JAVA VIRTUAL MACHINE FOR RESOURCE CONSTRAINED ENVIRONMENTS

A Thesis in
Computer Science and Engineering
by
Guangyu Chen

© 2006 Guangyu Chen

Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

August 2006
The thesis of Guangyu Chen was reviewed and approved by the following:

Mahmut T. Kandemir  
Associate Professor of Computer Science and Engineering  
Thesis Adviser  
Chair of Committee

Mary J. Irwin  
Professor of Computer Science and Engineering

Soundar R. T. Kumara  
Professor of Industrial and Manufacturing Engineering

Kenan Unlu  
Professor of Mechanical and Nuclear Engineering

Vijaykrishnan Narayanan  
Associate Professor of Computer Science and Engineering  
Graduate Program Officer of Computer Science and Engineering Department

*Signatures are on file in the Graduate School.
Abstract

Java-enabled mobile phones are dominating today’s market. Java is becoming the language of choice for personal mobile information devices such as smart phones and PDAs. This great success can be attributed to several factors, including portability, safety, ease of programming, and mature developer community. Java Virtual Machine (JVM) is the key component of Java technologies; its quality, in terms of energy efficiency, memory requirement, performance and reliability, has critical impact on the future success of Java technologies in the market of personal information devices. This thesis addresses the four critical issues in the design of a Java Virtual Machine for resource constrained devices: improving energy efficiency, reducing memory requirements, improving performance, and enhancing reliability.
# Table of Contents

List of Tables ........................................... x
List of Figures .......................................... xi
Acknowledgments ........................................ xiv

Chapter 1. Introduction .................................. 1
  1.1 Java Virtual Machine for Resource Constrained Systems .......................... 1
  1.2 Contribution of the Thesis ................................ 3
      1.2.1 Improving Energy Efficiency ................................ 3
      1.2.2 Reducing Memory Footprint ................................ 4
      1.2.3 Improving Performance ..................................... 5
      1.2.4 Enhancing Reliability ...................................... 6
  1.3 Organization of the Thesis ........................... 6

I Energy Aware Optimizations .......................... 7

Chapter 2. Tuning Garbage Collection for Reducing Memory System Energy ............ 8
  2.1 Introduction .......................................... 8
  2.2 KVM and Mark-and-Sweep Garbage Collector ........................................... 9
  2.3 Experimental Setup .................................... 12
      2.3.1 Banked Memory Architecture ............................... 12
      2.3.2 Energy Models ........................................... 14
      2.3.3 Benchmark Codes and Heap Footprints ..................... 16
  2.4 Energy Characterization and Optimization ............................................. 18
      2.4.1 Base Configuration ....................................... 18
      2.4.2 Impact of Mode Control ................................... 21
      2.4.3 Impact of Garbage Collection Frequency ..................... 22
      2.4.4 Impact of Object Allocation Style .......................... 26
      2.4.5 Impact of Compaction .................................... 28
      2.4.6 Impact of Number of Banks and Heap Size ................... 33
      2.4.7 Impact of Cache Memory ................................... 35
  2.5 Concluding Remarks .................................... 36
### Chapter 3. Energy Aware Adaptive Garbage Collection

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>39</td>
</tr>
<tr>
<td>3.2 Experimental Setup</td>
<td>42</td>
</tr>
<tr>
<td>3.3 Experimental Results</td>
<td>43</td>
</tr>
<tr>
<td>3.3.1 Heap Energy</td>
<td>43</td>
</tr>
<tr>
<td>3.3.2 Overall Energy</td>
<td>46</td>
</tr>
<tr>
<td>3.3.3 Execution Profiles</td>
<td>46</td>
</tr>
<tr>
<td>3.3.4 Performance Behavior</td>
<td>48</td>
</tr>
<tr>
<td>3.3.5 Bank Size Variations</td>
<td>48</td>
</tr>
<tr>
<td>3.3.6 Heap Size Variations</td>
<td>50</td>
</tr>
<tr>
<td>3.4 Related Work on Garbage Collection and Memory Energy Optimization</td>
<td>52</td>
</tr>
<tr>
<td>3.5 Concluding Remarks</td>
<td>53</td>
</tr>
</tbody>
</table>

### Chapter 4. Reducing Energy Consumption by Memory Compression

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>54</td>
</tr>
<tr>
<td>4.2 SoC Architecture</td>
<td>55</td>
</tr>
<tr>
<td>4.3 Compressing KVM Code and Class Libraries</td>
<td>57</td>
</tr>
<tr>
<td>4.4 Simulation Methodology</td>
<td>60</td>
</tr>
<tr>
<td>4.4.1 Energy Model</td>
<td>60</td>
</tr>
<tr>
<td>4.4.2 Base Configuration and Energy Distribution</td>
<td>61</td>
</tr>
<tr>
<td>4.5 Results</td>
<td>61</td>
</tr>
<tr>
<td>4.5.1 Sensitivity to the Decompression Cost</td>
<td>63</td>
</tr>
<tr>
<td>4.5.2 Sensitivity to the SPM Size</td>
<td>65</td>
</tr>
<tr>
<td>4.5.3 Sensitivity to the Block Size</td>
<td>66</td>
</tr>
<tr>
<td>4.5.4 Selective Compression</td>
<td>67</td>
</tr>
<tr>
<td>4.5.5 Clustering</td>
<td>69</td>
</tr>
<tr>
<td>4.6 Concluding Remarks</td>
<td>70</td>
</tr>
</tbody>
</table>

### Chapter 5. Offloading Computation/Compilation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Introduction</td>
<td>71</td>
</tr>
<tr>
<td>5.2 Target Platform and Benchmarks</td>
<td>75</td>
</tr>
<tr>
<td>5.3 Analysis of Execution and Compilation Strategies</td>
<td>80</td>
</tr>
<tr>
<td>5.3.1 Analysis of Static Strategies</td>
<td>81</td>
</tr>
<tr>
<td>5.3.2 Analysis of the Adaptive Execution Strategy</td>
<td>82</td>
</tr>
<tr>
<td>5.3.3 Analysis of Adaptive Compilation Strategy</td>
<td>91</td>
</tr>
<tr>
<td>5.3.4 Using Reconfigurable Datapath to Optimize Local Execution</td>
<td>92</td>
</tr>
<tr>
<td>5.3.5 Using Data Compression to Optimize of Remote Execution</td>
<td>95</td>
</tr>
<tr>
<td>5.4 Related Work</td>
<td>96</td>
</tr>
<tr>
<td>5.5 Concluding Remarks</td>
<td>99</td>
</tr>
</tbody>
</table>
Chapter 8. Field Level Analysis for Heap Space Optimization ............... 162
  8.1 Introduction .................................. 162
  8.2 Experimental Setup ............................. 163
  8.3 Field Access Characterization ...................... 164
    8.3.1 Field Lifetime Analysis .................... 164
      8.3.1.1 Life Cycle of Java Instances .......... 164
      8.3.1.2 Idle Field Ratio of Java Instances ....... 165
      8.3.1.3 Potential of Lifetime-Aware Space Optimization .... 165
      8.3.1.4 Determining the Last Access Time ........ 169
      8.3.1.5 Allocation Site Based Analysis .......... 171
    8.3.2 Disjointness Analysis ....................... 173
      8.3.2.1 Using Unions in Java .................. 173
      8.3.2.2 Memory Saving Potential of Using Unions .... 175
      8.3.2.3 Allocation Site Based Analysis .......... 177
  8.4 Realizing Memory Savings ........................ 177
  8.5 Related Work .................................. 184
  8.6 Concluding Remarks ............................. 188

III Reliability and Security 190
Chapter 9. Transient Error Tolerance .......................... 191
  9.1 Introduction .................................. 191
  9.2 Implementation ................................ 193
    9.2.1 Object Model ............................. 193
    9.2.2 Object Sharing ............................. 194
    9.2.3 Checkpointing and Rolling Back .............. 195
    9.2.4 Garbage Collection ......................... 196
      9.2.4.1 Local Collector ....................... 198
      9.2.4.2 Global Collector ....................... 199
    9.2.5 Handling Non-Dual-Executable Methods ....... 201
    9.2.6 Scheduling Java Threads .................... 204
  9.3 Experiments ................................... 204
    9.3.1 Experimental Setup ........................ 204
    9.3.2 Experimental Results ....................... 205
  9.4 Related Work ................................... 207
  9.5 Concluding Remarks ............................. 208
Chapter 10. Verifiable Annotations for JIT Compilation .......................... 209
  10.1 Introduction ............................................. 209
  10.2 Data Flow Analysis and Verification for Java Program .................. 211
    10.2.1 Flow Analysis Algorithms ............................ 211
    10.2.2 Verifying the Results of Flow Analysis .................. 213
  10.3 Array Bound Check Removal .................................. 214
    10.3.1 Definition of Frame States ............................ 215
    10.3.2 Forward Flow Analysis ................................ 216
    10.3.3 Annotation Generation and Verification .................... 222
  10.4 Null Pointer Check Removal .................................. 224
  10.5 Value Liveness Analysis ..................................... 227
    10.5.1 Backward Flow Analysis ............................... 228
    10.5.2 Annotation Generation and Verification .................... 228
  10.6 Experiments .............................................. 229
    10.6.1 Setup ............................................... 229
    10.6.2 Results .............................................. 230
  10.7 Related Work .............................................. 231
  10.8 Concluding Remarks ........................................ 234

References ..................................................... 235
List of Tables

2.1 Brief description of benchmarks used in our experiments. .................. 17
2.2 Energy and performance overhead of bank turning-off. .................... 25

3.1 The number of GC activations for each benchmark. ......................... 47
3.2 Percentage increase in execution cycles (with respect to the base collector). 50

4.1 Simulation parameters and their values for our base configuration. ...... 62
4.2 Energy consumptions for our applications under the base configuration. 63
4.3 Execution cycles and SPM misses for our applications under the base configuration. 65

