	
9	istore_1	

Figure 5.3. *Nested pattern folding example.*

JBCs/folding groups are issued in sequence starting with the topmost line. Assume both unfolded and folded instructions take 1 clock cycle to execute. The middle column shows JBCs, which contain nested folding groups, issued one at a time (without folding) consuming a total of 9 clock cycles. The last column shows these JBCs issued in folding groups consuming a total of 6 cycles.

As the two leading load instructions cannot be grouped with the inner pattern, pattern matching techniques fail to find a pattern that contains the four consecutive `load's`*. Although recursive grouping mechanisms could form a folding group that starts at* `iload_2` *and ends at* `istore 4`*, it will issue the two preceding* `load's` *separately as they do not have any matching operation or consumer.*

In both cases and as is shown in the third column of Figure 5.3, the result is the same: the two leading `load's` *have to be issued separately thereby destroying the outer pattern and taking extra unnecessary clock cycles.*

The `isub` *operation in the third column in Figure 5.3 cannot be easily issued before the preceding* `iadd` *operation, although the two operations are completely independent. Out of order issuing of operations that involve manipulating the stack demands accessibility to all stack elements, which complicates the stack design and limits the overall size of a processor's possible on-chip S-cache. In most cases, operations have to be issued one at a time and in order.*

The `nop` *operation in the second column of Figure 5.3 complicates folding the preced-*

ing `isub` operation with the following `istore_1`. Either a more complex folding unit should be used or more folding patterns with `nop's` located at all possible locations in a pattern, should be recognized by the algorithm. ■

5.3 The OPEX Bytecode Folding Algorithm

In this section we introduce a new and powerful folding algorithm that is free of the shortcomings described in the previous section: it handles nested patterns, allows multiple-issuing and thus OOO execution of foldable patterns per clock cycle, and is more robust compared to the two previously discussed approaches. Our main motivation is to allow multiple-issuing of JVM instructions. We will start with a categorization of JVM instructions followed by some necessary definitions. Then, we will explain the details of the algorithm.

5.3.1 Classification of JVM Instructions

JVM is characterized by a rich set of instructions that manipulate the stack [6, 7, 8]. We classify these instructions into 12 types according to the way they handle the stack[1]. Table 5.1 shows all those instructions that are foldable by our proposed algorithm. In contrast, Table 5.2 shows all those instructions that are considered unfoldable.

- **No-effect** (N) an instruction that does nothing. This group includes the `nop` no-operation, and the `wide` modifier which is just a directive that instructs the processing engine to expect longer indices.
- **Producer** (P) a simple instruction that loads the stack with a CN or a LV value. Operations with arrays, objects or CP sources are not classified under this category.
- **Consumer** (C) a simple instruction that removes a datum from the operand stack and stores it into an LV. Operations with array destinations are not classified under

[1] For the purpose of discussion in this chapter, we divide the instruction classes in Table 2.5 in Chapter 2 into finer groups. Correspondence between the classes in Table 2.5 and this chapter's categories are as follows: the scalar data transfer class is divided into no-effect, producer, and consumer categories; the ALU class corresponds to the operator and independent categories; the stack class is divided into destroyer, duplicator, and swapper categories; the object manipulation class is divided into the load and store categories; the control flow class corresponds to the branch category; and the complex group is the same in both chapters.

Table 5.1. *Foldable categories of JVM instructions.*

Type	Symbol	Description	Examples
no-effect	N	has no effect on stack	nop
producer	P_{CN_x}	loads stack with CN x	iconst_0,bipush
	P_{LV_x}	loads stack from LV x	iload_1
consumer	C_{LV_x}	stores stack top into LV x	istore_0
operator	O_A	an arithmetic operation	iadd,lsub
	O_L	a logic operation	iand,lor
	O_N	a type conversion operation	i2f,f2l,d2l
	O_M	a comparison operation	lcmp,fcmpg
independent	I_{LV_x,CN_y}	increments LV x with CN y	iinc
destroyer	$D([1,2])$	pops 1 or 2 words from stack top	pop
duplicator	$U([1,2])$	duplicates 1 or 2 stack entries	dup2
swapper	W	swaps 2 stack entries	swap
load	L_N	allocates a new object	new,newarray
	L_O	gets an object/class field	getfield
	L_C	checks object type	instanceof,checkcast
	L_P	loads from the CP	ldc,ldc_w
	L_A	loads an array element	iaload
store	S_A	stores an array element	iastore
	S_O	stores an object/class field	putfield
branch	B_Z	a compare-with-0 branch	ifeq
	B_C	a 2-number-comparison branch	if_icmpeq
	B_U	unconditional branch	goto,goto_w
	B_J	subroutine jump	jsr,jsr_w
	B_R	subroutine return	ret

this category.

- **Operator** (O) ALU instructions that operate on stack data: remove data from top of the stack, perform an ALU operation on them, and store the result back on the stack. Type conversion and comparison operations are also included in this group.
- **Independent** (I) the `iinc`, which increments a LV without affecting the stack.
- **Destroyer** (D) instructions that remove some stack entries.
- **Duplicator** (U) instructions that duplicate stack entries by inserting copies at specific locations.

- **Swapper** (*W*) instructions that alter the order of data on the stack by swapping their locations, without changing the data.
- **Load** (*L*) instructions that load the operand stack from the main memory (or the D-cache). This includes data retrieval from objects, classes, arrays, or the CP. These load instructions pop object references, array indices, and/or CP indices from the operand stack and push back the referenced element. This category also includes memory allocation for new objects and type checking for existing ones. These load instructions pop CP indices from the stack and push back object references. In JA-FARDD, instructions in this category trap to OS routines that perform the required instruction in software.
- **Store** (*S*) instructions that remove data from the operand stack and store them in the main memory (or the D-cache). Operations that store data into objects, classes, or arrays are classified under this category. Store instructions pop object references and/or array indices together with the value to be stored from the operand stack and push nothing back. Instructions in this category trap to OS routines that emulate the required instruction in software.
- **Branch** (*B*) instructions that branch to a specific target address either conditionally or unconditionally. Conditional branches make jumps based on comparing the top element of the stack with zero or the next element down the stack. Unconditional branches include instructions that jump to a target address without any condition (e.g., `goto`) and subroutine jump and return (e.g., `jsr` and `ret`). Branches that are stack-independent (e.g., `goto` and `ret`) could be classified under the independent category, but we prefer to keep them here.
- **Complex** (*X*) a complex instruction that is considered unfoldable by our algorithm. Such an instruction could be a multidimensional array creation, table jump, method invocation and return, exception throw, or synchronization operation. Table 5.2 summarizes the folding obstacle for each category. Instructions in this group may or may not write back the result on the operand stack. It is more efficient from a hardware realization point of view to emulate these in software by trapping them to an OS kernel. Reserved opcodes are also considered unfoldable for obvious reasons.

Table 5.2. *Unfoldable JVM instructions.*

Category	symbol	opcodes	reasons
Multidimensional array creation	M_A	multianewarray	variable operand count (in the range 1 – 255)
Table jump	M_T	lookupswitch,tableswitch	variable argument count (unlimited)
Method invocation	M_V	invokeinterface, invokespecial,invokestatic, invokevirtual	variable operand count (in the range 0 – 255)
Method return	M_R	ireturn,lreturn,freturn dreturn,areturn,return	discards the current stack
Exception throw	M_E	athrow	discards the current stack
Synchronization	M_M	monitorenter,monitorexit	requires OS support
Reserved opcodes	M_S	breakpoint,impdep1, impdep2	not implemented

5.3.2 Anchor Instructions

In the OPEX bytecode folding algorithm, we introduce the notion of *anchor instructions*. An *anchor instruction* is the principal instruction in a folding group that can alter the value or the order of the stack contents. Anchor instructions do this alteration by processing rather than by data transfer to/from LVs. An anchor could be an operator, a swapper, a duplicator, a destroyer, a load, a store, or a branch instruction. The anchor instruction together with the necessary producer(s) and consumer form a folding group. Producer instructions cannot be anchors. Consumers could be anchors in folding groups that contain only producers without any other type of anchor instructions. Each folding group needs one and only one anchor instruction. Complex bytecodes are not folded and require emulation. Table 5.3 shows, among other information, the equivalent notation for each anchor type and the estimated execution cycles. (Refer to Appendix A for folding notation.) For the clock cycles per instruction (*CPI*), we use numbers similar to those mentioned in [204] for comparison purposes.

By observing supported anchors in Table 5.3, we can see that they demand up to three producers. A zero producer-count anchor is included, which is the case of loading from the

Table 5.3. *Information related to each foldable JVM instruction category.*

Type	Percentage in benchmarks†	Estimated CPI	Effect on stack size‡	Anchor notation
no-effect	0.53	1	↔	n/a
producer	36.67	1	↑	n/a
consumer	7.24	1	↓	$A_{1,0}{}^{\star}$
operator	17.12	1	↕	$A_{[1,2],1}$
independent	2.23	1	↔	$A_{0,0}$
destroyer	0.07	1	↓	$A_{[1,2],0}$
duplicator	4.08	1	↑	$A_{[1,2],0}$
swapper	0.00	1	↔	$A_{2,0}$
load	14.55	5	↕	$A_{[0,2],1}$
store	2.01	5	↓	$A_{[1,3],0}$
branch	8.21	1	↓	$A_{[1,2],0}$ or $A^{\circ}_{0,1}$
complex	7.29	10	n/a	n/a

- † These figures were generated using SPECjvm98 benchmark suite as will be explained in Chapter 7.
- ‡ Effect on stack size: ↑, ↓, ↕, ↔ mean push, pop, push and pop, and no effect, respectively.
- ⋆ Consumers could be anchors in folding groups that contain only producers without any other type of anchor instructions.
- ∘ Branch instructions that have a consumer do not need producers and vice versa.

CP when the index is supplied as an argument to the JVM instruction. Furthermore, the three-producer case is included to cover store instructions having an array as a destination. The latter case mandates having a three-read-port LVF on any design that implements this algorithm. Also, some of the complex instructions require a producer count that has no upper limit (e.g., table jump operands). On the other hand, all anchors generate zero or one data item. That is, they all require none or one consumer at most.

5.3.3 Basics of the Algorithm

The OPEX bytecode folding algorithm works according to the following scenario (see Figure 4.3 for referenced architectural modules): snooping on the I-cache bus, the FIG unit searches for the first anchor instruction and marks it as a reference point; then scanning backward in the bytecodes read from the I-cache, the producers needed for the anchor

instruction are identified. (The properties of JBCs guarantee the existence of the necessary producers [1]). If the anchor instruction requires a consumer, the algorithm scans forward in the bytecodes read from the I-cache to see if there exists one immediately following the anchor instruction. If it is found, a window that starts at the first producer and ends at the consumer is marked as a complete folding group (*CFG*). If the consumer expected to follow the anchor immediately is not there, a folding group starting from the first producer up to the anchor only is marked as an incomplete folding group (*IFG*). The starting and ending locations of the folding groups in the BQ, together with other information (e.g., the type of the anchor and its position within the folding group, information about the folding pattern, type of operation to be performed by the EX, etc.) are added to the FIQ for later use. After that, the algorithm repeats itself looking for the next anchor instruction (in the JBCs read from the I-cache) and picking its producer(s) and consumer, if necessary, and so on.

Later in the execution stages, the FT translates all instructions within a folding group into a single RISC-like instruction. The number of clock cycles necessary to execute the folding group is equal to that needed for the anchor instruction alone in the case of issuing the bytecodes without folding. As a result, the cycles that would have been taken by the producers and consumers in the case of unfolded execution, are completely eliminated. Instructions generated from this dynamic translation process could be executed in order, OOO, or concurrently with other instructions. Thus, it is an advantage of our algorithm to permit OOO execution of JBCs.

Section 5.5 explains in detail the operation of the FIG unit and shows, using a state diagram and pseudocode, how the OPEX bytecode folding algorithm is incorporated in it.

Example 5.4 *Figure 5.4 shows the OPEX bytecode folding algorithm at work. Stage (a) identifies the anchor instruction* `iadd`*. Stage (b) extracts the necessary operands, whereas, stage (c) picks the consumer (if necessary). Once a group is found, it is marked for folding as in stage (d). The example shows how to handle two nested patterns, as in stage (e), where the outer pattern (*`iload_0, istore 4`*) forms a folding group. Furthermore, the two folded groups could be issued simultaneously or OOO. Also, observe that the existence of* `nop` *does not disrupt the folding process, which shows the generality and flexibility of the algorithm.* ■

We can observe from Example 5.4 that the producer `iload_0`, folded with the anchor

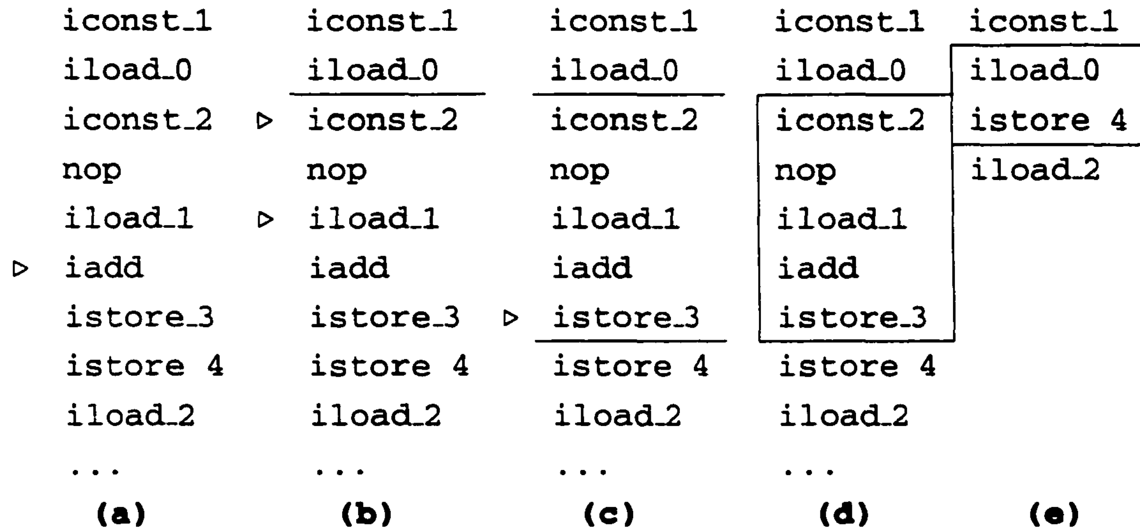


Figure 5.4. *An example showing the OPEX bytecode folding algorithm at work.* Bytecodes are executed starting at the top. (a) Scan for the first anchor instruction. (b) Scan backward and pick needed operands. The top horizontal line marks the beginning of a folding group. (c) Scan forward for the consumer. The bottom horizontal line marks the end of a folding group. (d) Mark a folding group as a compound instruction. (e) Issue the folding group and remove it from the BQ. A new folding group is identified and so on.

`istore 4`, physically exists in the BQ before the other folding group that was recognized earlier. Thus, our algorithm allows folding of nested patterns easily, which is another advantage over existing algorithms. However, folding nested patterns might result in additional data hazards! Section 5.4 clarifies situations that might lead to these hazards and shows how the algorithm could be modified slightly to handle such situations.

5.3.4 Tagging

As explained above, folding groups could either be complete or incomplete. CFGs are self-contained: they include the consumers that are necessary for their anchor instructions and do not need to access the stack at all (Figure 5.1(b)). On the other hand, IFG need consumers that are not present in the BQ. Thus, if the instructions within a window, recognized here as an IFG, were executed individually, one at a time, in a processor that does not employ any kind of folding, the final result would have been left on the stack (Figure 5.1(c)).

IFGs are made complete by using *tagged consumers* (T_{LV_t}) and *tagged producers* (Q_{LV_t}), where T denotes the tagged consumer, Q denotes the tagged producer, and LV_t is the tagged LV index. Tagged consumers are artificial consumers attached by the OPEX bytecode folding algorithm at runtime and at the hardware level to an IFG to make it complete. When this folding group is removed from the BQ, it is replaced in the BQ with a tagged producer. This tagged producer will then be used as a producer in a following folding group. (CFGs are removed from the BQ without any replacement.)

Tagged producers and consumers reference *tagged local variables* that are temporarily allocated, at runtime and at the hardware level, from the LVF to hold intermediate results transferred between folding groups. This requires the processor design to provide additional on-chip LVs that are invisible to the JVM. Tagged LVs play an important role in decoupling the virtual dependency between folding groups. The result produced by an IFG goes to a tagged LV (instead of the stack top), which in turn serves as a data source for succeeding tagged producers. It is worth observing that both the tagged consumer which is attached to an IFG, and the corresponding tagged producer which replaces the same IFG, reference the same tagged LV. A tagged LV stays marked as being in use until it is no longer referenced by any tagged producer in the BQ. From the implementation point of view, we selected two of the unused JVM opcodes and renamed them `tload` and `tstore` to act as tagged producer and consumer, respectively. (The internal implementation of the BQM ensures that tagged instructions take one byte each as will be explained in Chapter 6.)

Example 5.5 *Figure 5.5 illustrates the process of tagging as used in the OPEX bytecode folding algorithm. In this figure the folding group that contains the anchor* `isub` *is a CFG, whereas the group that contains* `iadd` *is an incomplete one. By inserting a tagged consumer (as in part b) and a tagged producer (as in parts c and d) the* `iadd` *instruction group is made complete.* ■

It is worth mentioning that introducing tags demands the use of an advanced instruction scheduling technique like reservation stations, where the operand source must be identified within the execution engine [42, 91, 197]. This will be fully explained in Chapter 6.

5.3.5 Model Details

Careful and extensive examination of different possible bytecode sequences revealed five different templates that could accommodate all possible folding patterns. These patterns

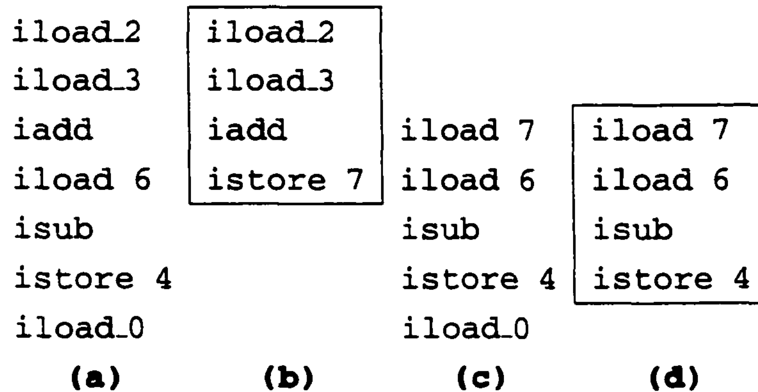


Figure 5.5. *Making incomplete folding groups complete by tagging.*
JBCs are executed starting at the top. (a) The BQ before folding. (b) A folding group with a tagged consumer. (LV_7 is a tagged LV). (c) The BQ after issuing the first folding group. (d) Another folding group with a tagged producer.

are used by the OPEX bytecode folding algorithm in the FIG. Table 5.4 shows how theses five different cases are issued and what the BQ looks like after issuing them:

- **Case 1** A CFG is identified for folding. An anchor finds the needed consumer in the BQ. Both are removed from the BQ with the needed producers. This case covers operator, load, and store anchors.
- **Case 2** IFGs are identified for folding. An anchor can not find the needed consumer in the BQ. The FIG unit realizes that this is an IFG when it encounters a producer or a subsequent anchor. The anchor together with the necessary operands are issued with an augmented tagged consumer. In the BQ, the anchor and its producers are replaced with a tagged producer. This case covers operator, load, and store anchors.
- **Case 3** The FIG unit encounters a consumer before any other anchor. Thus, it considers this consumer as an anchor to form a CFG. The consumer together with its required producer are removed from the BQ.
- **Case 4** An anchor that does not need a consumer always forms a CFG. In this case, the anchor together with its required producers are removed from the BQ. This case covers destroyer, swapper, duplicator, and some branch anchors.
- **Case 5** Independent and some branch anchors form CFG groups by themselves.

Table 5.4. *Different folding pattern templates recognized by the folding information generation unit (FIG).*

No	BQ before folding	Issuing form	BQ after folding	Example
1a	$\underbrace{P\cdots P}_{z}\overbrace{\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}C}^{\text{folding group}}X\cdots$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}C$	$\underbrace{P\cdots P}_{z}X\cdots$	$\text{iload_2}\ \overbrace{\text{iload_1 ineg istore_0}}^{\text{folding group}}\ \text{pop}\cdots$
1b	$\underbrace{P\cdots P}_{z}\quad\overbrace{A_{0,1}C}^{\text{folding group}}\quad X\cdots$	$A_{0,1}C$	$\underbrace{P\cdots P}_{z}X\cdots$	$\text{iload_2}\ \overbrace{\text{ldc 10 istore_0}}^{\text{folding group}}\ \text{ineg}\cdots$
2a	$\underbrace{P\cdots P}_{z}\overbrace{\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}}^{\text{folding group}}P\cdots$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}T$	$\underbrace{P\cdots P}_{z}QP\cdots$	$\text{iload_2}\ \overbrace{\text{iload_1 ineg}}^{\text{folding group}}\ \text{iconst_0}\cdots$
2b	$\underbrace{P\cdots P}_{z}\quad\overbrace{A_{0,1}}^{\text{folding group}}\quad P\cdots$	$A_{0,1}T$	$\underbrace{P\cdots P}_{z}QP\cdots$	$\text{iload_2}\quad\overbrace{\text{ldc 10}}^{\text{folding group}}\quad\text{iconst_0}\cdots$
2c	$\underbrace{P\cdots P}_{z}\overbrace{\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}}^{\text{folding group}}A_{z',[0,1]}\cdots$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}T$	$\underbrace{P\cdots P}_{z}QA_{z',[0,1]}\cdots$	$\text{iload_2}\ \overbrace{\text{iload_1 iconst_2 iadd}}^{\text{folding group}}\ \text{idiv}\cdots$
2d	$\underbrace{P\cdots P}_{z}\quad\overbrace{A_{0,1}}^{\text{folding group}}\quad A_{z',[0,1]}\cdots$	$A_{0,1}T$	$\underbrace{P\cdots P}_{z}QA_{z',[0,1]}\cdots$	$\text{iload_2}\quad\overbrace{\text{ldc 10}}^{\text{folding group}}\quad\text{ineg}\cdots$
3	$\underbrace{P\cdots P}_{z}\quad\overbrace{PC}^{\text{folding group}}\quad X\cdots$	PC	$\underbrace{P\cdots P}_{z}X\cdots$	$\text{iload_2}\ \overbrace{\text{iload_1 istore_3}}^{\text{folding group}}\ \text{iconst_1}\cdots$
4	$\underbrace{P\cdots P}_{z}\overbrace{\underbrace{P\cdots P}_{[1,3]}A_{[1,3],0}}^{\text{folding group}}X\cdots$	$\underbrace{P\cdots P}_{[1,3]}A_{[1,3],0}$	$\underbrace{P\cdots P}_{z}X\cdots$	$\text{iload_2}\ \overbrace{\text{iload_1 pop}}^{\text{folding group}}\ \text{istore_3}\cdots$
5	$\underbrace{P\cdots P}_{z}\quad\overbrace{A_{0,0}}^{\text{folding group}}\quad X\cdots$	$A_{0,0}$	$\underbrace{P\cdots P}_{z}X\cdots$	$\text{iload_2}\quad\overbrace{\text{iinc 2 1}}^{\text{folding group}}\quad\text{ineg}\cdots$

In this algorithm, *complex* instructions are issued individually without being attached to any folding group. Future extensions of the algorithm could consider the possibility of forming folding groups of these complex instructions.

5.4 Algorithm Hazards

In the OPEX bytecode folding algorithm, folding nested patterns results in OOO issuing of JBCs. For example, in Figure 5.4(d), instructions `iconst_2` and `iload_1` are issued before `iconst_1` and `iload_0`. Hence, for correctness, the algorithm must check dependency violations. Such dependencies are often called data hazards. There are three types of data hazards: *Read After Write (RAW)*, *Write After Read (WAR)*, and *Write After Write (WAW)* [42, 69, 209]. (The Read and Write operations refer to LV accesses). In this section, we investigate whether the OPEX bytecode folding algorithm introduces any data hazards, how to detect them, and how they can be resolved. This section is concerned only with

hazards that are caused by the OPEX bytecode folding algorithm. Regular RISC pipeline hazards could still occur between folding groups if they are executed OOO. Any pipelined and/or superscaler core designed to support our folding technique must be able to handle these usual hazards with standard techniques.

One necessary, but not sufficient, condition for RAW and WAW hazards to occur is OOO issuing of write instructions (with respect to subsequent write or read instructions in the original program sequence). In the JVM, writing to LVs is done by consumers (C) and reading from them is done by producers (P). Since in our algorithm, consumers are always extracted for folding and issuing in-order with respect to subsequent consumers and producers, RAW or WAW hazards can not be introduced by our algorithm. However, WAR hazards could occur! A consumer could be issued before a precedent producer, and if both instructions are targeting the same LV, this LV will be written by the consumer before being read by the producer. Formulated differently, a WAR hazard occurs if a consumer instruction attempts to write to a LV that is to be read by a pending producer instruction. As this contradicts the semantics of the instruction sequence, the program being executed will have the wrong result.

Example 5.6 *Consider the bytecode sequence shown in Figure 5.6(a) with* `?` *and* `X` *denoting unknown and don't care bytecode instructions, respectively. To demonstrate an OOO issuing, assume that two nested folding groups could be recognized from the shown bytecode sequence. The inner folding group (*`iload_1, istore_0`*) in lines* 2 *and* 3 *will be issued first, whereas the outer group (*`?, X`*) in lines* 1 *and* 4 *will be issued next. This means that the bytecode in line* 1 *will be issued OOO with respect to the bytecodes in lines* 2 *and* 3*. To check dependency violations, consider the following three cases:*

1. *A WAW occurs between the two groups if* `?` *is an* `istore_0` *as it will write to* LV_0 *after the* `istore_0` *in line* 3 *writes to it (Figure 5.6(b)). However,* `?` *can not be a consumer, otherwise it would have been folded with a precedent producer or precedent anchor and its needed producer(s), according to our algorithm. Thus, a WAW can not occur between the two groups.*
2. *A RAW occurs between the two groups if* `?` *is an* `istore_1` *as it will write to* LV_1 *after the* `iload_1` *in line* 2 *reads from it (Figure 5.6(c)). However,* `?` *can not be a consumer as explained above. Thus, a RAW can not occur between the two groups.*

3. *A WAR occurs between the two groups if ? is an* `iload_0` *as it will read from* LV_0 *after* `istore_0` *in line 3 writes to it (Figure 5.6(d)). This could occur according to our algorithm! Therefore, the OPEX bytecode folding algorithm could suffer from a WAR hazard.*

	(a)	(b)	(c)	(d)
1	?	istore_0	istore_1	iload_0
2	iload_1	iload_1	iload_1	iload_1
3	istore_0	istore_0	istore_0	istore_0
4	X	X	X	X

Figure 5.6. *Checking the OPEX bytecode folding algorithm for dependency violations.* JBCs are executed starting at the top. (a) A sample bytecode sequence with two recognized nested folding groups. ? and X denote unknown and don't care bytecode instructions, respectively. (b) ? is assumed to be an `istore_0` to check if a WAW could occur. (c) ? is assumed to be an `istore_1` to check if a RAW could occur. (d) ? is assumed to be an `iload_0` to check if a WAR could occur.

When issuing an IFG, a tagged consumer instruction is attached. The tagged LV allocated to this consumer is a free LV that is not currently being used by other producers or consumers. Therefore, the tagged LV associated with an IFG cannot be the source of a WAR hazard. Hence, hazards can only occur with CFGs. Notice that the independent folding category, i.e., `iinc`, could cause a hazard if it updates the value stored in a LV before a pending producer reads this LV.

The OPEX bytecode folding algorithm first detects hazards of this kind and then resolves them efficiently.

5.4.1 Hazard Detection

Hazard detection could be implemented in the FIG by scanning the read but not yet folded JBCs and checking to see if there exists any producer that has a source which is the same as the destination of the consumer in the current folding group. Formally, the necessary condition for WAR hazard can be stated as follows:

Let the first bytecode in the recognized folding group be located at position $n+1$ ($n+1 \leq BQ\ size$) in the BQ and the folded consumer instruction be $C\ l$. A WAR hazard occurs if $\exists\ P\ l$ at position i ($0 \leq i \leq n$) in the BQ.

Example 5.7 *Figure 5.7(a) shows a bytecode sequence that will suffer from a WAR hazard if the original OPEX bytecode folding algorithm is applied. Figure 5.7(b) applies the above detection algorithm to check the existence of a WAR hazard. The ⊳ symbols mark a detected hazard.* ■

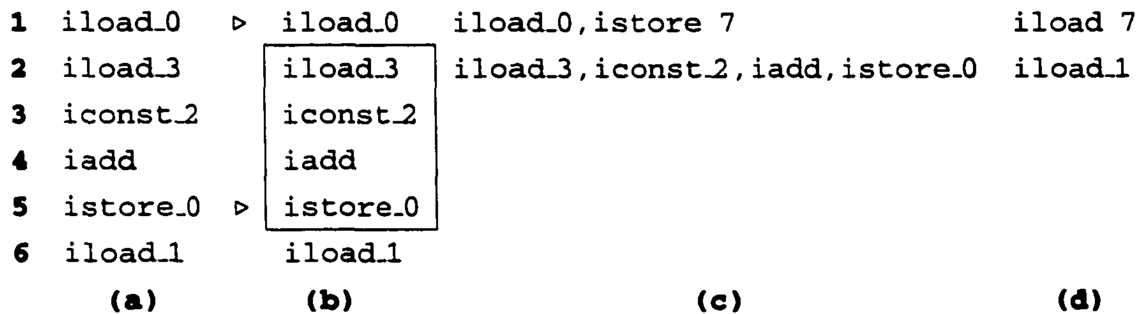


Figure 5.7. *WAR hazard detection and resolution.*

JBCs are executed starting at the top. (a) A sample code that will suffer from a WAR hazard if our folding algorithm is applied. (b) The box contains an identified folding group. The ⊳ symbols mark a detected hazard. (c) Issued folding groups in sequence. (LV_7 is a tagged LV.) (d) The BQ after issuing the folding group.

5.4.2 Resolution by Local Variable Renaming

This potential hazard can be resolved by LV renaming. Three steps are taken in sequence: **(1)** a folding group of the form (P_{LV_l}, $tstore\ LV_t$), where LV_l is the cause of contention and LV_t is a new tagged LV allocated at runtime, is issued; **(2)** every P_{LV_l} existing in the BQ, before the folding group that contains the contending consumer instruction, is replaced with $tload\ LV_t$. This ensures that the source value is moved to a safe place and will provide the correct value when read later; and **(3)** the folding group that contains the contending consumer instruction is issued.

Example 5.8 *Figures 5.7(c) and (d) show how to resolve the hazard detected in Example 5.7 by LV renaming.* ■

5.5 Operation of the FIG Unit

In this Section, we will explain the detailed operation of the FIG unit using a state diagram and an algorithm. As explained in Chapter 4, the FIG unit scans the bus between I-cache and the BQM looking for folding groups. This bus is made wide enough to accommodate several bytecodes, thus providing the capacity for folding more than one group. When a folding group is recognized, information about its location in the BQ is added to the FIQ for later use by the BQM in bytecode folding and issuing. Note that the FIG unit itself does not actually fold the bytecodes; it only generates information for the BQM to locate folding groups within the BQ. To be able to generate information about new folding groups without actually folding previously recognized groups or having access to the BQ itself, the FIG unit simply keeps internal state information that simulates bytecode folding and issuing.

It is worth mentioning that the OPEX bytecode folding algorithm does not check for bytecode errors. It is assumed that the JBC compiler produces well structured bytecodes, e.g., the guaranteed existence of the necessary operands [1]. Bytecode-related error checking is done at the software level by a bytecode verifier.

5.5.1 State Diagram

The operation of the FIG unit is summarized in the state diagram shown in Figure 5.8. States represent bytecodes read by the algorithm but not folded yet. Transitions between states are made in response to reading a new opcode (input). At that moment, if a new folding group is recognized, its template is associated with the transition (output). In the diagram, transition labels have the format: *read opcode folding category/recognized folding group template (if any)*. The five recognized folding templates correspond to the five cases detailed in Subsection 5.3.5 and summarized in Table 5.4.

5.5.2 Algorithm

The pseudocode for the operation of the FIG unit is given in Algorithm 5.1. The algorithm core is represented by the procedure *main*. It is invoked each time a new group of JBCs is read from the I-cache (*queueEnable* is asserted) or a folding group is issued (*dequeueEnable* is asserted). The algorithm requires and manipulates the following data structures:

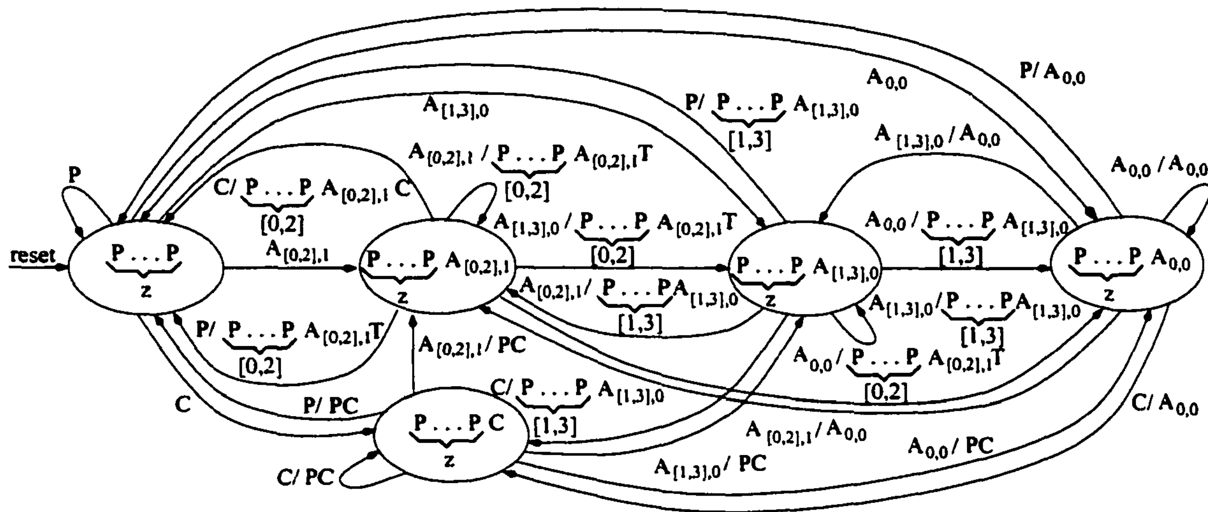


Figure 5.8. *State diagram for the operation of the FIG unit.*

States represent folding categories of scanned, but not folded, opcodes. Transition labels have the format: *read opcode folding category/recognized folding group template (if any)*. Symbols are used in a generic way to represent opcode folding categories, e.g., a transition labeled $A_{0,0}/A_{0,0}$ indicates that an input opcode of folding category $A_{0,0}$ results in recognizing a folding template that includes an opcode of the same category.

- **opcodeDB** an array of information about each JVM opcode including its folding category, folding subcategory (whether the opcode references a LV, a CN, etc.), anchor notation, corresponding instruction length, operand count, and the index of referenced LV ("-1" indicates that no actual LV is referenced).
- **FIQ** a queue of information about folding groups including their starting bytecode index in the BQ and bytecode count. (In an actual implementation of the algorithm, other information could also be stored in this queue, e.g., the type of the anchor and its position within the folding group, the folding pattern, the type of operation to be performed by the execution unit, etc.) The FIQ has two operations associated with it, *removeEntryAtHead()* and *addEntry(startBytecode, bytecodeCount)*.
- **auxBQ** an auxiliary bytecode queue that maintains opcode information, e.g., its folding category, instruction length, required number of operands, and whether a consumer is needed or not, about read but not yet folded bytecodes. This information is obtained initially from the opcodeDB and modified later by the algorithm (if needed).

Once information about a folding group is added to the FIQ, entries pertaining to it are removed from this queue. The two auxBQ pointers *auxBQstartPointer* and *auxBQcurrentPointer* keep track of the start of the current folding group and the last read opcode in the queue, respectively. As shown in Figure 5.9, these two pointers are mapped to the BQ using another two pointers *BQstartPointer* and *BQcurrentPointer*. Allowable operations on the auxBQ are *addEntry (opcodeDB[opcode]), deleteEntries (startingEntry, noOfEntries), duplicateEntries (startingEntry, noOfEntries, duplicateOpcode)*, and *swapEntries (startingEntry, noOfEntries)*. All these operations update the *auxBQcurrentPointer*.

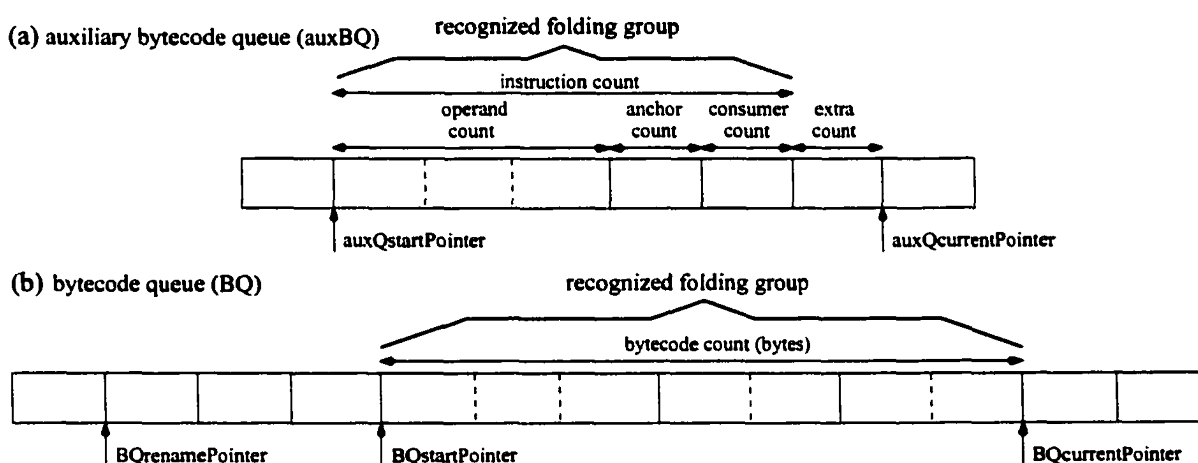


Figure 5.9. *Parameters and pointers relevant to the bytecode queue and the auxiliary bytecode queue as maintained by the FIG unit.*

With the assertion of the *queueEnable*, the bytecodes read from the I-cache and their count are passed to the *main* procedure in the parameters *cacheBus* and *fetchLength*, respectively. Scanning the *cacheBus* using the *inputPointer*, the *main* performs the following steps in sequence for each read opcode:

1. Insert opcode information including explicit LV reference in the auxBQ.
2. Call *stateTransition* procedure that realizes the state machine in Figure 5.8. (We are not including the code of this procedure as it is a literal realization of the diagram.) If a folding group is recognized, *stateTransition* returns *TRUE* and the algorithm continues to the following step. Otherwise, the algorithm jumps to step 8.

3. Set some parameters according to the template of the folding group as shown in Table 5.5. The starting bytecode and the size of the folding group can be determined from these parameters. The meanings of these parameters are shown, together with other relevant queue pointers, in Figure 5.9.
4. Set the folding group's instruction count and its starting location in the auxBQ.
5. If *stateTransition* indicates that a hazard is possible (by setting the *specialInstruction* parameter to *hazard*), call *WARprocess* to check the read but not yet folded bytecodes and to look for possible contention between source LVs and the current group destination LV. If such LVs are found, a rename entry is added to the FIQ with a pointer (*BQrenamePointer* in Figure 5.9) to the contending producer and a zero folding group size indicating a non-folding, but renaming operation. The renamed LV index is set to "−1" in the auxBQ to avoid future renames.
6. Call *foldingInformationGeneration* to determine the folding group potential starting bytecode and size in the BQ using the lengths of the JVM instructions in the folding group. This information is then added to the FIQ.
7. Call *adjustQueuePointers* to simulate the effect of issuing the folding group and removing it from the BQ by removing its corresponding entries in the auxBQ and updating the *BQcurrentPointer*. There are three exceptions from that: **(1)** if the folding group contains a swapper anchor instruction, only the opcode is deleted and the auxBQ is requested to swap the two operands; **(2)** if the folding group contains a duplicator anchor instruction, only the opcode is deleted and the auxBQ is requested to duplicate the operand according to the duplication opcode; and **(3)** if the folding group is incomplete, a tagged producer (represented by the artificial opcode *taggedProducer*) replaces it in the auxBQ. (Tagged consumers are attached at the BQM to IFGs when they are issued.)
8. Advance the *inputPointer*.

With the assertion of the *dequeueEnable* when a folding group is issued, the head of the FIQ is removed. This operation can take place concurrently with analyzing read JBCs.

Algorithm 5.1 *Pseudocode for the operation of the FIG unit.*

/ Global variables (maintained between procedure calls) */*

opcodeDB: opcodeInformationArray = initializeOpcodeDB();

Table 5.5. *Information generated by the state machine about recognized folding templates.*

Folding parameter	Description	Folding templates				
		$\underbrace{P \cdots P}_{[0,2]} A_{[0,2],1} C$	$\underbrace{P \cdots P}_{[0,2]} A_{[0,2],1} T$	$\underbrace{P \cdots P}_{[1,3]} A_{[1,3],0}$	$A_{0,0}$	PC
operandCount	number of operands needed	[0, 2]	[0, 2]	[1, 3]	0	1
anchorCount	if an anchor is included	1	1	1	1	0
consumerCount	if a consumer is included	1	0	0	0	1
extraCount	if opcodes are scanned, but are not part of the folding group	0	1	1	1	1
specialInstruction	special processing needed by the folding group	hazard	tag	duplicate, swap, or none	hazard	hazard

```
FIQ: foldingInformationArray;
auxBQ: opcodeInformationArray;
auxBQstartPointer, auxBQcurrentPointer: integer = 0;
BQstartPointer, BQcurrentPointer: integer= 0;
operandCount,anchorCount,consumerCount: integer;
extraCount,instructionCount,bytecodeCount: integer;
specialInstruction: string;
procedure main(queueEnable: boolean, dequeueEnable: boolean,
                        cacheBus: byte[],fetchLength: integer)
inputPointer: integer = 0;
opcode,instructionLength: integer;
begin
    if (queueEnable == TRUE) then
    begin
        while (inputPointer < fetchLength) do
        begin
            opcode = cacheBus[inputPointer];
            auxBQ.addEntry(opcodeDB[opcode]);
            instructionLength = opcodeDB [opcode].length;
            BQcurrentPointer = BQcurrentPointer + instructionLength;
            if ((instructionLength > 1) and (opcodeDB[opcode].subcategory = "lv")) then
            begin
                auxBQ[auxBQcurrentPointer - 1].LVindex = cacheBus[inputPointer+1];
            end if;
```

```
            if (stateTransition() == TRUE) then
            begin
                instructionCount = operandCount + anchorCount + consumerCount;
                auxBQstartPointer = auxBQcurrentPointer - extraCount - instructionCount;
                if (specialInstruction == "hazard") then
                begin
                    WARprocess();
                end if;
                foldingInformationGeneration();
                adjustQueuePointers();
            end if;
            inputPointer = inputPointer + instructionLength;
        end while;
    end if;
    if (dequeueEnable == TRUE) then
    begin
        FIQ.removeEntryAtHead();
    end if;
end main;

procedure WARprocess()
BQrenamePointer: integer = 0;
begin
    for i = 0 to auxBQStart - 1 do
    begin
        if (auxBQ[i].LVindex == auxBQ[auxBQCurrentPointer - extraCount].LVindex) then
        begin
            FIQ.addEntry(BQrenamePointer,0);
            auxBQ[i].LVindex := -1;
        end if;
        BQrenamePointer = BQrenamePointer + auxBQ[i].length;
    end for;
```

```
end WARprocess;

procedure foldingInformationGeneration()
begin
    bytecodeCount = 0;
    for i = 0 to instructionCount - 1 do
    begin
        bytecodeCount = bytecodeCount + auxBQ[auxBQstartPointer + i].length;
    end for;
    BQstartPointer = BQcurrentPointer - bytecodeCount ;
    if (extraCount == 1) then
    begin
        BQstart = BQstart - auxBQ[auxBQCurrentPointer - 1].length;
    end if;
    FIQ.addEntry(BQstart,bytecodeCount);
end foldingInformationGeneration;

procedure adjustQueuePointers()
begin
    if (specialInstruction == "swap") then
    begin
        auxBQ.swapEntries(auxBQstartPointer,1);
        auxBQ.deleteEntries(auxBQcurrentPointer - extraCount,1);
        BQcurrentPointer = BQcurrentPointer - 1;
    end If;
    else if (specialInstruction == "duplicate") then
    begin
        auxBQ.duplicateEntries(auxBQstartPointer,instructionCount-1,
                               auxBQ[auxBQCurrentPointer - extraCount].opcode);
        auxBQ.deleteEntries(auxBQcurrentPointer - extraCount,1);
        BQcurrentPointer = BQcurrentPointer + bytecodeCount - 2;
    end if;
```

```
    else if (specialInstruction == "tag") then
    begin
        auxBQ.addEntry(opcodeDB(taggedProducer));
        auxBQ.deleteEntries(auxBQstartPointer,instructionCount);
        BQcurrentPointer = BQcurrentPointer - bytecodeCount+2;
    end if;
    else
    begin
        auxBQ.deleteEntries(auxBQstartPointer,instructionCount);
        BQcurrentPointer = BQcurrentPointer - bytecodeCount;
    end else;
end adjustQueuePointers;
```

It is worth observing that as the algorithm is modeled as a finite state machine, its time complexity is linear. Additionally, verifying the coverage of the algorithm was performed by inspecting all out-going arcs in Figure 5.8.

5.6 Special Pattern Optimizations

The OPEX bytecode folding algorithm is further enhanced by combining some destroyer, swapper, or duplicator anchor instructions and issuing them simultaneously. Combining anchor instructions may result in eliminating the effect of one or both of them or producing a new anchor instruction. This process is best performed at the FIG unit and the corresponding information should be included in the FIQ. Table 5.6 summarizes the different optimizations supported and shows their combined effect. (Special pattern optimizations are not included in the state diagram and the algorithm in Section 5.5 for simplicity.)

5.7 Conclusions

We have presented and evaluated a new approach to Java bytecode folding using operand extraction. The algorithm is mainly motivated by our desire to issue folding groups concurrently and OOO to facilitate a superscalar support for Java programs. In the OPEX

Table 5.6. *Folding optimization by combining successive destroyer, duplicator, and/or swaper anchors.*

case	sequence	optimization	issued as
1	swap swap	ignore both effects	nop†
2	dup swap	ignore swap effect	dup
3	dup pop	ignore both effects	nop
4	dup pop2	pop only one word	pop
5	dup2 pop2	ignore both effects	nop
6	$D(t)$ $D(u)$ e.g., pop2 pop	combine their effects into one destroyer	$D(t+u)$ e.g., pop3‡

- † nop is ignored at runtime.
- ‡ destroyer(t+u), where $t + u \geq 3$ is not among the standard JVM instructions. It is assumed that the hardware will support such a combined instruction at runtime.

bytecode folding algorithm, principal operations in folding groups are first identified and their operands are picked from the BQ.

Our algorithm overcomes the stated shortcomings of existing folding techniques. It recognizes nested patterns as it first looks for anchors and then extracts their operands. To decouple virtual dependency between successive folding groups, IFGs are tagged. Eliminating virtual instruction dependency provides an opportunity for multiple-issuing and OOO execution of JBCs. Furthermore, the algorithm is robust: small changes in folding patterns can be handled easily. All these features allow folding as many patterns as possible which leads to a great reduction in the required on-chip stack size.

We introduced the notion of anchor instructions and formulated the anchor expression for each JVM instruction category. A state machine model together with the corresponding pseudocode were presented for the introduced folding algorithm. The algorithm was analyzed for possible hazards and a hazard detection and resolution mechanism was included.

We have developed a VHDL model of a processor that supports the OPEX bytecode folding algorithm. We use it as a testbed to experiment with the above techniques.

Further extensions to this algorithm could involve folding some of the complex instructions that are currently not considered.

The work in this chapter has been published in [65, 210, 211, 212, 213].

The next chapter studies JAFARDD's architecture and discusses its distinguishing features emphasizing the instruction pipeline modules.

Chapter 6

Architecture Details

6.1 Introduction

In this chapter, we detail the JAFARDD architecture and the mechanisms utilized to dynamically translate Java stack-dependent bytecodes into RISC-style stack-independent instructions. We especially focus on how virtual stack dependency is resolved by the use of the OPEX bytecode folding algorithm coupled with Tomasulo's algorithm.

The discussion flow in this chapter matches the flow of JBCs through each of the bytecode processing phases discussed in Section 4.4. Section 6.2 describes the front end portion of the pipeline. The folding administration unit is presented in Section 6.3. Whereas, Section 6.4 discusses the architecture of the queuing, folding, and dynamic translation modules and describes special hardware features incorporated to assist in dynamic translation. Dynamic scheduling and execution stages are included in Section 6.5. Conclusions are drawn in Section 6.6.

In discussing the operation of different processor modules, we adopt a unified approach. For every module, we present the following topics in sequence (if applicable): **(1)** *architectural design principles*, where we reflect the Java processing design principles listed in Section 4.2 on the module design; **(2)** *internal operations*, where the internal operations of the module are summarized; **(3)** *bytecode processing*, where we show how the module contributes in processing JBCs or folding groups; **(4)** *architectural block*, where the architecture (inputs, outputs, and internal data structures) of the module are specified; **(5)** *operation steps*, where the algorithm used by the module to perform its operations is explained; and **(6)** *operation control, priorities, and sequencing*, where the rules that govern the operation of the module to guarantee the correctness of the algorithm are discussed.

6.2 Front End

The front end phase contains only one pipeline stage: bytecode fetch. In this phase, the BF acts in conjunction with the BR to speculatively and adaptively fetch JBCs from the I-cache into the BQ.

6.2.1 BF Architectural Design Principles

Maintaining a high bytecode folding rate (as formulated in the Architectural Design Principle 10) implies that the fetch rate should be kept as high as possible to keep the pipeline continuously fed. Thus, the bytecode fetch policy should be sufficiently aggressive to cope with the variable JVM instruction length that results in a variable JBC issue size and the disruptions due to branches and cache misses. For that purpose, we propose an adaptive feedback bytecode fetch policy. (Details of the branch predication and speculative issuing are beyond the scope of this dissertation.)

6.2.2 An Adaptive Feedback Fetch Policy

JAFARDD adopts an adaptive feedback fetch policy in which the number of JBCs fetched every clock cycle is set according to the number of JBCs issued in the previous clock cycle (fedback from the BQM) and the current BQ size. The BF internally keeps track of the current BQ size and updates it on every queuing and dequeuing of JBCs.

The BF works in two modes: *transient* and *steady state*. When the BQ is empty or has to be flushed (at program start, method invocation, or a mispredicted branch), the BF enables the maximum allowable fetch capacity from the I-cache. On the other hand, at steady state, it tries to compensate for issued JBCs (up to the maximum fetch capacity).

Internally, the BQ has two zones (Figure 6.1): one for normal use and the other for overflow. Typical number of entries in the normal and overflow zones are 32 and 16, respectively. Normally, the BQ is filled to the end of the normal zone. However, some of the BQM-internal operations might result in the expansion of the BQ size (as will be explained later), which might require filling beyond the boundary of the BQ normal zone. In this case, JBCs are queued in the overflow zone.

The overflow zone is also used to resolve possible deadlock situations in the BQ. In situations when the normal zone is full and no folding groups could be constructed from

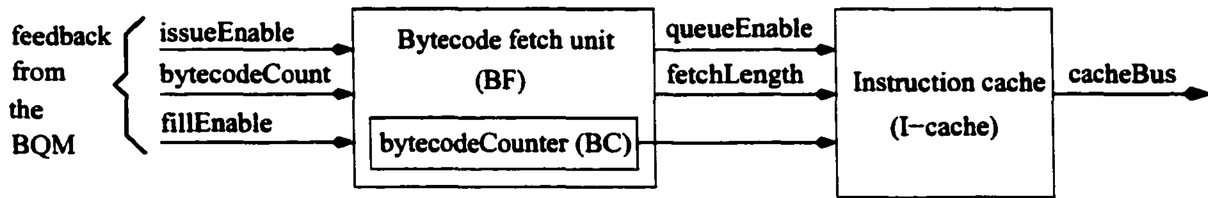


Figure 6.2. *Inputs and outputs of the BF and the I-cache.*

or jumps. The algorithm consists of four parts. The first one adjusts the internally maintained BQ size (*currentBQsize*) with the number of issued JBCs. The next one enables bytecode fetch to fill the normal BQ size. The next part enables fetching one extra bytecode to the BQ overflow zone. The algorithm gives the second part a higher priority over the third one. That is, if the *fillEnable* signal is active at a time when there is space in the nominal BQ, the *fillEenable* request is ignored. The last part of the algorithm handles the default case, where no JBCs are fetched.

Algorithm 6.1 *Pseudocode for the adaptive BF's feedback bytecode fetch algorithm.*

```
external CONSTANT normalBQsize: integer;
external CONSTANT overflowBQsize: integer;
external CONSTANT maxFetchSize: integer;
        VARIABLE currentBQsize: integer := 0;
procedure BF(issueEnable: in boolean, bytecodeCount: in integer,
      fillEnable: in boolean, queueEnable: out boolean,
      fetchLength: out integer, BC: out integer)
begin
   if (issueEnable == TRUE) then
   begin
      currentBQsize := currentBQsize - bytecodeCount;
   end if;
   if (currentBQsize < normalBQsize) then
   begin
      queueEnable := TRUE;
      fetchLength := normalBQsize - currentBQsize;
      fetchLength := min(maxFetchSize, fetchLength);
```

```
    currentBQsize := currentBQsize + fetchLength ;
    BC := BC + fetchLength after 1 clock cycle;
  end if;
  else if ((fillEnable == TRUE) and
        (currentBQsize < (normalBQsize + overflowBQsize))) then
  begin
    queueEnable := TRUE;
    fetchLength := 1;
    currentBQsize := currentBQsize + 1 ;
    BC := BC + 1 after 1 clock cycle;
  end if;
  else
  begin
    queueEnable := FALSE;
    fetchLength := 0;
  end else;
end;
```

6.3 Folding Administration

In Chapter 5 we discussed in detail the OPEX bytecode folding algorithm for Java processors. Here, we shall show the hardware implementation of the algorithm within the FIG in the off-pipeline folding administration stage and its interaction with the rest of the pipeline.

6.3.1 FIG Architectural Design Principles

According to the OPEX bytecode folding algorithm, the FIG scans the bus between I-cache and the BQM searching for folding groups. To maintain a high bytecode folding rate (as required for Architectural Design Principle 10), the I-cache bus must be wide enough to accommodate several JBCs, thus providing the capacity for folding more than one group every clock cycle. By this means, information about folding groups can be generated at a rate that surpasses the rate of their consumption (one every clock cycle) by other units in the pipeline (Architectural Design Principle 9). The size of the FIQ is selected in a way to

match the maximum number of folding groups that could be generated from the BQ. This requirement is compiled into the efficient FIG organization shown in Figure 6.3.

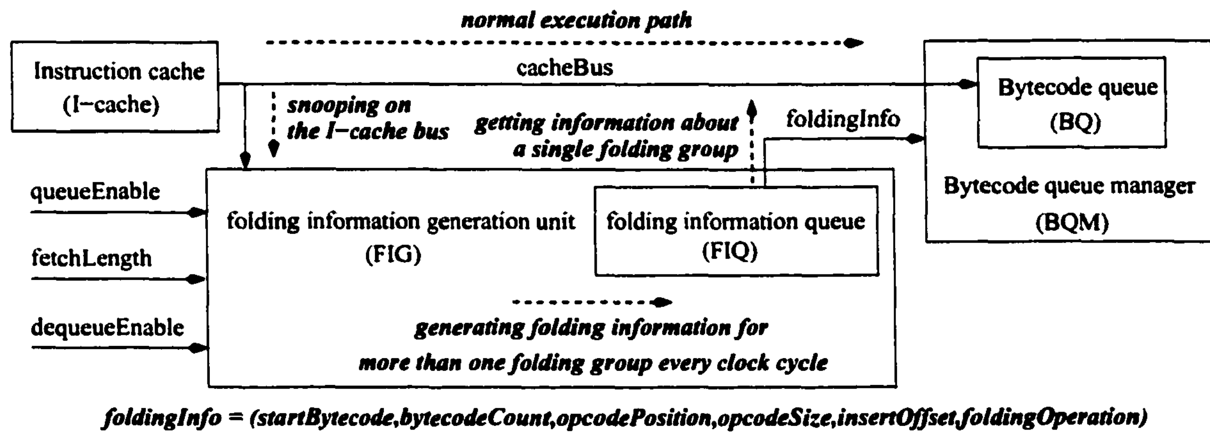


Figure 6.3. *Keeping the FIG out of the main pipeline flow path.*

6.3.2 Bytecode Processing within the FIG

For the bytecode processing done within the FIG, refer to Subsection 5.3.5.

6.3.3 FIG Architectural Module

As shown in Figure 6.3, there are four input signals to the FIG. *queueEnable* indicates that a group of JBCs of length *fetchLength* are being read on the *cacheBus*. When a folding group is to be issued *dequeueEnable* is asserted to enable removing the corresponding entry from the FIQ. The output of this module is the entry at the head of the FIQ (*foldingInfo*).

6.3.4 FIG Operation Steps

For the operation steps of the OPEX bytecode folding algorithm, refer to Section 5.5.

6.4 Queuing, Folding, and Dynamic Translation

Two modules work together to dynamically translate JBCs: the BQM in the bytecode queue and fold pipeline stage and the FT in the bytecode translation stage. The BQM uses infor-

mation generated by the FIG to fold a series of JBCs. The FT in turn translates these folded JBCs into RISC-like native binaries that could be handled by other non-Java modules. In what follows, we shall look at the operation and architecture of both modules.

6.4.1 Bytecode Queue Manager (BQM)

Administration of BQ operations is performed by the BQM. Here, we focus on the details of the BQM architecture and functionality.

6.4.1.1 BQM Architectural Design Principles

Architectural Design Principle 2 requires stack independency thus extending the functionality of the BQM beyond feeding instructions from a regular instruction queue into the pipeline. Hence, two subsequent requirements are needed specific for the BQM: **(1)** the capability of extensively and intelligently rearranging JBCs to compensate for the lack of a stack; and **(2)** the folding of JBCs as specified by the FIG. Therefore, we propose having an on-chip dynamic BQ that emulates typical stack operations, e.g., swapping and duplication (Architectural Design Principle 3). The BQM-internal architecture is structured in a way that facilitates integration with a stackless architecture that is capable of executing stack code. It also plays a significant role in dynamic scheduling and OOO execution provided by the Tomasulo's mechanism.

6.4.1.2 BQM-Internal Operations

A regular queue-like structure usually has two operations: queue and dequeue. A new set of BQM-internal operations is added to emulate some of the required stack operations, e.g., destroy top of stack, duplicate stack top, etc. For each folding group, the FIG specifies the BQM operation to be performed and includes it among the group information in the FIQ. In total, the BQM is capable of performing seven different internal operations. (An **Invalid** operation disables the BQM):

- (**Remove** – V): removes, without replacement, a CFG from the BQ and puts its JBCs on the BQM output.
- (**Augment** – A): removes an IFG from the BQ, makes it complete by augmenting a `tstore` instruction to it, and puts the new complete group on the BQM output. A

`tload` instruction replaces the removed JBCs in the BQ.

- **(Increment** – *I*): removes, without replacement, an `iinc` instruction from the BQ and puts it on the BQM output.
- **(Rename** – *N*): renames a LV, which is pending in the BQ, to a tagged one. Nothing changes in the BQ, but a renamed folding group that consists of a producer and a tagged consumer is put on the BQM output.
- **(Destroy** – *D*): eliminates a folding group from the BQ. Nothing is put on the BQM output and nothing replaces the disposed JBCs in the BQM.
- **(Duplicate** – *U*): duplicates a group of JBCs in the BQ. Nothing is put on the BQM output.
- **(Swap** – *W*): changes the order of JBCs storage in the BQ. Nothing is put on the BQM output.

Table 6.1 maps folding anchors to BQM-internal operations. (Refer to Appendix A for folding notation.) In this table we use the notation $BQM_{V,A,I,N,D,U,W}$ to describe the operation performed by the BQM in executing a certain anchor type. The symbols in the subscript denotes the seven supported BQM-internal operations. If a certain operation is to be performed, the corresponding subscript is included, otherwise, it is omitted. Note that the *rename* operation occurs only when there is a hazard. Close examination of the BQM notation for each operation helps us outline an architecture of the module. It is worth mentioning that some of the bytecode details are not recognized at this stage, e.g., normal LVs are not extracted from the producers or consumers. However, tagging (including tag selection) is done here.

6.4.1.3 Bytecode Processing within the BQM

We can distinguish two sets of BQM-internal operations in Subsection 6.4.1.2: terminating and non-terminating operations. Terminating operations finish the corresponding JBC processing at the BQM level without passing them to subsequent modules. These operations mimic different stack operations (e.g., `dup` and `pop`). Table 6.2 summarizes folding operations performed by each of these operations. Table 6.3 lists the JVM opcodes that are terminated internally at the BQM. Examples for terminating operations are included in Table 6.4. Non-terminating operations contribute to intermediate processing of JBCs.

Table 6.1. *Mapping different anchor instructions to the folding operations performed by the bytecode queue manager (BQM).*

Type	Anchor notation	Folding template	BQM notation
consumer	$A_{1,0}$	PC	$BQM_{V,N}$
operator	$A_{[1,2],1}$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}C$	$BQM_{V,N}$
	$A_{[1,2],1}$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}T$	BQM_A
independent	$A_{0,0}$	$A_{0,0}$	BQM_I
destroyer	$A_{[1,2],0}$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],0}$	BQM_D
duplicator	$A_{[1,2],0}$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],0}$	BQM_U
swapper	$A_{2,0}$	$PPA_{2,0}$	BQM_W
load	$A_{0,1}$	$A_{0,1}C$	$BQM_{V,N}$
	$A_{0,1}$	$A_{0,1}T$	BQM_A
	$A_{[1,2],1}$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}C$	$BQM_{V,N}$
	$A_{[1,2],1}$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}T$	BQM_A
store	$A_{[1,3],0}$	$\underbrace{P\cdots P}_{[1,3]}A_{[1,3],0}$	BQM_V
branch	$A_{0,0}$	$A_{0,0}$	BQM_V
	$A_{0,1}$	$A_{0,1}C$	$BQM_{V,N}$
	$A_{0,1}$	$A_{0,1}T$	BQM_A
	$A_{[1,2],0}$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],0}$	BQM_V

They produce intermediate results for the subsequent modules to continue processing. Table 6.5 summarizes folding operations performed by each of these operations and shows the corresponding output produced by the BQM. Examples for non-terminating operations are included in Table 6.6.

6.4.1.4 BQM Architectural Module

Figure 6.4 shows the inputs and outputs of the BQM module. BQM inputs can be divided into two sets. The first enables JBC queuing (*queueEnable*), provides the count of JBCs to be queued (*fetchLength*), and supplies the data to be queued (*cacheBus*). The second set is

Table 6.2. *Summary of folding operations done in the framework of terminating BQM-internal operation.*

BQM operation	BQ before folding	Folding template	BQ after folding
destroy	$\underbrace{P\cdots P}_{x}\overbrace{\underbrace{P\cdots P}_{[1,2]}A_{[1,2],0}}^{folding\ group}X\cdots$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],0}$	$\underbrace{P\cdots P}_{x}X\cdots$
duplicate	$\underbrace{P\cdots P}_{x}\underbrace{P\cdots P}_{[0,2]}\overbrace{\underbrace{P\cdots P}_{[1,2]}A_{[1,2],0}}^{folding\ group}X\cdots$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],0}$	$\underbrace{P\cdots P}_{x}\underbrace{P\cdots P}_{[1,2]}\underbrace{P\cdots P}_{[0,2]}\underbrace{P\cdots P}_{[1,2]}X\cdots$
swap	$\underbrace{P\cdots P}_{x}\overbrace{P_{LV_m}P_{LV_n}A_{2,0}}^{folding\ group}X\cdots$	$P_{LV_m}P_{LV_n}A_{2,0}$	$\underbrace{P\cdots P}_{x}P_{LV_n}P_{LV_m}X\cdots$

Table 6.3. *JVM opcodes terminated internally at the BQM.*

Category	Opcodes	Symbol	Folding group fields
DESTROYER			
pop stack element(s) u (and v)	pop	$D(1)$	$(u : X : 4)$
	pop2	$D(2)$	$\frac{(u:X:4),(v:X:4)}{(u:X:8)}$
DUPLICATOR			
duplicate stack element(s) u (and v)	dup	$U(1)$	$(u : X : 4)$
	dup_x1	$U(1)$	$(u : X : 4)$
	dup_x2	$U(1)$	$(u : X : 4)$
	dup2	$U(2)$	$\frac{(u:X:4),(v:X:4)}{(u:X:8)}$
	dup2_x1	$U(2)$	$\frac{(u:X:4),(v:X:4)}{(u:X:8)}$
	dup2_x2	$U(2)$	$\frac{(u:X:4),(v:X:4)}{(u:X:8)}$
SWAPPER			
swap stack elements v and u	swap	W	$(u : X : 1), (v : X : 1)$

Table 6.4. *Examples on the folding operations done in the framework of terminating BQM-internal operations.*

BQ before folding	Folding template	BQM operation	BQ after folding
iload_2 $\overbrace{\textit{iload_1 iload_3 pop2}}^{folding\ group}$ *istore_0*	$PPA_{2,0}$	destroy	*iload_2 istore_0*
iload_2 $\overbrace{\textit{iload_1 dup_x1}}^{folding\ group}$ *pop*	$PA_{1,0}$	duplicate	*iload_1 iload_2 iload_1 pop*
iload_2 $\overbrace{\textit{iload_1 iload_3 swap}}^{folding\ group}$ *pop*	$PPA_{2,0}$	swap	*iload_2 iload_3 iload_1 pop*

Table 6.5. *Summary of folding operations done in the framework of non-terminating BQM-internal operations and corresponding BQM output.*

BQM operation	BQ before folding†	Folding template (BQM output)	BQ after folding	BQM output		
				Opcode	Producers	Consumer
remove	$\underbrace{P\cdots P}_{z}\overbrace{\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}C}^{\text{folding group}}X\cdots$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}C$	$\underbrace{P\cdots P}_{z}X\cdots$	$A_{[1,2],1}$	$\underbrace{P\cdots P}_{[1,2]}$	C
	$\underbrace{P\cdots P}_{z}\overbrace{\underbrace{P\cdots P}_{[1,3]}A_{[1,3],0}}^{\text{folding group}}X\cdots$	$\underbrace{P\cdots P}_{[1,3]}A_{[1,3],0}$	$\underbrace{P\cdots P}_{z}X\cdots$	$A_{[1,3],0}$	$\underbrace{P\cdots P}_{[1,3]}$	—
	$\underbrace{P\cdots P}_{z}\ \overbrace{A_{0,1}C}^{\text{folding group}}\ X\cdots$	$A_{0,1}C$	$\underbrace{P\cdots P}_{z}X\cdots$	$A_{0,1}$	—	C
	$\underbrace{P\cdots P}_{z}\ \overbrace{PC}^{\text{folding group}}\ X\cdots$	PC	$\underbrace{P\cdots P}_{z}X\cdots$	nop	P	C
	$\underbrace{P\cdots P}_{z}\ \overbrace{A_{0,0}}^{\text{folding group}}\ X\cdots$	$A_{0,0}\neq I$	$\underbrace{P\cdots P}_{z}X\cdots$	$A_{0,0}$	—	—
augment	$\underbrace{P\cdots P}_{z}\overbrace{\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}}^{\text{folding group}}Y\cdots$	$\underbrace{P\cdots P}_{[1,2]}A_{[1,2],1}T_{LV_t}$	$\underbrace{P\cdots P}_{z}QY\cdots$	$A_{[1,2],1}$	$\underbrace{P\cdots P}_{[1,2]}$	T_{LV_t}
	$\underbrace{P\cdots P}_{z}\ \overbrace{A_{0,1}}^{\text{folding group}}\ Y\cdots$	$A_{0,1}T_{LV_t}$	$\underbrace{P\cdots P}_{z}QY\cdots$	$A_{0,1}$	—	T_{LV_t}
increment	$\underbrace{P\cdots P}_{z}\ \overbrace{A_{0,0}}^{\text{folding group}}\ X\cdots$	$A_{0,0}=I_{LV_x,CN_y}$	$\underbrace{P\cdots P}_{z}X\cdots$	iinc	$P_{LV_x}P_{CN_y}$	C_{LV_x}
rename	$\cdots P\cdots C\cdots$	PT_{LV_t}	$Q_{LV_t}\cdots C\cdots$	nop	P	T_{LV_t}

† Y is either P or $A_{z',[0,1]}$

Table 6.6. *Examples on the folding operations done in the framework of non-terminating BQM-internal operations.*

BQM operation	BQ before folding	Folding template	BQM output	BQ after folding
remove	iload_2 $\overbrace{\text{iload_1 ineg istore_0}}^{\text{folding group}}$ pop	$PA_{1,1}C$	iload_1 ineg istore_0	iload_2 pop
	iload_2 $\overbrace{\text{iload_1 putstatic 24}}^{\text{folding group}}$ pop	$PA_{1,0}$	iload_1 putstatic 24	iload_2 pop
	iload_2 $\overbrace{\text{jsr 26 astore_0}}^{\text{folding group}}$ pop	$A_{0,1}C$	jsr 26 astore_0	iload_2 pop
	iload_2 $\overbrace{\text{iload_1 istore_0}}^{\text{folding group}}$ pop	PC	iload_1 istore_0	iload_2 pop
	iload_2 $\overbrace{\text{goto}}^{\text{folding group}}$ pop	$A_{0,0}\neq I$	goto	iload_2 pop
augment	iload_2 $\overbrace{\text{aload_1 getfield 1 1}}^{\text{folding group}}$ pop	$PA_{1,1}T_{LV_8}$	iload_1 getfield 1 1 tstore 8	iload_2 tload 8 pop
	iload_2 $\overbrace{\text{ldc 10}}^{\text{folding group}}$ pop	$A_{0,1}T_{LV_8}$	ldc 10 tstore 8	iload_2 tload 8 pop
increment	iload_2 $\overbrace{\text{iinc 5 7}}^{\text{folding group}}$ pop	$A_{0,0}=I$	iinc 5 7	iload_2 pop
rename	iload_2 $\overbrace{\text{iload_1} \cdots \text{istore_1}}$	PT_{LV_8}	iload_1 tstore 8	iload_2 tload 8 $\cdots$ istore_1

read from the FIQ when the signal *dequeueEnable* is asserted to select a JBC window for issuing on the *issueBytecodes* output port. This window is defined by a starting bytecode (*startBytecode*) and a length (*bytecodeCount*). This input group also includes a pointer to the folding group opcode and its arguments (*opcodePosition*) and its size *opcodeSize* to enable reading the opcode on the *issueOpcode* output port. *insertOffset* is used in case of duplication or swapping to indicate the insertion point of JBC. (The folding algorithm sets the *opcodePosition* to -1 in case of consumer anchors, which results in reading a `nop` opcode on the *issueBytecodes*). BQM operation is controlled by the FIG via the *foldingOperation* signal. Non-terminating BQM-internal operations read folding groups on the *issueBytecodes* output bus, whereas a terminating operation reads `nop's` on this output bus, which eventually injects a bubble into the pipeline. The RS minicontroller (explained in Subsection 6.4.1.5) negates the *dequeueEnable* signal to insert a bubble in the pipeline when there is no empty RS entry.

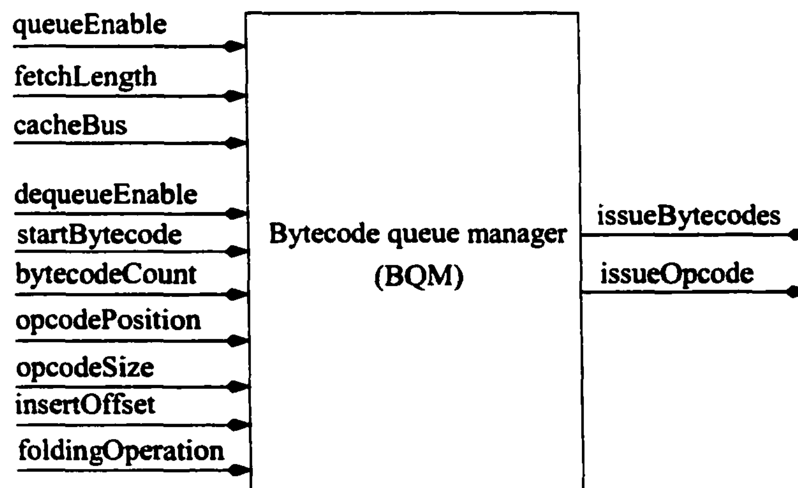


Figure 6.4. *Inputs and outputs of the BQM.*

Internally, the BQ is a linear first-in first-out array of entries (*BQentries*) with *currentBQpointer* pointing to the last bytecode in the BQ. Each entry in the BQ holds a JBC, a tagged flag (`Q`), and an ignore flag (`I`). The tagged flag (`Q`) indicates that the corresponding bytecode is an inserted, tagged producer and has to be prefixed with a `tload` bytecode while being dispatched, to form a tagged producer. The `tload` bytecode itself is not saved internally in the BQ to simplify operation and to save space. Only the tag is saved in the

BQ. When a 2-byte producer is replaced with a `tload` bytecode, the ignore flag (`I`) is asserted to mark the high order byte to be discarded at issue time.