5.1 Energy consumption values for processor core and memory. ............... 77
5.2 Description of our benchmarks. ..................................... 79
5.3 Summary of the static and dynamic (adaptive) strategies. ................. 82
5.4 Distribution of channel conditions and size parameter values in 24 different scenarios. ................................................. 88
5.5 Local and remote compilation energies. .................................. 93

6.1 Garbage collection strategies. ........................................... 114
6.2 Java benchmarks. ...................................................... 116
6.3 Heap-related behavior of our benchmarks. .................................. 117
6.4 Minimum heap sizes to execute benchmarks without an out-of-memory exception. 118
6.5 The number of compressions and decompressions using MCCL. .......... 121
6.6 The number of compressions and decompressions for MCC and MCCL. ... 122

7.1 Important characteristics of the benchmarks. ............................... 142
7.2 Determining the potential levels for the fields of three example classes. 147

9.1 The Java benchmarks for our experiments. ................................ 205
9.2 Minimum heap space requirement for our transient fault tolerant JVM. ... 206

10.1 Backward flow function for the value liveness analysis. .................... 228
10.2 Our Java applications. ............................................... 230
10.3 Results for the null check removal experiments. ........................... 231
10.4 Results for the array bound check removal experiments. .................. 232
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Operation of garbage collector and compactor</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Major components of our SoC. Note that cache memories are optional.</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Simulation environment</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Heap footprints</td>
<td>19</td>
</tr>
<tr>
<td>2.5</td>
<td>Energy distribution</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td>Normalized energy consumption in heap and stack due to mode control (M&amp;S).</td>
<td>22</td>
</tr>
<tr>
<td>2.7</td>
<td>Percentage distribution of active banks (M&amp;S).</td>
<td>23</td>
</tr>
<tr>
<td>2.8</td>
<td>Normalized energy consumption in heap and ROM with different GC frequencies.</td>
<td>24</td>
</tr>
<tr>
<td>2.9</td>
<td>Normalized energy consumption in heap (active bank allocation versus default allocation).</td>
<td>26</td>
</tr>
<tr>
<td>2.10</td>
<td>Energy consumption in heap due to mode control (M&amp;C) normalized with respect to M&amp;C without mode control.</td>
<td>28</td>
</tr>
<tr>
<td>2.11</td>
<td>Comparison of M&amp;C and M&amp;S, and different compacting collectors.</td>
<td>30</td>
</tr>
<tr>
<td>2.12</td>
<td>Energy distribution in core due to GC.</td>
<td>34</td>
</tr>
<tr>
<td>2.13</td>
<td>Impact of number of banks and heap size.</td>
<td>36</td>
</tr>
<tr>
<td>2.14</td>
<td>Impact of cache memory</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>Garbage collector is invoked at T1, T2, T3 and T4.</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>Garbage collection (c_1) and (c_2) are invoked at (t_1) and (t_2) respectively.</td>
<td>41</td>
</tr>
<tr>
<td>3.3</td>
<td>Normalized energy consumption (static+dynamic) in the heap memory.</td>
<td>44</td>
</tr>
<tr>
<td>3.4</td>
<td>Normalized energy consumption breakdown.</td>
<td>45</td>
</tr>
<tr>
<td>3.5</td>
<td>Execution footprints of Calculator, Dragon, and ManyBalls under different collection frequencies.</td>
<td>49</td>
</tr>
<tr>
<td>3.6</td>
<td>Results with different bank sizes: Top: Kvideo, Bottom: Dragon.</td>
<td>51</td>
</tr>
<tr>
<td>3.7</td>
<td>Results with different heap sizes (Kvideo).</td>
<td>52</td>
</tr>
<tr>
<td>4.1</td>
<td>SoC memory architecture</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>SoC memory architecture with decompressor</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Normalized energy consumption</td>
<td>64</td>
</tr>
<tr>
<td>4.4</td>
<td>Normalized energy consumption in read-only memory with varying decompression costs.</td>
<td>66</td>
</tr>
<tr>
<td>4.5</td>
<td>Normalized energy consumption in read-only memory with varying SPM sizes.</td>
<td>67</td>
</tr>
<tr>
<td>4.6</td>
<td>Normalized energy consumption in read-only memory with varying block sizes.</td>
<td>68</td>
</tr>
<tr>
<td>4.7</td>
<td>Compression rate of Modified X-match algorithm with varying block sizes.</td>
<td>68</td>
</tr>
</tbody>
</table>
4.8 Comparison of different compression strategies. ................... 69
4.9 Effectiveness of clustering. ........................................... 70

5.1 An execution environment based on composable Java components. ........ 74
5.2 Communication system architecture and the energy parameter for each compo-
nent. ................................................................. 78
5.3 Energy consumption of eight benchmarks with static execution strategies. ...... 83
5.4 Curve fitting for execution energy estimation using Local1. .................... 85
5.5 Average of normalized energy consumptions. ..................................... 87
5.6 Energy consumptions of different execution/compilation strategies. ............. 89
5.7 Co-processor with reconfigurable datapath. .................................... 95
5.8 Impact of reconfigurable datapath on energy consumptions. ..................... 96
5.9 The breakdown of remote execution energy costs. ................................ 97

6.1 Referencing an object through a handle. .................................... 105
6.2 Expansion of the permanent space. .......................................... 106
6.3 Formats of an object in different states. .................................... 107
6.4 Compressing the heap. .................................................... 107
6.5 Zero-removal compression algorithm. ....................................... 109
6.6 Example compression. ...................................................... 110
6.7 Decompressing an object. ................................................... 111
6.8 Decompression algorithm. ................................................. 112
6.9 Breaking down a large array into subarrays. .................................. 113
6.10 The sizes of the live objects. .............................................. 119
6.11 Heap memory usage of MCCL. ........................................... 123
6.12 Heap memory usage of MCCL (continued). ............................... 124
6.13 Heap memory usage of MCCL (continued). ................................ 125
6.14 Runtime overheads due to MCCL with the minimum heap size for MCCL. .... 127
6.15 Runtime overheads due to MCCL and MC with the minimum heap size for MC. . 128
6.16 Runtime overheads due to MCCL and MC with 150% of the minimum heap size for MC. . . .................................................. 129
6.17 Impact of the threshold parameter \( k \) on performance. .................... 131
6.18 Impact of the threshold parameter \( k \) on heap behavior. .................... 132
6.19 Division of a Java object into reference and non-reference zones. ............ 135
6.20 Decompression of an object when no handle is used. ........................ 135
6.21 Runtime overheads due to MCCL+. ...................................... 136

7.1 Frequent value distribution. .............................................. 144
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2 Zero and non-zero frequent values.</td>
<td>145</td>
</tr>
<tr>
<td>7.3 Object formats for Scheme-1.</td>
<td>151</td>
</tr>
<tr>
<td>7.4 Object formats for Scheme-2.</td>
<td>152</td>
</tr>
<tr>
<td>7.5 Maximum heap occupancy.</td>
<td>155</td>
</tr>
<tr>
<td>7.6 Heap occupancy profiles for two benchmarks with s1 input: jess and raytrace.</td>
<td>156</td>
</tr>
<tr>
<td>7.7 Average heap occupancy for Scheme-2.</td>
<td>157</td>
</tr>
<tr>
<td>7.8 Percentage increases due to Scheme-1 (S1) and Scheme-2 (S2) in execution cycles over the base results.</td>
<td>158</td>
</tr>
<tr>
<td>7.9 Increase in the maximum heap occupancy of Scheme-2 when a product-based potential field selection scheme is used.</td>
<td>160</td>
</tr>
<tr>
<td>7.10 Maximum heap occupancy of Scheme-2 with the s10 input.</td>
<td>161</td>
</tr>
<tr>
<td>8.1 Instance and field lifetimes.</td>
<td>163</td>
</tr>
<tr>
<td>8.2 CDFs for idle field ratio.</td>
<td>166</td>
</tr>
<tr>
<td>8.3 Heap occupancies for object instances.</td>
<td>168</td>
</tr>
<tr>
<td>8.4 Normalized maximum and average heap occupancy when using object/field lifetime analysis.</td>
<td>170</td>
</tr>
<tr>
<td>8.5 Comparing the heap occupancy of FFL against OFL.</td>
<td>171</td>
</tr>
<tr>
<td>8.6 Percentage of last used fields as a function of the number of accesses.</td>
<td>172</td>
</tr>
<tr>
<td>8.7 Idle space-time product for allocation sites.</td>
<td>174</td>
</tr>
<tr>
<td>8.8 Determining the union format for allocation sites.</td>
<td>178</td>
</tr>
<tr>
<td>8.9 Normalized maximum and average heap occupancies when using unions.</td>
<td>179</td>
</tr>
<tr>
<td>8.10 Reduction in allocated memory size when using merged unionization at the first 50 allocation sites.</td>
<td>180</td>
</tr>
<tr>
<td>8.11 Modifying the source code of jess to reduce heap memory requirements.</td>
<td>183</td>
</tr>
<tr>
<td>8.12 Modifying the source code of raytrace to reduce heap memory requirements.</td>
<td>185</td>
</tr>
<tr>
<td>8.13 Heap occupancies of the original and modified applications.</td>
<td>186</td>
</tr>
<tr>
<td>9.1 Object model for our transient fault tolerant JVM.</td>
<td>194</td>
</tr>
<tr>
<td>9.2 The architecture of the transient fault tolerant JVM.</td>
<td>196</td>
</tr>
<tr>
<td>9.3 Writing to an object in $H_g$.</td>
<td>197</td>
</tr>
<tr>
<td>9.4 Global garbage collection.</td>
<td>202</td>
</tr>
<tr>
<td>9.5 Normalized execution times of our transient fault tolerant JVM.</td>
<td>207</td>
</tr>
<tr>
<td>10.1 JIT compilation with/without verifiable annotations.</td>
<td>211</td>
</tr>
<tr>
<td>10.2 The match function for the array bound check removal annotation.</td>
<td>217</td>
</tr>
<tr>
<td>10.3 Forward flow analysis for the array bound check removal.</td>
<td>223</td>
</tr>
<tr>
<td>10.4 Annotation generation algorithm.</td>
<td>225</td>
</tr>
<tr>
<td>10.5 Compilation with annotation verification.</td>
<td>226</td>
</tr>
<tr>
<td>10.6 Backward flow algorithm for value liveness analysis.</td>
<td>227</td>
</tr>
<tr>
<td>10.7 Compilation with the value lifetime annotation.</td>
<td>229</td>
</tr>
</tbody>
</table>
Acknowledgments

I am most grateful and indebted to my thesis advisor, Mahmut Kandemir, for the large doses of guidance, patience, and encouragement he has shown me during my time here at Penn State. I am also grateful and indebted to Vijaykrishnan Narayanan, for inspiration and enlightening discussions on a wide variety of topics. I am especially indebted for the financial support which they have provided to me over the years. I thank my other committee members, Mary J. Irwin, Soundar R. T. Kumara, and Kenan Unlu, for their insightful commentary on my work.
Chapter 1

Introduction

1.1 Java Virtual Machine for Resource Constrained Systems

The market for mobile devices and phones is continuing to increase at a rapid rate. For example, the handheld mobile device market in the U.S. is currently increasing at an annual rate of 22% [19]. In contrast to the PC market, several products and companies compete for this market share. Since the ability to support dynamic software content is a major factor in determining the success of mobile devices, many of the mobile device manufacturers are increasingly adopting Java technology. A recent report projects that Java will be the dominant terminal platform in the wireless sector, being supported by over 450 million handsets in 2007, corresponding to 74% of all wireless phones that will ship that year [95].

Using the Java technology on personal information devices has several important benefits [10]. First, Java is cross-platform compatible. As a result, Java code can run smoothly without modification on a wide range of devices. Such cross-platform compatibility is especially important for the diverse mobile device market that is shaped by a variety of devices executing different operating systems. Second, Java enhances user experience by supporting rich GUI components and by providing dynamic downloading capability. Further, Java security model allows users to run these applications on their devices safely. Finally, Java has a very mature developer community. The developer talent needed for Java devices already exists and is readily available.

Java applications are executed by Java Virtual Machines (JVMs). The quality of JVM (performance, memory footprint, energy efficiency, etc) has critical impact on the user feelings about the Java applications. Performance is the primary designing concern for the JVMs running on the systems with plenty of resources (such as desktop systems and high end servers). When designing a JVM for embedded mobile systems (such as PDAs and mobile phones), however, memory footprint and energy efficiency are as important concerns as the performance. Energy
efficiency determines how long the users can use their devices without recharging the batteries. Memory footprint determines whether a particular application can be used on a given device. Specifically, an application cannot not run properly on a device whose available memory is smaller than the memory footprint of this application. Note that many mobile devices have stringent budget as memory has severe implications on the cost, form factor, and energy consumption of the device. The budget for the memory has a significant impact on the overall cost of a device. For example, currently, a Palm m130 PDA with an 8MB memory costs $199 as compared to $299 cost for a similar configuration with a 16MB memory [14]. Similarly, as compared to the base price of $99 for a Palm Zire PDA with 2MB memory, upgrading its memory to 8MB adds an additional $59 [1]. In addition to the cost factor, a larger memory also demands a larger form factor. In fact, many cell phone companies need to resort to more costly packaging techniques to incorporate larger memories in a smaller space. A less expensive solution would be to reduce the memory demand through application tuning and optimization [96]. Furthermore, the reduction in memory requirements for a single application can be exploited in a multiprogrammed environment to support more concurrent applications. Finally, the larger the memory the more the energy demand (in both active and idle states) [92]. Due to these underlying reasons, many low-end mobile devices such as cell phones (e.g., Casio CdmaOne C452CA [4]) typically support less than 1MB of memory, of which the Java Virtual Machine (JVM) may have access to a even smaller portion.

Ideally, users want a JVM for embedded system to run faster, occupy smaller memory, and consume less energy. In reality, however, energy consumption, memory footprint and execution time of a JVM are usually entangled together, which makes the tradeoffs among these three factors very complex. For example, reducing the size of available memory reduces the overall power consumption of the system. On the other hand, reducing memory size may cause the garbage collector to be invoked more frequently, and thus may increase the execution time and energy consumption of the application. The goal of our proposed research is to develop the technologies that can achieve the best balance point among the three factors: energy consumption, memory footprint, and execution time.

In addition to achieving the optimal tradeoffs among performance, energy consumption, and memory footprint, I also propose a scheme for implementing a transient-fault-tolerant JVM
for resource constrained systems. Today’s personal mobile systems are becoming more and more powerful. Many desktop applications have been migrated to the small devices. Many non-trivial Java applications, such as web browsers and email processors, are popular on the devices such as mobile phones and PDAs. In the near future, one can even see mission-critical applications (e.g., on-line transaction processing) running on mobile-phone-like devices. Following this trend, fault-tolerance will be an important issue for Java-enabled embedded environments. Many solutions have been proposed for improving the reliability of computer systems. For example, dual-execution techniques enable us detect the transient errors in the datapath [63, 107]. However, such solutions require special hardware support, and thus, increase the overall cost of the system. Embedded systems are usually sold in huge quantities and thus tend to be more sensitive to the per device cost as compared to their high-performance counterparts. Consequently, the existing fault-tolerance solutions for high-end systems may not be attractive for low-cost embedded systems.

1.2 Contribution of the Thesis

In this thesis, I address the four critical issues in the design of a Java Virtual Machine for resource constrained devices: improving energy efficiency, reducing memory requirements, improving performance, and enhancing reliability.

1.2.1 Improving Energy Efficiency

I proposed strategies for reducing the energy consumption of a JVM from two different angles: memory management and Java bytecode execution. In Chapter 2, I propose using the garbage collector to detect and safely turn off the memory banks that do not contain any live objects to conserve energy. In this research, I observed that the frequency with which garbage collection is invoked significantly affects the overall energy consumption of the memory subsystem. As the optimal garbage collection frequency is dependent on the particular Java application and the user input, it is hard to determine the optimal frequency statically. In Chapter 3, I further propose an adaptive strategy that automatically determines at runtime the optimal garbage collection frequencies for different Java applications. Besides garbage-collection-controlled memory...
banks, I also proposed using code compression to reduce leakage energy consumption of the main memory (Chapter 4).

Java applications are shipped in platform independent bytecode. JVM executes byte-codes either through interpretation or via Just-In-Time (JIT) compilation. In addition to interpretation and JIT compilation, I exploit two new execution strategies – remote execution and remote compilation – with the goal of reducing the energy consumption of wireless connected mobile embedded devices (Chapter 5). In the remote execution strategy, the mobile device sends the arguments and the name of a Java method to a resource-rich server over the wireless connection; the server executes the method and sends back the return values. In the remote compilation strategy, the mobile device sends the name of a Java method to the server, and the server compiles the method into the native code of the mobile device and sends back the compiled code. These two strategies exploit the energy tradeoffs between local computation and remote communication to reduce the overall energy consumption of mobile embedded devices. Since the cost for data communication is dependent on the dynamically changing network conditions, e.g., the distance to the server and the number of mobile devices within the range, I also proposed runtime strategies that dynamically decide where to execute each execution/compilation task - locally on the mobile client or remotely on the server.

1.2.2 Reducing Memory Footprint

The memory footprint (i.e., the size of memory space required to execute an application) is one of the primary factors that limit the number of Java applications that can be executed on resource-constrained embedded devices. Although the cost of memory is decreasing, having a large memory in a low cost embedded device is still a luxury. In Chapter 6, I propose and evaluated a technique that reduces the memory requirement of Java applications by compressing the objects in the heap space. In this work, I added an additional compression phase to a conventional mark/sweep garbage collector. Without this compression phase, the JVM usually terminates the application with an “out of memory” exception when the application uses up all the space in the heap. In contrast, by using the heap compression, the JVM gains an extra opportunity to continue with code execution. Specifically, when the application uses up the heap space and the garbage collector cannot collect sufficient free space to allocate the new object, a
heap compressor is invoked to compress the live objects in the heap in order to create extra free space in order to satisfy the current allocation request. Later, when the contents of a compressed object are needed by the application, a decompressor will be invoked to decompress that object. My research results show that, on average, the proposed heap compression technique reduces memory demand of embedded Java applications by 35% with a small cost of 2% performance degradation. In addition to heap compression, I also investigate the characteristics of the life cycles of the fields of Java objects (Chapter 8). This research reveal that, if the heap space were managed in a finer granularity than an object, the memory demands of many Java applications could be significantly reduced. In Chapter 7, I exploit the values that frequently appear in certain fields of Java objects to reduce the memory requirements of embedded Java applications.