BQM maintains a pool of tags in a linear array of entries (*tagEntries*). The tag entry count is maintained in the constant (*tagCount*). When needed, a free tag (*freeTag*) is selected to be used for a new tagged producer or consumer.

6.4.1.5 BF Minicontroller

As shown in Subsection 6.2.4, a minicontroller is needed to generate two signals (*issueEnable* and *fillEnable*) to control the BF based on the folding operation (*foldingOperation*) indicated by the FIG. This controller is located in the same pipeline stage as the BQM and structured as a simple combinational logic circuit (Figure 6.5). The following logic equations summarize the assertion conditions for the output control signals.

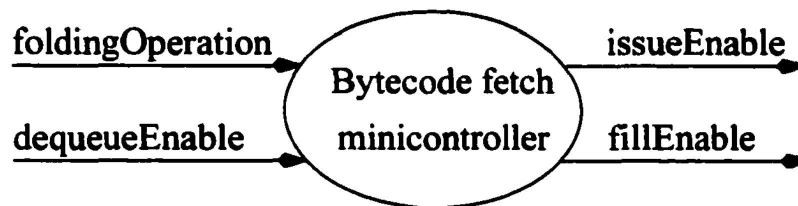


Figure 6.5. *Inputs and outputs of the BF minicontroller.*

$$issueEnable \leftarrow (foldingOperation != invalid) \cdot (foldingOperation != rename) \cdot (foldingOperation != duplicate) \quad (6.1)$$

$$fillEnable \leftarrow (foldingOperation == invalid) \quad (6.2)$$

6.4.1.6 BQM Operation Steps

Allowable operations on the BQ are *deleteEntries (startingEntry, noOfEntries)*, *duplicateEntries (startingEntry, insertingPoint, boundryEntry)*, and *swapEntries (startingEntry, insertingPoint, boundryEntry)*. All these operations update the *currentBQpointer*. Each

BQM operation is decomposed into a set of ministeps. Some of the ministeps are shared among different operations. Table 6.7 lists all distinct ministeps used by different BQM-internal operations. For each ministep, the table shows the accomplished action(s). As shown in this table, some of these ministeps are parameterized.

Figure 6.6 visualizes the operations done for JBC folding and issuing. As Figure 6.7 explains, issuing from non-terminating operations is a multiplexing process: producer(s), opcodes, and consumers from different folding groups are multiplexed on one output bus.

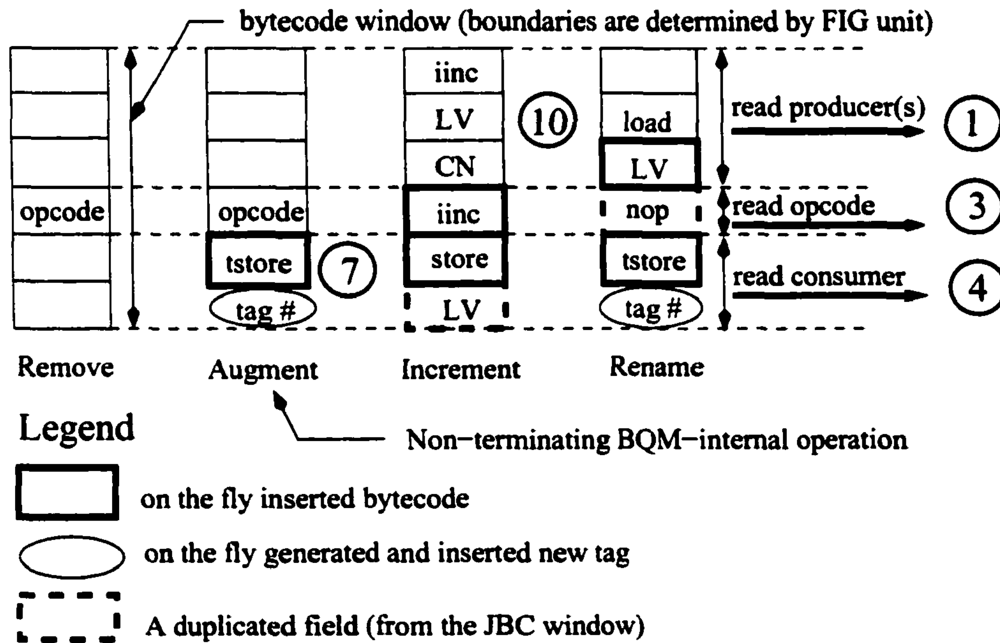


Figure 6.6. *Arrangements made within the BQM to prepare for JBC folding and issuing.* The diagram shows only non-terminating BQM operations that result in reading JBCs on the BQM output. Numbers in circles correspond to the ministeps numbers in Table 6.7.

Figure 6.8 shows how the BQ looks like after executing each of the BQM-internal operations. Only flags (`I` and `Q`) that are asserted by the operations are shown in the figure. These flags then control the producer assembling process as depicted in Figure 6.9.

Figure 6.10 visualizes the tag-related BQM ministeps. Internally, the BQM, maintains a pool of tags. It keeps track of the count of bytecodes referencing each tag so that a free tag can be selected when needed. Issuing a tagged bytecode decrements the count attached to the corresponding tag, while duplicating a tagged bytecode increments the attached count.

Table 6.7. *Ministeps involved in performing BQM-internal operations.*

No	Ministep ([parameters, ···])	Symbol	Action(s) or bookkeeping†
1	Read producers	*RP*	1 **for** i = startBytecode **to** opcodePosition - 1 **do** 2 **if** (BQentries[i].I == 0) **then** 3 **begin** 4 **if** (BQentries[i].Q = = 1) **then** 5 issueBytecodes[m++] := tload; 6 issueBytecodes[m++]:= BQentries[i].JBC; 7 **end if;**
2	Decrement count attached to tags in the folding group	*DC*	1 **for** i = startBytecode **to** opcodePosition - 1 **do** 2 **if** (BQentries[i].Q = = 1) **then** 3 tagEntries[BQentries[i].JBC]- -;
3	Read opcode (opcode position, opcode size)	*RO(pos, size)*	1 **for** i = 0 **to** size -1 **do** 2 issueOpcode[i] := BQentries[pos+i].JBC;
4	Read consumer	*RC*	1 **for** i = opcodePosition+opcodeSize **to** startBytecode+bytecodeCount-1 **do** 2 issueBytecodes[m++] := BQentries[i].JBC;
5	Remove read entries (starting JBC, number of entries)	*RE(pos, size)*	1 BQentries.deleteEntries(pos,size);
6	Get a free tag	*GT*	1 **for** i = 0 **to** tagCount - 1 **do** 2 **if** (tagEntries[i] = = 0) **then** 3 **begin** 4 freeTag := i; 5 tagEntries[i]++; 6 **end if;**
7	Generate a tagged consumer	*GC*	1 issueBytecodes[m++] := tstore; 2 issueBytecodes[m++] := freeTag;
8	Insert a tagged producer	*IP*	1 BQentries[startBytecode] := (freeTag,1,0);
9	Rename a LV	*RL*	1 **if** (BQEntries[startBytecode-1].JBC = = "iload") **then** 2 BQentries[startBytecode-1].I :=1; 3 BQentries[startBytecode] := (freeTag,1,0);
10	Read a producer to rename its LV	*RR*	1 **if** (BQentries[startBytecode-1].JBC == "iload") **then** 2 issueBytecodes[m++] := BQentries[startBytecode-1].JBC; 3 issueBytecodes[m++] := BQentries[startBytecode].JBC;
11	Duplicate entries	*DE*	1 BQentries.duplicateEntries(startBytecode,insertOffset,opcodePosition); 2 currentBQpointer := currentBQpointer + bytecodeCount - opcodeSize;
12	Increment count attached to tags in the folding group	*IC*	1 **for** i = startBytecode **to** startBytecode - bytecodeCount - 1 **do** 2 **if** (BQentries[i].Q = = 1) **then** 3 tagEntries[BQentries[i].JBC]++;
13	Swap entry orders	*SE*	1 BQentries.swapEntries(startBytecode,insertOffset,opcodePosition);

† The index m, which is initialized to zero, is used to specify a position on the output bus (*issueBytecodes*) for the next JBC to be written.

In summary, the proposed BQM architecture not only compensates for the lack of the processor stack, but also allows for efficient processing of JBCs. The smart BQM permits random access to the queue elements which facilitates insertion/duplication and dele-

Table 6.8. *Sequencing description of the ministeps involved in performing each of the BQM-internal operations.*

Operation	Sequencing description
remove	$DC, (RP\|RO(opcodePosition, opcodeSize)\|RC), RE(startBytecode, bytecodeCount)$
augment	$DC, (RP\|RO(opcodePosition, opcodeSize)\|(GT, GC)), RE(startBytecode, bytecodeCount - 2), IP$
increment	$(RP\|RO(startBytecode, 1)), RE(startBytecode, bytecodeCount)$
rename	$((GT, RL)\|RR), GC$
destroy	$DC, RE(startBytecode, bytecodeCount)$
duplicate	$IC, DE, RE(opcodePosition, opcodeSize)$
swap	$SE, RE(opcodePosition, opcodeSize)$

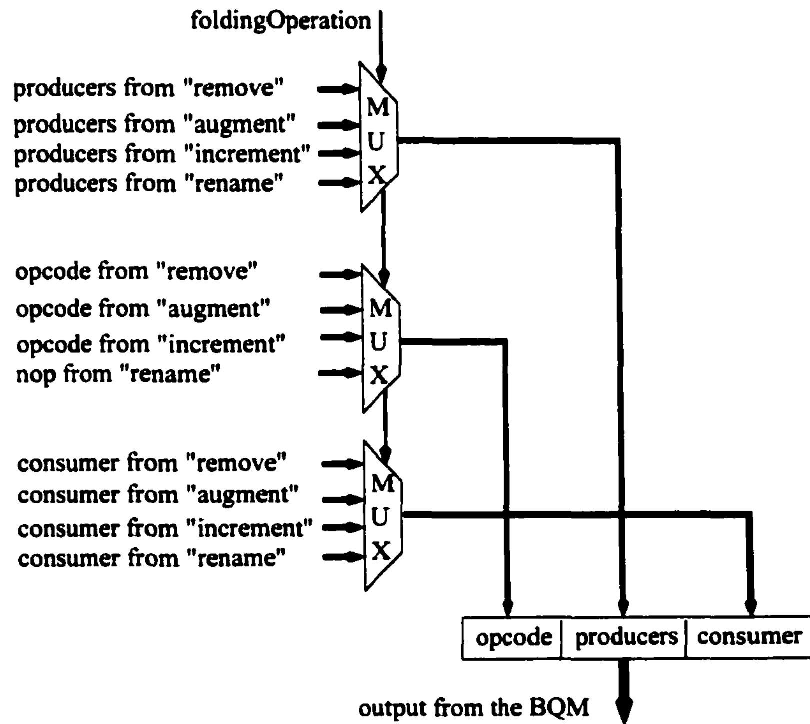


Figure 6.7. *BQM output assembled from queued JBCs.*

The BQM-internal operation, specified by the FIG, controls this assembly process.

tion/removal of the JBCs from the queue.

6.4.1.7 BQM Operation Control, Priorities, and Sequencing

BQM-internal operations are done one at a time. For each folding group, an operation is specified by the FIG. The only exception from that is when there is a need for LV renaming,

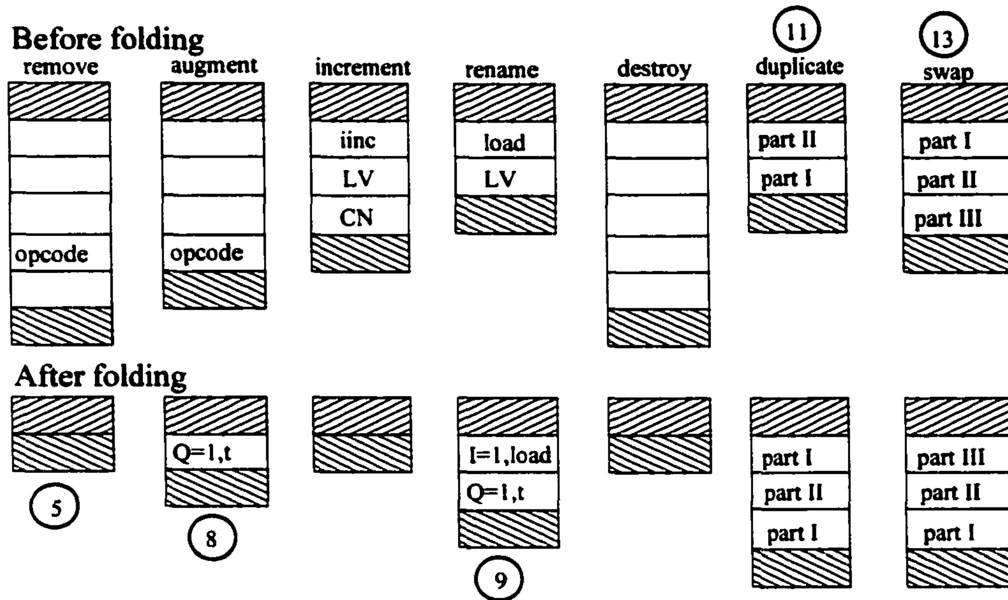


Figure 6.8. *How the BQ looks like after executing each BQM-internal operation.*

Numbers in circles correspond to the ministep numbers in Table 6.7.

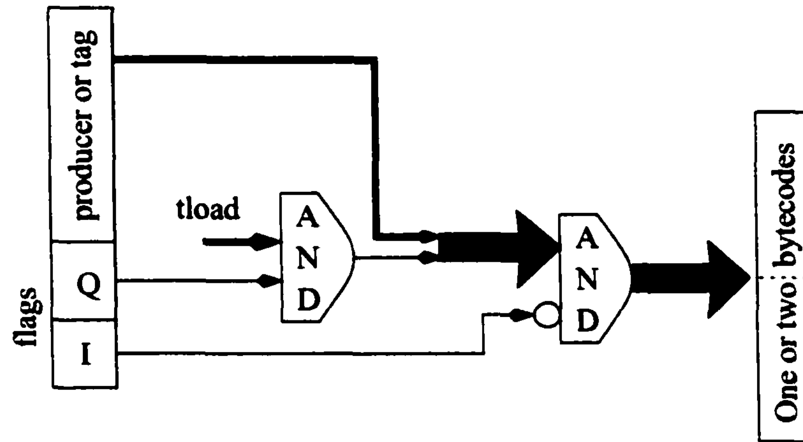


Figure 6.9. *Generating producers while executing BQM-internal operations that put JBCs on the bytecode queue manager (BQM) output.*

The Q and I flags control this process. A null result is dropped from the output.

where the FIG issues two consecutive commands to the BQM. The first one is for renaming (BQM_N) and the other one is for normal folding group removal (BQM_V). This is indicated in Table 6.1 by ($BQM_{V,N}$). Table 6.8 shows a possible sequencing description of each of

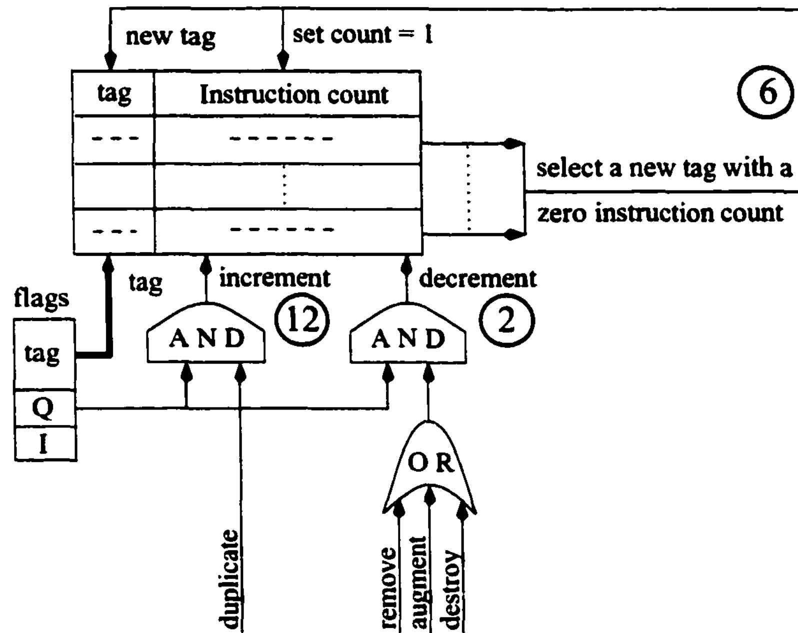


Figure 6.10. *Tag handling inside the bytecode queue manager (BQM).*
Numbers in circles correspond to the ministeps numbers in Table 6.7.

the BQM-internal operations in terms of the ministeps listed in Table 6.7. The process of queuing JBCs read from the *cacheBus* is not shown as it is a straightforward appending operation that updates the *currentBQpointer*. Notice that these ministeps are hardwired and thus could be parallelized easily (if the semantic allows that).

6.4.2 Folding Translator (FT)

An important property of JBCs is that statically determinable type state enables simple on the fly translation of JBCs into efficient machine code [1]. The FT utilizes this property to dynamically translate a folding group to a decoded RISC instruction format that is usable by the subsequent stages.

6.4.2.1 FT Architectural Design Principles

The FT is situated in the middle of the pipeline as the boundary between Java-specific modules and Java-independent (RISC-style) ones. This facilitates incorporating a RISC core that could work independently from Java, addressing Architectural Design Principle 6.

Because JBCs are fetched and locally stored as pure JVM instructions, JBCs generated by typical Java compilers can be executed on JAFARDD without any modifications as required by Architectural Design Principle 7.

6.4.2.2 Produced Instruction Format

The FT dynamically translates a folding group's producers and consumers into a set of LV indices. When these indices are assembled with the folding group anchor, and an output from the BQM, a typical RISC instruction is obtained as in Figure 6.11. Modules following the FT look at these indices as if they are RISC register specifiers. Fields in this instruction together with their sizes and meaning are shown in Figure 6.12 and explained as follows[1]:

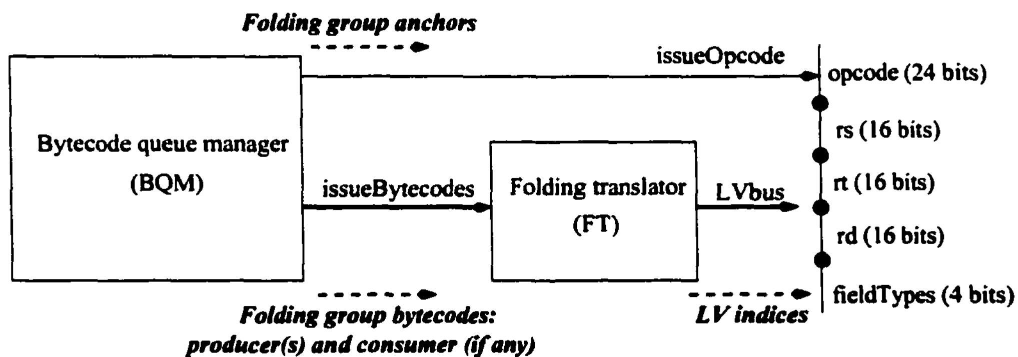


Figure 6.11. *Inputs and outputs of the FT and the instruction format at its output and the entrance of the dynamic scheduling and execution pipeline stage.*

opcode (24)	rs (16)	rt (16)	rd (16)	fieldTypes (4)
folding anchor and its arguments	1st source	2nd source	3rd source or destination	source types

Figure 6.12. *JAFARDD native instruction format.*

Field sizes are shown in bits.

[1] Field sizes are selected in a way that matches the JVM specification. However, much smaller sizes could be used according to the benchmarking results presented in Chapter 2. For example, source fields could be sufficiently encoded in 8 bits.

- *opcode (24 bits)* JVM operation and 0 to 2 arguments. JVM opcodes can be up to 3 bytes in length.
- *rs (16 bits)* First source operand.
- *rt (16 bits)* Second source operand.
- *rd (16 bits)* Third source operand or LV destination index.
- *fieldTypes* [4] *(4 bits)* Four decoding bits divided into three groups that specify the type of values saved in the *rs*, *rt*, and *rd* fields, respectively (two bits correspond to *rd*). We use the vector notation $fieldTypes[rs\ rt\ rd_0\ rd_1]$ to specify the values stored in this field. The decoding bits for the *rs* and *rt* fields are encoded in the same way: 0 indicates that the value saved in the field is to be interpreted as an LV index (default), whereas 1 indicates an immediate value. When any of these fields is not used a 0 is saved in the field and in the corresponding decoding bit, as it is not a destructive action to read an unwanted LV. This is different for the *rd* decoding bits. These bits not only have to specify if the field is being used as an index for a source or a destination, but also have to specify whether the field is being used or not. It is destructive to write to unwanted LV since a tag will be associated with this LV. Thus, the decoding bits for the *rd* are split into two bits. The left specifies whether it is a source (0– default) or a destination (1) and the right shows source type, like the case of the other two fields. When the *rd* field is not used, its decoding bits are set to 00.

6.4.2.3 Bytecode Processing Within the FT

The FT is responsible for translating dynamically folding groups' producers and consumers into instruction fields. This dynamic translation process is summarized in Table 6.9 for each relevant folding template. Starting from a folding group, this table shows the values generated for each instruction field at the output of the FT. Figure 6.13 illustrates the folding translation process by an example. Table 6.10 includes more examples.

Table 6.11 shows the JBCs that are dynamically translated by the FT and how they are mapped to the instruction fields in Figure 6.12. `iinc` is treated as a special case as it holds its necessary LV producer and consumer indices in the operation arguments, provided in the bytecode stream, instead of being provided by other JVM instructions. (In stack based JVM realization, `iinc` does not reference the stack.) Later this opcode is interpreted as an addition. The `wide` bytecode is treated by the FT as a directive that affects the width of

Table 6.9. *Mapping different folding templates to the folding translator unit (FT) output.*

Folding template	Opcode	Translated fields			
		rs	rt	rd	Field types
$\frac{P_{LV_x}}{P_{CN_a}} \frac{P_{LV_y}}{P_{CN_b}} \frac{P_{LV_z}}{P_{CN_c}} A_{3,0}$	$A_{3,0}$	$\frac{x}{a}$	$\frac{y}{b}$	$\frac{z}{c}$	$\frac{LV_x \to 0}{CN_a \to 1}\ \frac{LV_y \to 0}{CN_b \to 1}\ 0 \frac{LV_z \to 0}{CN_c \to 1}$
$\frac{P_{LV_x}}{P_{CN_a}} \frac{P_{LV_y}}{P_{CN_b}} A_{2,1} C_{LV_m}$	$A_{2,1}$	$\frac{x}{a}$	$\frac{y}{b}$	m	$\frac{LV_x \to 0}{CN_a \to 1}\ \frac{LV_y \to 0}{CN_b \to 1}$ 10
$\frac{P_{LV_x}}{P_{CN_a}} \frac{P_{LV_y}}{P_{CN_b}} A_{2,0}$	$A_{2,0}$	$\frac{x}{a}$	$\frac{y}{b}$	0	$\frac{LV_x \to 0}{CN_a \to 1}\ \frac{LV_y \to 0}{CN_b \to 1}$ 00
$\frac{P_{LV_x}}{P_{CN_a}} A_{1,1} C_{LV_m}$	$A_{1,1}$	$\frac{x}{a}$	0	m	$\frac{LV_x \to 0}{CN_a \to 1}$ 0 10
$\frac{P_{LV_x}}{P_{CN_a}} A_{1,0}$	$A_{1,0}$	$\frac{x}{a}$	0	0	$\frac{LV_x \to 0}{CN_a \to 1}$ 0 00
$A_{0,1} C_{LV_m}$	$A_{0,1}$	0	0	m	0 0 10
$A_{0,0} \neq I$	$A_{0,0}$	0	0	0	
$\frac{P_{LV_x}}{P_{CN_a}} C_{LV_m}$	*nop*	$\frac{x}{a}$	0	m	$\frac{LV_x \to 0}{CN_a \to 1}$ 0 10
$A_{0,0} = I_{LV_x, CN_a}$	$A_{0,0}$	x	a	x	0 1 10

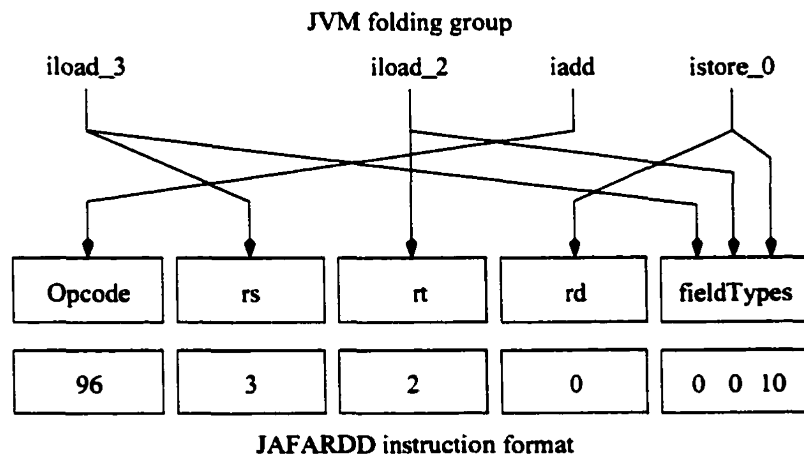


Figure 6.13. *An example of translating a folding group into a JAFARDD instruction.*

the operand. Other JBCs (e.g., nop), if passed to FT, are considered as don't cares.

6.4.2.4 FT Architectural Module

Figure 6.11 shows the inputs and outputs of the FT module. *issueBytecodes* provides the folding group producers and consumer. The anchor instruction itself (*issueOpcode*) bypasses the FT to the next modules. As such, the FT does not have to identify the opcode and separate it from the producers and consumer, which simplifies its design. The excep-

Table 6.10. *Examples on dynamic translation for different folding templates.*

Folding template	Folding group example	Opcode	Translated fields			
			rs	rt	rd	fieldTypes
$P_{LV_5} P_{LV_3} P_{CN_1} A_{3,0}$	aload 5 iload_3 iconst_1 iastore_2	iastore_2	5	3	1	0 1 01
$P_{LV_3} P_{CN_1} A_{2,1} C_{LV_2}$	iload_3 iconst_1 iadd istore_2	iadd	3	1	2	0 1 10
$P_{LV_1} P_{CN_1} A_{2,0}$	aload_1 iconst_1 putfield 0 20	putfield 0 20	1	1	0	0 1 00
$P_{LV_5} A_{1,1} C_{LV_7}$	iload 5 ineg tstore 7	ineg	5	0	7	0 0 10
$P_{CN_1} A_{1,0}$	iconst_1 putstatic 0 20	putstatic 0 20	1	0	0	1 0 00
$A_{0,1} C_{LV_9}$	ldc 10 tstore 9	ldc 10	0	0	9	0 0 10
$A_{0,0} \neq I$	goto	goto	0	0	0	0 0 00
$P_{CN_3} C_{LV_2}$	iconst_3 istore_2	nop	3	0	2	1 0 10
$A_{0,0} = I_{LV_3,CN_2}$	iinc 3 2	iinc	3	2	3	0 1 10

Table 6.11. *Correspondence between no-effect, producer, consumer, and increment JVM categories and the dynamically produced instruction fields.*

Subcategory	Opcodes†	Symbol	Folding group operands	Translated value	Target instruction fields(s)		
					rs	rt	rd
NO EFFECT							
no effect	nop	N	n/a	n/a			
PRODUCERS							
load stack with a CN	bipush x:1	P_{CN_x}	$(S(x, byte, 1, 2) : int : 2)$	$S(x, byte, 1, 2)$	✓	✓	✓
	iconst_$< a >$	P_{CN_a}	$(Z(a, int, 1, 2) : int : 2)$	$Z(a, int, 1, 2)$	✓	✓	✓
	sipush x:2	P_{CN_x}	$(x : int : 2)$	x	✓	✓	✓
load stack from a LV	iload_$< b >$	P_{LV_b}	$(LV_b : int : 4)$	$Z(b, int, 1, 2)$	✓	✓	✓
	iload x:1	P_{LV_x}	$(LV_x : int : 4)$	$Z(x, int, 1, 2)$	✓	✓	✓
	wide iload x:2	P_{LV_x}	$(LV_x : int : 4)$	x	✓	✓	✓
	tload t:1	P_{LV_t}	$(LV_t : int : 4)$	$Z(t, int, 1, 2)$	✓	✓	✓
CONSUMERS							
store stack into a LV	istore_$< b >$	C_{LV_b}	$\Rightarrow (LV_b : int : 4)$	$Z(b, int, 1, 2)$			✓
	istore x:1	C_{LV_x}	$\Rightarrow (LV_x : int : 4)$	$Z(x, int, 1, 2)$			✓
	wide istore x:2	C_{LV_x}	$\Rightarrow (LV_x : int : 4)$	x			✓
	tstore t:1	C_{LV_t}	$\Rightarrow (LV_t : int : 4)$	$Z(t, int, 1, 2)$			✓
INDEPENDENT							
increment a LV with a CN	iinc x:1 y:1	I_{LV_x, CN_y}	$(LV_x : int : 4)$,	$Z(x, int, 1, 2)$	✓		
			$(S(CN_y, byte, 1, 2) : int : 2)$	$Z(y, int, 1, 2)$		✓	
			$\Rightarrow (LV_x : int : 4)$,	$Z(x, int, 1, 2)$			✓
	wide iinc x:2 y:2	I_{LV_x, CN_y}	$(LV_x : int : 4)$,	x	✓		
			$(CN_y : short : 2)$	y		✓	
			$\Rightarrow (LV_x : int : 4)$	x			✓

- † $< c >$ is to be replaced with a possible value for c. $a \in \{-1 \cdots 5\}$. -1 is represented as $m1$. $b \in \{0 \cdots 3\}$

tions from that are consumers and the `iinc` anchor, which is duplicated by the BQM to be included in the producer JBCs in addition to being read as the opcode (as required by the

FT operation). The output of the FT is the *LVbus* that holds a set of LV indices to be read or written by the LVF.

Internally, the FT has a combinational design that translates the input format into the output format. No internal data structures are needed for the operation of this module.

6.4.2.5 Non-Translated Instructions

Complex operations in the JVM as summarized in Table 5.2 are not folded, and thus are not translated in JAFARDD. The table mentions the folding obstacle for each category. These bytecodes only represent 5 to 8% of executed JBCs in our benchmark studies [86, 87]. In the rare case where the hardware encounters a nonfoldable JBC, execution is trapped to the OS for software emulation. This approach fully complies with Architectural Design Principle 1.

6.5 Dynamic Scheduling and Execution

Like most modern processors, JAFARDD dynamically schedules instructions for execution by incorporating the reservation station mechanism. In this section, we highlight the structure of the local variable file (*LVF*), reservation stations (*RSs*), and execution units (*EXs*). We will emphasize on how they help to achieve a high degree of parallelism, and dynamic scheduling and execution in JBC processing.

6.5.1 Local Variable File (LVF)

The FT translates a folding group's producers and consumer into a set of LV indices. Modules following the translator interprets these indices as if they are RISC register specifiers. In this subsection, we focus on the details of the LVF architecture and its functionality.

6.5.1.1 LVF Architectural Design Principles

To address Architectural Design Principle 3, an LVF is included on chip to permit manipulating JVM's LVs as regular RISC registers as well as providing the venue for adding some LVF-specific operations. In addition to the regular register file operations, JAFARDD's LVF performs advanced operations that compensate for the lack of a stack, exploit ILP

among JBCs and facilitate the interaction with Tomasulo's algorithm implementation as specified by Architectural Design Principles 2, 4, and 5.

6.5.1.2 LVF-Internal Operations

A regular register-file-like structure usually has two operations: operand read and result write. In the frame work of Tomasulo's algorithm as adopted in this work, result write is changed to result capture (from the CDB). An operation is needed for reserving the destination LV to indicate the absence of the value. In addition, two operations are added to the LVF to compensate for the lack of a stack. These two operations are LV-to-LV copy and immediate write into a LV. Thus, in total, the LVF performs five different internal operations controlled by a set of enable signals:

- LV-to-LV copy (**Copy** – C): copies a LV to another one, which is useful for folding groups that consist of a LV producer and a LV consumer, e.g., (`iload_2 istore_0`).
- Immediate write (**Write** – W): writes a constant directly into one of the LVs, which is useful for folding groups that consist of a CN producer and a LV consumer, e.g., (`iconst_2 istore_0`).
- Source operand read (**Read(n)** – $D(n)$): reads three LVs, together with their tags. As a LV read is a harmless operation, this operation is always enabled. However, n indicates the number of operands actually required by the instruction.
- Register reserve (**Reserve** – V): marks an LV as busy waiting for a result. A tag is attached to it and at the same time forwarded to the next pipeline stage. If this operation is not enabled (as in the case of anchors that need no consumers), a zero tag is forwarded to the following pipeline stages.
- Result capture (**Capture** – T): The LVF is always snooping on the CDB for broadcasted values. Once a value is captured, its tag is compared with all entries. If a match is found, the value is stored and the internal tag is nullified.

Table 6.12 lists anchors that need access to the LVF and maps their requirements onto the LVF-internal operations. (Destroyer, swapper, and duplicator anchors are not shown in this table as they are executed and terminated within the BQM and do not affect the LVF.) The table also shows folding templates for those anchors. In this table, we use the notation $LV_{C,W,D(n),V,T}$ to describe the operation performed by the LVF in executing a certain anchor

type. The symbols in the subscript denote the five supported LVF operations. If the LVF is required to perform a certain operation, the corresponding subscript is included, otherwise, it is omitted. As *read* operations involve multiple operands, we include the number of operands to be read attached to the symbol D. (Note that although *read* is always enabled, we only include the number of LVs that are needed to be read in the notation.) Also observe that a *reserve* operation usually needs a *capture* operation to follow, if the attached tag is not over-written by another *reserve operation*. Thus, both *reserve* and its corresponding *capture*, indicated by $LV_{V,T}$, do not take place in the same clock cycle. Close examination of the LV notation for each operation helps us outline the architecture of the module.

Table 6.12. *Mapping different anchor instructions to the operations performed by the local variable file (LVF).*

Type	Anchor notation	Folding template	LVF notation	Folding group example
consumer	$A_{1,0}$	$P_{CN_x} C_{LV_y}$ $P_{LV_x} C_{LV_y}$	LV_W LV_C	iconst_0 tstore 7 iload_1 istore_0
operator	$A_{[1,2],1}$	$\underbrace{P \cdots P}_{[1,2]} A_{[1,2],1} C$	$LV_{D([1,2]),V,T}$	iload_1 ineg istore_0
independent	$A_{0,0}$	$A_{0,0}$	$LV_{D(1),V,T}$	iinc 3 1
load	$A_{0,1}$ $A_{[1,2],1}$	$A_{0,1} C$ $\underbrace{P \cdots P}_{[1,2]} A_{[1,2],1} C$	$LV_{V,T}$ $LV_{D([1,2]),V,T}$	ldc 10 istore_0 aload_1 getfield 0 2 istore_0
store	$A_{[1,3],0}$	$\underbrace{P \cdots P}_{[1,3]} A_{[1,3],0}$	$LV_{D([1,3])}$	iload_2 putstatic 0 2
branch	$A_{0,0}$ $A_{0,1}$ $A_{[1,2],0}$	$A_{0,0}$ $A_{0,1} C$ $\underbrace{P \cdots P}_{[1,2]} A_{[1,2],0}$	LV_{-} $LV_{V,T}$ $LV_{D([1,2])}$	goto jsr 2 astore_0 iload_0 ifeq 30

6.5.1.3 Early Assignment of Instruction/LV Tags

In the original Tomasulo algorithm, the tag attached to a destination LV is equal to the index of the RS that hosts the pending instruction. Thus, dispatching an instruction along the pipeline mandates getting an available RS index back to the LVF. There are two approaches to get the tag: **(1)** send a request to the RS controller for an available RS index. The pipeline is stalled until the tag comes back to the LVF. Then the instruction proceeds to the assigned

RS unit. Successor instructions that access tagged LVs are dispatched with these tags. The weak point of this approach is the pipeline stall due to the interaction between the dispatching mechanism and the RS controller; (2) allow the current instruction and the following ones to proceed without getting the RS tag. The tag is assigned to the instruction when it enters its RS. At that moment, the tag is sent back to the LVF, and is also forwarded to already dispatched instructions which reference that destination. Although this approach does not need to stall the pipeline, controlling the flow of the tag would be quite complex.