1.2.3 Improving Performance

The performance of Java applications directly affects user trends about Java-enabled devices. JIT compilation has been used in many JVM implementations to improve the performance of Java applications. In JVM implementations for high performance systems, the JIT compilers perform extensive optimizations to improve the quality of the generated native code. However, an optimizing compiler is not very suitable for resource-constrained environments due to its large memory requirements. Therefore, the JIT compilers used in today’s embedded JVMs are usually light-weighted, which means that they can only perform a very limited set of optimizations. To address this problem, I propose a verifiable annotation mechanism to support optimizing JIT compilation in an embedded environment (Chapter 10). Specifically, one can employ an offline analyzer to perform expensive analyses. The results of these analyses are then incorporated into the Java class files as annotations. At runtime, these annotations guide the compiler to generate optimized code without performing expensive analyses. Annotation-guided JIT compilation has been studied by several researchers. My work differs from these prior efforts in that the annotations I proposed can be verified at runtime by a lightweight verifier against the security model specified by the JVM specification. As a result, a malicious annotation generator cannot cheat the JIT compiler into generating native code that violates the JVM security model. Such verification is necessary for environments where information security is important.
1.2.4 Enhancing Reliability

In Chapter 9, I propose a software-based fault tolerant embedded JVM to address the needs of personal embedded systems. The proposed transient fault tolerant JVM does not require expensive hardware support. Instead, it relies on the memory page protection mechanism, which is currently available in the memory management unit of many embedded processors. It can detect most errors in both the memory and the processor core, and recover from most of the errors due to transient faults in the processor core. The memory overheads due to the fault tolerance mechanism built in this JVM are moderate and the performance overheads are within tolerable limits.

1.3 Organization of the Thesis

The rest of this thesis is divided into three parts. The first part is focused on energy optimization, including garbage collector controlled memory banks (Chapter 2), energy-aware adaptive garbage collection (Chapter 3), compressing read-only memory for reducing energy consumption (Chapter 4), and energy-aware bytecode execution strategies (Chapter 5). The second part is focused on reducing memory footprint, including compressing Java objects in the heap (Chapter 6), exploiting the frequent values in the fields of Java objects (Chapter 7), and an investigation on the potential of saving heap memory by exploiting and the life cycles of the fields of Java objects (Chapter 8). The last part discusses the issues relevant to reliability and security. Specifically, Chapter 9 presents the design of a JVM that can tolerate transient hardware errors, and Chapter 10 presents verifiable annotations that can be safely used by a JIT compiler to improve performance.
Part I

Energy Aware Optimizations
Chapter 2

Tuning Garbage Collection for Reducing Memory System Energy

2.1 Introduction

The energy consumption in the memory system is a significant portion of overall energy expended in execution of a Java application [125]. There are two important components of memory energy: dynamic energy and leakage energy. Dynamic energy is consumed whenever a memory array is referenced or precharged. Recent research has focused on the use of memory banking and partial shutdown of the idle memory banks in order to reduce dynamic energy consumption [53, 85]. However, leakage energy consumption is becoming an equally important portion as supply voltages and thus threshold voltages and gate oxide thicknesses continue to become smaller [38]. Researchers have started to investigate architectural support for reducing leakage in cache architectures [132, 79]. In this chapter, we show that it is possible to also reduce leakage energy in memory by shutting down idle banks using an integrated hardware-software strategy.

The garbage collector (GC) [73] is an important part of the JVM and is responsible for automatic reclamation of heap-allocated storage after its last use by a Java application. Various aspects of the GC and heap subsystems can be configured at JVM runtime. This allows control over the amount of memory in the embedded device that is available to the JVM, the object allocation strategy, how often a GC cycle is triggered, and the type of GC invoked. We exploit the interaction of these tunable parameters along with a banked-memory organization to effectively reduce the memory energy (leakage and dynamic) consumption in an embedded Java environment. Garbage collection is a memory-intensive operation and directly affects application performance, and its impact on performance has been a popular research topic (e.g., see [73] and the references therein). In an embedded/portable environment, however, its impact on energy should also be taken into account. There are three questions we need to take into consideration when designing garbage collectors for energy sensitive systems:
• Since garbage collector itself consumes energy, how to reduce energy consumption during GC?

• Since garbage collector scans the memory, very detailed information about current memory usage can be obtained with a relatively small overhead right after each GC invocation. How can we make use of this information to reduce memory energy consumption?

• Some garbage collectors move objects to compact the heap. Is it possible to relocate objects during compaction phase to further enhance memory energy savings?

This chapter studies the energy impact of various aspects of a mark-and-sweep (M&S) garbage collector, which is commonly employed in current embedded JVM environments, in a multi-bank memory architecture. The experiments are carried out using two different (compacting and non-compacting) collectors in Sun’s embedded JVM called KVM [109, 84]. Further, the virtual machine is augmented to include features that are customized for a banked-memory architecture. We also measure the sensitivity of energy behavior to different heap sizes, cache configurations, and number of banks. In order to investigate the energy behavior, we gathered a set of thirteen applications frequently used in hand-held and wireless devices. These applications include utilities such as calculator and scheduler, embedded web browser, and game programs. We observe that the energy consumption of an embedded Java application can be significantly more if the GC parameters are not tuned appropriately. Further, we notice that the object allocation pattern and the number of memory banks available in the underlying architecture are limiting factors on how effectively GC parameters can be used to optimize the memory energy consumption.

2.2 KVM and Mark-and-Sweep Garbage Collector

K Virtual Machine (KVM) [84, 109] is Sun’s virtual machine designed with the constraints of inexpensive embedded/mobile devices in mind. It is suitable for devices with 16/32-bit RISC/CISC microprocessors/controllers, and with as little as 160 KB of total memory available, 128 KB of which is for the storage of the actual virtual machine and libraries themselves. Target devices for KVM technology include smart wireless phones, pagers, mainstream personal digital assistants, and small retail payment terminals. The KVM technology does not support Java
Native Interface (JNI). The current implementation is interpreter-based and does not support JIT (Just-in-Time) compilation.

An M&S collector makes two passes over the heap. In the first pass (called mark pass), a bit is marked for each object indicating whether the object is reachable (live). After this step, a sweep pass returns unreachable objects (garbage) to the pool of free objects. M&S collectors are widely used due to their ease of implementation and simple interface. As compared to other garbage collectors such as reference counting and generational collectors [73], the M&S collector has both advantages and disadvantages. For example, no write-barrier overhead is necessary in M&S collectors while reference counting collectors rely on write-barrier mechanism to maintain reference counters. Similarly, generational collectors rely on write-barriers to keep track of inter-generational references. Further, in many real implementations of reference counting and generational collectors, M&S collectors are still used to resolving cyclic references and for collecting objects in older generations respectively.

The KVM implements two M&S collectors, one without compaction and one with compaction [109]. In the non-compacting collector, in the mark phase, all the objects pointed at by the root objects, or pointed at by objects that are pointed at by root objects are marked live. This is done by setting a bit in the object’s header called MARK BIT. In the sweep phase, the object headers of all objects in the heap are checked to see if the MARK BIT was set during the mark phase. All unmarked objects (MARK BIT=0) are added to the free list and for the marked objects (MARK BIT = 1), the MARK BIT is reset. While allocating a new object, the free list is checked to see if there is a chunk of free memory with enough space to allocate the object. If there is not, then garbage collector is called. After garbage collection (mark and sweep phases), object allocation is tried again. If there is still not any space in the heap, an out-of-memory exception is thrown. Because this collector does not move objects in memory, the heap can easily get fragmented and the virtual machine may run out of memory quickly.

In an embedded environment, this heap fragmentation problem brings up two additional issues. First, because the memory capacity is very limited, we might incur frequent out-of-memory exceptions during execution. Second, a fragmented heap space means more active banks (at a given time frame) and, consequently, more energy consumption in memory. Both
of these motivate for compacting live objects in the heap. Compacting heap space, however, consumes both execution cycles and extra energy which also need to be accounted for.

A garbage collector with compaction works as follows. When a new object is to be allocated in the heap, the free list is checked to see whether there is a chunk of free memory with enough space to allocate the object. If there is not, the garbage collector is called. During garbage collection (after sweep phase), it is checked whether the largest free chunk of memory (obtained after sweep phase) satisfies the size to be allocated. If not, the collector enters compaction phase. During compaction, all live objects are moved to one end of the heap so that the free blocks can be combined to form a large free in the other end of the heap. After compaction, object allocation is attempted again. If there still is not any space, an out-of-memory exception is signaled.

The default compaction algorithm in KVM is a Break Table-based algorithm [73]. In this method, instead of using an additional field in the object’s header to store the new address, a table containing relocation information is constructed in the free space. Thus, there is no extra space used to keep track of the updated addresses of the objects that are moved during the compaction. This table is called the Break Table. The live objects are moved to one end of the heap, and as they are moved, an entry is made in the Break Table consisting of two fields: (i) the old start address of the object and (ii) the total free space found until then. The Break Table may need to be shifted around if it gets in the way as live objects get compacted. If the Break Table rolls, it is sorted. After the objects are shifted, the pointers within the live objects are updated to point to the new address of the object. Advantages of this algorithm are that no extra space is needed to maintain the relocation information, objects of all sizes can be handled, and the order of object allocation is maintained. The disadvantage is that both sorting the break table and updating the pointers are costly operations both in terms of execution time and energy.

In the rest of this chapter, we refer to the compacting and the non-compacting collectors as M&S and M&C, respectively. It should be noted that both the collectors are not optimal in the sense that they do not reclaim an object immediately after the object becomes garbage as an object is not officially garbage until it is detected to be so.

Fig. 2.1 shows the operation of garbage collection and compaction in our banked memory architecture that contains four banks for the heap. Each step corresponds a state of the heap after an object allocation and/or garbage collection/compaction. Step 0 corresponds to initial state
where all banks are empty and thus turned off. In Step 1, object A is allocated and in Step 10, two more objects (B and C) are allocated. In Step 50, object B becomes garbage and three new objects (D, E, and F) are allocated. In Step 100, both D and E become garbage and G is allocated. Note that at this point all the banks are active despite the fact that Bank 2 holds only garbage. In Step 200, the garbage collector is run and objects B, D, and E are collected and their space is returned to free space pool. Subsequently, since Bank 2 does not hold any live data, it can be turned off. In Step 500, object C in Bank 1 becomes garbage. Finally, Step 1000 illustrates what happens when both garbage collection and compaction are run. Object C is collected, live objects A, G, and F are clustered in Bank 0, and Banks 1 and 3 can be turned off. Two points should be emphasized. Energy is wasted in Bank 2 between steps 100 and 200 maintaining dead objects. Thus, the gap between the invocation of the garbage collection and the time at which the objects actually become garbage is critical in reducing wasted energy. Similarly, between steps 500 and 1000, energy is wasted in Banks 1 and 3 because the live objects that would fit in one bank are scattered in different banks. This case illustrates that compaction can bring additional energy benefits as compared to just invoking the garbage collector.