The problems with the above two mechanisms stem from the tag's counterflowing in the pipeline from the RS unit back to the LVF. In our architecture, we use a novel tag requisition technique: a separate tag generation unit (*TG*) located in the same pipeline stage as the LVF (Figure 4.3) is introduced. For each destination LV specified by a translated folding group, a renaming tag is obtained from the TG to be assigned to the instruction and to the destination LV. This tag propagates together with instruction operands to the hosting RS and EX. The rename tag assignment remains valid until the instruction is retired. The TG listens on the CDB for broadcasted tags to return them to the internally maintained free tag pool.

Figure 6.14 shows the input and output signals of the TG. It always holds at its output a *newTag*. When the LVF captures this tag, it raises the *tagTaken* signal to ask for a new tag to be generated. The pipeline does not need to be stalled during this process.

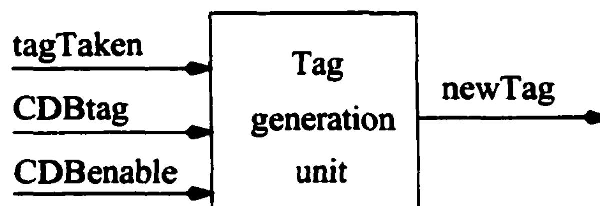


Figure 6.14. *Inputs and outputs of the tag generation unit (TG).*

Internally, the TG keeps a pool of tags with a status flag that indicates whether the tag is in use or not. It also snoops on the CDB. When the *CDBenable* is asserted, the TG captures the broadcasted tag (*CDBtag*) and internally frees it for reuse. With the number of tags equaling the total number of reservation stations' entries plus one, the TG is guaranteed to have an unused tag every clock cycle.

This tag-early assignment approach has a number of merits: (1) it saves the overhead

(time and circuitry) in communicating with the RS unit and getting the tag back to the LVF; **(2)** a new tag is generated every clock cycle and in the same pipeline stage as the LVF, which means a zero-cycle tag generation; and **(3)** it eliminates the need for information counterflow in the pipeline, which simplifies the design and speeds up processing. The cost for a separate TG is the extra storage needed at each RS to store the tag together with the instruction, as the tag will no longer be equal to the RS unit index. We believe that the advantages outweigh the disadvantages.

6.5.1.4 LVF Architectural Module

Figure 6.15 shows the inputs and outputs of the LVF module. By observing supported anchors in Table 6.12 and their LVF notation, it can be concluded that there is a need to read up to three LVs at once. The zero producer-count anchor is the case of load anchors reading from the CP with the index supplied as the opcode argument. Store operations that have arrays as destinations need three producers. This mandates having a three-read port LVF. (Some non-foldable instructions require a producer count that has no upper limit, e.g., `tableswitch`). On the other hand, from the internal LV operation description, we see that we need only single write, copy, and CDB ports. Table 6.13 shows how each LVF-internal operation makes use of different input indices, enable signals values and tags, and output values and tags. *newTag* and *tagTaken* are used to communicate with the TG. Such arrangement supports a high degree of parallelism as multiple operations can be enabled in one clock cycle as will be explained shortly. *copy* and *write* operations are of a special nature as they can not be enabled concurrently. The benchmarking of the OPEX folding algorithm indicated that these two operations were not executed as often as other operations as shown in Chapter 2. Thus, we elected to reuse one of the *copy* ports to index the destination LV for *write* operations.

Table 6.14 shows the instruction field needed to be connected to each LVF index port for each folding template. From this table the following connections can be made: *copyIndex* to rs, *writeIndex* to rd, *readIndex*[3] to $[rs, rt, rd]$, and *reserveIndex* to rd. These connections are to be permanent wired even if the corresponding LVF operation is not needed since different enable signals control the information transfer process.

Internally, the LVF entries are organized as a linear array (*LVentries*) like any regular register file. Each entry in this array has a tag field for integration with the RSs. Entries in

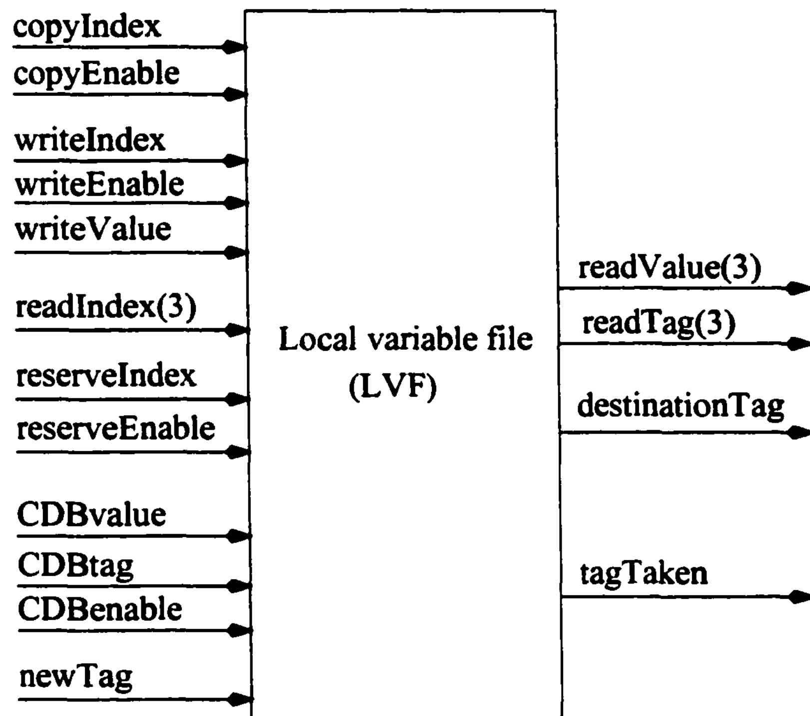


Figure 6.15. *Inputs and outputs of the local variable file (LVF).*
The suffix (3) attached to all *read* signals indicates an aggregation of three signals indexed from 0 to 2.

Table 6.13. *Port usage for different LVF-internal operations.*

LV Operation	LV port(s)						
	Enable signal	Input		Output		LV indices	
		Value	Tag	Value	Tag	Source	Destination
Copy	copyEnable	n/a	n/a	n/a	n/a	copyIndex	writeIndex
Write	writeEnable	writeValue	n/a	n/a	n/a	n/a	writeIndex
Read[i]†	always enabled	n/a	n/a	readValue[i]	readTag[i]	readIndex[i]	n/a
Reserve*	reserveEnable	n/a	n/a	n/a	destinationTag	n/a	reserveIndex
Capture	CDBEnable	CDBValue	CDBTag	n/a	n/a	n/a	LV $x^\ddagger$

- † We will use the suffix [i] to designate a certain port from an aggregation of ports.
- ‡ $x \in \{x|LVentries[x].tag == CDBtag\}$. The CDBtag is used as a matching tag, i.e., the LV that holds a matching tag is targeted by the operation.
- ⋆ When reserve is not enabled, the destination tag (*destinationTag*) is cleared.

Table 6.14. *Mapping different folding templates to the local variable file (LVF) inputs.*

Folding template	LVF enable signals			LVF indices			
	copy	**write**	**reserve**	**copy**	**write**	**read[3]**	**reserve**
$\frac{P_{LV_x}}{P_{CN_a}} \frac{P_{LV_y}}{P_{CN_b}} \frac{P_{LV_z}}{P_{CN_c}} A_{3,0}$	0	0	0	0	0	[rs,rt,rd]	0
$\frac{P_{LV_x}}{P_{CN_a}} \frac{P_{LV_y}}{P_{CN_b}} A_{2,1} C_{LV_m}$	0	0	1	0	0	[rs,rt,0]	*rd*
$\frac{P_{LV_x}}{P_{CN_a}} \frac{P_{LV_y}}{P_{CN_b}} A_{2,0}$	0	0	0	0	0	[rs,rt,0]	0
$\frac{P_{LV_x}}{P_{CN_a}} A_{1,1} C_{LV_m}$	0	0	1	0	0	[rs,0,0]	*rd*
$\frac{P_{LV_x}}{P_{CN_a}} A_{1,0}$	0	0	0	0	0	[rs,0,0]	0
$A_{0,1} C_{LV_m}$	0	0	1	0	0	[0,0,0]	*rd*
$A_{0,0} \neq I$	0	0	0	0	0	[0,0,0]	0
$\frac{P_{LV_x}}{P_{CN_a}} C_{LV_m}$	$\frac{LV_x \to 1}{CN_a \to 0}$	$\frac{LV_x \to 0}{CN_a \to 1}$	0	*rs*	*rd*	[0,0,0]	0
$A_{0,0} = I_{LV_x, CN_a}$	0	0	1	0	0	[rs,0,0]	*rd*

this array have the simple format (*value, tag*). A zero tag indicates a ready value, whereas a non-zero tag signals the absence of a valid value.

Entries in the LVF are split into two groups: one group includes the JVM visible LVs, whereas the other group includes the tagged LVs allocated by the BQM and used by the folding algorithm. From the processor design point of view, there is no architectural difference in manipulating these two LV types.

6.5.1.5 LVF Minicontroller

The minicontroller in Figure 6.16 is needed to generate the three control signals that control the LVF: *reserveEnable*, *copyEnable*, and *writeEnable*. (As will be shown later, the *reserveEnable* signal is also propagated forward to control the RSs.) The LVF minicontroller generates these signals based on the *issueOpcode* (generated by the BQM), the instruction *fieldTypes* (generated by the FT), and whether the LVF is enabled by the control signal *dequeueEnable*. This controller is structured as a simple combinational logic circuit. Table 6.14 shows the control signal values for different folding templates. From this table, we deduce the following logic equations for different control signals:

$$nil \leftarrow (opcode == nop) \tag{6.3}$$

$$reserveEnable \leftarrow fieldTypes[rd_0] \cdot !nil \cdot dequeueEnable \tag{6.4}$$

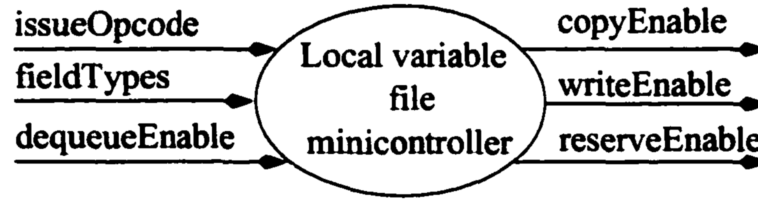


Figure 6.16. *Inputs and outputs of the LVF minicontroller.*

$$copyEnable \leftarrow \overline{fieldTypes[rs]} \cdot nil \cdot dequeueEnable \quad (6.5)$$

$$writeEnable \leftarrow fieldTypes[rs] \cdot nil \cdot dequeueEnable \quad (6.6)$$

6.5.1.6 LVF Operation Steps

Table 6.15 gives the sequential steps of each LVF-internal operation. Operations are performed according to their precedence hierarchy shown in the table. Operations with low precedence numbers are evaluated before operations with higher numbers, while operations with the same precedence cannot be executed in the same clock cycle.

6.5.1.7 LVF Operation Control, Priorities, and Sequencing

A set of conditions must be satisfied for a certain LVF-internal operation to occur. In what follows we give the logic equation for each operation (all events are enabled at the clock rising edge):

$$copy \leftarrow copy_enable \cdot \overline{reserve_enable} \quad (6.7)$$

$$write \leftarrow write_enable \cdot \overline{reserve_enable} \quad (6.8)$$

$$read \leftarrow 1 \quad (6.9)$$

$$reserve \leftarrow reserve_enable \quad (6.10)$$

$$capture \leftarrow CDB_enable \cdot \overline{copy_enable} \cdot \overline{write_enable} \cdot \overline{reserve_enable} \quad (6.11)$$

To allow for higher degree of parallelism, the LVF supports multiple operations in a single clock cycle. However, the LVF architecture, operation efficiency and correctness impose the following constraints on performing different operations in a single clock cycle:

Table 6.15. *Steps required for each LVF-internal operation.*

Precedence	Operation	Action(s) or bookkeeping
1	capture	1 $\forall x \in \{x \mid LVentries[x].tag == CDBtag\}$ **do** 2 **begin** 3 LVentries[x].value := CDBvalue; 4 LVentries[x].tag := 0; 5 **end for;**
2	copy	1 LVentries[writeIndex].value := LVentries[copyIndex].value; 2 LVentries[writeIndex].tag := LVentries[copyIndex].tag;
2	write	1 LVentries[writeIndex].value := writeValue; 2 LVentries[writeIndex].tag := 0;
3	read	1 **for** i = 0 **to** 2 **do** 2 **if** ((writeEnable == 1) **and** (writeIndex == readIndex[i])) **then** 3 **begin** 4 readValue[i] := writeValue; 5 readTag[i] := 0; 6 **end if;** 7 **else if** ((copyEnable == 1) **and** (writeIndex == readIndex[i])) **then** 8 **begin** 9 readValue[i] := LVentries[copyIndex[i]].value; 10 readTag[i] := LVentries[copyIndex[i]].tag; 11 **end if;** 12 **else if** ((CDBenable == 1) **and** (LVentries[readIndex[i]].tag==CDBtag)) **then** 13 **begin** 14 readValue[i] := CDBvalue; 15 readTag[i] := 0; 16 **end if;** 17 **else** 18 **begin** 19 readValue[i] := LVentries[readIndex(i)].value; 20 readTag[i] := LVentries[readIndex(i)].tag; 21 **end else;**
4	reserve	1 LVentries[reserveIndex].tag := newTag; 2 destinationTag := newTag; 3 tagTaken := 1 until next clock falling edge;

- *copy* and *write* cannot be done in the same clock cycle because the latter uses the *writeIndex* port for the destination LV.
- *copy* and *write* have higher priorities than *capture*. This means that if *capture* is enabled with *copy* and/or *write* and targets the same destination LV as any of them, capture is ignored.
- If *capture* and *copy*, or *write* are enabled targeting the same LV as any of the sources for the *read* operation in the same clock cycle, the value being written into the des-

tination LV is directly forwarded to the output read port without waiting for it to be saved in the corresponding LV first. The previous priority rule also applies here; *copy* and *write* cancel a *capture* operation.

- *reserve* overrides all operations targeting the same LV.
- *read* does not need to wait for *capture*, *copy*, or *write* to finish before it starts. The input to the latter operations, if any of them is enabled and targeting the same destination LV, could be forwarded to the *read* mechanism. This is extremely useful, especially if the *capture* operation is enabled and the captured tag is the same as that for any of the read LVs. In this case, it is more efficient to forward the captured value to the output with a null tag. (In fact, if this case is not handled, a deadlock situation might happen! If a tag that has just been broadcasted on the CDB is forwarded from the LVF to an RS, the entry will stall in its RS forever or perhaps the RS may capture the wrong value.)
- The above rules imply that an operation output could be overridden by another higher priority operation. For optimization, a lower priority operation is disabled if a higher priority operation is active and targeting the same LV.

The following formula puts the above constraints into a sequencing description that shows the allowable LV operations in a single clock cycle (refer to Appendix B for description of the used notation).

$$(\overline{V} \cdot \overline{P} \cdot \overline{W} \rightarrow T)|(\overline{V} \rightarrow (P \oplus W))|(((P \oplus W);T) \Rightarrow D)|V \qquad (6.12)$$

6.5.2 Reservation Stations (RSs)

RSs were first introduced by Tomasulo for the floating point unit of the IBM 360/91 [42, 197, 214] to facilitate OOO instruction issues. A number of extensions of this algorithm were then suggested [215, 216, 217, 218, 219]. In this subsection, we present the RS design used in JAFARDD.

6.5.2.1 RS Architectural Design Principles

RSs provide a venue and mechanism for LV renaming, dynamic scheduling, and pipeline hazard handling which permit achieving a higher degree of ILP (Architectural Design Principles 4 and 5). These three features are also effective tools for stack dependency resolution (Architectural Design Principle 2).

6.5.2.2 Tomasulo's Algorithm for Java Processors

Instead of being blocked at the local variable access pipeline stage waiting for a valid operand to become available (a RAW hazard), or for a potential false data hazard to be resolved (WAW or WAR), or for a busy EX to be freed, an instruction can be dispatched conditionally. The instruction together with any available operands, are buffered in an RS. A not-yet available LV operand is renamed to the index of the instruction that will produce the expected operand. A free EX will start processing when the source operands are available. When processing is completed, the EX will forward the result to the awaiting RSs, and will write back to an LV if there is no WAW conflict. The result forwarding and writing are done via the CDB.

To meet the requirements of Tomasulo's algorithm and the OPEX bytecode folding algorithm, instructions and operands are tagged. Instructions are tagged by assigning a tag of value t to them at the local variable access pipeline stage. Operands are tagged by expanding their representation in the LVF from just an indexed value (v) to an indexed, order set: (v, t), where v is the operand value and t is an associated LV tag. If v holds the operand value, t is set to 0. Otherwise, t is set equal to the tag associated with the instruction that is to produce the operand value, v. When an instruction is issued, it is tagged with a tag t. As Tomasulo's algorithm involves early reading of the operands (known as *rename bound*), when an instruction is issued to the instruction shelve pipeline stage, either $(v, 0)$ (if the LV is not tagged) or $(0, t)$ (if the LV is tagged) is forwarded with the instruction. When an instruction is dispatched to its EX, the instruction LV tag is memorized by that EX. When the instruction is executed and the result is available, the corresponding EX broadcasts the result with the instruction tag t on the CDB to all modules. If a broadcasted tag matches that of a tagged LV in the LVF or a to-be-read operand in an RS, the result is captured and the corresponding tag is internally nullified. Having all its operands, an instruction can then start execution if the required EX is free.

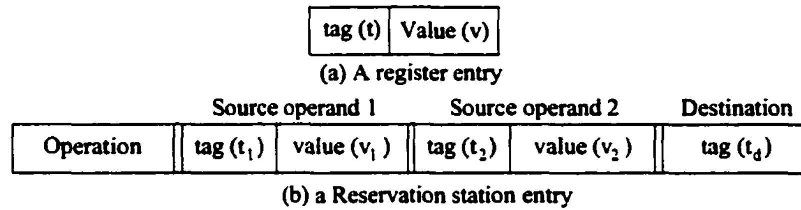


Figure 6.17. *A register entry and a reservation station entry in Tomasulo's algorithm.*

A zero tag means the corresponding value is ready.

6.5.2.3 LV Renaming

Figure 6.18 shows the LV renaming design space from [91, 220, 221]. From the implementation point of view, as our architecture does not require any changes in the Java compilation phase or need any JBC preprocessing, LV renaming is done *dynamically* during the issuing and execution phases.

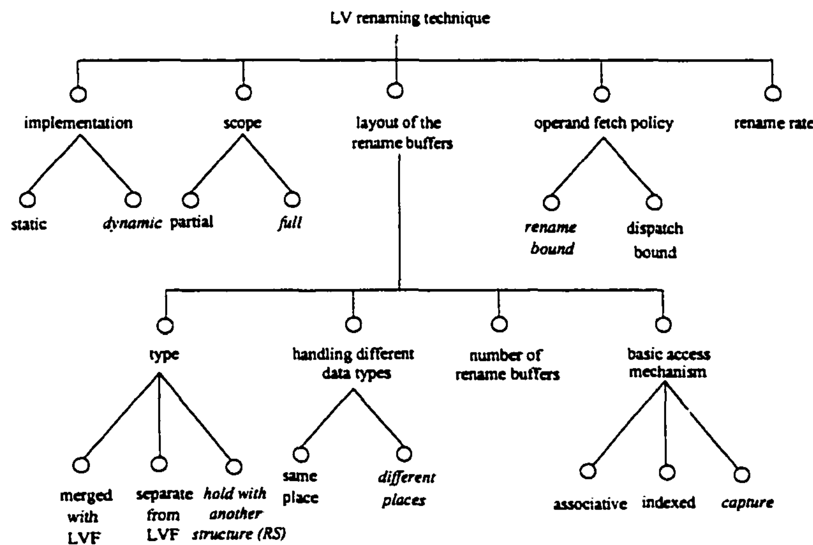


Figure 6.18. *Design space for LV renaming.*

Options selected for use in our architecture are shown in italic.

By specifying a unified LV structure for fixed and floating point data, the JVM specification makes it easy to incorporate a *full* renaming scope. Consequently, JAFARDD does not incorporate explicit separation in the renaming process; it renames all data types.

In JAFARDD, it is quite natural to *hold rename buffers with another structure (RSs)*. We

also have *different places* (RSs) for handling different data types. The *number of rename buffers* is dependent on the implementation. Operands in our processor are not fetched by searching the renaming buffers. If operands are not available in their corresponding LVs, they are *captured* when broadcasted, instead. Thus, a required operand cannot be held in more than one location at any one time. This has the advantage of saving the time and space cost for associative search, indirect indexing or parallel access to multiple entries.

In JAFARDD, operands are fetched during renaming (*rename bound*). If the operand is not available, the associated tag is read.

The *rename rate* usually equals the issue rate of the processor to avoid any bottlenecks. As JAFARDD issues one folding group per cycle, it does one rename per cycle.

6.5.2.4 Instruction Shelving

Figure 6.19 shows our design strategies in the instruction shelving design space [91, 220, 222]. Our design shelves all data and instruction types thus providing a *full* shelving scope.

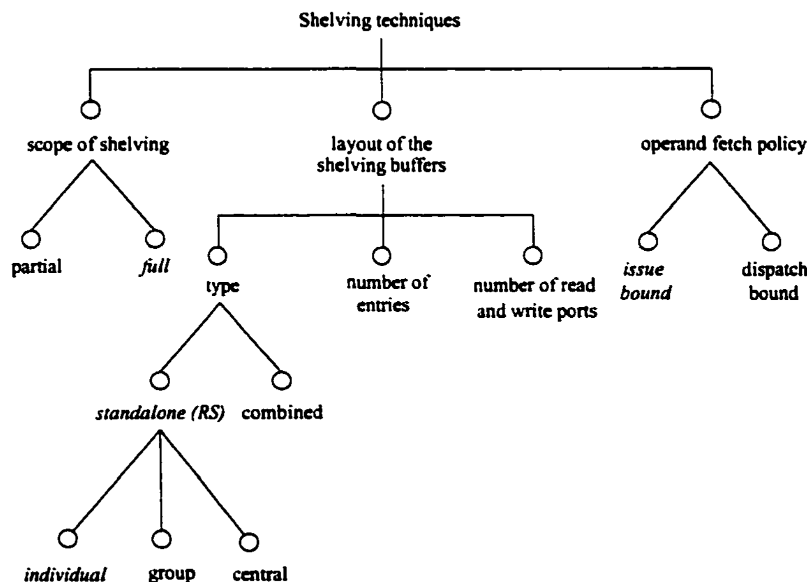


Figure 6.19. *Design space for shelving.*

Options selected for use in our architecture are shown in italic.

As *individual* stations are dedicated for each EX, RSs are *standalone* buffers used exclusively for shelving. The *number of shelving buffers* is dependent on the implementation. As

RSs are not shared between EXs, one port for reading and another for writing are enough.

An *issue bound* policy is adopted to fetch operands. During instruction issuing, the source and destination LV indices of the issued instructions are forwarded to the LVF for fetching the referenced operands or their associated tags, and for assigning tags to the destination LVs, respectively.

6.5.2.5 Instruction Dispatching

Instruction dispatching consists of scheduling instructions held in a particular RS unit for execution and sending them to the allocated EX. (In this dissertation, we say that an instruction is issued or dispatched when it is proceeding to the FT or an EX, respectively.) Figure 6.20 shows our design strategies on the dispatching design space tree [91, 220].

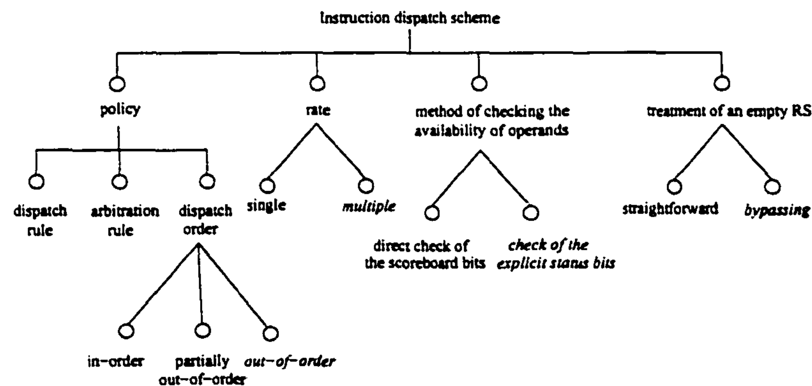


Figure 6.20. *Design space for instruction dispatch scheme.*

Options selected for use in our architecture are shown in italic.

The *dispatch rule* specifies when instructions are considered executable. Employing RSs takes care of resolving false data dependency. Furthermore, if a suitable EX is available, then those instructions whose operands are available are executable. Our *arbitration rule* selects the earliest instruction in the sequence when more than one instruction is executable. We employ an OOO instruction dispatch order policy. A non-executable instruction does not block the dispatch of subsequent executable instructions. Instead, any executable instruction held in the shelving buffer is eligible for dispatch. This implies, from the implementation point of view, that all of the shelving buffer entries have to be inspected for executability.

As our architecture has multiple EXs with individual RSs, it can afford a *multiple instruction dispatch rate.*

Another issue for the dispatching design space is the method to check whether all operands of instructions held in the shelving buffers are available. Since JAFARDD fetches operands during instruction issue, *a check of the explicit status bits* (the tag field) is enough at dispatch time.

An important issue related to instruction dispatch is the treatment of all-empty or all-non-executable RSs. We selected an advanced approach for handling the situation when instructions, with all available operands, arrive at all-empty or all-non-executable RSs: *bypassing*. Here, some additional circuitry permits instructions to bypass an empty RS unit and immediately be forwarded to the EXs to eliminate any unnecessary delay.

6.5.2.6 RS Internal Operations

In addition to the RS basic operations (instruction shelving and dispatch, and result capture), JAFARDD also incorporates two advanced operations (bypass and forward), i.e., the RS unit performs five different internal operations controlled by a set of conditions:

- Instruction shelving (**Shelve** – $S(j)$): an instruction missing one or more of its operands or targeting a busy EX is shelved (queued) in the free j^{th} RS entry. Operand values and/or tags, operation, and the destination tag are all stored in this entry.
- Instruction dispatch (**Dispatch** – $D(j)$): an instruction, with all operands available, is dispatched (dequeued) from the jth RS entry for execution by a free EX.
- Result capture from the CDB (**Capture** – $T(j, i)$): the CDB announces a newly released result together with its destination tag to be captured by awaiting RSs. If the associated tag matches the jth instruction's i^{th} operand tag, the broadcasted result is captured and saved as the corresponding operand value.
- Empty RS bypass (**Bypass** – B): if all RSs are empty or there is no executable entry, the new executable instruction bypasses the RS unit directly to a free EX. This arrangement optimizes the operation of the pipeline.
- Result forwarding (**Forward** – $F(i)$): a result is forwarded from the CDB to the ith operand of an instruction arriving at the input of the RS unit (called input instruction). There are two scenarios for that: in one situation, called *Forward and shelve* ($F_S(i)$),

the result is forwarded to an input instruction that has an operand with a tag matching the broadcasted one before the instruction is shelved. Whereas, in the other scenario, called *Forward and bypass* ($F_B(i)$), the result is forwarded to an input instruction to bypass empty RSs. $F(i)$ denotes CDB result forwarding to the input instruction in any of these scenarios. Forwarding is a crucial process to the operation of the RS. If an input instruction that has a tag matching the CDB tag is stored in the RS without result forwarding, the pending operand will miss the opportunity to capture the CDB result. Later, this pending tag might find a match with another broadcasted tag, which would lead to an incorrect operation.

6.5.2.7 RS Unit Architectural Module

JAFARDD includes a number of RS units. Internally, each RS unit has a set of entries stored in the array *RSentries*, each having slots for operands *sourceValue*[3] and *sourceTag*[3], an operation *operation*, and a result tag *resultTag*. Each RS entry can accommodate up to three operands and the maximum number of entries is defined by the design constant *RSsize*.

Inputs and outputs of an RS unit are shown in Figure 6.21. There are three input groups. The first group inputs the instruction fields to be shelved: *operandValue*, *readTag*, *destinationTag*, and *issueOpcode*. The signal *RSenable* enables the reading of these fields. The second group of signals (*CDBvalue*, *CDBtag*, and *CDBenable*) captures the result from the CDB. The third group contains the signal *EXbusy*, which indicates whether the corresponding EX is busy or free. The outputs of this module are directed towards its EX: *EXsource*, *EXoperation*, and *EXdestinationTag*. Signal *EXenable* enables reading these values to the corresponding EX. The *operandValue* is generated from the LVF's *readValue* and the immediate fields (rs, rt, and rd) via the immediate/LV selector shown in Figure 6.22. The immediate fields are passed to the selector in the array *immValue*. This circuit selects between the immediate value from the instruction or the value read from the LVF according to the *fieldTypes* embedded in the translated instruction. Each RS unit outputs 3 flags: *RSready* indicates the availability of at least one instruction to be dispatched, *RSfull* flags the lack of internal space in the RS unit, and *RSempty* is asserted when there is a no entry in the RS unit. *RSempty* will not be asserted with any of the other two flags. Two internal counters *entryCount* (holds the number of current entries in an RS unit) and *readyCount* (holds the number of entries that are ready for dispatch) are used to to generate these flags.

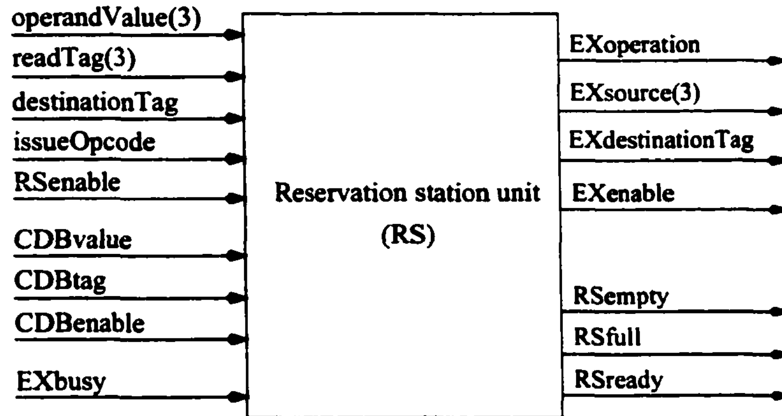


Figure 6.21. *Inputs and outputs of a reservation station (RS) unit.*

6.5.2.8 RS Unit Minicontroller

The minicontroller in Figure 6.23 is needed to control all RS units and consequently the pipeline. This controller generates a set of enable signals, *RSenable*'s, that are used to control different RS units. The RS minicontroller is structured as a simple combinational circuit that uses the *issueOpcode*, and *reserveEnable* signals carried forward from previous stages, and the *RSfull*, and *RSready* signals from individual RS units to select a single appropriate RS unit, and *EXBusy* signals from all EXs. Empty spaces in RSs and operand data types are a major factor in this selection process. (JVM opcodes indicate the type of the expected operands, e.g., `iadd` is used to add two integer operands.) The RS minicontroller also generates the signal *dequeueEnable* that enables the BQM to dequeue another folding group. This signal is negated to insert a bubble in the pipeline when there is no free RS to host the current instruction and the instruction can not be dispatched directly to a suitable EX (when either the EX is busy or the instruction operands are not available).

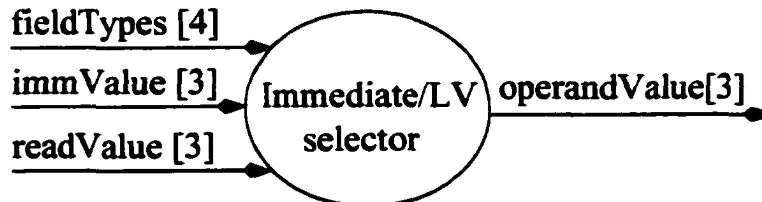


Figure 6.22. *Inputs and outputs of the immediate/LV selector.*

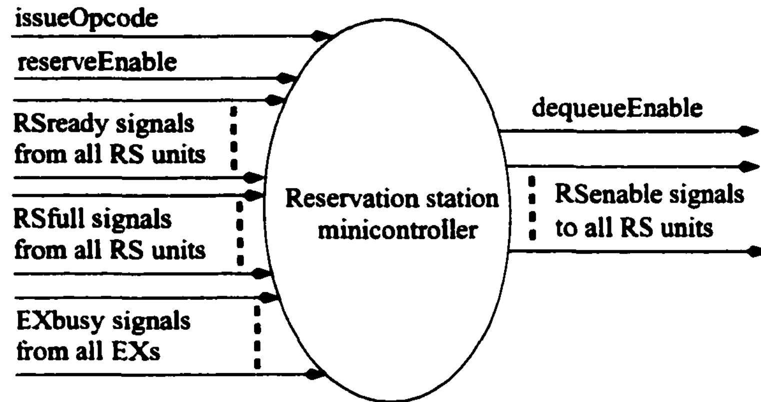


Figure 6.23. *Inputs and outputs of the RS unit minicontroller.*

6.5.2.9 RS Operation Steps

Table 6.16 gives the steps of RS unit-internal operations. Operations start execution when the appropriate condition is asserted. (Refer to formulas in Subsection 6.5.2.10.)

6.5.2.10 RS Operation Control, Priorities, and Sequencing

A set of conditions need to be satisfied for a certain RS unit-internal operation to occur. In what follows we give the optimized logic equation for each operation. Events are enabled at the clock rising edge. (In these formulas, j is an RS entry and i is an operand in this entry). For more readable equations, the logic condition $taggedInput(i)$ is defined to indicate that the ith operand is tagged.

$$RSempty \leftarrow (entryCount == 0) \tag{6.13}$$

$$RSfull \leftarrow (entryCount == RSsize) \tag{6.14}$$

$$RSready \leftarrow (readyCount \mathrel{!}= 0) \tag{6.15}$$

$$taggedInput(i) \leftarrow (readTag(i) \mathrel{!}= 0) \tag{6.16}$$

$$dispatch \leftarrow RSready \cdot \overline{EXbusy} \cdot RSenable \tag{6.17}$$

$$\begin{aligned} shelve \leftarrow\ & ((EXbusy + \sum_{i=0}^{2} taggedInput(i) \cdot \\ & \overline{forward(i)}) \cdot \overline{RSfull} + \\ & RSready \cdot \overline{EXbusy}) \cdot RSenable \end{aligned} \tag{6.18}$$

Table 6.16. *Steps required for RS unit-internal operations.*

Operation	Action(s) or bookkeeping
dispatch	**1 for** k = 0 **to** entryCount **do** 2 **if** (**not** ($\exists i$ (RSentries[k].sourceTag[i] != 0)) **then** 3 **begin** 4 j = k; 5 break; 6 **end if;** 7 EXsource := RSentries[j].sourceValue; 8 EXoperation := RSentries[j].operation; 9 EXdestinationTtag := RSentries[j].resultTag; 10 entryCount - -; 11 EXenable := 1 until next clock falling edge;
forward	1 $F = \{k \| readTag[k] == CDBtag\}$;
shelve	1 $\forall i \in F$ **do** 2 **begin** 3 RSentries[entryCount].sourceValue[i] := CDBvalue; 4 RSentries[entryCount].sourceTag[i] := 0; 5 **end for;** 6 $\forall i \notin F$ **do** 7 **begin** 8 RSentries[entryCount].sourceValue[i] := readValue[i]; 9 RSentries[entryCount].sourceTag[i] := readTag[i]; 10 **end for;** 11 RSentries[entryCount].operation := issueOpcode; 12 RSentries[entryCount].resultTag := destinationTag; 13 **if** (**not** ($\exists i$ (RSentries[entryCount].sourceTag[i] != 0))) **then** 14 readyCount ++; 15 entryCount ++;
bypass	1 $\forall i \in F$ **do** 2 EXsource[i] := CDBvalue; 3 $\forall i \notin F$ **do** 4 EXvalue[i] := readValue[i]; 5 EXoperation := issueOpcode; 6 EXdestinationTag := destinationTag;
capture	1 **for** j = 0 **to** entryCount - 1 **do** 2 **begin** 3 $S = \{k \| RSentries[j].sourceTag[k] != 0\}$; 4 **if** (S != ϕ) **then** 5 if ($\forall i \in$ S (RSentries[j].sourceTag[i] == CDBtag)) 6 readyCount ++; 7 $\forall i \in \{k \| RSentry[j].sourceTag[k] == CDBtag\}$ **do** 8 **begin** 9 RSentries[j].sourceValue[i] := CDBvalue; 10 RSentries[j].sourceTag[i] := 0; 11 **end for;** 12 **end for;**

$$
\begin{aligned}
bypass &\leftarrow (\prod_{i=0}^{2}\overline{(\overline{taggedInput(i)} + forward(i))}) \cdot \\
&\quad (RSempty + \overline{RSready}) \cdot \\
&\quad \overline{EXbusy} \cdot RSenable && (6.19) \\
forward(i) &\leftarrow (readTag(i) == CDBtag) \cdot \\
&\quad CDBenable && (6.20) \\
forward_S(i) &\leftarrow forward(i) \cdot shelve && (6.21) \\
forward_B(i) &\leftarrow forward(i) \cdot bypass && (6.22) \\
capture(j,i) &\leftarrow (RSentries(j) \cdot tag(i) == CDBtag) \cdot \\
&\quad CDBenable && (6.23)
\end{aligned}
$$

To achieve higher degree of parallelism, RS units enable multiple internal operations in a single clock cycle. However, the RS unit's operation correctness and efficiency impose the following constraints on performing different operations in a single clock cycle:

- *capture* and *dispatch* cannot apply to the same RS entry in the same clock cycle. If an entry becomes ready for dispatch after capturing the CDB result in a certain clock cycle, it cannot be dispatched in that same clock cycle due to the first-in first-out scheduling policy. Also, capturing the CDB result to a certain instruction then dispatching it will certainly require a longer clock period.
- when *shelve* and *dispatch* target the same RS entry in the same clock cycle, the *shelve* operation is converted to a *bypass* one.
- when *shelve* and *capture* cannot target the same RS entry in the same clock cycle, the *capture* operation is converted to a *forward* one.
- *bypass* cannot be done in the same clock cycle with *shelve* or *dispatch*, as an input could either bypass the RS or be shelved, and the RS unit output could be either an RS entry or a new input that bypasses the RS unit. Consequently, the following item applies.
- *forward*$_B$ cannot be done in the same clock cycle with *shelve* or *dispatch* as *shelve*/*dispatch* and *bypass* cannot take place in the same clock cycle.
- *forward* has to be done first before a related *shelve* or *bypass* is done.