2.3 Experimental Setup

2.3.1 Banked Memory Architecture

The target architecture we assume is a system-on-a-chip (SoC) as shown in Fig. 2.2. The processor core of the system is based on the microSPARC-IleⅠep embedded processor. This core is a 100MHz, 32-bit five-stage pipelined RISC architecture that implements the SPARC architecture v8 specification. It is primarily targeted for low-cost uniprocessor applications. The target architecture also contains on-chip data and instruction caches that can be selectively enabled. Further, it contains an on-chip ROM and an on-chip SRAM. Fig. 2.2 also shows both logical and physical views of the portion of the memory system of interest. This portion is divided into three logical parts: the KVM code and class libraries, the heap that contains objects and method areas, and the non-heap data that contains the runtime stack and KVM variables. Typically, the KVM code and the class libraries reside in a ROM. The ROM size we use is 128 KB for the storage of the actual virtual machine and libraries themselves [84]. Since, not all
libraries are used by every application, banked ROMs can provide energy savings. We activate the ROM partitions only on the first reference to the partition. A ROM partition is never disabled once it has been turned on. This helps to reduce the leakage energy consumption in memory banks not used throughout the application execution. While it may be possible to optimize the energy consumed in the ROM further using techniques such as clustering of libraries, in this study, we mainly focus only on the RAM portion of memory (SRAMs are commonly used in embedded environments as memory modules) which holds the heap. The heap (a default size of 128KB) holds both application bytecodes and application data, and is the target of our energy management strategies. An additional 32KB of SRAM is used for storing the non-heap data.

We assume that the memory space is partitioned into banks and depending on whether a heap bank holds a live object or not, it can be shutdown. Our objective here is to shutdown as many memory banks as possible in order to reduce leakage and dynamic energy consumption. Note
Fig. 2.2. Major components of our SoC. Note that cache memories are optional.

Fig. 2.3. Simulation environment.

that the operating system is assumed to reside in a different set of ROM banks for which no optimizations are considered here. Further, we assume a system without virtual memory support.

2.3.2 Energy Models

For obtaining detailed energy profiles, we have customized an energy simulator and analyzer using the Shade [48] (SPARC instruction set simulator) tool-set and simulated the entire KVM executing a Java code, as shown in Fig. 2.3. Shade is an instruction-set simulator and custom trace generator. Application programs are executed and traced under the control of a user-supplied trace analyzer. Current implementations run on SPARC systems and, to varying degrees, simulate the SPARC (Versions 8 and 9) and MIPS I instruction sets.
Our simulator tracks energy consumption in the processor core (datapath), on-chip caches, and the on-chip SRAM and ROM memories. The datapath energy is further broken into energy spent during execution and energy spent during GC. The GC energy, itself, is composed of energy spent in mark phase, sweep phase, and compaction phase (if used). Similarly, the memory energy is divided into three portions: energy spent in accessing KVM code and libraries, energy spent in accessing heap data, and energy spent in accessing the runtime stack and KVM variables. The simulator also allows the user to adjust the various parameters for these components. Energies spent in on-chip interconnects are included in the corresponding memory components.

The energy consumed in the processor core is estimated by counting (dynamically) the number of instructions of each type and multiplying the count by the base energy consumption of the corresponding instruction. The energy consumption of the different instruction types is obtained using a customized version of our in-house cycle accurate energy simulator [123]. The simulator is configured to model a five-stage pipeline similar to that of the microSPARC-IIep architecture. The energies consumed by caches are evaluated using an analytical model that has been validated to be highly accurate (within 2.4% error) for conventional cache systems [77]. All energy values reported in this chapter are based on parameters for 0.10 micron, 1V technology.

In our model, a memory bank is assumed to be in one of three modes (i.e., states) at any given time. In the read/write mode, a read or write operation is being performed by the memory bank. In this mode, dynamic energy is consumed due to precharging the bitlines and also in sensing the data for a read operation. For a write operation, dynamic energy is consumed due to the voltage swing on the bitlines and in writing the cells. In the active mode, the bank is active (i.e., holds live data) but is not being read or written. In this mode, we consume dynamic precharge energy as there is no read or write into the bank. In addition, leakage energy is consumed in both these modes. Finally, in the inactive mode, the bank does not contain any live data. Thus, the bank is not precharged.
control mechanism to reduce the leakage current. Thus, a bank in this mode consumes only a small amount of leakage energy and no dynamic energy.

In optimizing leakage current, we modify the voltage down converter circuit [75] already present in current memory chip designs to provide a gated supply voltage to the memory bank. Whenever the Sleep signal is enabled, the supply to the memory bank is cut off, thereby essentially eliminating leakage in the memory bank. Otherwise, the Gated $V_{DD}$ signal follows the input supply voltage ($V_{DD}$). The objective of our optimization strategy is to put as many banks (from the heap portion of memory) as possible into the inactive mode (so that their energy consumption can be optimized). This can be achieved by compacting the heap, co-locating objects with temporal affinity, invoking the garbage collector more frequently, adopting bank-aware object allocation strategies, or a combination of these as will be studied in detail in Section 2.4. When a bank in the inactive mode is accessed to allocate a new object, it incurs a penalty of 350 cycles to service the request. The turn-on times from the inactive mode are dependent on the sizing of the driving transistors. Note that the application of this leakage control mechanism results in the data being lost. This does not pose a problem in our case as the leakage control is applied only to unused (inactive) banks.

### 2.3.3 Benchmark Codes and Heap Footprints

In this study, we used thirteen applications ranging from utility programs used in handheld devices to wireless web browser to game programs. These applications are briefly described in Table 2.1. The first number in the third column of each application gives the maximum live footprint of the application; i.e., the minimum heap size required to execute the application without an out-of-memory error if garbage is identified and collected immediately. The actual heap size required for executing these applications are much larger using the default garbage collection mechanism without compaction. For example, Kwml requires a minimum heap size of 128KB to complete execution without compaction. The second number in the third column of each application in the figure gives the effective live heap size; that is, the average heap size occupied by live objects over the entire duration of the application’s execution. A more detailed characterization of the live heap size over the entire application execution is shown in Fig. 2.4. It should be noted that y-axis in these graphs represents the total size of live objects currently in the
heap, not the actual memory usage of each application. Lack of variation in some graphs does not necessarily mean the memory usage of the application remains unchanged. Instead, it means that the objects of the application die as quickly as they are created. Actually, y-axis indicates the minimal memory requirement of each application, which determines the potential of shutting down portions of the heap memory. However, the ability to exploit this potential depends on various factors. These factors include the bank size, the garbage collection frequency, object allocation style, compaction style, and compaction frequency as will be discussed in the next section.

<table>
<thead>
<tr>
<th>Application</th>
<th>Brief Description</th>
<th>Footprint</th>
<th>Base Energy (mJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculator</td>
<td>Arithmetic calculator</td>
<td>18,024</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.cse.psu.edu/~gchen/kvmgc/">www.cse.psu.edu/~gchen/kvmgc/</a></td>
<td>14,279</td>
<td></td>
</tr>
<tr>
<td>Crypto</td>
<td>Light weight cryptography API in Java</td>
<td>89,748</td>
<td>8.40</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.bouncycastle.org">www.bouncycastle.org</a></td>
<td>60,613</td>
<td></td>
</tr>
<tr>
<td>Dragon</td>
<td>Game program with Sun’s KVM</td>
<td>11,983</td>
<td>5.92</td>
</tr>
<tr>
<td></td>
<td>comes with Sun’s KVM</td>
<td>6,149</td>
<td></td>
</tr>
<tr>
<td>Elite</td>
<td>3D rendering engine for small devices</td>
<td>20,284</td>
<td>3.67</td>
</tr>
<tr>
<td></td>
<td>home.rochester.rr.com/ohommes/Elite</td>
<td>11,908</td>
<td></td>
</tr>
<tr>
<td>Kshape</td>
<td>Electronic map on KVM</td>
<td>39,684</td>
<td>13.52</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.jshape.com">www.jshape.com</a></td>
<td>37,466</td>
<td></td>
</tr>
<tr>
<td>Kvideo</td>
<td>KPG (MPEG for KVM) decoder</td>
<td>31,996</td>
<td>1.52</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.jshape.com">www.jshape.com</a></td>
<td>14,012</td>
<td></td>
</tr>
<tr>
<td>Kwml</td>
<td>WML browser</td>
<td>57,185</td>
<td>34.97</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.jshape.com">www.jshape.com</a></td>
<td>49,141</td>
<td></td>
</tr>
<tr>
<td>Manyballs</td>
<td>Game program with Sun’s KVM</td>
<td>20,682</td>
<td>6.19</td>
</tr>
<tr>
<td></td>
<td>comes with Sun’s KVM</td>
<td>13,276</td>
<td></td>
</tr>
<tr>
<td>MathFP</td>
<td>Fixed-point integer math library routine</td>
<td>11,060</td>
<td>6.91</td>
</tr>
<tr>
<td></td>
<td>home.rochester.rr.com/ohommes/MathFP</td>
<td>8,219</td>
<td></td>
</tr>
<tr>
<td>Mini</td>
<td>A configurable multi-threaded mini-benchmark</td>
<td>31,748</td>
<td>1.46</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.cse.psu.edu/~gchen/kvmgc/">www.cse.psu.edu/~gchen/kvmgc/</a></td>
<td>16,341</td>
<td></td>
</tr>
<tr>
<td>Missiles</td>
<td>Game program with Sun’s KVM</td>
<td>26,855</td>
<td>4.28</td>
</tr>
<tr>
<td></td>
<td>comes with Sun’s KVM</td>
<td>17,999</td>
<td></td>
</tr>
<tr>
<td>Scheduler</td>
<td>Weekly/daily scheduler</td>
<td>19,736</td>
<td>9.63</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.cse.psu.edu/~gchen/kvmgc/">www.cse.psu.edu/~gchen/kvmgc/</a></td>
<td>17,685</td>
<td></td>
</tr>
<tr>
<td>Starcruiser</td>
<td>Game program with Sun’s KVM</td>
<td>13,475</td>
<td>4.58</td>
</tr>
<tr>
<td></td>
<td>comes with Sun’s KVM</td>
<td>11,360</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1. Brief description of benchmarks used in our experiments. The two footprint values of each application are the maximal and effective footprint sizes (in bytes) respectively.
2.4 Energy Characterization and Optimization