Now, we will formulate the above constraints into a sequencing description that shows the allowable RS unit-internal operations in a single clock cycle (refer to Appendix B for description of the used notation):

$$(D(k); (F_B \gg B))|(\overline{B} \rightarrow (F_S \gg S(l)))|T(m, i)$$
$$where\ l \neq k \neq m \tag{6.24}$$

6.5.3 Generic Execution Unit (EX)

In this subsection, we present the architectural module for a generic EX that provides a framework for all EXs. In the following subsection we will discuss in detail the structure of the LS as it is directly related to JBC execution. In fact, the provided LS design is tailored to JVM requirements. Details about the design of the other EX types are out of the scope of this dissertation.

6.5.3.1 EX Architectural Design Principles

Multiple EXs are included in the processor to allow ILP (Architectural Design Principles 4 and 5). These EXs are of three types: ALUs, load/store units (LSs), and branch units (BRs). ALU types are single-cycle integer units (SIs), multi-cycle integer units (MIs), or floating point units (FPs).

6.5.3.2 Generic EX Architectural Module

Figure 6.24 shows the block diagram for a generic EX. Inputs to this module include the operation (*EXoperation*), source operands (*EXsource*), tag to be associated with the result (*EXdestinationTag*), and an enable signal for the EX (*EXenable*). The EX also gets a signal (*writeEnable*) to indicate when it can write the result on the CDB so as to avoid collision with other EX outputs. Outputs of this module include a signal to the CDB arbiter asking for permission to write on the CDB (*outEnable*), the result (*CDBresult*), its tag (*CDBtag*), and a signal that indicates whether the unit is busy (*EXbusy*).

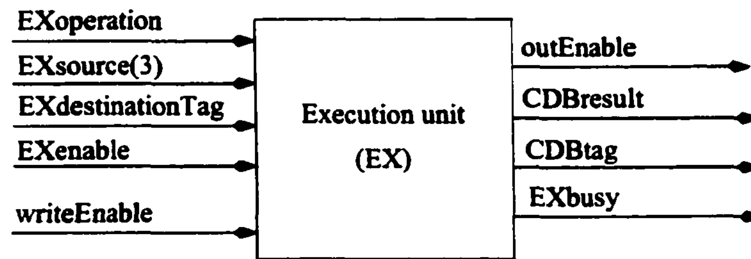


Figure 6.24. *Inputs and outputs of a generic execution unit (EX).*

6.5.4 Load/Store Execution Unit (LS)

The LS handles instructions that manipulate objects (including arrays as they are objects in Java) or need access to the CP. We classify such instructions under the *load* and *store* anchor categories.

The JVM specification requires having objects and the CP residing in the main memory. Maintaining this requirement, JAFARDD therefore needs to access the main memory or the D-cache. Furthermore, execution of such instructions is either very complicated, requires services from the underlying OS, or both. For example, these instructions might require looking up a class dynamically from the list of classes loaded in the system, allocating memory space for an object, dynamically loading a class file, etc [6, 7, 8]. Some of these tasks need to be coordinated with the underlying OS. Thus, the significant complexity and/or the dependency on the underlying OS motivate their implementation in software.

6.5.4.1 LS Architectural Design Principles

Specialized LSs serve as the gateway for the hardware to reach emulating software routines for object-oriented instructions (Architectural Design Principle 8).

6.5.4.2 Bytecode Processing within the LS

Table 6.17 shows the JBCs that are executed by the LS and how they integrate with folding groups producers and consumers.

Table 6.17. *JVM load/store anchor operations internally implemented by the LS.*

Subcategory	Opcodes	Symbol†	Folding group fields	Register assignment‡		
				R_3	R_1	$R_4^{\star}$
LOAD ANCHORS						
load the ith element from an array with the reference r into LV_x	aaload	L	(r:reference:4),(i:int:4)⇒(LV_x:reference:4)	r	i	$LV_x.tag$
	baload	L	(r:reference:4),(i:int:4)⇒(LV_x:int:4)	r	i	$LV_x.tag$
	caload	L	(r:reference:4),(i:int:4)⇒(LV_x:char:4)	r	i	$LV_x.tag$
	daload	L	(r:reference:4),(i:int:4)⇒(LV_x:double:8)	r	i	$LV_x.tag$
	faload	L	(r:reference:4),(i:int:4)⇒(LV_x:float:4)	r	i	$LV_x.tag$
	iaload	L	(r:reference:4),(i:int:4)⇒(LV_x:int:4)	r	i	$LV_x.tag$
	laload	L	(r:reference:4),(i:int:4)⇒(LV_x:long:8)	r	i	$LV_x.tag$
	saload	L	(r:reference:4),(i:int:4)⇒(LV_x:int:4)	r	i	$LV_x.tag$
allocate a new object of type $CP(j)$ or TY_p and size c (in case of arrays), load the new element reference in LV_x	anewarray j:2	L_{CP_j}	(c:int:4)⇒(LV_x:reference:4)	c	j	$LV_x.tag$
	new j:2	L_{CP_j}	⇒(LV_x:reference:4)	—	j	$LV_x.tag$
	newarray p:1	L_{TY_p}	(c:int:4)⇒(LV_x:reference:4)	c	p	$LV_x.tag$
retrieve an item of type $CP(j)$ from object/class with the reference r into LV_x	arraylength	L	(r:reference:4)⇒ (LV_x:int:4)	r	—	$LV_x.tag$
	getfield j:2	L_{CP_j}	(r:reference:4)⇒ ($LV_x : X : \frac{4}{8}$)	r	j	$LV_x.tag$
	getstatic j:2	L_{CP_j}	⇒ ($LV_x : X : \frac{4}{8}$)	—	j	$LV_x.tag$
check that an object of type r is type compatible with the type $CP(j)$.	checkcast j:2	L_{CP_j}	(r:reference:4)⇒ (LV_x:reference:4)	r	j	$LV_x.tag$
	instanceof j:2	L_{CP_j}	(r:reference:4)⇒ (LV_x:int:4)	r	j	$LV_x.tag$
load an item from $CP(j)$ into LV_x	ldc j:1	L_{CP_j}	⇒(LV_x:int/float/reference:4)	—	j	$LV_x.tag$
	ldc_w j:2	L_{CP_j}	⇒(LV_x:int/float/reference:4)	—	j	$LV_x.tag$
	ldc2_w j:2	L_{CP_j}	⇒(LV_x:long/double:8)	—	j	$LV_x.tag$
STORE ANCHORS						
store an element v into the ith element of an array with the reference r	aastore	S	(r:reference:4),(i:int:4),(v:reference:4)	r	i	v
	bastore	S	(r:reference:4),(i:int:4),(T(v,int,4,1):byte:1)	r	i	v
	castore	S	(r:reference:4),(i:int:4),(T(v,int,4,2):char:2)	r	i	v
	dastore	S	(r:reference:4),(i:int:4),(v:double:8)	r	i	v
	fastore	S	(r:reference:4),(i:int:4),(v:float:4)	r	i	v
	iastore	S	(r:reference:4),(i:int:4),(v:int:4)	r	i	v
	lastore	S	(r:reference:4),(i:int:4),(v:long:8)	r	i	v
	sastore	S	(r:reference:4),(i:int:4),(T(v,int,4,2):short:2)	r	i	v
store an item v of type $CP(j)$ into an object/class with the reference r	putfield j:2	S_{CP_j}	(r:reference:4),(v:X:4/8)	r	j	v
	putstatic j:2	S_{CP_j}	(v:X:$\frac{4}{8}$)	—	j	v

- † L_{CP_j}, S_{CP_j} are loads and stores that have CP index j as an argument. L_{TY_p} is a load that has a type p as its argument.
- ‡ R_1 and R_4 are two bytes in length, whereas R_3's length is one byte. When a register is loaded with a byte, the other byte is unused.
- ⋆ $LV_x.tag$ is the *EXdestinationTag* that is attached with LV_x.

6.5.4.3 LS Architectural Module

As shown in Figure 6.25, the LS shares the following signals with a generic EX: *EXoperation* (opcode and its argument), *EXsource*[3], *EXdestinationTag*, *EXenable*, and *EXbusy*. As the LS is not directly responsible for writing the result, it is not connected to the signals: *writeEnable*, *CDBresult*, *outEnable*, and *CDBtag*. Two extra flags are added: *trap* to activate the OS trap mechanism and *done* that tells the unit that the trap routine is completed.

Internally, the LS includes four registers to pass inputs and operation to the trap mech-

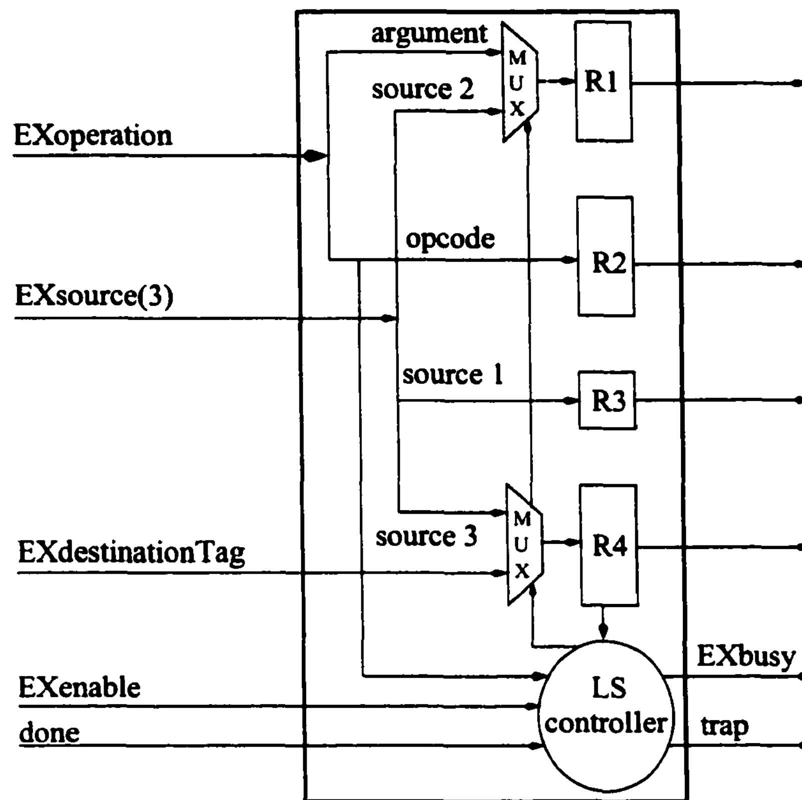


Figure 6.25. *Inputs and outputs to the LS unit.*

anism. Values stored in these registers are set according to the operation to be performed. Figure 6.25, Table 6.17, and Table 6.18 suggest a possible mapping of folding groups inputs, destination tag, and operation onto these registers. As can be seen, R_1 stores the second operand or the opcode argument, R_2 stores the opcode, R_3 stores the first operand, and R_4 stores the third operand and the destination tag. An internal controller manages the register multiplexors and generates the appropriate values for the output flags: *trap* and *EXbusy* based on the *EXoperation*, *done* and *EXenable* signals.

6.5.4.4 LS Operation Steps

To simplify our discussion, we make some assumptions: (1) Nested load/store operations are not supported. That is a new LS operation will not be allowed to start until a previous one completes; (2) LS operations are type-safe. This means that folding group producers and consumer are of types that match the operation to be performed. We assume that the

Table 6.18. *Mapping of folding group inputs and destination tag onto the LS internal registers.*

Folding template†	LS register contents			
	R_2 opcode	R_3 source[1]	R_1 source[2] or argument	$R_4^{\ddagger}$ source[3] or destination tag
$P_{LV_x} \frac{P_{LV_y}}{P_{CN_b}} LC_{LV_m}$	L	LV_x	$\frac{LV_y}{CN_b}$	$LV_m.tag$
$\frac{P_{LV_x}}{P_{CN_a}} \frac{L_{CP_j}}{L_{TY_p}} C_{LV_m}$	$\frac{L_{CP_j}}{L_{TY_p}}$	$\frac{LV_x}{CN_a}$	$\frac{j}{p}$	$LV_m.tag$
$L_{CP_j} C_{LV_m}$	L_{CP_j}	—	j	$LV_m.tag$
$P_{LV_x} LC_{LV_m}$	L	LV_x	—	$LV_m.tag$
$P_{LV_x} \frac{P_{LV_y}}{P_{CN_b}} \frac{P_{LV_z}}{P_{CN_c}} S$	S	LV_x	$\frac{LV_y}{CN_b}$	$\frac{LV_z}{CN_c}$
$P_{LV_x} \frac{P_{LV_z}}{P_{CN_c}} S_{CP_j}$	S_{CP_j}	LV_x	j	$\frac{LV_z}{CN_c}$
$\frac{P_{LV_z}}{P_{CN_c}} S_{CP_j}$	S_{CP_j}	—	j	$\frac{LV_z}{CN_c}$

- † L_{CP_j}, S_{CP_j} are loads and stores that have CP index j as an argument. L_{TY_p} is a load that has a type p as its argument.
- ‡ $LV_x.tag$ is the *EXdestinationTag* that is attached with LV_x.

JBCs run through a software bytecode verifier that ensures the type correctness of the class file and any runtime type mismatch will be handled by a software exception mechanism; **(3)** Emulation routines are ready to handle exceptions that may occur; **(4)** Quick operations are not supported. (Quick processing is out of the scope of this dissertation); **(5)** We ignore the effect of trapping on the pipeline. In fact, trapping might result in pipeline flushing, etc.

We use an instruction emulation trapping approach in executing the load/store instructions. When the hardware encounters any of these instructions, the LS throws a hardware exception that traps to an OS routine, which carries the execution steps. Three options exist for emulation: **(1)** emulate the instruction execution using primitive JBCs. This option mandates extending the JVM to perform some low level operations, e.g., direct main memory access; **(2)** injecting native RISC instructions at the second half of the pipeline; Both of these options use processor instructions (hardwired approach) in emulation; and **(3)** use a firmware approach that executes a microprogram stored into an on-chip ROM. Regardless of the selected approach the trapping mechanism follows the steps below:

1. A setup phase is done first when an operation reaches the LS. Depending on the instruction, the operation operands (provided by the folding group producers), the operation argument (provided in the bytecode stream following the operation), the

operation itself, and/or the destination tag are copied into LS's internal registers.

2. Once the needed data are ready in the corresponding registers, the LS controller generates a signal to trigger the CPU to make an instruction emulation trap. We elected to have all operations trap to the same memory address and leave it to the software to take different actions for different operations [42].
3. Depending on which JBC caused the trap, the exception handler invokes a particular software/firmware routine that emulates the trapped JBC. In performing the desired operation, the invoked subroutine will have full access to the LS internal registers.
4. The invoked subroutine will perform a series of steps to accomplish its task. Table 6.19 shows highlights of the steps needed for each load/store subcategory.
5. When an operation that produces a result completes, the result will be broadcasted on the CDB together with the tag for pending LVs and RSs to capture. (Load instructions produce a result, whereas store ones do not).
6. When the operation completes the result is written back. After half a clock cycle, EX indicates that it is not busy and thus is ready to handle another operation.

6.5.5 Common Data Bus (CDB)

When an EX has its result ready, it broadcasts it on the CDB. However, having multiple EXs raises a possibility for bus contention. Thus, a CDB arbiter is required to resolve this. The arbiter works as follows (Figure 6.26): more than one EX indicate a need to write on the CDB by asserting the *outEnable* signal. The CDB arbiter schedules simultaneous requests in a round robin fashion that lets only one of them write on the bus at anytime. The EXs know their turn by receiving these *writeEnable* signals from the arbiter. At that moment, the enabled EX can go ahead and write the result on the CDB. Concurrently, the arbiter asserts the *CDBenable* output signal to inform all processor modules of the event. Result broadcast takes just one clock cycle after which the EX removes the data from the bus, and then the arbiter enables another unit and so on.

It is worthwhile to observe that not all operations require writing to the CDB. Anchors that require no consumer, i.e., have the anchor notation: $A_{u,0}$, do not write to LVs.

Table 6.19. *Steps needed to perform LS operations.*

Subcategory	Action(s) or Bookkeeping†
LOAD ANCHORS	
load an array element	1 Calculate the effective address 2 Validate the operands-destination compatibility 3 Load the array element from the D-cache 4 Extend the loaded value to 4 bytes‡
allocate a new object	1 Construct an index into the CP of the current class‡ 2 Retrieve the item at the CP index‡ 3 Resolve symbolic reference to get a class type, or get the object type from the opcode arguments 5 Allocate memory for the new object 6 Initialize object fields to their initial values 7 Load a reference to the instance
retrieve an item from an object/class field	1 Construct an index into the CP of the current class‡ 2 Retrieve the item at the CP index‡ 3 Resolve symbolic reference to get a class type 4 Retrieve the object field from the D-cache
check object type	1 Construct an index into the CP of the current class‡ 2 Retrieve the item at the CP index‡ 3 Resolve symbolic reference to get a class type 4 Perform type checking
load an item from CP	1 Construct an index into the CP of the current class‡ 2 Retrieve the item at the CP index‡ 3 Resolve the symbolic reference to get a class type 4 Load the CP element from the D-cache
STORE ANCHORS	
store an element into an array	1 Calculate the effective address 2 Validate the operand-destination compatibility 3 Truncate the value to be stored 4 Store element into the D-cache‡
store an item into an object/class field	1 Construct an index into the CP of the current class‡ 2 Retrieve the item at the CP index‡ 3 Resolve symbolic reference to get a class type 4 Store the object field into the D-cache

- † OS routines involve exception processing. If any operand is invalid, an exception is thrown.
- ‡ Operation is performed only if needed.

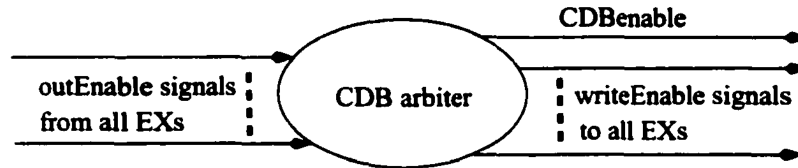


Figure 6.26. *Inputs and outputs to the CDB arbiter.*

6.6 Conclusions

In this chapter, the details of JAFARDD's architecture were presented. JAFARDD adapts the well known Tomasulo algorithm to folding based Java processors. Our architecture uses the OPEX bytecode algorithm for folding JBCs. In addition to its flexibility, this algorithm recognizes nested patterns. These two characteristics enable folding as many patterns as possible, which diminishes the need for a stack. Reservation stations are employed for dynamic scheduling.

On closer inspection for Java processor requirements presented in Chapter 4, and the OPEX bytecode folding and the Tomasulo algorithms, we have identified, and thus incorporated, several techniques for tuning the Tomasulo algorithm to our architecture: (1) folding information generation off-line (2) IFG processing; (3) folding-dependent hazard resolution; (4) BQ management; (5) on-chip LVF; (6) specialized LS units and OS traps.

The features presented in this chapter are specific for Java processing. However, other languages that share some similar characteristics with Java can also make use of those hardware features in improving their performance.

In Subsection 6.5.2, we fine-tuned the reservation station structure to suit the needs of JBC processing. We identified and justified our architecture on the design space of renaming, shelving, and dispatching. The idea of early-tag assignment is equally applicable to any other architecture. We conclude that though our discussion was specific to JBC processing, the presented features can also be used to improve the performance of other architectures.

The work in this chapter has been published in [198, 199, 200, 201].

The next chapter examines JAFARDD from a global perspective. It illustrates JAFARDD's microoperations by a comprehensive example and assesses its performance using a benchmark study.

Chapter 7

A Processing Example and Performance Evaluation

7.1 Introduction

In this chapter, we look at JAFARDD from a global perspective. JAFARDD's microoperations are illustrated by a comprehensive example in Section 7.2. This example focuses on the folding and translation processes. Section 7.3 defines the formula used to evaluate the speedup obtained in JAFARDD. The experimental framework is explained in Section 7.4. Sections 7.5 and 7.6 show and analyze the different folding patterns recognized by JAFARDD. Potential performance enhancement is estimated in Section 7.7. Section 7.8 evaluates the operation of the different modules in the processor. We then look at how different stack features are compensated for in our stackless architecture in Section 7.9. Section 7.10 compares JAFARDD's approach to other similar hardware techniques for speeding up Java. Finally, Section 7.11 draws some conclusions on JAFARDD's performance.

7.2 A Comprehensive Illustrative Example

To show how various modules work together, consider the following Java code segment: (`x = 5; this.f += (x++)/4;`), where x is an integer method variable, f is an integer object field, and *this* is a pointer to the current class.

Figure 7.1 traces the execution of the corresponding JBCs at each pipeline step (a step takes one clock cycle). Here, we assume that JBCs are already queued in the BQ and as such all related folding information is already generated and saved in the FIQ. For clarity of

presentation, each shown entry in the BQ contains one complete JVM instruction. (In a real implementation, each byte occupies a single entry in the BQ.) Both queues grow bottom up. Both the BQ and the FIG get input from the I-cache bus. The folding information is saved in the FIQ as an ordered set (*st,bc,op,os,io,f_operation*), where *st* is the starting location of the folding group in the BQ, *bc* is the group bytecode count, *op* is the opcode position in the BQ, *os* is the opcode size, *io* is the BQ insert offset for duplicated and swapped JBCs, *f_operation* is the desired BQM-internal folding operation.

Dequeued FIQ outputs are used to control the BQ, whereas dequeued JBCs from the BQ are passed to the FT. The FT translates folding group JBCs read from the BQ into RISC-like binaries. The folding group opcode and immediate values are buffered in the buffer *B* for one clock cycle to allow the LVF to supply data/tags needed for the folding group. Information from the FT to the LVF specifies the operation to be done by the latter. Information saved in the buffer and read at the LVF output forms a single RISC instruction to be passed as an input to the RS unit. For example, `idiv 5 4, t3` means integer divide 5 by 4 and save the result into the LV that holds the tag 3 (tagged LVs are prefixed with *t*). For clarity of presentation, we show one unified RS unit and only one general EX that is capable of performing all desired operations. RS fields are: opcode, (value (v_i), tag (t_i)) for the three operands, and a destination tag (*td*). The BQ, LVF, TG, and RS monitor the CDB for broadcasted results and tags.

In Figure 7.1, step (a) shows the processor state just before executing this piece of code. LV_0 is used to hold the value of `this` which is set by the JVM to 20. LV_1 holds the value of x. The code finishes execution in step (p).

The trace includes the 18 JBCs of the 11 JVM instructions in the code segment. On a stack processor that directly executes the JBCs without folding or dynamic scheduling (assuming a seven-stage pipeline like JAFARDD and that the pipeline is ideal to the extent that each instruction takes one clock cycle), this code takes $11 + a\ setup\ time\ of\ (7-1) = 17$ cycles and requires a stack height of 4 words. Thus, the Java instruction per clock cycle (*JPC*) metric for this machine is 0.65. On JAFARDD, three `tload` instructions are added bringing the total JVM instruction count to 14. It takes 14 cycles (including pipeline setup time) to execute this code, resulting in a JPC of 1. Thus, JAFARDD's JPC is 54% better than a stack processor's.

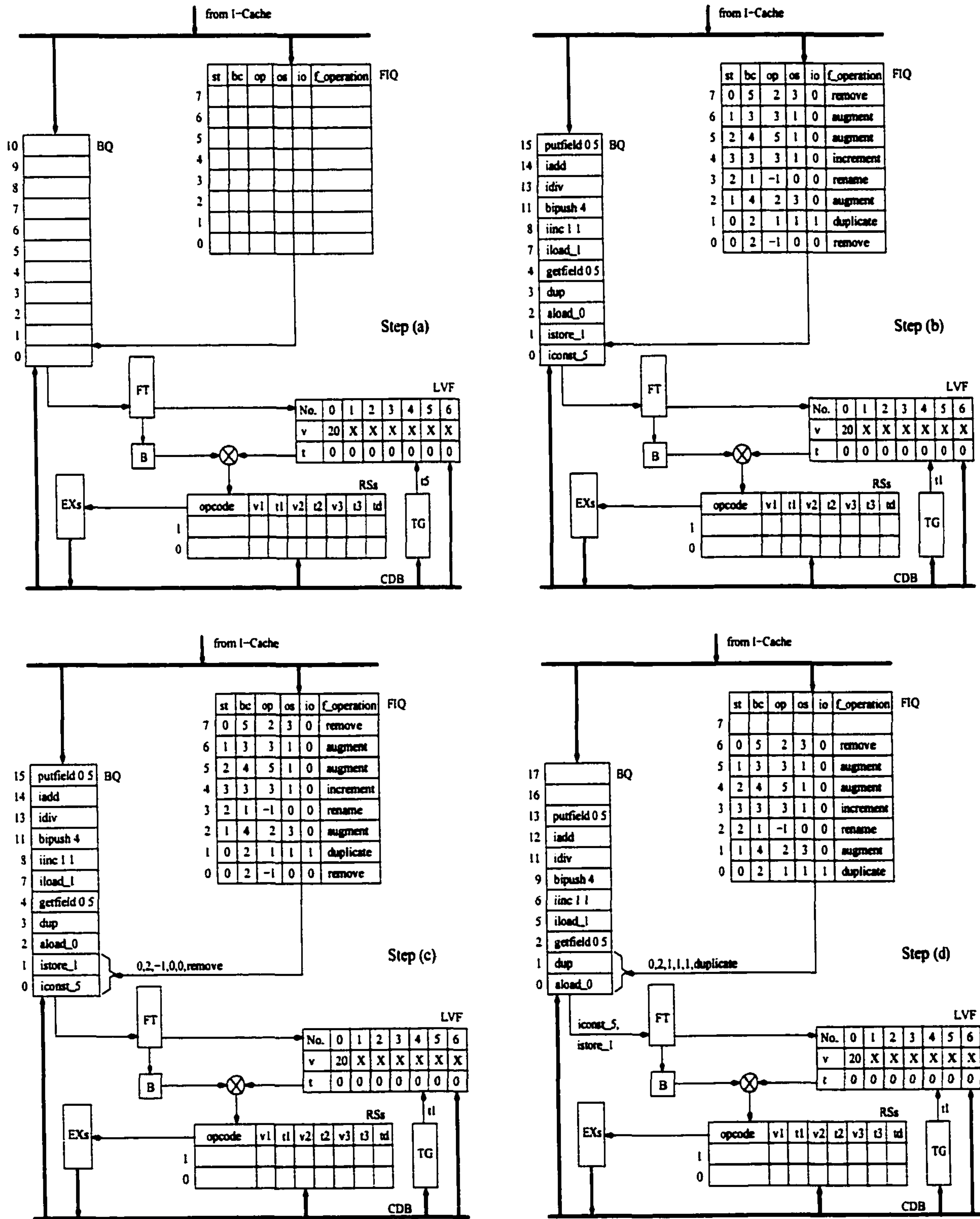


Figure 7.1. *A JBC trace example.*

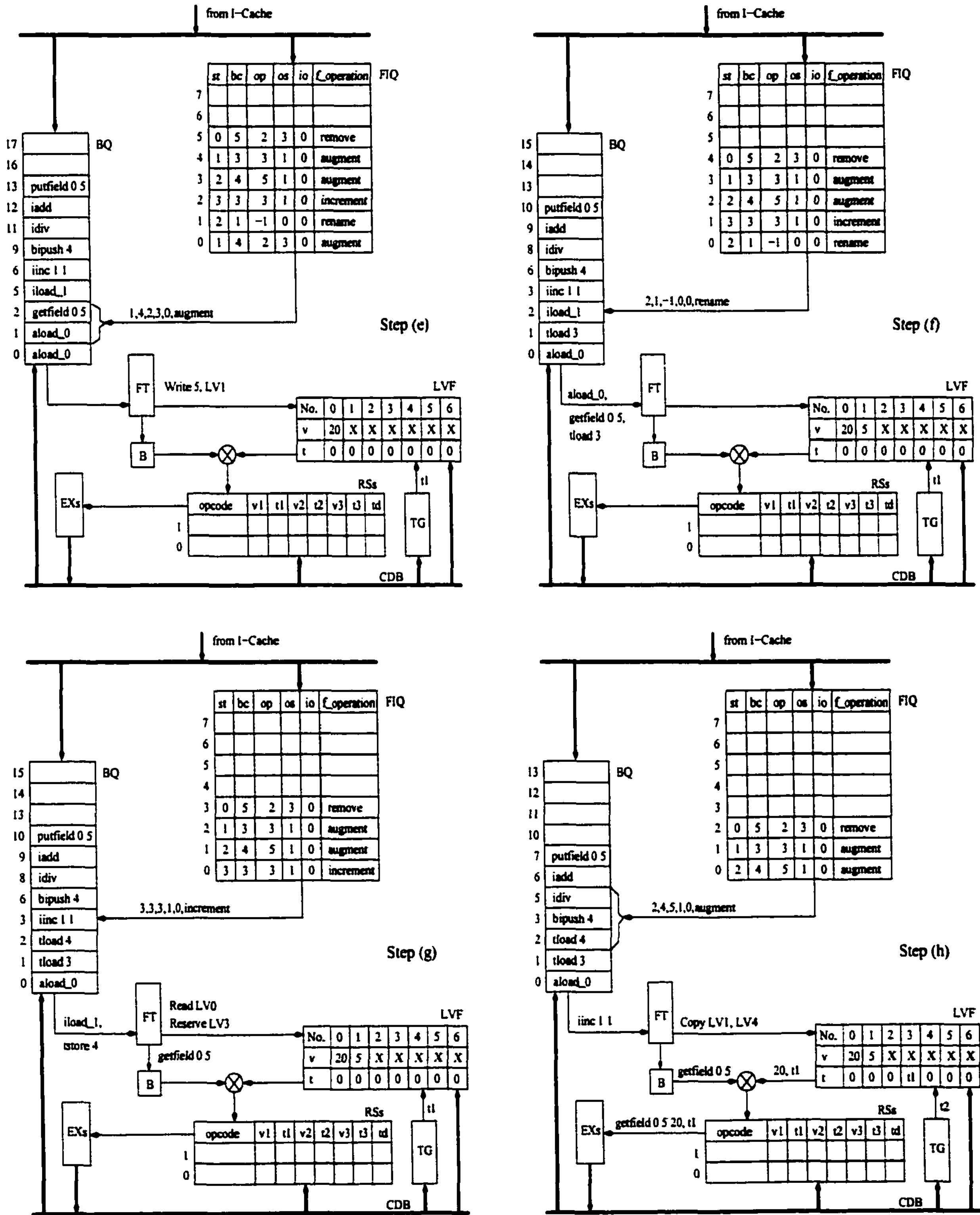


Figure 7.1. *(continued) A JBC trace example.*

Figure 7.1. *(continued) A JBC trace example.*

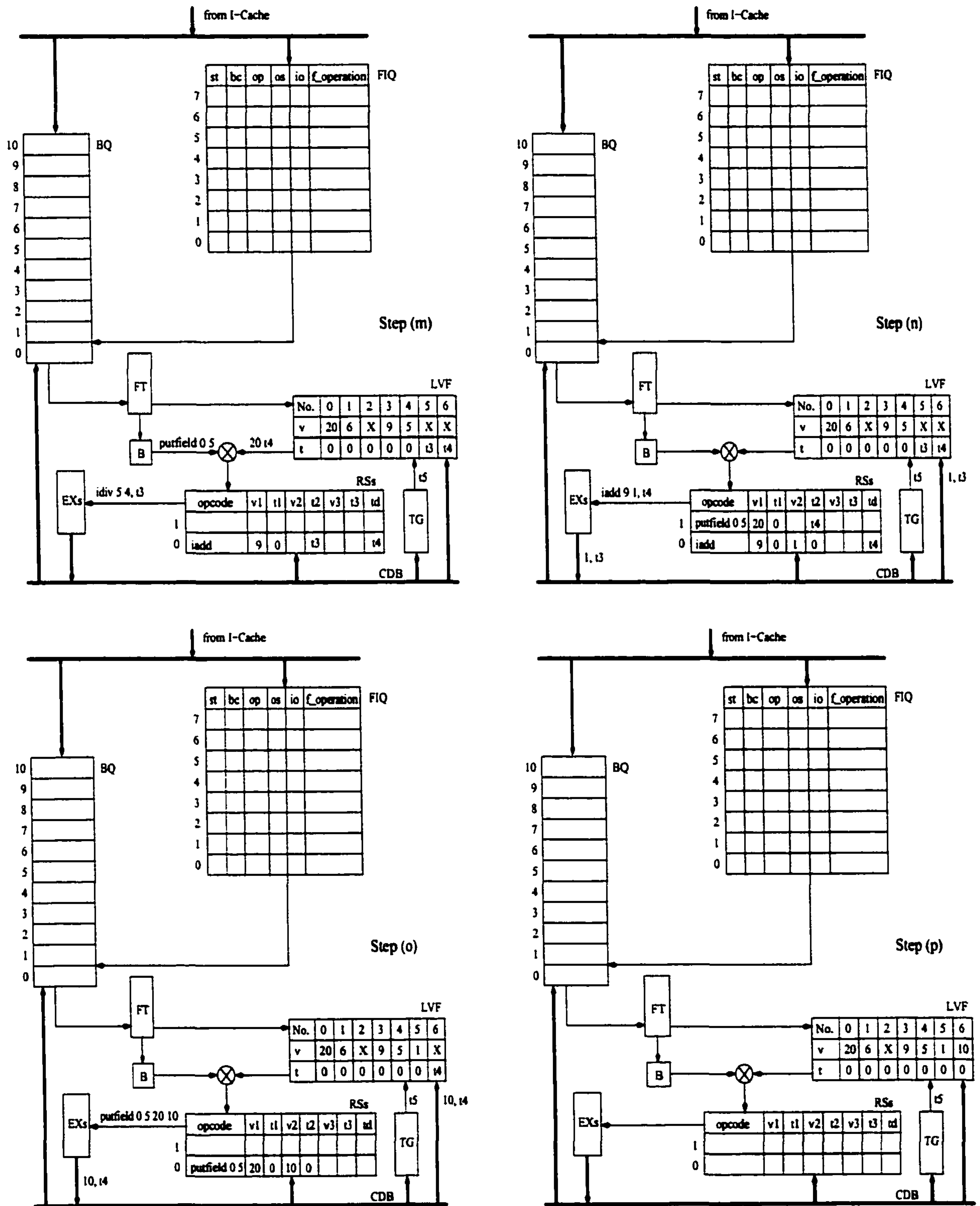


Figure 7.1. *(continued) A JBC trace example.*

7.3 Speedup Formula

Here we introduce the speedup formula used to assess the performance gain associated with the architectural features introduced in JAFARDD.

OPEX bytecode folding, the key feature of JAFARDD, eliminates the CPU clock cycles needed for executing the non-anchor instructions in a folding group and the anchor instructions that were removed by optimization. With this saving, JAFARDD achieves an overall program execution speedup.

The most indicative performance evaluation factor, the program execution time, is given by [42, 69]:

$$CPU\ execution\ time = CPI * C * T \tag{7.1}$$

where CPI is the average clock cycles per JVM instruction, C is the total dynamic instruction count, and T is the clock period. We can formulate the speedup as:

$$Speedup = \frac{CPU\ execution\ time_{\text{Before folding}}}{CPU\ execution\ time_{\text{After folding}}} \tag{7.2}$$

One way to model the effect of folding is to set individual CPI's (CPI_i) for eliminated stack instructions to zero. This has the advantage of keeping the instruction count (C) in Equation 7.1 unaffected by folding. Furthermore, if we assume the usage of the same clock period (T) (for the purpose of comparison), the speedup formula can be reduced to:

$$Speedup = \frac{CPI_{\text{Before folding}}}{CPI_{\text{After folding}}}. \tag{7.3}$$

Thus, we can adopt the average CPI alone as the performance measure. To focus on issues directly related to the OPEX bytecode folding, we assume that JAFARDD is cache-miss free and has an ideal instruction pipeline that could resolve control and data hazards without stalling. The CPI can then be calculated as follows: **(1)** divide the instruction set into several categories (see Table 5.1). All instructions in a category, say i, have the same individual CPI, say CPI_i; **(2)** measure the relative frequency, f_i, of executed instructions for each category; and **(3)** program average CPI can be given by the formula [42, 69]:

$$CPI = \sum_{i=1}^{n} CPI_i * f_i. \tag{7.4}$$

We define the following subfractions (relative to the total executed instructions): a_i is the portion of the anchor instructions that are folded; b_i is the portion of the unfolded complex instructions; c_i is the portion of the unfolded non-anchor instructions; d_i is the portion of the non-anchor instructions that are folded; e_i is the portion of the anchor instructions eliminated through optimization. After folding, two of these factors, d_i and e_i, are removed. For a certain instruction class, the prefolding f_i ($f_{i,pre}$) can be defined in terms of these subfractions as follows:

$$f_{i,pre} = a_i + b_i + c_i + d_i + e_i \tag{7.5}$$

whereas, the postfolding f_i ($f_{i,post}$) can be defined as:

$$f_{i,post} = a_i + b_i + c_i. \tag{7.6}$$

Using these introduced factors, the speedup can be calculated as:

$$Speedup = 1 + \frac{\sum_{i=1}^{n} CPI_i * (d_i + e_i)}{\sum_{i=1}^{n} CPI_i * (a_i + b_i + c_i)}. \tag{7.7}$$

This formula illustrates how much performance improvement can be achieved in JAFARDD through the use of the OPEX bytecode folding algorithm. From the formula, we can easily see that performance could be enhanced by increasing the portion of the non-anchor instructions that are folded and the anchor instructions that are eliminated through optimization.

The above formula assumes a practical pipeline where different instructions have different latencies. Long instruction latencies might lead to pipeline stalls. Now, let us consider a processor model that is more optimized in a way that all instructions take one clock cycle. (This could be achieved by employing proper pipelining and latency hiding techniques, e.g., having multiple pipelined functional units with an efficient OOO scheduler. Tomasulo's algorithm approach employed by JAFARDD is a good example of this.) In this case, the optimum speedup ($Speedup_{\mathrm{op}}$) could be achieved by setting all CPI_i 's to unity:

$$Speedup_{\mathrm{op}} = 1 + \frac{\sum_{i=1}^{n} (d_i + e_i)}{\sum_{i=1}^{n} (a_i + b_i + c_i)}. \tag{7.8}$$

This speedup represents the upper bound on performance improvement that could be obtained in JAFARDD. Further speedup could be achieved by multiple issuing of JBC folding groups.

7.4 Experimental Framework

To evaluate the performance of JAFARDD, a trace driven simulation was used. The results of evaluating JAFARDD using SPECjvm98 benchmark suite are summarized and explained in the following subsections.

7.4.1 Study Platform

The platform used to perform this study was a Sun Ultra 5_10 workstation with an UltraSPARC-IIi 270 Mhz processor and 128 MB of RAM. The OS was Solaris 2.6. The Solaris JDK interpreter version 1.2.2_05 with native threads from Sun Microsystems was used to generate a dynamic trace.

7.4.2 Trace Processing

This trace was processed using a Perl script to calculate the fractions and subfractions (used in Equations 7.5 and 7.6). These numbers were used to produce the results presented below.

7.4.3 Benchmarking

In benchmarking JAFARDD, we wanted to use a standard suite to fairly compare our results with existing folding based architectures. Currently, almost all Java-related published research uses SPECjvm98. Since it was released by SPEC in late 1998, SPECjvm98 has become the de facto benchmark for Java [223]. Before that, researchers used non-standard benchmarks. Sun Microsystems has used javac and raytracer [92, 95]. Chang *et al.* have reported the performance of their, as well as Sun Microsystems', folding algorithms using a combination of benchmarks (e.g., javac and CaffineMark) [203, 204, 205, 206, 207, 208]. For the purpose of validating our results and their consistency with others, we observe that javac and raytracer are included in SPECjvm98 [223]. In addition, in Chapter 2 we have experimented with CaffineMark and obtained results consistent to those produced by SPECjvm98 [86, 87, 88]. Therefore, we are confident that the comparisons between our results and others' findings are valid, consistent, and meaningful.

As shown in Table 7.1, SPECjvm98 includes a set of real-program benchmarks intended to evaluate the performance and efficiency of the JVM client platform. The SPECjvm98

suite tests arithmetic (integer and floating point) and logical instructions, library calls, and I/O as well as other common features. However, it does not directly experiment with the GUI, graphics, or networking components. Each benchmark can run with three different input sizes (classes) known as $s1$, $s10$, and $s100$ [223]. For the purpose of the experiment reported here, we restricted the results to the size $s10$.

Table 7.1. *SPECjvm98 Java benchmark suite summary.*

Program	Description	Instruction count
jess	Java expert system shell based on NASA's CLIPS expert system	75×10^6
jack	A Java parser generator with lexical analysis based on the Purdue Compiler Constructor Tool Set. This is an early version of what is now called JavaCC	207×10^6
check	Simple program to test various JVM features	2×10^6
mpegaudio	Decoder to decompress audio files that conform to the ISO MPEG Layer-3 audio specification	1099×10^6
mtrt	Dual-threaded raytracer algorithm	126×10^6
compress	Modified Lempel-Ziv method (LZW) to compress/decompress large files	862×10^6
javac	The JDK 1.0.2 Java compiler	43×10^6
db	Performs multiple database functions on a memory resident database	63×10^6
total	The entire SPECjvm98 suite	2476×10^6

7.5 Generated Folding Patterns

JAFARDD recognizes a set of folding patterns with different folding depths. Figure 7.2 divides the executed instructions into groups with respect to being anchor or not. From the figure we can see that the instructions executed the most are the producers followed by the non-load/non-store anchors. This is consistent over all benchmarks. The high percentage of existing producers indicates a good potential for speedup by folding.

Figure 7.3 shows the percentages of occurrence of different folding cases (see Table 5.4). Overall, *Case 2*, which deals with tagging, is executed the most in the majority of the benchmarks. Thus, it can be speculated that runtime performance will be improved by adopting the tagging technique. The second most executed case is *Case 4*, which deals with anchors that do not need consumers (e.g., branches). These results are reflected on the

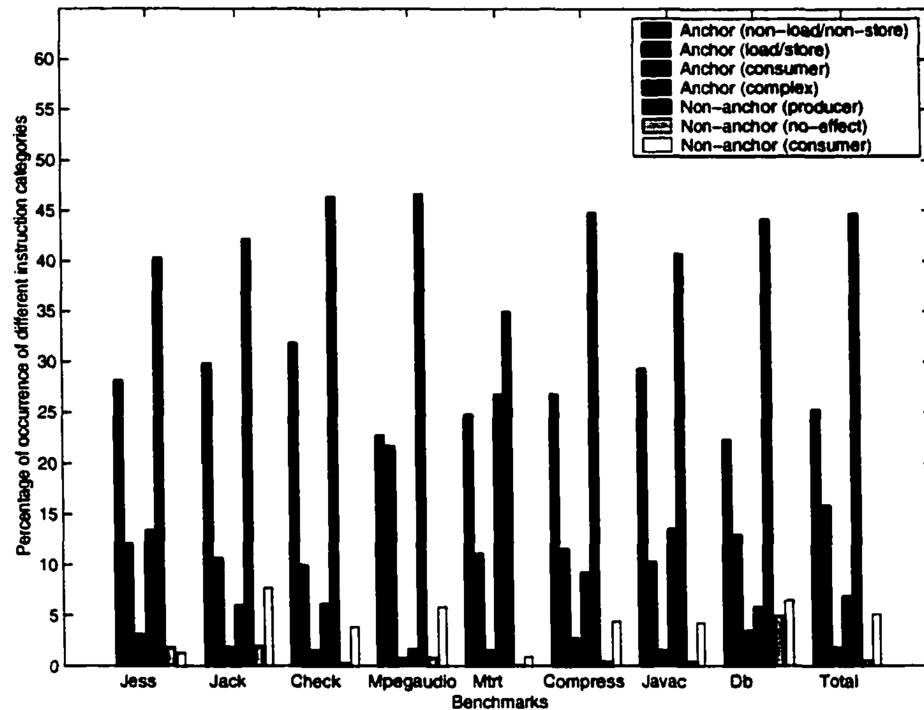


Figure 7.2. *Percentages of occurrence of different instruction categories.*

design of the FT that translates folding groups into RISC-style instructions.

Figure 7.4 further looks into the patterns that are folded the most and their percentages of usage for each benchmark. The variety of folded patterns that are found in the benchmarks under study indicates the generality of the architecture. From the figure, we see that the most common folding pattern is the binary operator in an IFG (i.e., PPOT). This shows the improvement gained by tagging. Also of a noticeable value, especially with tagging, are unary and binary load anchors in IFGs (i.e., PPLT and PLT). The significant percentage of folded load anchor instructions shows the importance of having a LS unit in JAFARDD. Store anchor instructions are not as popular as load anchors, though. Branches are executed often across all benchmarks, which means that a capability of folding them with necessary producers will significantly enhance performance. Destroyer and duplicator anchors are the least executed and thus folded. Also, anchors that do not need any producers are executed rarely. In general, we observe that binary anchors are executed the most. Looking at all benchmarks, we see a consistent behavior (except in Jess and Jack).

Figure 7.5 categorizes and counts folding groups by the number of producers and con-

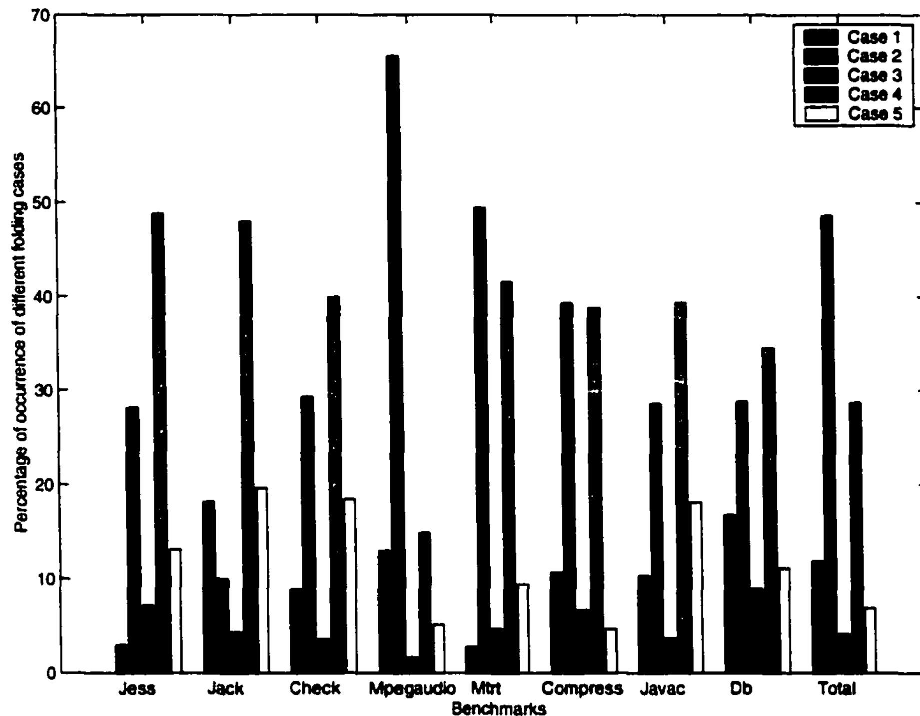


Figure 7.3. *Percentages of occurrence of different folding cases recognized by the folding information generation (FIG) unit.*

sumers. We can see from the figure that the binary anchors, which need a consumer, i.e., $u = 2,\ v = 1$, (whether actually there exists a consumer following the anchor or not) are executed the most. We can also observe that most of the folded patterns involve binary anchor instructions. This result also reflects on the design of the FT.

7.6 Analysis of Folding Patterns

Preparing folding patterns to be issued concurrently and handling nested patterns are the main strengths of the OPEX bytecode folding algorithm. In this section, we see how efficient JAFARDD performs these tasks.

Figure 7.6 shows the percentages of tagged folding patterns relative to all folding patterns recognized by JAFARDD for each benchmark and in total. Percentages within each benchmark range between 10% to 66% with an absolute average around 50%. This shows that plenty of folding patterns are issued incomplete with a tagged consumer and more

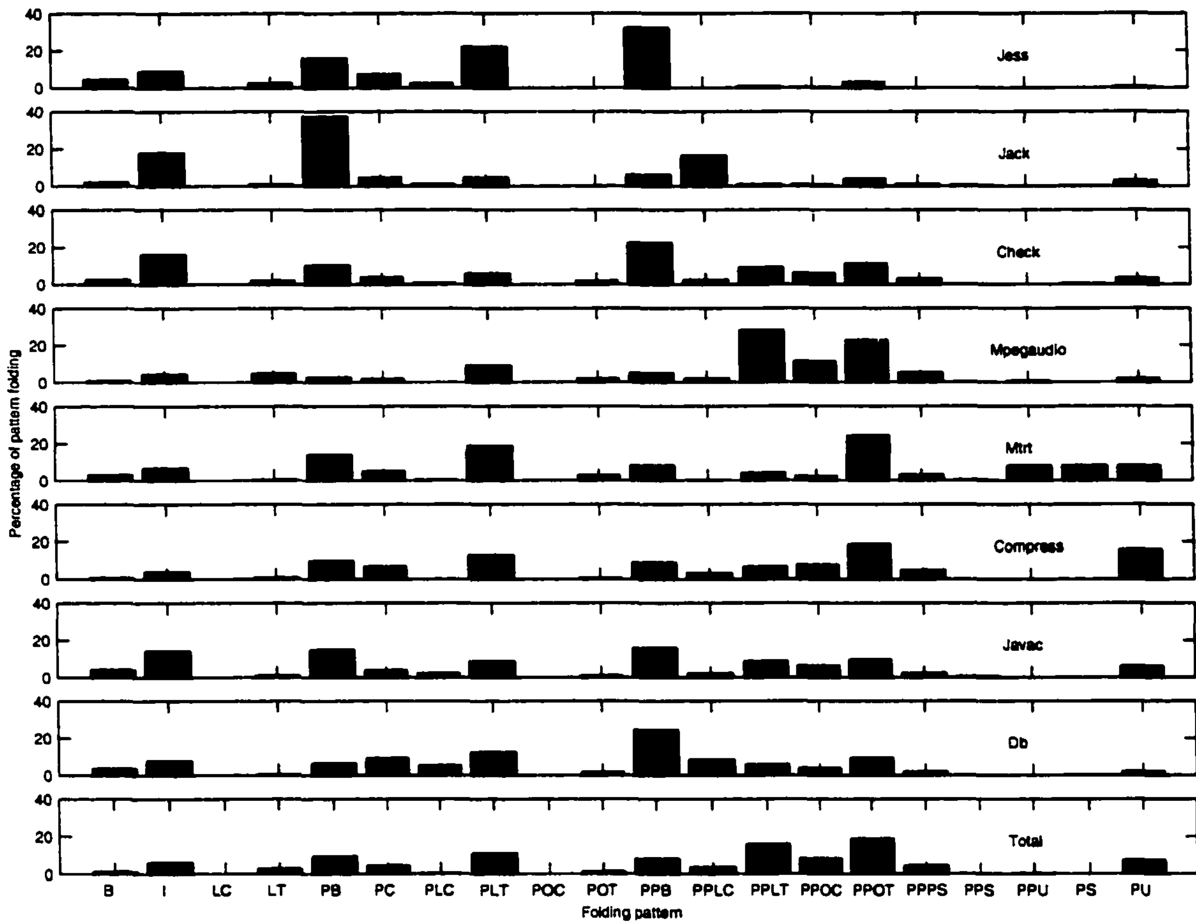


Figure 7.4. *Percentages of patterns that are recognized and folded by JAFARDD.* Refer to Table 5.1 for the meaning of the symbols in different patterns. Note that in calculating the total percentages, absolute instruction counts were used.

patterns take their producers from tagged LVs. The main advantage of tagging is that it decouples dependent folding groups which permits concurrent and/or OOO execution. Without the process of tagging, these considerable percentages of folding groups must wait for the completion of the value-producing instruction and must be issued in order.

Figure 7.7 shows the percentages of nested patterns recognized in JAFARDD for each benchmark and in total. Percentages range between 23.36% to 42.17% with an average of 36.30%. This shows how the OPEX bytecode folding algorithm facilitates folding of nested patterns. Other folding techniques fail to recognize outer nested patterns therefore issue them unfolded, reducing the overall efficiency.

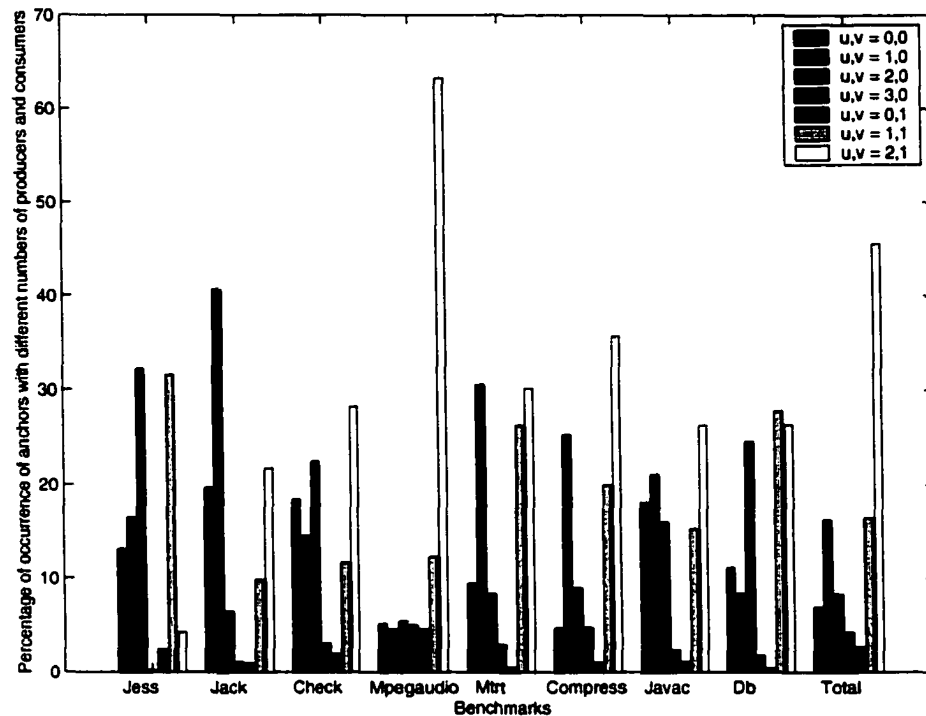


Figure 7.5. *Percentages of occurrence of anchors with different numbers of producers (u) and consumers (v).*

We also measured the percentages of eliminated patterns by optimization for each benchmark and in total. In general, the optimization patterns do not seem to occur significantly in the studied benchmarks because of the low percentages of swapper, duplicator, and destroyer anchors, which can be optimized in the produced JVM trace. Although JAFARDD recognizes these patterns, other hardware implementations of the OPEX bytecode folding algorithm may choose to ignore these patterns to save realization complexity.

7.7 Performance Enhancements

Figure 7.8 shows the percentages of eliminated instructions (i.e., producers and consumers folded in JAFARDD) with respect to all stack instructions (stack instructions include all producers and consumers) and with respect to all instructions in the benchmarks under study. For all benchmarks, the percentages of eliminated stack instructions are over 93%. Chang *et al.* report that Sun Microsystems' folding algorithm can fold up to 60% of all stack

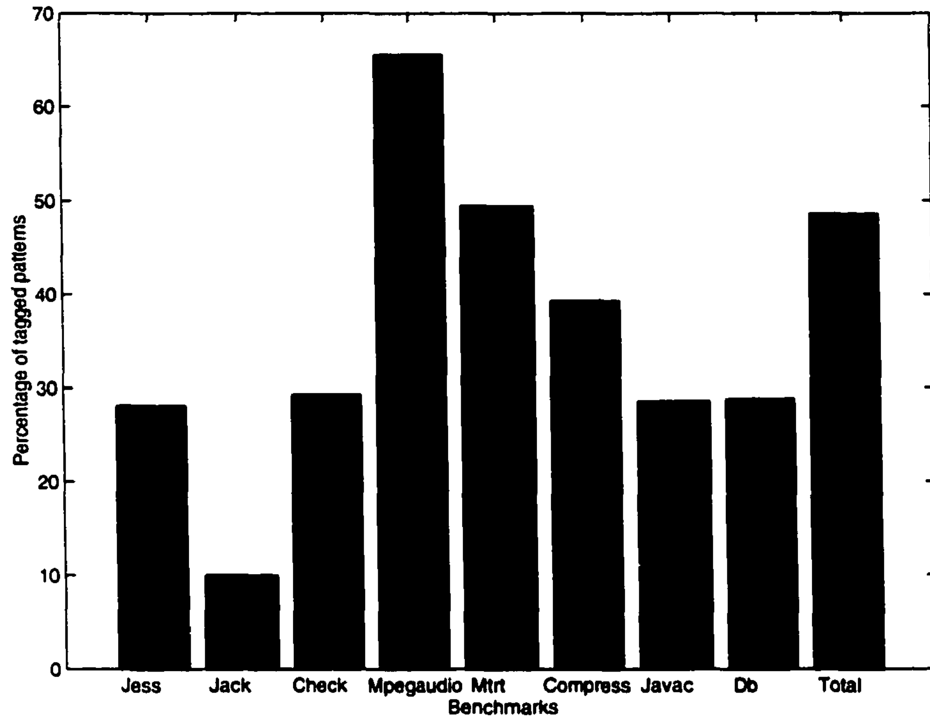


Figure 7.6. *Percentages of tagged patterns among all folded patterns.*

instructions [22, 32, 92, 95, 154, 204]. Chang *et al.*'s 4–foldable strategy can eliminate up to 84% of all stack instructions.

Figure 7.9 shows the actual and optimum (with unity CPI_i for all categories) speedup for JAFARDD. We can see that JAFARDD achieves an actual speedup between 1.10 and 1.37 and an optimum speedup between 1.56 and 2.25. Chang *et al.* report that their 2–, 3–, and 4–foldable strategies result in a maximum actual program speedup of 1.22, 1.32, and 1.34, respectively, as compared to a stack machine without folding [203].

So, in addition to previously mentioned advantages, JAFARDD performance surpasses other techniques in percentages of eliminated instructions and speedup.

7.8 Processor Modules at Work

Figure 7.10 shows the percentages of occurrence of different BQM-internal operations (Tables 6.5 and 6.2). Overall, *Remove* and *Augment* are executed the most. These operations produce folding groups that are translated naturally to typical RISC instructions. Therefore,

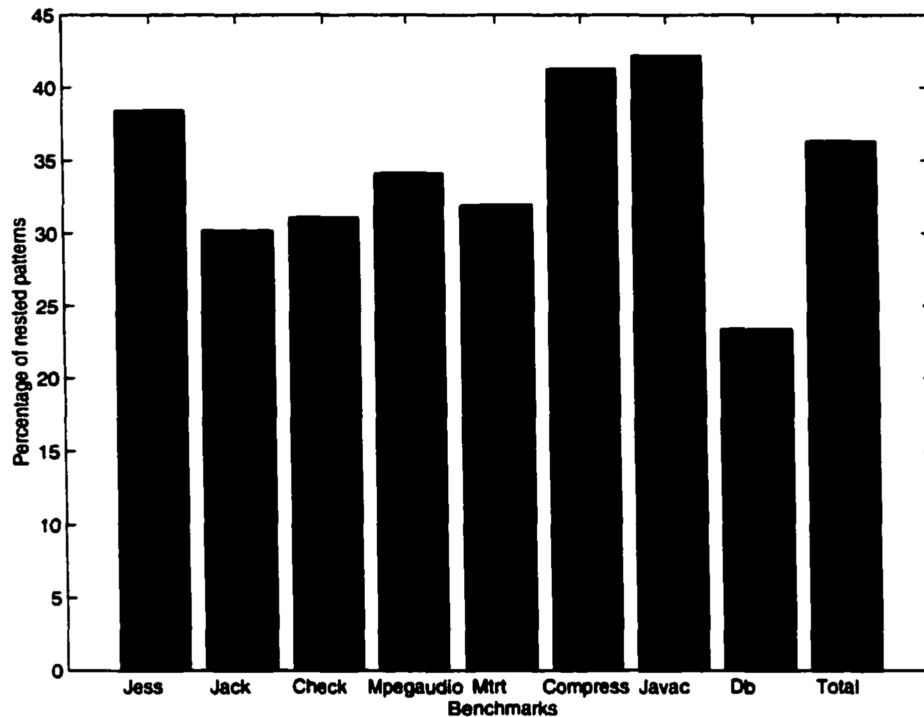


Figure 7.7. *Percentages of nested patterns among all folded patterns.*

these introduced operations mapped efficiently to a RISC processing core.

Figure 7.11 examines the frequency of different patterns translated by the FT (Table 6.9). The most common folding pattern is the binary operator that needs a consumer, perfectly matching the RISC-style.

Figure 7.12 shows the percentages of occurrence of the LVF-internal operations (Table 6.12). Overall, *Capture* and *Reserve* are executed the most. This proves the strong link between the Java-dependent portion of the pipeline and the reservation station structure.

The frequencies of processing different patterns at the LS are shown in Figure 7.13. Load anchors with producers are the most common folding patterns, and these map well onto RISC-style instructions.

EX usage is benchmarked in Figure 7.14. Operations that are terminated at the BQM or LVF have a low percentage of occurrence, and most folding groups require access to EXs. This figure suggests that each type of EX is used significantly.

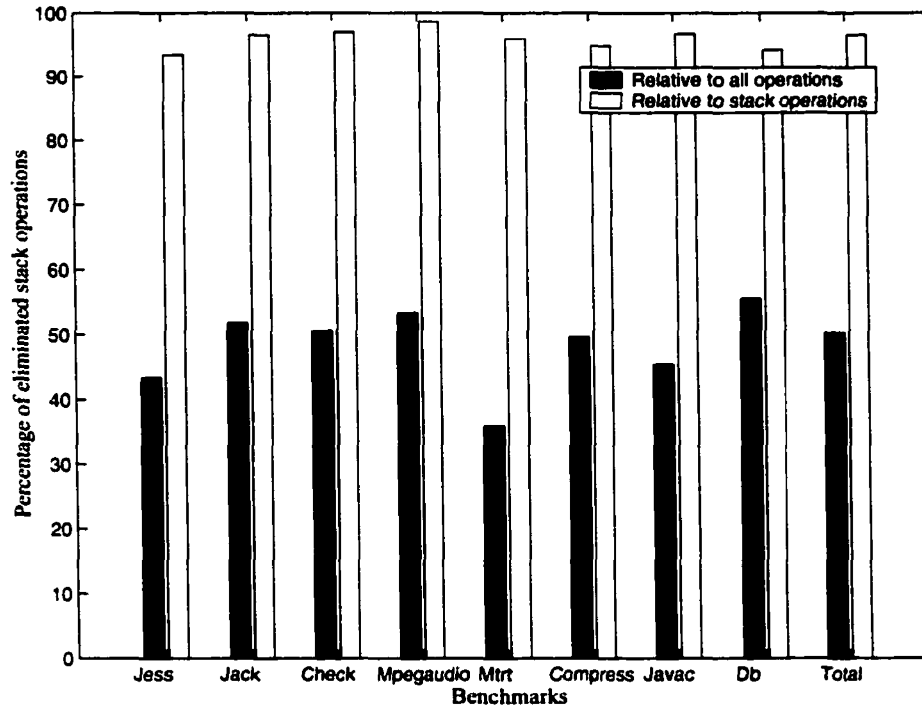


Figure 7.8. *Percentages of eliminated instructions relative to all instructions and relative to stack instructions (producers and non-anchor consumers) only.*

7.9 Global Picture

JAFARDD executes the JVM stack code without physically having a physical stack. This means that the functionality of the stack has to be compensated for and/or handled by other modules. Both the OPEX bytecode folding and Tomasulo's algorithms orchestrate this process. In this section we look at the different stack-related functionalities that any hardware supporting Java is required to provide and/or compensate for. We then look at how JAFARDD fulfills these requirements. In other words, we will show here how the missing stack features are distributed over JAFARDD's modules.

Table 7.2 shows for each JVM instruction category, the JVM basic hardware requirements, e.g., stack operation, and if LVs, EXs, and D-cache are needed to be accessed or not. The table also shows the various modules in JAFARDD that share in processing each of these categories. From the table, it is clear that the JAFARDD architecture compensates for all stack-missing operations and surpasses the JVM basic requirements.

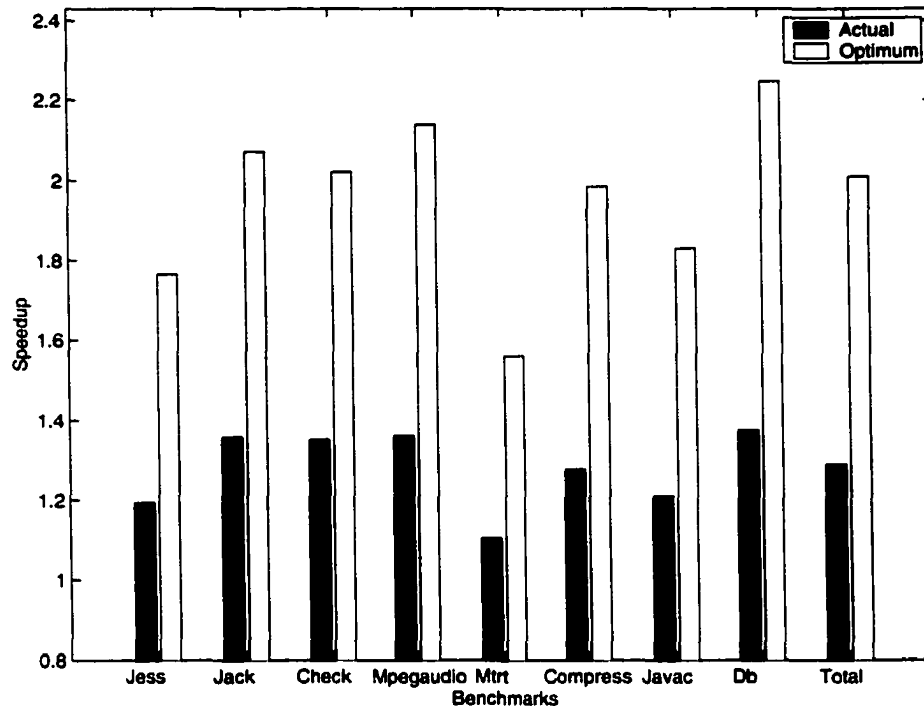


Figure 7.9. *Speedup of folding.*

Actual denotes the speedup obtained by using CPI_i's equal to or greater than unity, whereas *optimum* refers to the optimum case where all CPI_i's are set to unity.

7.10 Alternative Architectures

Besides JAFARDD's approach, there exists a number of hardware approaches that support Java. In this section, the hardware translation approach is compared to two other commonly used Java hardware approaches: direct stack execution of JBC on a stack machine and hardware interpretation to native RISC binaries.

Sun Microsystems, Patriot Scientific corporations, and others provide processors that directly realize the JVM in hardware [92, 95, 99]. Radhakrishnan *et al.* proposed using hardware support for JBC interpretation [126]. Their architecture performs the translation of JBCs to native code in hardware, thus eliminating much of the overhead of software translation. Translation is done using microcode ROM, which contains the translated code for each supported JBC.

We selected these two approaches specifically as JAFARDD could have advantages

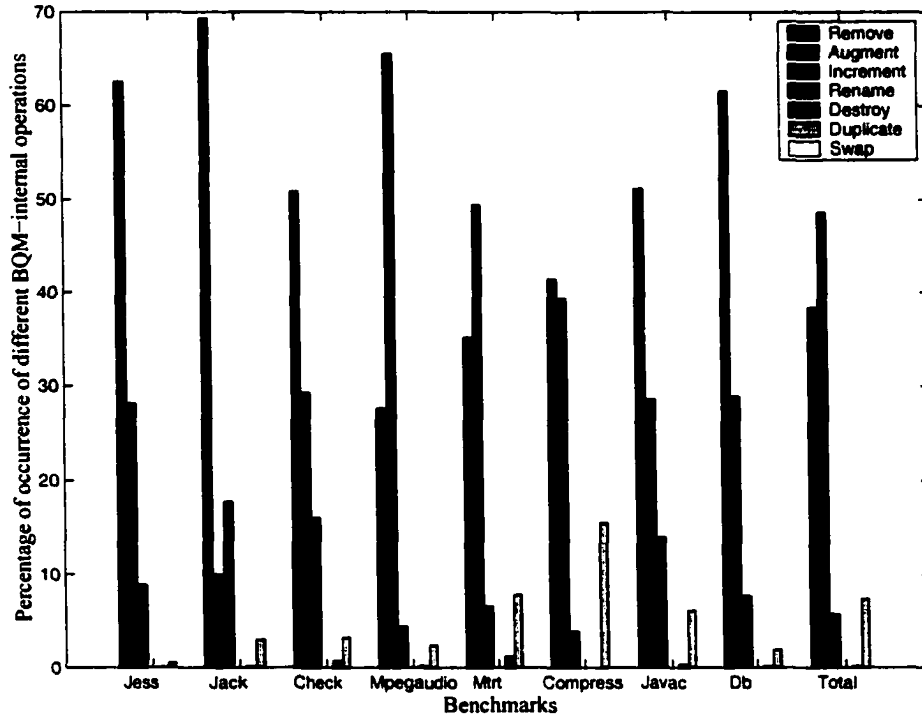


Figure 7.10. *Percentages of occurrence of different folding operations performed by the bytecode queue manager (BQM).*

from each approach while avoiding some of their drawbacks. JAFARDD folds JBC into groups as proposed in the direct execution approach, but does not execute the JBCs natively. On the other hand, to translate JBCs to a native RISC format is similar to the hardware interpretation approach in having a RISC core as the main execution core. However, JAFARDD folds the JBCs before translating them and does not employ any form of microcode lookup.

Table 7.3 lists key features and options in supporting Java in hardware and compares the three approaches.

Some of these approaches enhance Java performance by eliminating some of the stack virtual data dependency (first approach) or permitting efficient RISC code execution (second approach). However, both techniques have their own limitations. The direct stack execution approach relies on a Java-specific core thus restricting the generality of the processor. On the other hand, although Radhakrishnan's proposal has some general advantages, the microcoded control is tightly coupled to the JVM specification. It also involves a microcode ROM lookup of individual JBCs, which may limit speedup gains.

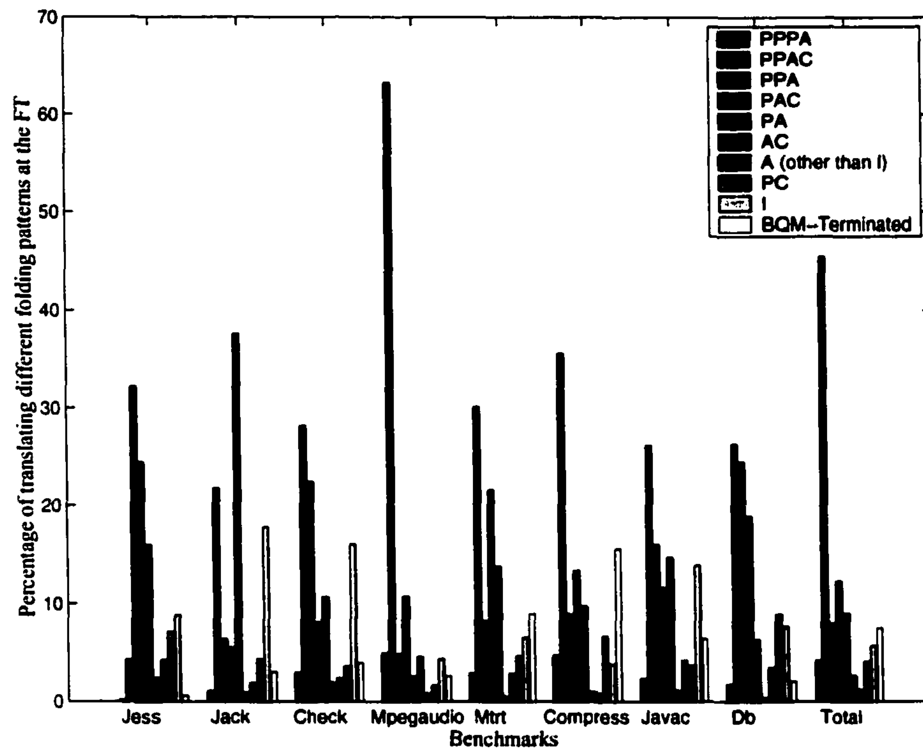


Figure 7.11. *Percentages of occurrence of different folding patterns at the output of the folding translator unit (FT).*

Considering the generality in executing different languages and the independency of the native instructions on the JVM specification as metrics of flexibility, we can conclude that JAFARDD is more flexible than the other two approaches.