2.4.1 Base Configuration

Unless otherwise stated, our default bank configuration has eight banks for the heap, eight banks for the ROM, and two banks for the runtime stack (as depicted in Fig. 2.2). All banks are 16KB. In this base configuration, by default, all banks are either in the active or read/write states, and no leakage control technique is applied. The overall energy consumption of this cacheless configuration running with M&S (GC without compaction) is given in the last column of Table 2.1. The energy distribution of our applications is given in Fig. 2.5. The contribution of the garbage collector to the overall datapath energy is 4% on average across the different benchmarks (not shown in the figure). We observe that the overall datapath energy is small compared to the memory energy consumption. We also observe that the heap energy constitutes 39.5% of the overall energy and 44.7% of the overall memory (RAM plus ROM) energy on the average.

The memory energy consumption includes both the normal execution and garbage collection phases and is divided into leakage and dynamic energy components. On average, 75.6% of the heap energy is due to leakage. The leakage energy is dependent on the duration of the application execution while the dynamic energy is primarily determined by the number of references. Considering this energy distribution, reducing the heap energy through leakage control along with efficient garbage collection and object allocation can be expected to be very effective.

We also note from Fig. 2.5 that overall ROM energy is less than the overall heap energy. This is mainly due to the following reasons. First, the dynamic energy for accessing a ROM is less than the corresponding value for a same size RAM. This difference results from the smaller capacitive load on the wordlines and bitlines. In the ROM, only the memory cells that store a value of zero contribute a gate capacitance to the wordline. Further, only these cells contribute a drain capacitance to the bitline [29]. In addition, the number of bitlines is reduced by half with respect to the RAM configuration and a single-ended sense amplifier is used for the ROM array as opposed to a differential sense amplifier in the RAM array. Our circuit simulations show that the per access energy of a RAM array can thus be as large as 10 times that of a ROM.
Fig. 2.4. Heap footprints (in bytes) of our applications. For each graph, x-axis denotes the time and y-axis gives the cumulative size of live objects.
array. However, the difference is dependent on the actual bit pattern stored in the array. In our experiments, we conservatively used a dynamic energy cost for accessing the ROM to be half that of a corresponding RAM array access. Since the effective transistor width in the ROM array is also smaller than that in a correspondingly sized RAM array, the leakage energy of the ROM is also smaller. Another reason that the ROM energy is less than the heap energy is because of using a ROM configuration that implements a simple but effective energy optimization. In particular, we use a banked ROM configuration and activate the supply voltage selectively to only those banks that contain libraries that are accessed by the application. Note that this incurs a penalty at runtime when the bank is accessed the first time. However, we found this overhead to be negligible.

Another interesting observation is the relative leakage and dynamic energy consumption breakdowns in the heap memory and the ROM. We found that the dynamic energy of the ROM is 63.7% of overall ROM energy which is much higher than the corresponding value in the heap. This difference is due to high access frequency of the ROM banks that contain the KVM code as well as class libraries.
2.4.2 Impact of Mode Control

Turning off a heap bank when it does not contain any live object can save energy in two ways. First, leakage energy is reduced as a result of the leakage reduction strategy explained earlier. Second, the precharge portion of dynamic energy is also eliminated when the bank is powered off. Fig. 2.6 gives the heap energy consumption due to M&S when mode control (leakage control) is employed, normalized with respect to the heap energy due to M&S when no mode control is used (i.e., all partitions are active all the time). We observe from this figure that turning off unused banks reduces the heap energy consumption by 31% on the average (with savings ranging from 2% to 65%). On average, 90% of these savings come from leakage energy reduction. Fig. 2.7 explains the energy savings due to leakage control. This figure shows the percentage distribution of active banks for the applications in our suite. We observe that many applications execute with a small number of active banks most of the time, meaning that the remaining banks are turned off. We also observe, however, that some applications use all eight banks at some point during their executions. Considering this behavior and the heap footprints of live data shown in Fig. 2.4, it can be clearly seen how badly live objects can be scattered throughout our 128KB heap memory (although their cumulative sizes are much smaller than 128KB).\(^1\)

Fig. 2.6 also shows that the normalized runtime stack energy. This energy gain in runtime stack is achieved by not activating one of the banks of the runtime stack when it does not contain any useful data. Since we have two banks allocated to runtime stack (and the KVM variables) and many applications in our suite can operate most of the time with one bank only, on the average, we achieve around 50% energy saving on these banks.

These energy savings, however, do not come for free. As discussed earlier, accessing a powered off bank requires an extra 350 cycles for the supply voltage to be restored. During this time, a small amount of energy is also expended. Fig. 2.2 shows the extra execution cycles and extra energy as both absolute values and percentages of overall execution time and memory energy, respectively. We can see from this figure that both of these overheads are negligible.

\(^1\)As an example, Fig. 2.7 shows that, for more than 70% of the execution time of Kshape, five or six banks (16KB each) are turned on. However, in Fig. 2.4, we find that the total size of live objects of Kshape never exceeds 40KB. Some banks cannot be turned off because they contain live objects, although the total size of live objects contained in this bank is much smaller than the bank size.
Therefore, we can conclude that applying leakage control mechanism to the inactive heap banks can reduce energy consumption significantly without too much impact on execution time.

2.4.3 Impact of Garbage Collection Frequency

The M&S collector is called by default when, during allocation, the available free heap space is not sufficient to accommodate the object to be allocated. It should be noted that between the time that an object becomes garbage and the time it is detected to be so, the object will consume heap energy as a dead object. Obviously, the larger the difference between these two times, the higher the wasted energy consumption if collecting would lead to powering off the bank. It is thus vital from the energy perspective to detect and collect garbage as soon as possible. However, the potential savings should be balanced with the additional overhead required to collect the dead objects earlier (i.e., the energy cost of garbage collection).

In this subsection, we investigate the impact of calling the garbage collector (without compaction) with different frequencies. Specifically, we study the influence of a k-allocation collector that calls the GC once after every k object allocations. We experimented with five different values of k: 10, 40, 75, 100, and 250. The top graph in Fig. 2.8 illustrates the heap

Fig. 2.6. Normalized energy consumption in heap and stack due to mode control (M&S).
energy (normalized with respect to M&S heap energy without mode control) of the $k$-allocation collector. The impact of pure mode control is reproduced here for comparison.

We clearly observe that different applications work best with different garbage collection frequencies. For example, the objects created by Dragon spread over the entire heap space very quickly. However, the cumulative size of live objects of this benchmark most of the time is much less than the available heap space. Consequently, calling the GC very frequently (after every 10 object allocations) transitions several banks into the inactive state and reduces heap energy by more than 40%. Reducing the frequency of the GC calls leads to more wasted energy consumption for this application. In Kvideo, we observe a different behavior. First, the energy consumption is reduced by reducing the frequency of collector calls. This is because each garbage collection has an energy cost due to fact that mark and sweep operations access memory. In this application, the overhead of calling GC in every 10 allocations brings an energy overhead that cannot be compensated for by the energy saving during execution. Therefore, calling the GC less frequently generates a better result. Beyond a point ($k=75$), however, the energy starts to increase as the garbage collections become so less frequent that significant energy is consumed due to dead but not collected objects. Applications like Mini, on the other hand, suffer
Fig. 2.8. Normalized energy consumption in heap and ROM memory when M&S with mode control is used with different garbage collection frequencies.
Table 2.2. Energy and performance overhead of bank turning-off.

greatly from the GC overhead and would perform best with much less frequent garbage collector calls. Overall, it is important to tune the garbage collection frequency based on the rate at which objects become garbage to optimize energy consumption.

The GC overhead also leads to increased energy consumption in the ROM, runtime stack, and processor core. The energy increase in the ROM is illustrated on the bottom graph of Fig. 2.8. Each bar in this graph represents the energy consumption in the ROM normalized with respect to the energy consumption of the ROM with M&S with mode control. It can be observed that the case with $k = 10$ increases the energy consumption in ROM significantly for many of the benchmarks. On the other hand, working with values of $k$ such as 75, 100, and 250 seems to result in only marginal increases, and should be the choice, in particular, if they lead to large reductions in heap energy. We also found that the energy overheads in the core and runtime stack were negligible and have less than 1% impact on overall energy excluding cases of $k = 10$. To summarize, determining globally optimal frequency demands a tradeoff analysis between energy saving in the heap and energy loss in the ROM. Except for cases when $k = 10$, the energy savings in the heap clearly dominate any overheads in the rest of the system.

A major conclusion from the discussion above is the following. Normally, a virtual machine uses garbage collector only when it is necessary, as the purpose of garbage collection is to create more free space in the heap. In an energy-sensitive, banked-memory architecture, on
the other hand, it might be a good idea to invoke the collector even if the memory space is not a concern. This is because calling GC more frequently allows us to detect garbage earlier, and free associated space (and turn off the bank). This early detection and space deallocation might result in large number of banks being transitioned to the inactive state.

### 2.4.4 Impact of Object Allocation Style

M&S in KVM uses a global free list to keep track of the free space in the heap. When an object allocation is requested, this free list is checked, the first free chunk that can accommodate the object is allocated, and the free list is updated. While in a non-banked architecture, this is a very reasonable object allocation policy, in a banked-memory based system it might be possible to have better strategies. This is because the default strategy does not care whether the free chunk chosen for allocation is from an already used (active) bank or inactive bank. It is easy to see that everything else being equal, it is better to allocate new objects from already active banks.

To experiment with such a strategy, we implemented a new bank allocation method where each bank has its own private free list. In an object allocation request, first, the free lists of active banks are checked and, only if it is not possible to allocate the space for the object from one
of these lists, the lists of inactive banks are tried. This strategy is called the active-bank-first allocation.

Fig. 2.9 gives the energy consumption for three different versions. M&S with leakage control (denoted Mode Control), active-bank-first allocation (denoted Active Bank), and a version that combines active-bank-first allocation with a strategy that activates the GC only when the new object cannot be allocated from an already active bank (denoted Active Bank+). All values in this figure are normalized with respect to the heap energy consumption of M&S without mode control. We see from these results that Active Bank does not bring much benefit over Mode Control in most cases (except that we observe a 6% heap energy improvement in MathFP).

This can be explained as follows. Objects with long life time are typically allocated early (before the first GC is invoked) and occupy the first few banks. The younger objects that occupy banks with higher addresses seldom survive the next garbage collection. From the traces of bank occupation, we observe that after each GC, the banks with lower address are always occupied and the higher addresses are typically free. Consequently, the default allocation acts like active-bank-first allocation. MathFP is an exception to this allocation behavior. In MathFP, after each GC, the occupied banks are not always contiguous. In this case, active-bank-first allocation can save energy by postponing the turning on a new bank. In contrast, in benchmarks such as Kwml and Scheduler, the energy overhead of maintaining multiple free lists shows up as there is almost no gain due to the allocation strategy itself.

Thus, it is important to modify the default garbage collection triggering mechanism in addition to changing allocation policy to obtain any benefits. Active Bank+ combines the active-bank-first allocation mechanism along with a strategy that tries to prevent a new bank from being turned on due to allocation. As it combines an energy aware allocation and collection policy, Active Bank+ can lead to significant energy savings as shown in Fig. 2.9. The causes for these savings are three fold. First, Active Bank+ invokes the GC more frequently, and thus banks without live objects are identified and turned off early. Second, during allocation, it reduces the chances of turning on a new bank. Third, it allocates permanent objects more densely, thereby increasing the the opportunities of turning off banks.
Fig. 2.10. Energy consumption in heap due to mode control (M&C) normalized with respect to M&C without mode control.

2.4.5 Impact of Compaction

As explained earlier in this chapter, the compaction algorithm in KVM performs compaction only when, after a GC, there is still no space for allocating the object. In a resource-constrained, energy-sensitive environment, compaction can be beneficial in two ways. First, it might lead to further energy savings over a non-compacting GC if it can enable turning off a memory bank that could not be turned off by the non-compacting GC. This may happen as compaction tends to cluster live objects in a smaller number of banks. Second, in some cases, compaction can allow an application to run to completion (without out-of-memory error) while the non-compacting algorithm gives an out-of-memory error. In this subsection, we study both these issues using our applications.

Let us first evaluate the energy benefits of mode control when M&C (the default compacting collector in KVM) is used. The results given in Fig. 2.10 indicate that mode control is very beneficial from the heap energy viewpoint when M&C is employed. Specifically, the heap energy of the M&C collector is reduced by 29.6% over the M&C without mode control. The top graph in Fig. 2.11 compares heap energy of M&S and M&C with mode control. Each bar in this graph represents heap energy consumption normalized with respect to M&S without mode...
control. It can be observed that M&C does not bring significant savings over M&S (denoted Mode Control in the graph). First, moving objects during compaction and updating reference fields in each object consumes energy. In addition, compacting may increase the applications running time, which also means more leakage energy consumption. Therefore, a tradeoff exists when compaction is used. In our implementation, to lessen the performance impact, compaction is performed only when the object to be allocated is larger than any of the available free chunks, or if it can turn off more banks. Kwml is one of the benchmarks where compaction brings some energy benefits over M&S with mode control. The execution trace of this code indicates that there are many scenarios where Mode Control does not turn off banks because all banks contain some small-sized permanent objects. M&C, on the other hand, turns off some banks after garbage collection due to the fact that it both compacts fragmented live objects with short life times and clusters permanent objects in a smaller number of banks. In some benchmarks such as Dragon, on the other hand, M&C does not create sufficient number of free banks to offset the extra energy overhead due to additional data structures maintained.

The original allocation policy in the compacting version distinguishes between permanent and dynamic objects as mentioned earlier. In the banked-memory architecture, the default allocation policy is slightly modified to allocate the permanent objects and regular objects in separate banks. This eliminates the need to move the already allocated dynamic objects when a new permanent object is allocated. However, this strategy requires activating at least two banks when both permanent and dynamic objects are present. The active-bank-first allocation strategy, on the other hand, co-locates both the permanent and dynamic objects together and saves energy. However, it incurs the cost of moving the already allocated dynamic objects to a new bank when a new permanent object is allocated. Fortunately, this operation is very infrequent. Consequently, as opposed to the case without compaction, the Active Bank version (that is, allocating object from an already active bank if it is possible to do so) combined with M&C generates better results than M&C with default allocation, and consumes 10% less heap energy on the average. That is, compacting the heap improves the energy impact of the active-bank-first allocation strategy. Finally, as before, the Active Bank+ outperforms other versions for most of the cases.
Fig. 2.11. Top: Comparison of M&C and M&S. Bottom: Comparison of different compacting collectors.
The bottom graph in Fig. 2.11 compares heap energy consumption of three different compaction algorithms. M&C is the default compactor in KVM. The M&C+ version differs from M&C in that it performs compaction after each garbage collection (whether or not it is actually needed from the viewpoint of free space). Our results show that in some benchmarks such as Kshape and Scheduler, it generates better results than both M&S (denoted Mode Control in the figure) and M&C. This result means that unlike general-purpose systems, in an energy-sensitive system, extra compactions might bring energy benefits for some applications.

M&C2, on the other hand, is a collector that uses the Lisp2 Algorithm [73], as opposed to the default Break Table-based algorithm in KVM. In the Lisp2 algorithm, during compaction, first, the new addresses for all objects that are live are computed. The new address of a particular object is computed as the sum of the sizes of all the live objects encountered until this one, and is then stored in an additional ‘forward’ field in the object’s header. Next, all pointers within live objects which refer to other live objects are updated by referring to the ‘forward’ field of the object they point to. Finally, the objects are moved to the addresses specified in the “forward” field, and then the ‘forward’ field is cleared so that it can be used for the next garbage collection.

The advantages of this algorithm are that it is can handle objects of varying sizes, it maintains the order in which objects were allocated, and it is a fast algorithm with an asymptotic complexity of $O(M)$, where $M$ is the heap size. Its disadvantage is that it requires an additional four-byte pointer field in each object’s header that increases the heap footprint of the application.

There are two potential energy benefits due to this compaction style. First, objects can be relocated accounting for temporal affinities and object lifetimes, instead of sliding-only compaction as in M&C. For example, clustering objects with similar lifetime patterns increases the potential for deactivating an entire bank (when the objects it holds die together). Secondly, reference fields can be updated more efficiently as compared to M&C and M&C+, where updating each reference field needs to look up the Break Table. Finally, the extra forward field can be used as a stack in the marking phase to reduce the overhead during the scanning phase.

In case that the heap is severely fragmented, M&C2 will out perform M&C+ because it treats each object individually, and does not need to copy the Break Table (in this case, the Break Table will be large) when moving objects. On the other hand, when most live objects are placed contiguously, M&C+ will perform better because it can move objects in fewer chunks.
Further, the smaller Break Table reduces the look up cost (whose time complexity increases logarithmically with respect to the Break Table size) when updating each reference field during compaction. Obviously, if the total number of reference fields is large, M&C+’s performance will suffer a lot during the updating phase.

Crypto is an example application with rather big heap footprint that benefits from M&C2’s cheaper marking and updating. In contrast, Elite is an application with very small footprint. Due to the 4-byte’s overhead in each objects, M&C2 turns on a new bank much earlier than M&C+. Specifically, M&C2 turns on the third bank about 5.6 seconds after program initialization while the corresponding value for M&C+ is 6.2 seconds. Initializing the forwarding fields of each objects also consumes some extra energy.

As we mentioned earlier, a compacting GC can run an application in a smaller heap memory than a corresponding non-compacting version. For example, Missiles can run using a 32KB heap when M&C is employed while requiring a minimum of 64KB heap when executing using M&S. Comparing the energy consumption for systems with these configurations, we found that the M&S that uses a 64KB heap with four 16KB-banks consumes a heap energy of 1.02mJ, which is much larger than 0.71mJ, the heap energy consumed by M&C2 when using a 32KB heap using two 16KB-banks. Similarly, Kwml can run using a 64KB heap when M&C is employed, while requiring a minimum of 128KB heap when executing using M&S. For this application, the M&S that uses a 128KB heap with eight 16KB-banks consumes a heap energy of 13.15mJ, which is much larger than 7.66mJ, the heap energy consumed by M&C2 when using a 64KB heap using four 16KB-banks.