7.11 Conclusions

In this chapter, we illustrated the operations of JAFARDD by a detailed example and evaluated its performance via benchmarking. We also presented a global view of the architecture and showed how it compensates for the a stack. Comparison with other relevant proposals shows the flexibility of JAFARDD's architecture.

We have developed a VHDL model of JAFARDD. We use it as a testbed to experiment with the techniques introduced in this dissertation. Results show that the pipeline modules interface and cooperate as expected. Traces of the tags show that the integrated reservation

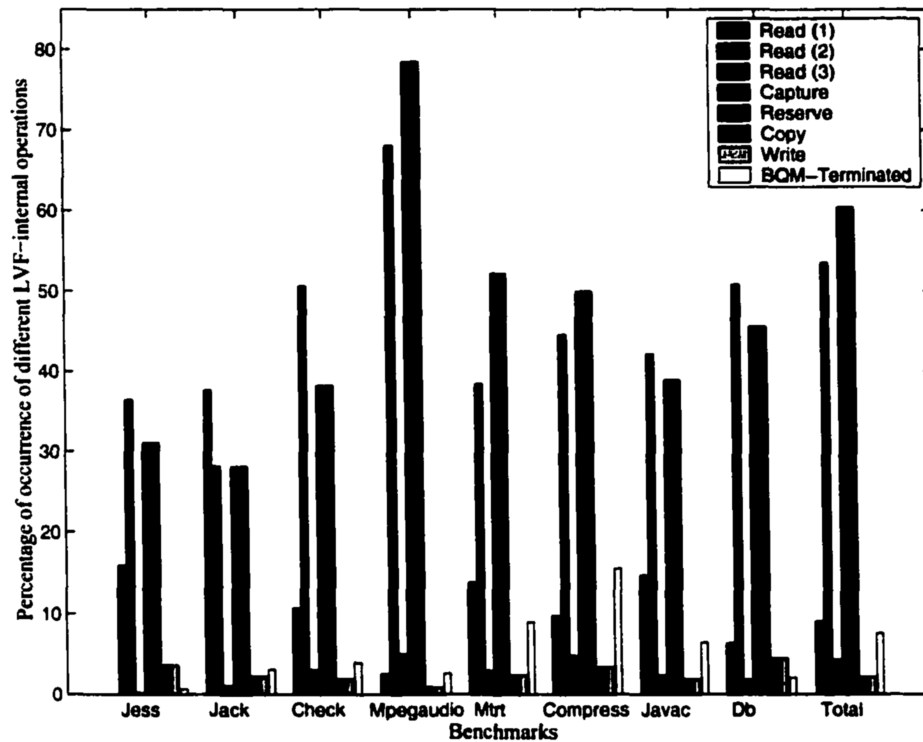


Figure 7.12. *Percentages of occurrence of different operations performed by the local variable file (LVF).*

stations and the OPEX bytecode folding algorithm is a promising approach. Moreover, benchmarking the operations of individual modules strongly manifests the effectiveness of the introduced techniques and the relevance of the incorporated internal operations.

The benchmark evaluation shows that the introduced techniques in JAFARDD are useful in speeding up Java execution. The OPEX bytecode folding algorithm speeds up JBC execution by an average of about 1.29. Furthermore, it eliminates an average of about 97% of the stack instructions and an average of about 50% of the overall executed instructions. Benchmark traces show that an average of about 48% of the patterns are tagged and an average of about 36% of the total patterns are nested, which illustrates the performance gain that could be achieved in JAFARDD.

The work in this chapter has been published in [65, 198, 199, 200, 201, 210, 211, 212, 213].

The next chapter concludes this dissertation and presents future research directions.

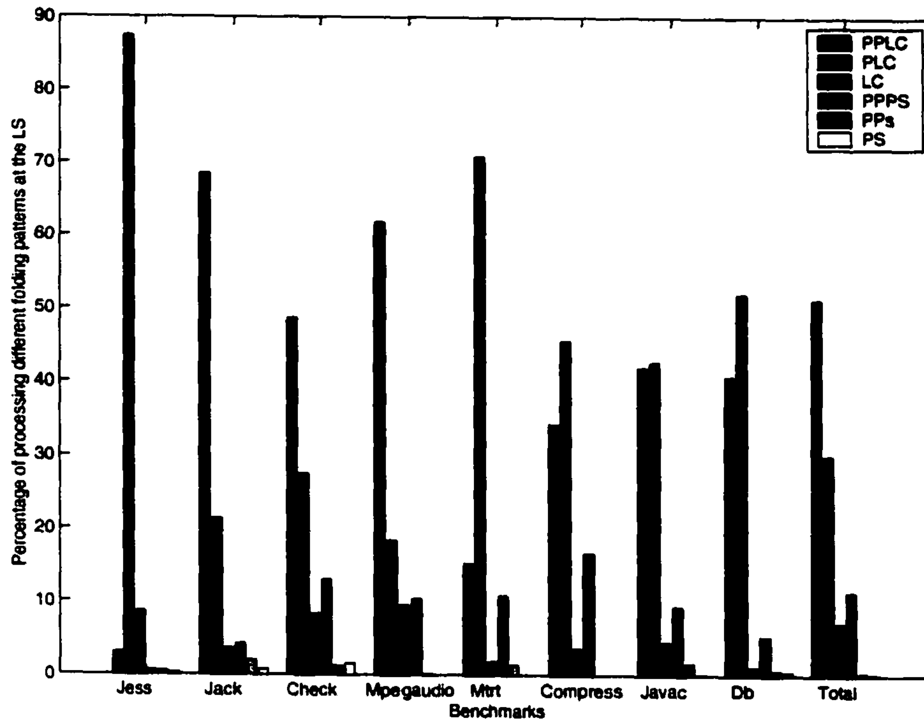


Figure 7.13. *Percentages of occurrence of different folding patterns processed by the load/store unit (LS).*

Table 7.2. *Associating JVM instruction categories with their basic requirements and mapping them to relevant modules in JAFARDD.*

Category†	JVM basic requirement				JAFARDD's modules								
					Folding translation			RISC core					
	Stack‡	LV‡	EX	D-cache⋆	FIG	BQM	FT	LVF	TG	RS	EX	D-cache‡	CDB
no-effect	↔				✓		✓						
producer	↓	✓		✓	✓	✓	✓	✓					
consumer	↑	✓		✓	✓	✓	✓	✓	✓				✓
operator	↕		ALU		✓					✓	ALU		
independent	↔	✓	ALU	✓	✓		✓	✓	✓	✓	ALU		✓
destroyer	↑				✓	✓							
duplicator	⇓				✓	✓							
swapper	⇌				✓	✓							
load	↕			✓	✓					✓	LS	✓	
store	↑			✓	✓					✓	LS	✓	
branch	↕	✓	ALU	✓	✓					✓	BR		

- †All instructions require access to the I-cache and the BF.
- ‡ Effect on stack: ↓, ⇓, ↑, ↕, ↔, ⇌ mean push, duplicate, pop, push and pop, no effect, and swap, respectively.
- ⋆ D-cache holds object memory and CP. In JVM specification, the LV area resides in the D-cache. In JAFARDD it resides in the LVF.

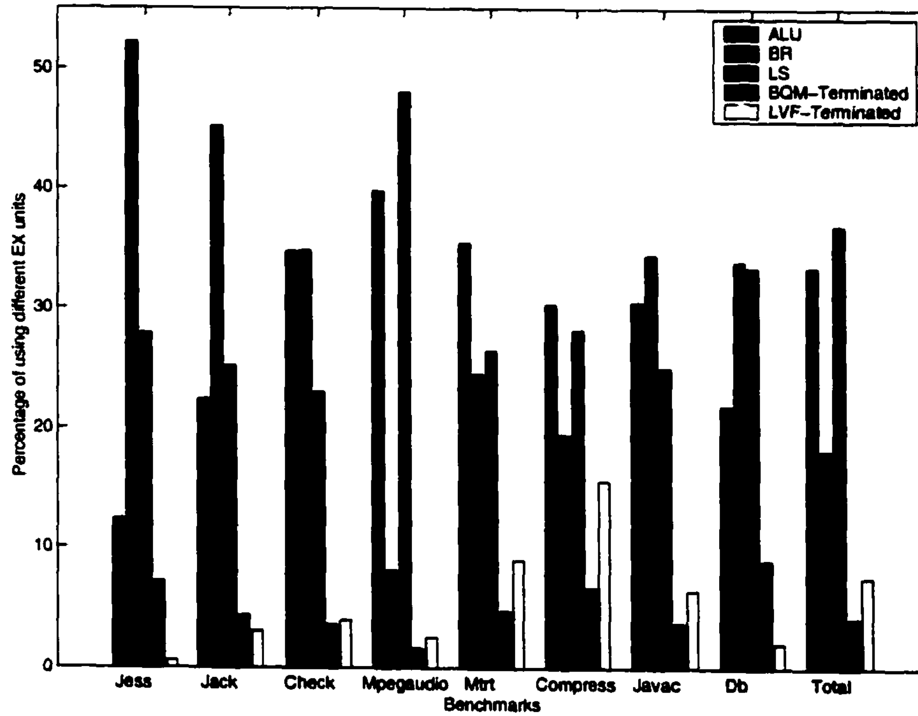


Figure 7.14. *Percentages of usage of different execution units (EXs).*

Table 7.3. *Comparison between the three approaches in supporting Java in hardware: direct stack execution, hardware interpretation, and hardware translation.*

Feature/option	Direct stack execution	Hardware interpretation	Hardware translation
general or specialized	specialized	general	general
stack cache included	✓		
uses JBC folding	✓		✓
employs RISC core		✓	✓
native language	JVM	RISC-like	RISC-like
translation methodology		microcoded	hardwired
dynamic scheduling		✓	✓
ILP exploration		✓	✓
flexibility	No	some	Yes

Chapter 8

Conclusions

8.1 Summary

This dissertation explored the feasibility of accommodating Java execution in the framework of modern processors. We presented the JAFARDD processor, a Java Architecture based on a Folding Algorithm, with Reservation Stations, Dynamic Translation, and Dual Processing, to accelerate JBC execution.

JAFARDD dynamically translates JBCs into RISC instructions to facilitate the utilization of a typical general-purpose RISC core at the back end. This approach enables the exploitation of the ILP among the translated instructions using well established techniques, and facilitates migration to Java-enabled hardware.

JAFARDD's design was guided with a set of global architectural design principles identified earlier in the research. Towards this end, Table 8.1 shows how each of these design principles were addressed while designing the various JAFARDD modules.

In this conclusions chapter, we detail the contributions of this dissertation (Section 8.2) and highlight some directions for future research (Section 8.3).

8.2 Contributions

This work has made a number of contributions to the improvement of Java execution through hardware support. These contributions are detailed in the following subsections. It is worth mentioning that some of the innovative ideas/techniques presented are not only applicable to Java processing, but could also in general be extended to any processor design.

Table 8.1. *How JAFARDD addresses the global architectural design principles.*

No.	Global architectural design principle	Corresponding JAFARDD module and feature	
1	Common case support	**FT:**	translates only frequently used JBCs
		LS:	traps non-foldable JBCs to OS routines
2	Stack dependency resolution	**BQM:**	folds JBCs as specified by the FIG
		BQM:	extensively and intelligently rearranges JBCs to emulate typical stack operations
		LVF:	performs internal operations that emulate typical stack operations
		RS:	processes tagged IFGs
3	Working data elements storage on-chip	**BQM:**	included on-chip
		LVF:	included on-chip
4	Exploitation of ILP	**LVF:**	performs internal operations that exploit ILP among JBCs
		RS:	provides a venue and mechanism for LV renaming and pipeline hazard handling
		EX:	multiple units are included
5	Dynamic scheduling	**LVF:**	interacts with Tomasulo's hardware
		RS:	provides a venue and mechanism for dynamic scheduling
		EX:	multiple units are included
6	Utilization of a Java-independent RISC core	**FT:**	dynamically translates folding groups into RISC instructions
7	Minimal changes to the software	**FT:**	does not require any JBC preprocessing
8	Hardware support for object-oriented processing	**LS:**	serves as the gateway for the hardware to reach emulating software routines
9	Folding information generation off the critical path	**FIG:**	generates folding groups at a rate that surpasses the rate of their consumption
10	Maintaining a high bytecode folding rate	**BF:**	An adaptive feedback fetch policy
		FIG:	I-cache bus is wide enough to accommodate several folding groups

8.2.1 Java Workload Characterization

Using a benchmarking based experimental framework, we conducted a comprehensive behavioral analysis of a typical Java workload. This study collected many aspects of the Java workload behavior from an execution trace, analyzed statistics obtained, and made detailed

recommendations for a Java processor's architectural requirements. We examined access patterns for data types, addressing modes, instruction encoding, instruction set utilization, instruction execution time, method invocation behavior, and the impact of object orientation. The results obtained help identify performance-critical aspects that are candidates for hardware support to narrow the semantic gap between the virtual machine and the native one. This study surpasses other similar ones in terms of the number of aspects studied and the coverage of the recommendations made.

8.2.2 A Methodical Design Space Analysis

We explored different hardware design options and alternatives that are suitable for Java, and their trade-offs. We especially focused on the design methodology, execution engine organization, parallelism exploitation, and support for high-level language features. The pros and cons of each option and alternative were discussed. Using this information, a computer architect could easily infer the best design configurations that meet target application specification, cost, and desired performance. The visual representation of the design trees helps identify innovative design ideas that would provide enhanced execution for Java. These ideas include a modified Tomasulo's algorithm, an improved folding algorithm, a dual processing architecture, and others.

8.2.3 The Identification of Global Architectural Design Principles

The behavioral analysis results and the design space exploration ideas were compiled into a list of global architectural design principles. These principles ensure JAFARRD can execute Java efficiently and are taken into consideration as the various instruction pipeline modules are designed. Some of these principles are recommended for addressing JVM performance-hindering features: stack dependency resolution, working data elements storage on-chip, hardware support for object-oriented processing, and folding information generation off the critical path. Other principles aim at achieving a better performance: common case support, exploitation of ILP, dynamic scheduling, utilization of a Java-independent RISC core, making minimal changes to the software systems, and maintaining a high bytecode folding rate.

8.2.4 Stack Dependency Resolution

Results gathered about the JVM behavior confirmed that data dependency in stack operations limits the performance of Java processors. Individual stack operations are executed one at a time thus consuming extra stack access cycles and creating virtual data dependency that limits performance and prohibits any form of ILP.

JAFARDD resolves the stack dependency by a number of novel techniques. We introduced the OPEX bytecode folding algorithm (Subsection 8.2.5) that eliminates the need for a stack. We also designed the processor internal modules in a way that compensates for the lack of the stack. For example, both the on-chip BQ and the LVF support a set of internal operations that emulate the required stack operations. Additionally, we tailored Tomasulo's algorithm for the execution of JBCs (Subsection 8.2.10). These combined features enabled a stackless processor design that is capable of executing Java stack code.

8.2.5 An Operand Extraction Bytecode Folding Algorithm

Stack operation folding has been suggested in the literature to enhance Java performance by grouping contiguous instructions that have true data dependencies into compound instructions. In this work, we extended existing folding algorithms by removing the restriction that the folded instructions must be consecutive. To the best of our knowledge, our folding algorithm is the only that permits nested pattern folding, tolerates variations in folding groups, and detects and resolves folding hazards. By incorporating it into any Java processor design, the need for, and therefore the limitations of, a stack are eliminated.

Benchmarking shows excellent performance gains as compared to other folding algorithms. The OPEX bytecode folding algorithm speeds up JBC execution by an average of about 1.29. Furthermore, it eliminates an average of about 97% of the stack instructions and an average of about 50% of the overall executed instructions. Benchmark traces show that an average of about 48% of the patterns are tagged and an average of about 36% of the total patterns are nested, which illustrate the performance gain that could be achieved. Issuing tagged patterns decouples the virtual dependency between successive folding groups. Whereas, folding nested patterns enables recognizing more patterns which decreases the number of JBCs issued unfolded. Processing both tagged and nested patterns indeed eliminates the need for a stack.

It is worth mentioning that although the OPEX bytecode folding algorithm was presented in the context of a hardware implementation, it can be adopted with minor changes in software implementations. It could be included, for example, in a JIT framework.

8.2.6 An Adaptive Bytecode Fetch Policy

We addressed the issue of variable length instructions which complicates JBC fetching and decoding, by an adaptive feedback fetch policy. In this policy, the number of bytecodes fetched every clock cycle is set according to the number of those issued in the previous clock cycle and those being queued. This policy helps keep the fetch rate as high as possible, thus maintaining a high folding rate and ensuring the pipeline is continuously fed.

8.2.7 Dynamic Binary Translation

The statically determinable type state of the JBCs enables simple on the fly translation into efficient native code. We utilized this property to dynamically translate folding groups into decoded RISC instruction formats. This facilitates incorporating a RISC core that could work independently from Java. As a result, no software preprocessing is required for JAFARDD. JBCs generated by typical Java compilers can run directly on it. This approach is able to generate highly optimized native code and narrows the semantic gap between the JVM and the processor, without adding extra semantic contents to the native code.

8.2.8 A RISC Core Driven by a Deep and Dynamic Pipeline

JAFARDD is designed to employ a deep and dynamic pipeline that provides high throughput and efficient mechanisms for parallelism exploitation. This pipeline drives a typical back end RISC core. This approach executes simple instructions quickly and enables parallelism exploitation among translated instructions using well established RISC techniques.

8.2.9 On-Chip Local Variable File

Results gathered about the JVM behavior confirmed that manipulating the LVs is one of the critical performance issues. We included the LVs on chip to permit manipulating them as

regular RISC registers. The novel LVF included provides the venue for adding some JVM-specific internal operations that compensates for the lack of a stack, exploits parallelism among JBCs, and facilitates the interaction with the Tomasulo algorithm's implementation.

8.2.10 A Modified Tomasulo's Algorithm

In this work, we tailored Tomasulo's Algorithm for use with JVM stack code execution. We used reservation stations as a venue and mechanism for LV renaming, dynamic scheduling, and pipeline hazard handling, which permit a higher degree of parallelism. They allow JBCs to be issued and executed OOO. These three features are also effective tools for stack dependency resolution and eliminate the need for a stack.

One of the salient contributions of our approach is the early assignment of instruction tags. Instead of generating tags at the reservation stations and letting them counterflow in the pipeline to the LVF, a separate tag generation unit is located in the same pipeline stage as the LVF to provide tags earlier in the process. This tag-early assignment approach has a number of merits: it saves the overhead in obtaining the tags, a new tag is generated every clock cycle and in the same pipeline stage as the LVF (which means a zero-cycle tag generation) and it eliminates the need for information counterflow in the pipeline (which simplifies the design and speeds up processing).

8.2.11 Complex Instruction Handling via a Load/Store Unit

The specialized LS handles instructions that manipulate objects or need access to the CP. Execution of such instructions is either very complicated, requires services from the underlying OS, or both. Thus, the significant complexity and/or the dependency on the underlying OS motivated us to implement them in software. The LS serves as the gateway for the hardware to reach emulating software routines for these instructions efficiently. It interacts smoothly with the pipeline organization and the incorporated Tomasulo's Algorithm.

8.2.12 A Dual Processing Architecture

Our dual architecture pipeline allows the execution of both Java and non-Java code on the same core. From a programmer's perspective, there are two logical views: a JVM, which is a stack machine, supporting all Java features, and a general RISC machine capable of

exploiting parallelism and contains features in addition to those required by the JVM. This duality in views maintains the generality of the processor, provides backward compatibility to execute non-Java code, and facilitates the migration to Java-enabled hardware.

8.3 Directions for Future Research

This dissertation has made many significant contributions towards improving Java execution using hardware support. However, there are still many unexplored areas that could potentially benefit from further research. Some of these areas include the following:

- **Providing better support for unfoldable bytecodes** JAFARDD emulates JVM complex instructions in software. Further research could aim at folding some of these bytecodes or seeking more efficient support for them. This could provide better performance for method invocations and other performance-critical aspects of the JVM.
- **Executing folding groups concurrently and OOO** This would lead to a superscalar support for Java programs.
- **Investigating the suitability of a VLIW paradigm for JBC execution** Research could be initiated to design the FT module for organizing translated folding groups into VLIW instructions.
- **Processing Java and non-Java code concurrently** Although our architecture supports dual processing, it does not permit Java and non-Java code to be interleaved in the pipeline. An advanced scheduler is needed to regulate the flow of both codes into the back end of the pipeline. A concurrent dual processing environment will improve the performance of multitasking/multithreading and I/O intensive environments.
- **Studying branching behavior of JVM programs** The performance of control flow instructions is well known to have a great impact on the overall program execution performance. Therefore further studies of the branch behavior is needed for designing better hardware support for Java.
- **Enhancing quick opcode processing** Implementing self-modifying code, resulting from Java's quick opcode processing, is an open research area. Self-modifying code in hardware can be costly. It requires flushing the structures that improve performance, including the pipeline, prefetch buffers, and decoded instruction cache.

- **Supporting garbage collection in hardware** A garbage collector mechanism cannot be interrupted. Otherwise the list of references and unreferenced memory might be corrupted. Consequentially, garbage collection must run atomically, i.e., once it starts, it must run to completion without interruption [224]. This might limit performance, as the application will have to stay idle, which imposes restrictions on real-time processing. Memory consumption can also be a problem in the case of performing an incremental garbage collection. Providing support for garbage collector helps non-Java languages as well as Java.
- **Assisting Java's HLL features** Hardware constructs that could aid Java high-level features, without affecting its generality, are vital for the design. Other HLLs running on such cores would also benefit from such features.
- **Accelerating exception processing** Yet another possibility for improving Java performance could be achieved by faster processing of Java exceptions. Methods used for achieving precise interrupts can be useful here.
- **Running Java threads on a simultaneous multithreading architecture** Another promising research direction would be to investigate the suitability of a simultaneous multithreading platform for Java execution.
- **Evaluating the design at the device level** The presented design of JAFARDD was performed at the behavioral level. This design needs to be brought down to the device level so that the power consumption, operating speed, and the required die area can be estimated. This might trigger further fine tuning for the design to be suitable for power-aware devices and embedded systems.
- **Tuning JAFARDD for embedded systems** Embedded system research is an exciting field. JAFARDD needs to be further tuned to suit such applications. For example, the power and area requirements need to be optimized.
- **Exploring the applicability of the presented concepts to other processors** The ideas presented in this work are not only applicable for Java, but could also be extended to other processor designs. For example, design of network processing units could benefit from the bytecode folding concepts in packet processing.

Bibliography

[1] J. Gosling, "Java intermediate bytecodes," in *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representation (IR'95)*, San Francisco, CA, USA, Jan. 22, 1995, ACM SIGPLAN Notices, pp. 111–118.

[2] M. Campione, K. Walrath, and A. Huml, *The Java Tutorial: A Short Course on the Basics*, Addison-Wesley, third edition, 2001.

[3] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*, The Java Series, Addison-Wesley, third edition, 2000.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, The Java Series, Addison-Wesley, second edition, June 2000.

[5] J. Gosling and H. McGilton, "The Java language environment," A White Paper, Sun Microsystems, May 1996.

[6] J. Meyer and T. Downing, *Java Virtual Machine*, O'Reilly and Associates, 1997.

[7] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, The Java Series, Addison-Wesley, second edition, Apr. 1999.

[8] B. Venners, *Inside The Java 2 Virtual Machine*, McGraw-Hill, 1999.

[9] A. S. Tanenbaum and J. R. Goodman, *Structured Computer Organization*, Prentice-Hall, fourth edition, 1999.

[10] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioǧlu, and E. Altman, "LaTTe: a Java VM just-in-time compiler with fast and efficient register allocation," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT 99)*, Newport Beach, CA, USA, Oct. 12–16, 1999, pp. 128–138.

[11] M. J. Murdocca and V. P. Heuring, *Principles of Computer Architecture*, Prentice-Hall, 2000.

[12] B. Case, "Java virtual machine should stay virtual," *Microprocessor Report*, vol. 10, no. 5, pp. 14–15, Apr. 15, 1996.

[13] C.-H. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhall, and W.-M. W. Hwu, "Optimizing NET compilers for improved Java performance," *IEEE Computer*, vol. 30, no. 6, pp. 67–75, June 1997.

[14] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, "Compiling Java just in time," *IEEE Micro*, pp. 36–43, May 1997.

[15] C.-H. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhall, and W.-M. W. Hwu, "A study of the cache and branch performance issues with running Java on current hardware platforms," in *Proceedings of the 42nd IEEE International Computer Conference (IEEE CompCon '97)*, San Jose, CA, USA, Feb. 23–26, 1997, pp. 211–216.

[16] C.-H. A. Hsieh, J. C. Gyllenhall, and W.-M. W. Hwu, "Java bytecode to native code translation: The caffeine prototype and preliminary results," in *Proceedings of the 29th IEEE/ACM Annual International Symposium on Microarchitecture (MICRO-29)*, Paris, France, Dec. 2–4, 1996, pp. 90–97.

[17] R. Radhakrishnan, "Improving performance of contemporary programming paradigms," A Ph. D. Research Proposal, Department of Electrical and Computer Engineering, The University of Texas at Austin, Apr. 1999.

[18] A. Murthy, N. Vijaykrishnan, and A. Sivasubramaniam, "How can hardware support Just-in-Time compilation?," in *Proceedings of the First Annual Workshop on Hardware Support for Objects and Microarchitectures for Java, held in conjunction with the International Conference on Computer Design (ICCD '99)*, Austin, TX, USA, Oct. 10, 1999, pp. 15–19.

[19] A. Krall and R. Grafl, "CACAO– a 64 bit JavaVM just-in-time compiler," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1017–1030, Nov. 1997.

[20] A. Krall, "Efficient Java VM just-in-time compilation," in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Paris, France, Oct. 12–18, 1998.

[21] P. Halfhill, "How to soup up Java. Part I," *Byte Magazine*, vol. 23, no. 5, pp. 60–74, May 1998.

[22] B. Case, "Implementing the Java virtual machine. Java's complex instruction set can be built in software or hardware," *Microprocessor Report*, vol. 10, no. 4, pp. 12–17, Mar. 25, 1996.

[23] P. Koopman, *Stack Computers: The New Wave*, Ellis Horwood, 1989.

[24] C. K. Yuen, "Stack and RISC," *Computer Architecture News*, vol. 27, no. 1, pp. 3–9, Mar. 1999.

[25] R. Radhakrishnan, J. Rubio, and L. K. John, "Characterization of Java applications at the bytecode level and at Ultra-SPARC machine code levels," in *Proceedings of the International Conference on Computer Design (ICCD '99)*, Austin, TX, USA, Oct. 11–13, 1999, pp. 281–284.

[26] P. Wayner, "How to soup up Java. Part II: Nine recipes for fast, easy Java– a maturing Java brings a wide variety of programming solutions to developers," *Byte Magazine*, vol. 23, no. 5, pp. 76–80, May 1998.

[27] N. Vijaykrishnan, "Java technology-based microprocessors: The past, the future in pioneers' progress with picoJava technology home page,

Sun Microsystems, [online document]," [Aug. 28, 2002] Available at: http://www.sun.com/microelectronics/picoJava/pioneers/vol3/licensee-PennState.html, Spring 1999.

[28] N. Vijaykrishnan and M. I. Wolczko, Eds., *Java Microarchitectures*, Kluwer Academic Publishers, Norwell, MA, USA, Apr. 2002.

[29] I. H. Kazi, B. Stanley, D. J. Lilja, A. Verma, and S. Davis, "Techniques for obtaining high performance in Java programs," Tech. Rep. HPPC-99-01, Department of Electrical Engineering and Department of Computer Science, High-Performance Parallel Computing Research Group, University of Minneapolis, Minnesota, Minnesota, USA, 1999.

[30] R. Radhakrishnan, "Microarchitectural techniques to improve performance of Java bytecode execution," A Research Proposal, Department of Electrical and Computer Engineering, The University of Texas at Austin, Apr. 1999.

[31] M. W. El-Kharashi and F. Elguibaly, "Java microprocessors: Computer architecture implications," in *Proceedings of the 1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM'97)*, Victoria, BC, Canada, Aug. 20–22, 1997, vol. 1, pp. 277–280.

[32] J. Turley, "Sun reveals first Java processor core," *Microprocessor Report*, vol. 10, no. 14, pp. 28–31, Oct. 28, 1996.

[33] J. Wharton, B. Case, D. S. Hardin, M. Hopkins, J. Novistsky, and M. Tremblay, "Software or silicon– what's the best route to Java?," Hotchips Panel Session, Aug. 18–20, 1996.

[34] M. Levy, "Java to go: Part 1. Accelerators process byte codes for portable and embedded applications," *Microprocessor Report*, vol. 15, no. 7, pp. 1,5–7, Feb. 12, 2001.

[35] M. Levy, "Java to go: Part 2. InSilicon takes Java acceleration to JVXtremes," *Microprocessor Report*, vol. 15, no. 10, pp. 7–8, Mar. 5, 2001.

[36] M. Levy, "Java to go: Part 3. Chicory system's Java accelerator pours a HotShot," *Microprocessor Report*, vol. 15, no. 13, pp. 9–11, Mar. 26, 2001.

[37] M. Levy, "Java to go: Part 4. Heterogeneous multiprocessing for Java applications," *Microprocessor Report*, vol. 15, no. 15, pp. 4–8, June 4, 2001.

[38] M. Levy, "Java to go: The finale," *Microprocessor Report*, vol. 15, no. 15, pp. 9–15, June 4, 2001.

[39] B. Case, "Java performance advancing rapidly," *Microprocessor Report*, vol. 10, no. 7, pp. 17–19, May 27, 1996.

[40] J. Bayko, "Great microprocessors of the past and present (V 12.3.0) in John

Bayko (Tau)'s Home Page, [online document]," [Aug. 28, 2002] Available at: http://www3.sk.sympatico.ca/jbayko/cpu.html, Mar. 2002.

[41] D. Takahashi, "Java chips make a comeback in Red Herring's Home Page, [online document]," [Aug. 28, 2002] Available at: http://www.redherring.com/industries/2001/0712/1630019763.html, July 12, 2001.

[42] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, third edition, 2002.

[43] K. Sakamura, "A Java-enabled evolution," *IEEE Micro*, pp. 2–3, July 2001.

[44] Sun Microsystems, Inc., "Sun blazes another trail– introducing the microJava 701 microprocessor," Press Releases, Sun Microsystems, Mountain View, CA, USA, Oct. 1997.

[45] Sun Microsystems, Inc., "The JavaChip processor: Redefining the processor market," A White Paper WPR-0011-03, Sun Microsystems, Nov. 1997.

[46] Sun Microsystems, Inc., "The benefits of the Java technology to microprocessors," A White Paper, Sun Microsystems, Nov. 1997.

[47] Sun Microsystems, Inc., "The burgeoning market for Java processors. Inside the networked future: The unprecedented opportunity for Java systems," A White Paper WPR-96-043, Sun Microsystems, Oct. 1996.

[48] S. H. Leibson, "Jazz joins VLIW juggernaut. CMP and Java as an HDL take system-on-chip design to parallel universe," *Microprocessor Report*, vol. 14, no. 4, pp. 34–40, Mar. 27, 2000.

[49] H. Ploog, R. Kraudelt, N. Bannow, T. Rachui, F. Golatowski, and D. Timmermann, "A two step approach in the development of a Java silicon machine (JSM) for small embedded systems," in *Proceedings of the First Annual Workshop on Hardware Support for Objects and Microarchitectures for Java, held in conjunction with the International Conference on Computer Design (ICCD'99)*, Austin, TX, USA, Oct. 10, 1999, pp. 45–49.

[50] M. Baentsch, P. Buhler, T. Eirich, F. Horing, and M. Oestreicher, "JavaCard– from hype to reality," *IEEE Concurrency*, pp. 36–43, Oct. 1999.

[51] R. W. Atherton, "Moving Java to the factory," *IEEE Spectrum*, pp. 18–23, Dec. 1998.

[52] P. Dibble, "The reality of real-time Java," *Computer Design*, pp. 70–76, Aug. 1998.

[53] Y.M. Lee, B.-C. Tak, H.-S. Maeng, and S.-D. Kim, "Real-time Java virtual machine for information appliances," *IEEE Transactions on Consumer Electronics*, vol. 46, no. 4, pp. 949–957, Nov. 2000.

[54] C.-M. Lin and T.-F. Chen, "Dynamic memory management for real-time embedded

Java chips," in *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*, Cheju Island, South Korea, Dec. 12–14, 2000, pp. 49–56.

[55] S. Hachiya, "Java use in mobile information devices: Introducing JTRON," *IEEE Micro*, pp. 16–21, July 2001.

[56] S. A. Ito, L. Carro, and R. P. Jacobi, "Making Java work for microntroller applications," *IEEE Design and Test of Computers*, pp. 100–110, Sept. 2001.

[57] M. Berekovic, H. Kloos, and P. Pirsch, "Hardware realization of a Java virtual machine for high performance multimedia applications," *Journal of VLSI Signal Processing System*, vol. 22, no. 1, pp. 31–43, Aug. 1999.

[58] C. E. McDowell, B. R. Montague, M. R. Allen, E. A. Baldwin, and M. E. Montoreano, "JAVACAM: Trimming Java down to size," *IEEE Internet Computing*, pp. 53–59, May 1998.

[59] R. Grehan, "JavaSoft's embedded specification overdue, but many tool vendors aren't waiting," *Computer Design*, pp. 14, 16, 18, Apr. 1998.

[60] M. Berekovic, H. Kloos, and P. Pirsch, "Hardware realization of a Java virtual machine for high performance multimedia applications," in *Proceedings of the 1997 IEEE Workshop on Signal Processing Systems (SIPS 97)– Design and Implementation*, Leicester, UK, Nov. 1997, pp. 479–488.

[61] J. R. Larus and M. D. Hill, "Implementing the Java VM in hardware," Course Notes, Computer Sciences Department, University of Wisconsin-Madison, Sept. 1996.

[62] J. Engel, *Programming for the Java Virtual Machine*, Addison-Wesley, 1999.

[63] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the AFIPS 1967 Spring Joint Computer Conference*, Atlantic City, NJ, USA, Apr. 18–20, 1967, vol. 30, pp. 483–485.

[64] Sparc International, Inc., *The Sparc Architecture Manual*, Prentice-Hall, 1992.

[65] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "A robust stack folding approach for Java processors: An operand extraction-based algorithm," *Journal of Systems Architecture*, vol. 47, no. 8, pp. 697–726, Dec. 2001.

[66] Sun Microsystems, Inc., "Sun community source licensing for picoJava technology, [online document]," [Aug. 28, 2002] Available at: http://www.sun.com/microelectronics/communitysource/, 1999.

[67] Pendragon Software Corporation, "Caffeinemark 3.0, [online document]," [Aug. 28, 2002] Available at: http://www.webfayre.com/pendragon/cm3, 1997.

[68] D. N. Antonioli and M. Pilz, "Analysis of the Java class file format," Tech. Rep. ifi-98.04, Department of Computer Science, University of Zurich, Apr. 1998.

[69] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, second edition, 1998.

[70] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan, "Java runtime systems: Characterization and architectural implications," *IEEE Transactions on Computers*, vol. 50, no. 2, pp. 131–146, Feb. 2001.

[71] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, and A. Sivasubramanium, "Architectural issues in Java runtime systems," in *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA-6)*, Toulouse, France, Jan. 8–12, 2000, pp. 387–398.

[72] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, and A. Sivasubramaniam, "Architectural issues in Java runtime systems," Tech. Rep., Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA, 1999.

[73] R. Radhakrishnan, J. Rubio, L. K. John, and N. Vijaykrishnan, "Execution characteristics of Just-In-Time compilers," Tech. Rep. TR-990717-01, Department of Electrical and Computer Engineering. The University of Texas at Austin, Austin, TX, USA, July 1999.

[74] T. Li, L. K. John, V. Narayanan, A. Sivasubramaniam, J. Sabarinathan, and A. Murthy, "Using complete system simulation to characterize SPECjvm98 benchmarks," in *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, NM, USA, May 8–11, 2000, pp. 22–33.

[75] A. Barisone, F. Bellotti, R. Berta, and A. DeGloria, "UltraSparc instruction level characterization of Java virtual machine workload," in *Workload Characterization For Computer System Design (Digest of The Workshop on Workload Characterization (WWC-99), Oct 9, 1999, Austin, TX, USA)*, L. K. John and A. M. G. Maynard, Eds., chapter 1, pp. 1–24. Kluwer Academic Publishers, 2000.

[76] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey, "A methodology for benchmarking Java grande applications," in *Proceedings of the ACM 1999 Java Grande Conference*, San Francisco, CA, USA, June 12–14, 1999, pp. 81–88.

[77] C. Daly, J. Horgan, J. Power, and J. Waldron, "Platform independent dynamic Java virtual machine analysis: the Java grande forum benchmark suite," in *Proceedings of the ACM 2001 Java Grande Conference*, Palo Alto, CA, USA, June 2–4, 2001, pp. 106–115.

[78] J.-S. Kim and Y. Hsu, "Analyzing memory reference traces of Java programs," in *Workload Characterization For Computer System Design (Digest of The Workshop on Workload Characterization (WWC-99), Oct. 9, 1999, Austin, TX, USA)*, L. K. John and A. M. G. Maynard, Eds., chapter 1, pp. 25–48. Kluwer Academic Publishers, 2000.

[79] S. Liang and D. Viswanathan, "Comprehensive profiling support in the Java vir-

tual machine," in *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, May 3–7, 1999, pp. 229–240.

[80] Ø. Strøm, A. Klauseie, and E. J. Aas, "A study of dynamic instruction frequencies in byte compiled Java programs," in *Proceedings of the 25th EUROMICRO Conference (EUROMICRO'99)*, Milan, Italy, Sept. 8–10, 1999, vol. 1, pp. 232–235.

[81] M. Thomas, V. Getov, M. Williams, and R. Whitney, "Benchmarking Java on the IBM SP2," in *Proceedings of the ACM 1999 Java Grande Conference*, San Francisco, CA, USA, June 12–14, 1999.

[82] J. Waldron, "Dynamic bytecode usage by object oriented Java programs," in *Proceedings of the Technology of Object-Oriented Languages and Systems*, Nancy, France, June 7–10, 1999, p. 246.

[83] J. Kreuzinger, R. Zulauf, A. Schulz, T. Ungerer, M. Pfeffer, U. Brinkschulte, and C. Krakowski, "Performance evaluations and chip-space requirements of a multithreaded Java microcontroller," in *Proceedings of the Second Annual Workshop on Hardware Support for Objects and Microarchitectures for Java, held in conjunction with the International Conference on Computer Design ICCD'00*, Austin, TX, USA, Sept. 17, 2000, pp. 32–36.

[84] D. Gregg, J. Power, and J. Waldron, "Benchmarking the Java virtual architecture," in *Java Microarchitectures*, N. Vijaykrishnan and M. I. Wolczko, Eds., chapter 1, pp. 1–18. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[85] J. Rubio, "Characteristics of Java applications at the bytecode level," M.S. thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA, May 1999.

[86] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "A quantitative study for Java microprocessor architectural requirements. Part I: Instruction set design," *Microprocessors and Microsystems*, vol. 24, no. 5, pp. 225–236, Sept. 1, 2000.

[87] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "A quantitative study for Java microprocessor architectural requirements. Part II: High-level language support," *Microprocessors and Microsystems*, vol. 24, no. 5, pp. 237–250, Sept. 1, 2000.

[88] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "Quantitative analysis for Java microprocessor architectural requirements: Instruction set design," in *Proceedings of the First Annual Workshop on Hardware Support for Objects and Microarchitectures for Java, held in conjunction with the International Conference on Computer Design (ICCD'99)*, Austin, TX, USA, Oct. 10, 1999, pp. 50–54.

[89] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "Architectural requirements for Java processors: A quantitative analysis," Tech. Rep. ECE98-5, Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, Canada, Nov. 1998.

[90] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "Multithreaded processors: The

upcoming generation for multimedia chips," in *Proceedings of the IEEE Symposium on Advances in Digital Filtering and Signal Processing (DFSP'98)*, Victoria, BC, Canada, June 5–6, 1998, pp. 111–115.

[91] D. Sima, T. Fountain, and P. Kacsuk, *Advanced Computer Architecture: A design Space Approach*, Addison-Wesley, 1997.

[92] H. McGhan and J. M. O'Connor, "picoJava: A direct execution engine for Java bytecode," *IEEE Computer*, vol. 31, no. 10, pp. 22–30, Oct. 1998.

[93] S. Hangel and M. O'Connor, "Performance analysis and validation of the picoJava processor," *IEEE Micro*, pp. 66–72, May 1999.

[94] F. Azhari, "The picoJava processor core," A presentation at JavaOne 1999, June 13–18, 1999.

[95] J. M. O'Connor and M. Tremblay, "picoJava-I: The Java virtual machine in hardware," *IEEE Micro*, pp. 45–53, Mar. 1997.

[96] J. Turley, "microJava pushes bytecode performance. Sun's microJava 701 based on new generation of picoJava core," *Microprocessor Report*, vol. 12, no. 2, pp. 9–12, Nov. 17, 1997.

[97] P. Wayner, "Sun gambles on Java chips: Are Java chips better than general purpose CPUs? or will new compilers make them obsolete?," *Byte Magazine*, vol. 21, no. 11, pp. 79, 80, 82, 84, 86, 88, Nov. 1996.

[98] M. Levy, "DCT marches into Java processors. Lightfoot and Bigfoot processors offer new twist to Java execution," *Microprocessor Report*, vol. 16, no. 4, pp. 14–22, Jan. 28, 2002.

[99] Patriot Scientific Corporation, "PTSC Java accelerator chip, [online document]," [Aug. 16, 2000] Available at: http://www.ptsc.com/psc1000.

[100] J. Turley, "New embedded CPU goes ShBoom," *Microprocessor Report*, vol. 10, no. 5, pp. 1,6–10, Apr. 15, 1996.

[101] D. S. Hardin, "Crafting a Java virtual machine in silicon," *IEEE Instrumentation and Measurement Magazine*, vol. 4, pp. 54–56, Mar. 2001.

[102] S. Dey, D. Panigrahi, L. Chen, C. N. Taylor, K. Sekar, and P. Sanchez, "Using a soft core in a SOC design: Experiences with picoJava," *IEEE Design and Test of Computers*, pp. 60–71, July 2000.

[103] A. Kim and M. Chang, "Designing a Java microprocessor core using FPGA technology," *Computing and Control Engineering Journal*, vol. 11, no. 3, pp. 135–141, June 2000.

[104] A. Kim and M. Chang, "Designing a Java microprocessor core using FPGA technology," in *Proceedings of the 11th Annual IEEE International ASIC Conference*, Rochester, NY, USA, Sept. 13–16, 1998, pp. 13–17.

[105] S. Kimura, H. Kida, K. Takagi, T. Abematsu, and K. Watanabe, "An application specific Java processor with reconfigurabilities," in *Proceedings of the Asia and South Pacific Design Automation Conference 2000 (ASP-DAC 2000)*, Yokohama, Japan, Jan. 25–28, 2000, pp. 25–26.

[106] J. M. P. Cardoso and H. C. Neto, "Macro-based hardware compilation of Java bytecodes into a dynamic reconfigurable computing system," in *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, USA, Apr. 20–23, 1999, pp. 2–11.

[107] T. Halfhill, "JSTAR coprocessor accelerates Java," *Microprocessor Report*, vol. 14, no. 4, pp. 29–33, Mar. 27, 2000.

[108] K. B. Kent and M. Serra, "Hardware architecture for Java in a hardware/software co-design of the virtual machine," in *To Appear in the Proceedings of the EUROMICRO Symposium on Digital System Design (DSD'2002)*, Dortmund, Germany, Sept. 4–6, 2002.

[109] K. B. Kent and M. Serra, "Context switching in a hardware/software co-design of the Java virtual machine," in *Proceedings of the Designer's Forum of Design Automation and Test in Europe (DATE) 2002*, Paris, France, Mar. 4–6, 2002, pp. 81–86.

[110] K. B. Kent and M. Serra, "Hardware/software co-design of a Java virtual machine," in *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, Paris, France, June 21–23, 2000, pp. 66–71.

[111] Sun Microsystems, Inc., "MAJC architecture tutorial [online document]," A White Paper WPR-0064-01, [Aug. 28, 2002] Available at: http://www.sun.com/processors/whitepapers/majctutorial.pdf, Sept. 1999.

[112] Sun Microsystems, Inc., "MAJC architecture features [online document]," A White Paper, [Aug. 28, 2002] Available at: http://www.sun.com/processors/whitepapers/majcintro.html.

[113] Sun Microsystems, Inc., "MAJC 5200 product bulletin MAJC home page, Sun Microsystems, [online document]," [Aug. 28, 2002] Available at: http://www.sun.com/processors/MAJC/03.SME.125ProdBul.pdf.

[114] Sun Microsystems, Inc., "A high performance microprocessor for multimedia computing [online document]," A White Paper, [Aug. 28, 2002] Available at: http://www.sun.com/processors/whitepapers/5200wp.html.

[115] Sun Microsystems, Inc., "MAJC processor family in MAJC home page, Sun Microsystems, [online document]," [Aug. 28, 2002] Available at: http://www.sun.com/processors/MAJC/.

[116] S. Sudharsanan, "MAJC-5200: A high performance microprocessor for multimedia computing," in *Proceedings of the 15th Workshop on Parallel and Distributed Pro-*

cessing (IPDPS 2000), Cancun, Mexico, May 2–4, 2000, Lecture Notes in Computer Science, Springer-Verlag, Berlin (LNCS 1800), pp. 163–170.

[117] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse, "The MAJC architecture: A synthesis of parallelism and scalability," *IEEE Micro*, pp. 12–25, Nov. 2000.

[118] M. Tremblay, "Wirespeed multimedia computing," A Keynote Presentation at Media Processors 2000, an IS&T/SPIE Conference, Jan. 27–28, 2000.

[119] M. Tremblay, "MAJC-5200: A VLIW convergent MPSOC," A Presentation at Microprocessor Forum, Oct. 4–8, 1999.

[120] M. Tremblay, "MAJC: An architecture for the new millennium," in *Proceedings of the Hot Chips 11 Symposium*, Palo Alto, CA, USA, Aug. 15–17, 1999, pp. 275–288, A Tutorial.

[121] T. Halfhill, "Sun reveals secrets of "magic". New MAJC architecture has VLIW, chip multiprocessing up its sleeve," *Microprocessor Report*, vol. 13, no. 11, pp. 13–17, Aug. 23, 1999.

[122] L. Gwennap, "MAJC gives VLIW a new twist: New Sun instruction set is powerful but simpler than IA-64," *Microprocessor Report*, vol. 13, no. 12, pp. 12–15, Sept. 13, 1999.

[123] B. Case, "Sun makes MAJC with mirrors. Dual on-chip mirror-image processor cores cooperate for high performance," *Microprocessor Report*, vol. 13, pp. 18–21, Oct. 25, 1999.

[124] P. Sriram, S. harsanan, and Amit Gulati, "MPEG-2 video decompression on a multiprocessing VLIW microprocessor," in *Proceedings of the IEEE International Conference on Consumer Electronics (ICCE 2000)*, Los Angeles, CA, USA, June 13–15, 2000, pp. 483–485.

[125] D. Pountain, "StrongARM tactics," *Byte Magazine*, vol. 21, no. 1, pp. 153–154, Jan. 1996.

[126] R. Radhakrishnan, R. Bhargava, and L. K. John, "Improving Java performance using hardware translation," in *Proceedings of the International Conference on Supercomputing (ICS01)*, Sorrento, Italy, June 18–21, 2001, pp. 427–439.

[127] R. Radhakrishnan, *Microarchitectural Techniques to Enable Efficient Java Execution*, Ph.D. thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA, Aug. 2000.

[128] R. Radhakrishnan and L. John, "A decoupled translate execute (DTE) architecture to improve performance of Java execution," in *Proceedings of the First Annual Workshop on Hardware Support for Objects and Microarchitectures for Java, held in conjunction with the International Conference on Computer Design (ICCD'99)*, Austin, TX, USA, Oct. 10, 1999, pp. 25–29.

[129] J. Glossner and S. Vassiliadis, "The Delft-Java engine: Microarchitecture and Java acceleration," in *Java Microarchitectures*, N. Vijaykrishnan and M. I. Wolczko, Eds., chapter 6, pp. 105–121. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[130] C. John Glossner, *The Delft-Java Engine*, Ph.D. thesis, Delft University of Technology, Delft, The Netherlands, Nov. 2001.

[131] J. Glossner and S. Vassiliadis, "Delft-Java dynamic translation," in *Proceedings of the 25th EUROMICRO Conference (EUROMICRO'99) Informatics: Theory and Practice for The New Millennium*, Milan, Italy, Sept. 8–10, 1999, vol. 1, pp. 57–62.

[132] C. J. Glossner and S. Vassiliadis, "Delft-Java link translation buffer," in *Proceedings of the 24th EUROMICRO Conference (EUROMICRO'98)*, Vasteras, Sweden, Aug. 25–27, 1998, vol. 1, pp. 221–228.

[133] C. J. Glossner and S. Vassiliadis, "The Delft-Java engine: An introduction," in *Proceedings of the Third International Euro-Par Conference (Euro-Par'97 Parallel Processing)*, Passau, Germany, Aug. 26–29, 1997, Lecture Notes in Computer Science, Springer-Verlag, Berlin (LNCS 1300), pp. 766–770.

[134] T. Halfhill, "Imsys hedges bets on Java: Rewritable-microcode chip has instruction sets for Java, Forth, C/C++," *Microprocessor Report*, vol. 15, no. 4, pp. 1–4, Aug. 14, 2000.

[135] T. Halfhill, "GP1000 has rewritable microcode. Imsys processor executes Java bytecodes concurrent microcode processes," *Microprocessor Report*, vol. 13, no. 2, pp. 14–17, Dec. 28, 1998.

[136] Imsys AB, "Low cost and low power solutions for embedded Java applications, [online document]," [Aug. 28, 2002] Available at: http://www.javamachine.com/homeeng.htm.

[137] T. Halfhill, "Embedded Java chips get real: Bytecode-native aJ-100 handles real-time processing," *Microprocessor Report*, vol. 15, no. 2, pp. 31–36, Aug. 7, 2000.

[138] aJile Systems, Inc., "aJ-100 reference manual version 2.0," Manual, aJile Systems, Nov. 2000.

[139] J. Turley, "Embedded news: Rockwell unearths Java JEM," *Microprocessor Report*, vol. 11, pp. 10–11, Oct. 15, 1997.

[140] R. Radhakrishnan, L. K. John, R. Bhargava, and D. Talla, "Improving Java performance in embedded and general-purpose processors," in *Java Microarchitectures*, N. Vijaykrishnan and M. I. Wolczko, Eds., chapter 5, pp. 79–104. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[141] J. Turley, "Most significant bits: Siemens nabs Java for smart cards," *Microprocessor Report*, vol. 11, no. 11, pp. 4–5, 9, Aug. 4, 1997.

[142] Ø. Strøm, *VLSI Realization of an Embedded Microprocessor Core with Support for*

Java Instructions, Ph.D. thesis, Norwegian University of Science and Technology, Trondheim, Norwegian, Oct. 2000.

[143] Ø. Strøm and E. J. Aas, "A novel microprocessor architecture for executing byte compiled Java code," in *Proceedings of the International Conference on Chip Design Automation (ICDA 2000), a Part of the 16th IFIP World Computer Congress*, Aug. 22–24, 2000.

[144] Ø. Strøm and E. J. Aas, "A novel approach for executing bytecode compiled Java code in hardware," *IEEE Computer Society's Technical Committee on Computer Architecture (TCCA) Newsletter*, vol. 28, no. 2, pp. 21–23, June 2000.

[145] Sun Microsystems, Inc., "picoJava-II programmer's reference," Manual, Sun Microsystems, Mountain View, CA, USA, Mar. 1999.

[146] Sun Microsystems, Inc., "picoJava-II microarchitecture guide," Manual, Sun Microsystems, Mountain View, CA, USA, Mar. 1999.

[147] Sun Microsystems, Inc., "microJava-701 Java processor," Data Sheet, Sun Microsystems, Mountain View, CA, USA, Jan. 1998.

[148] Sun Microsystems, Inc., "microJava-701 Java processor," User Guide, Sun Microsystems, Mountain View, CA, USA, Jan. 1998.

[149] Sun Microsystems, Inc., "picoJava-II Java processor core," Data Sheet, Sun Microsystems, Mountain View, CA, USA, Apr. 1998.

[150] Sun Microsystems, Inc., "picoJava-I Java processor core," Data Sheet, Sun Microsystems, Mountain View, CA, USA, Dec. 1997.

[151] M. Tremblay and J. M. O'Connor, "picoJava: A hardware implementation of the Java virtual machine," Hotchips Presentation, Aug. 18–20, 1996.

[152] N. Vijaykrishnan, *Issues in the Design of a Java Processor Architecture*, Ph.D. thesis, College of Engineering, University of South Florida, Tampa, FL, USA, July 1998.

[153] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla, "Object-oriented architectural support for a Java processor," in *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, New York, NY, USA, July 20–24, 1998, Lecture Notes in Computer Science, Springer-Verlag, Berlin (LNCS 1445), pp. 330–354.

[154] Sun Microsystems, Inc., "picoJava-I microprocessor core architecture," A White Paper WPR-0014-01, Sun Microsystems, Oct. 1996.

[155] M. Lentczner, "Java's virtual world," *Microprocessor Report*, vol. 10, no. 4, pp. 8–11,17, Mar. 25, 1996.

[156] Sun Microsystems, Inc., "Sun microelectronics' picoJava-I posts outstanding performance," A White Paper WPR-0015-01, Sun Microsystems, Oct. 1996.

[157] K. Skadron K. Scott, "BLP: Applying ILP techniques to bytecode execution," in *Proceedings of the Second Annual Workshop on Hardware Support for Objects and Microarchitectures for Java, held in conjunction with the International Conference on Computer Design ICCD'00*, Austin, TX, USA, Sept. 17, 2000, pp. 27–31.

[158] Kevin Scott and Kevin Skadron, "BLP: Applying ILP techniques to bytecode execution," Tech. Rep. CS-2000-05, Department of Computer Science, University of Virginia, Charlottesville, VA, USA, Feb. 2000.

[159] R. Radhakrishnan, D. Talla, and L. K. John, "Allowing for ILP in an embedded Java processor," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-2000)*, Vancouver, BC, Canada, June 10–14, 2000, pp. 294–305.

[160] R. Radhakrishnan, J. Rubio, and L. John, "Instruction level parallelism in Java," Tech. Rep. TR-990719-01, Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA, 1999.

[161] Y. Li, S. Li, X. Wang, and W. Chu, "JAViR– exploiting instruction level parallelism for Java machine by using virtual registers," in *Proceedings of the Second European IASTED International Conference on Parallel and Distributed Systems*, Vienna, Austria, July 1–3, 1998.

[162] N. Vijaykrishnan and N. Ranganathan, "Tuning branch predictors to support virtual method invocation in Java," in *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, May 3–7, 1999, pp. 217–228.

[163] T. Li and L. K. John, "OS-aware prediction: Alleviating user/kernel branch aliasing in Java processing," Tech. Rep. TR-010720-01, Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, USA, July 2001.

[164] T. Li, L. K. John, and N. Vijaykrishnan, "Understanding control flow transfer and its predictability in Java processing," Tech. Rep. TR-010108-01, Department of Electrical and Computer Engineering, Engineering, University of Texas, Austin, TX, USA, Jan. 2001.

[165] T. Li, L. K. John, and N. Vijaykrishnan, "Improving branch predictability in Java processing," Tech. Rep. TR-010215-01, Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, USA, Feb. 2001.

[166] T. Li and L. K. John, "Understanding control flow transfer and its predictability in Java processing," in *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Tucson, AZ, USA, Nov. 4–6, 2001, pp. 65–76.

[167] T. Li, L. K. John, N. Vijaykrishnan, and A. Sivasubramaniam, "Branch behavior of Java runtime systems and its microarchitectural," Tech. Rep. TR-000625-01, De-

partment of Electrical and Computer Engineering, Engineering, University of Texas, Austin, TX, USA, June 2000.

[168] A. Kumar, "The HP PA-8000 RISC CPU," *IEEE Micro*, pp. 27–32, Mar. 1997.

[169] J. A. Fisher, "Very long instruction word architectures and ELI-512," in *Proceedings of the 10th Annual International Symposium on Computer Architecture (ISCA-1983)*, Stockholm, Sweden, June 13–17, 1983, pp. 140–150.

[170] B. R. Rau, "Dynamically scheduled VLIW processors," in *Proceedings of the 26th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-26)*, Austin, TX, USA, Dec. 1-3, 1993, pp. 80–92.

[171] K. Buchenrieder, R. Kress, A. Pyttel, A. Sedlmeier, and C. Veith, "Scalable processor architecture for Java with explicit thread support," *Electronics Letters*, vol. 33, no. 18, pp. 1532–1534, June 19, 1997.

[172] I. Watson, G. Wright, and A. El-Mahdy, "VLSI architecture using lightweight threads (VAULT)," in *Proceedings of the First Annual Workshop on Hardware Support for Objects and Microarchitectures for Java, held in conjunction with the International Conference on Computer Design (ICCD'99)*, Austin, TX, USA, Oct. 10, 1999, pp. 40–44.

[173] K. Watanabe, W. Chu, and Y. Li, "Exploiting Java instruction/thread level parallelism with horizontal multithreading," in *Proceedings of the 6th Australian Computer Systems Architecture Conference (ACSAC 2001)*, Gold Coast, Queensland, Australia, Jan. 29–Feb. 4, 2001, pp. 122–129.

[174] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer, "A multithreaded Java microcontroller for thread-oriented real-time event-handling," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT 99)*, Newport Beach, CA, USA, Oct. 12–16, 1999, pp. 34–39.

[175] D. S. Hardin, A. P. Mass, M. H. Masters, and Nick M. Mykris, "An efficient hardware implementation of Java bytecodes, threads, and processes for embedded and real-time applications," in *Java Microarchitectures*, N. Vijaykrishnan and M. I. Wolczko, Eds., chapter 3, pp. 41–54. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[176] Y. Luo and L. K. John, "Workload characterization of multithreaded Java servers,," Tech. Rep. TR-010815-01, Department of Electrical and Computer Engineering, Engineering, University of Texas, Austin, TX, USA, Aug. 2001.

[177] C.-M. Chung and S.-D. Kim, "A dualthreaded Java processor for Java multithreading," in *Proceedings of the 1998 International Conference on Parallel and Distributed Systems (ICPADS'98)*, Tainan, Taiwan, Dec. 14–16, 1998, pp. 693–700.

[178] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen,

"Simultaneous multithreading: A platform for next-generation processors," *IEEE Micro*, pp. 12–19, Sept. 1997.

[179] R. Espasa and M. Valero, "Exploring instruction– and data-level parallelism," *IEEE Micro*, pp. 20–27, Sept. 1997.

[180] J. L. Lo, S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen, "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 322–354, Aug. 1997.

[181] K. S. Loh and W. F. Wong, "Multiple context multithreaded superscalar processor architecture," *Journal of Systems Architectures*, vol. 46, no. 3, pp. 243–258, Jan. 2000.

[182] E. B. Fernandez and T. Lang, *Software-Oriented Computer Architecture*, IEEE Computer Society Press, 1986.

[183] IEEE, "IEEE trial-use standard for extended high-level languages implementations for microprocessors," IEEE Standard 755, 1985.

[184] K. W. Ng and K. Y. Mok, "The high level language and operating system support features of advanced microprocessors. Part I: High level language support features," *Microprocessing and Microprogramming*, vol. 19, pp. 203–218, 1987.

[185] K. W. Ng, "The high level language and operating systems support features of advanced microprocessors. Part II: Operating system support features," *Microprocessing and Microprogramming*, vol. 19, pp. 277–289, 1987.

[186] V. Milutinovic, *Advances in Computer Architecture*, John Wiley and Sons, second edition, 1982.

[187] V. Milutinovic, *Advanced Microprocessors and High-Level Language Computer Architecture*, IEEE Computer Society Press, 1986.

[188] N. Vijaykrishnan and N. Ranganathan, "Supporting object accesses in a Java processor," *IEE Proceedings– Computers and Digital Techniques*, vol. 147, no. 6, pp. 435–443, Nov. 2000.

[189] N. Vijaykrishnan and N. Ranganathan, "Object addressing support for a Java processor," in *Proceedings of the 6th International Conference on Advanced Computing (ADCOM'98)*, Pune, India, Dec. 14–16, 1998, pp. 61–67.

[190] T. Ogasawara, H. Komatsu, and T. Nakatani, "A study of exception handling and its dynamic optimization in Java," in *Proceedings of the OOPSLA'01 Conference on Object-Oriented Programming Systems, Languages and Applications*, Tampa Bay, FL, USA, Oct. 14–18, 2001, pp. 83–95.

[191] A. Thekkath and H. M. Levy, "Hardware and software support for efficient exception handling," in *Proceedings of the 6th International Conference on Architectural Sup-*

port for Programming Languages and Operating Systems (ASPLOS-VI), San Jose, CA, USA, Oct. 4–7, 1994, pp. 110–119.

[192] J. Cohen, "Garbage collection of linked data structures," *ACM Computing Surveys*, vol. 13, no. 3, pp. 341–367, Sept. 1981.

[193] O. Agesen, D. Detlefs, and J. E. B. Moss, "Garbage collection and local variable type-precision and liveness in Java virtual machines," in *Proceedings of the Programming Language Design and Implementation*, Montreal, Canada, June 17–19, 1998, pp. 269–279.

[194] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko, "Tuning garbage collection in an embedded Java environment," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA-8)*, Cambridge, MA, USA, Feb. 2–6, 2002.

[195] A. Berlea, S. Cotofana, I. Athanasiu, J. Glossner, and S. Vassiliadis, "Garbage collection for the Delft Java processor," in *Proceedings of the 18th IASTED International Conference on Applied Informatics (AI'2000)*, Innsbruck, Austria, Feb. 14–17, 2000, pp. 232–238.

[196] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "Hardware adaptations for Java: A design space approach," Tech. Rep. ECE99-1, Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, Canada, Jan. 1999.

[197] R. M. Tomasulo, "An efficient algorithm for exploring multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, Jan. 1967.

[198] M. W. El-Kharashi, F. Gebali, and K. F. Li, "Stack dependency resolution for Java processors based on hardware folding and translation: A bytecode processing analysis," in *Java Microarchitectures*, N. Vijaykrishnan and M. I. Wolczko, Eds., chapter 4, pp. 55–77. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[199] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "Adapting Tomasulo's algorithm for bytecode folding based Java processors," *ACM Computer Architecture News*, pp. 1–8, Dec. 2001.

[200] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "A Java processor architecture with bytecode folding and dynamic scheduling," in *Proceedings of the 2001 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM'01)*, Victoria, BC, Canada, Aug. 26–28, 2001, vol. 1, pp. 307–310.

[201] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "A Java processor architecture based on hardware bytecode folding and translation, and Tomasulo's algorithm with reservation stations," Tech. Rep. ECE01-5, Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, Canada, July 2001.

[202] W. Munsil and C.-J. Wang, "Reducing stack usage in Java bytecode execution," *Computer Architecture News*, vol. 26, no. 1, pp. 7–11, Mar. 1998.

[203] L.-C. Chang, L.-R. Ton, M.-F. Kao, and C.-P. Chung, "Stack operations folding in Java processors," *IEE Proceedings– Computers and Digital Techniques*, vol. 145, no. 5, pp. 333–340, Sept. 1998.

[204] H.-M. Tseng, L.-C.Chang, L.-R. Ton, M.-F. Kao, S.-S Shang, and C.-P. Chung, "Performance enhancement by instruction folding strategies of a Java processor," in *Proceedings of the International Conference on Computer Systems Technology for Industrial Applications– Internet and Multimedia (CSIA'97)*, Hsinchu, Taiwan, Apr. 23–25, 1997, pp. 286–293.

[205] L.-R. Ton, L.-C. Chang, M.-F. Kao, H.-M. Tseng, S.-S Shang, R.-L. Ma, D.-C. Wang, and C.-P. Chung, "Instruction folding in Java processor," in *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS'97)*, Seoul, Korea, Dec. 11–13, 1997, pp. 138–143.

[206] L.-R. Ton, L.-C. Chang, and C.-P. Chung, "Exploiting Java bytecode parallelism by enhanced POC folding model," in *Proceedings of the 6th International Euro-Par Conference (Euro-Par 2000 Parallel Processing)*, Munich, Germany, Aug. 29–Sept. 1, 2000, Lecture Notes in Computer Science, Springer-Verlag, Berlin (LNCS 1900), pp. 994–997.

[207] A. Kim and M. Chang, "Advanced POC model-based Java instruction folding mechanism," in *Proceedings of the 26th EUROMICRO Conference (EUROMICRO'00)*, Maastricht, The Netherlands, Sept. 5–7, 2000, vol. 1, pp. 332–338.

[208] A. Kim and M. Chang, "An advanced instruction folding mechanism for a stackless Java processor," in *Proceedings of the International Conference on Computer Design ICCD'00*, Austin, TX, USA, Sept. 17–20, 2000, pp. 565–566.

[209] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[210] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "A new methodology for stack operations folding for Java microprocessors," in *High Performance Computing Systems and Applications*, N. J. Dimopoulos and K. F. Li, Eds., chapter 17, pp. 235–251. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[211] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "An operand extraction-based stack folding algorithm for Java processors," in *Proceedings of the Second Annual Workshop on Hardware Support for Objects and Microarchitectures for Java, held in conjunction with the International Conference on Computer Design (ICCD'00)*, Austin, TX, USA, Sept. 17, 2000, pp. 22–26.

[212] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "A novel approach for stack operations folding for Java processors," *IEEE Computer Society's Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 104–107, Sept. 2000.

[213] M. W. El-Kharashi, F. Elguibaly, and K. F. Li, "Generalized stack folding for Java

processors: An operand extraction-based algorithm," Tech. Rep. ECE00-2, Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, Canada, July 2000.

[214] W.-M. Hwu and Y. Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," in *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA-1986)*, Tokyo, Japan, June 2–5, 1986, pp. 297–306.

[215] G. S. Sohi, "Instruction issue logic for high performance, interruptible, multiple functional unit, pipelined computers," *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 349–359, Mar. 1990.

[216] A. R. Peszkun and G. S. Sohi, "The performance potential of multiple functional unit processors," in *Proceedings of the 15th Annual International Symposium on Computer Architecture (ISCA-1988)*, Honolulu, Hawaii, USA, May 30–June 2, 1988, pp. 37–44.

[217] G. S. Sohi and S. Vajapeyam, "Instruction issue logic for high performance, interruptable pipelined processors," in *Proceedings of the 14th Annual International Symposium on Computer Architecture (ISCA-1987)*, Pittsburg, PA, USA, June 2–5, 1987, pp. 27–34.

[218] S. Weiss and J. E. Smith, "Instruction issue logic in pipelined supercomputers," *IEEE Transactions on Computers*, vol. 33, no. 11, pp. 1013–1022, Nov. 1984.

[219] P. N. Golla and E. C. Lin, "A dynamic scheduling logic for exploiting multiple functional units in single chip multithreaded architectures," in *Proceedings of the 1999 ACM Symposium on Applied Computing*, San Antonio, TX, USA, Feb. 29–Mar. 2, 1993, pp. 466–473.

[220] D. Sima, "Superscalar instruction issue," *IEEE Micro*, pp. 28–39, Sept. 1997.

[221] D. Sima, "The design space of register renaming techniques," *IEEE Micro*, pp. 70–83, Sept. 2000.

[222] D. Sima, "The design space of shelving," *Journal of Systems Architectures*, vol. 45, no. 11, pp. 863–885, May 1999.

[223] Standard Performance Evaluation Corporation (SPEC), "SPECjvm98 benchmark (V 1.04) in SPEC web site, [online document]," [Aug. 28, 2002] Available at: http://www.spec.org/osg/jvm98, Feb. 2001.

[224] P. R. Wilson, "Uniprocessor garbage collection techniques," in *Proceedings of the 1992 International Workshop on Memory Management*, St. Malo, France, Sept. 16–18, 1992, pp. 90–97.

Appendix A

Folding Notation

To ensure uniformity in this dissertation, the following folding notation is used (unless otherwise mentioned):

- Java programs are compiled into an intermediate representation targeting the JVM. Each JVM instruction is structured as a series of bytes called *Java bytecodes (JBCs), or bytecodes* for short, consisting of a one-byte opcode followed by zero or more arguments. Some opcodes could be prefixed with the one-byte `wide` modifier. The JVM instruction `iadd` is an example of an opcode that has no arguments. Another example for a JVM instruction is `wide iinc 92 300`. This instruction consists of the `wide` modifier, the opcode `iinc`, and the two arguments `92 300` are saved in two bytecodes each, giving a total of 6 bytecodes.
- The terms operands and arguments are used differently here. Operands are values popped from the stack, whereas arguments are taken from the JVM instruction.
- JVM instructions are described using the notation:

  $$[modifier]\ mnemonic\ argument : length\ [argument : length \cdots]$$

  Length is in bytes.
- $A_{u,v}$ describes an anchor instruction, where u ($u \in \{0, 1, 2, 3\}$) describes the number of required operands that are to be popped from the stack (if folding is not applied) and v ($v \in \{0, 1\}$) describes the number of results that are to be pushed on the stack (if folding is not applied). v indicates whether a consumer is required for the instruction (when $v = 1$) or not (when $v = 0$). When a certain type of anchors could have multiple values for u or v, we will use the set notation $[i, j]$ to denote the set $\{i, i + 1, \cdots, j\}$

- In the expression $\underbrace{P \cdots P}_{[1,2]} A_{[1,2],0}$, P's count (1 or 2 as in $[1,2]$) is equal to the number of operands required for A as indicated by the first suffix of A ($[1,2]$).
- The symbol $K_{I_x}(j)$ is used to describe instruction categories. It denotes a subcategory I of instruction category K that takes a parameter j (if applicable) and targets LV x or CN x. Symbols without suffixes denote all category members.
- In this dissertation, P is used to refer to both tagged and untagged producers, whereas Q is used to refer to tagged producers only. Similarly, C is used to refer to both tagged and untagged consumers, whereas T is used to refer to tagged consumers only.
- X means don't care.
- n/a means not applicable.
- $z \geq 0, z' \leq z + 1$.
- $\frac{a}{b}$ means a or b.
- — indicates a field that is not used.
- $u \rightarrow v$ denotes that u implies v.
- Folding group fields are described using the notation:

$$(operand : type : length),\ [(operand : type : length), \cdots] \Rightarrow result$$

Length is in bytes. X means any data type can be used. Some field manipulation functions will be used: **(1)** $Z(v, t, l, f)$ zero-extends v of type t and length l bytes to f bytes; **(2)** $S(v, t, l, f)$ sign-extends v of type t and length l bytes to f bytes; and **(3)** $T(v, t, l, f)$ truncates v of type t and length l bytes to f bytes. Although JVM constants are to be pushed as integers (4 bytes) onto the stack, our design only extends them as 2-byte integers to fit the processor internal instruction format. At EXs, constant values are indeed extended to 4 bytes. This restriction in constant sizes does not affect the correctness of executing Java programs as our benchmarking showed that all constants used had two or fewer bytes.

Appendix B

Sequencing Notation

Assume that A and B are different operations and C is a logical condition. The following notation is used in sequencing description. We use the word *enable* to mean enabled by an external signal or by satisfying a certain condition.

- Conditional execution: $A \rightarrow B$: If B is enabled, it can be executed if A is enabled also. A could be a compound logical expression that includes negation, anding $(\cdot)$ and/or oring $(+)$
- Conditional enable: $A \leftarrow C$: A is enabled if C is satisfied. C can be a complex logical condition.
- Exclusive enable: $A \oplus B$: A and B can not be enabled at the same time. The enabled one is executed.
- Sequential execution: A, B: A is executed first, followed by B (if both are enabled).
- Concurrent execution: $A|B$: Both A and B are executed simultaneously (if enabled).
- Priority execution: $A; B$: A has higher priority than B. That is, if both are enabled, A is executed and B is not.
- Input forwarding: $A \Rightarrow B$: If A is enabled, its input is forwarded to B (if B is enabled) whether A itself is executed or not.
- Sequential execution with output forwarding: $A \gg B$: If A is enabled, its output is forwarded to B. B has to be enabled in this case. Both A and B have to be executed in a sequential manner with A first.
- Parenthesizes have higher precedence.
- $\sum_{i=1}^{n} x(i)$ means $x(1) + \cdots + x(n)$.
- $\prod_{i=1}^{n} x(i)$ means $x(1) \cdot \cdots \cdot x(n)$.