It is also interesting to study how much energy the compaction itself contributes relative to the other portions of the collector. Fig. 2.12 shows the core energy breakdown due to the garbage collection activity with M&C and M&C2. Both M&C and M&C2 have four major phases. For M&C the phases are mark, sweep, relocate, and update. We combine the last two into a part called ‘compaction’ as they are invoked only during compaction. For M&C2, the phases are mark and the three phases associated with compaction: compute, update, and relocate. We see that in M&C that the mark phase consumes the bulk of the energy, mainly, because it is more expensive than the sweep operation. The contribution of compact energy varies from application to application depending on the number of times the compaction is invoked. When we consider
M&C2, however, the energy behavior changes. First, since there is no explicit sweep activity, the energy consumption is distributed mainly between compact and mark phases. Second, since this collector performs compaction every time GC is called (as opposed to M&C which performs compaction only when an object still cannot be allocated after GC), the compaction energy constitutes a larger portion in most of the benchmarks.

2.4.6 Impact of Number of Banks and Heap Size

The heap size and bank size can influence the effectiveness of mode control. Let us consider the example in Fig. 2.1 once again to illustrate the influence of bank size. Instead of four banks, if we had only two larger banks (Bank 0 + Bank 1 and Bank 2 + Bank 3), at step 200, the garbage collector would not be able to turn off any of the banks. Similarly, the heap size would also influence the frequency of GC invocations in the default version. For example, if the heap size is reduced by half in Fig. 2.1 (i.e., only Banks 0 and 1 are available), the garbage collector will be invoked at step 50 to find space to accommodate object D. Further reducing the heap size will also reduce overall leakage energy as we have fewer leaking transistors. Thus, it is important to evaluate the energy impact of varying these parameters.

Fig. 2.13 shows the impact of varying the bank and heap sizes when using the M&S garbage collector with Mode Control and Active Bank+. It must be noted that many applications cannot complete with a smaller heap size. Only six of the applications can execute using both 64KB and 32KB heap sizes. In general reducing the heap size reduces the overall energy consumption. There are two reasons for this behavior. With a smaller heap, the effort expended in allocating an object and also marking and sweeping the heap during garbage collection reduce. It must be reiterated that the M&S algorithm uses a non-stack implementation which means that the cost of garbage collection is proportional to the size of the heap. However, a smaller heap will also increase the frequency of garbage collection making the overhead of garbage collection more significant. As an example, the number of collections increase from 2 to 20, when heap size is reduced from 128KB to 32KB when executing Scheduler. For most cases, this overhead is more than compensated for by the energy savings due to more frequent garbage collection. In other words, the dead objects are collected closer to when they become dead.
Fig. 2.12. Energy distribution in core due to GC. Top: M&C; Bottom: M&C2.
Secondly, when we increase the number of banks, mode control has the ability to exploit a finer granularity of turn-off. In addition, a smaller bank has a smaller capacitive load and hence a smaller per access dynamic energy cost. This leads to smaller heap energy consumption when using smaller banks. On the average, for a 64KB heap, when using 8KB banks the energy consumption is only 65% of the energy consumed when using 16KB banks. Similar trends are observed for a 32KB heap. We also observe that the Active Bank+ scheme brings energy benefits over simple mode control across all configurations (around 20%, on the average, across all benchmarks and configurations). We also evaluated the impact of bank and heap sizes on other garbage collection algorithms. We found similar trends in their behavior.

While smaller banks are found to be beneficial, the overheads of banking make them unattractive at very small granularities. A more detailed characterization of the influence of number of banks for different size memory arrays can be found in [128]. In order to achieve additional savings, it might be important to exploit finer granular turn-offs such as those at the word level [79] instead of the bank level.

2.4.7 Impact of Cache Memory

The presence of a cache influences the energy behavior in two ways. First, the number of references to the memory, both the ROM and RAM, are reduced. This reduces the dynamic energy consumed in the memory. In particular, we find that the heap energy reduces to 23% of the overall energy in the presence of the 4KB data and 4KB instruction caches. Note that embedded cores typically have small caches. Second, the cache can account for a significant portion of the overall system energy. In particular, the instruction cache is a major contributor as it is accessed every cycle. In the context of this work, it is important to evaluate how the cache influences the effectiveness of mode control strategy and the additional gains that energy-aware allocation and collection provide over pure mode control. Fig. 2.14 shows the normalized heap energy in the presence of a 4K 2-way instruction cache and a 4KB 2-way data cache when a 64KB heap is used. Pure mode control with M&S reduces 15% of heap energy on the average across all benchmarks. An additional 28% heap energy saving is obtained through the energy-aware active bank allocation and garbage collection before new bank activation. The corresponding figures
Fig. 2.13. Impact of number of banks and heap size (M&S). Note that Crypto and Kwml do not run with 32KB and 64KB heap sizes. Kvideo, Manyballs, Kshape, Missiles and Mini do not run with 32KB heap size.

when M&C2 is used are 14% and 25%, respectively. These results show that the proposed strategies are effective even in the presence of a cache.

2.5 Concluding Remarks

In this chapter, we characterized the energy impact of GC parameters built on top of Sun’s embedded Java virtual machine, KVM. Further, we showed how the GC can be tuned to exploit banked memory architectures for saving energy. The major conclusions from this work are as follows:

- Our characterization of energy consumption shows that the heap energy consumption is 39.5% of the overall energy consumption of an embedded system-on-a-chip when executing typical Java applications. Further, we observe that the leakage energy is the dominant portion, accounting for 75.6% of the heap energy.
In an energy-constrained environment, the GC can be used to identify unused heap memory banks and apply energy control mechanisms. Our results show that GC-controlled energy mode control can save 31% of the heap energy on the average.

The duration between when an object becomes dead and when it is garbage collected determines the wasted leakage energy in maintaining these dead objects. Thus, the frequency of garbage collection has a profound impact on how much of this wasted energy can be reduced. The more frequent the garbage collection, the less the wasted energy. Thus, in a banked memory environment, it will be beneficial from an energy perspective to invoke garbage collection even before the traditional invocation time (when space cannot be found for allocating an object). However, garbage collection itself incurs an energy cost that must be considered.

The energy savings of GC-controlled energy optimization are influenced by both the object allocation and garbage collection policies. In particular, we find that a strategy that
allocates objects only on active banks (if possible) and activates garbage collection before turning on a new bank provides consistent improvements over pure mode control.

- Clustering live objects in small number of banks using compaction can reduce heap energy. While some applications benefit from this clustering, the energy overhead of moving the live objects during compaction negates the potential benefits in others. As in the case of garbage collection frequency, compacting more often than when only running out of heap space to allocate an object provides energy savings in some benchmarks. The compaction style also influences the overhead and overall effectiveness of compaction. Specifically, taking object reference relations into account (M&C2) improves the energy impact of compaction in some cases.

- The proposed GC-controlled energy management is effective across different heap, bank and cache configurations. We observe that while decreasing heap size can prevent some applications from running to completion, it generally reduces the overall heap energy consumption. In addition, our experiments show that smaller bank sizes result in less energy consumption due to reduced capacitive load when accessing the banks and the increased potential for finer granular leakage control. Finally, when caches are enabled, the GC-controlled energy management is still shown to be effective.
Chapter 3

Energy Aware Adaptive Garbage Collection

3.1 Introduction

In the previous chapter, I propose using the garbage collector to detect and safely turn off the memory banks that do not contain any live objects to conserve energy. In this research, one can observe that the frequency with which garbage collection is invoked significantly affects the overall energy consumption of the memory subsystem. As the optimal garbage collection frequency is dependent on the particular Java application and the user input, it is hard to determine the optimal frequency statically. An important issue then is to come up with a suitable garbage collection frequency that balances the potential leakage energy savings with the additional overheads. It is not difficult to see that if we fix the GC frequency at a specific value over all applications, this value may be suitable for only some of these applications. This is because each application has a different object allocation and garbage generation pattern. Our experimental results detailed in Section 3.3 shows that this is really the case. Another option would be tuning the frequency of garbage collection dynamically (i.e., during execution).

In this chapter, we present an adaptive garbage collection strategy that tunes the frequency of the garbage collection according to the dynamic behavior of the application at runtime. A basic principle is that whenever there is an opportunity to turn off a bank that holds only dead objects, the garbage collector should be invoked. Simple application of this principle, however, may cause frequent turn on and turn off of the same bank (thrashing) if too few objects are collected at each collection (Fig. 3.1). To avoid thrashing, we need an additional principle: garbage collector should not be invoked unless a certain number of objects have become garbage.

It should be noted, however, without actually invoking GC, it is not possible to tell exactly how many (and what size) objects have become garbage (since the last collection phase).
Fig. 3.1. Garbage collector is invoked at T1, T2, T3 and T4.

However, due to the fact that many object allocation/deallocation patterns exhibit some regularity in one form or another [73], we can use past (history) information to estimate the size of the dead (unreachable) objects.

Let $k_1$ and $k_2$ be the object creation rate and the garbage generation rate, respectively. Assume two successive garbage collections, $c_1$ and $c_2$, that are invoked at times $t_1$ and $t_2$, respectively (Fig. 3.2). Assume further that after $c_1$, the total size of the (live) objects in the heap is $a_1$ and that, at $t_2$, just before $c_2$, the total size (including the unreachable objects) in the heap is $h_2$. Finally, let us assume that after $c_2$ the total size of the objects in the heap (excluding the unreachable ones) is $a_2$. Based on this, the object creation rate during the time period $[t_1, t_2]$ is:

$$k_1 = (h_2 - a_1)/(t_2 - t_1).$$

Similarly, the garbage generation rate during $[t_1, t_2]$ can be expressed as:

$$k_2 = (h_2 - a_2)/(t_2 - t_1).$$

In our strategy, we use $k_1$ and $k_2$ to estimate the object creation and garbage generation rates after $t_2$ until the next collection, say $c_3$, is invoked. After $c_3$, the values of $k_1$ and $k_2$ will be updated to adapt their values to (potentially) changing heap behavior.

Following each allocation $i$ that occurs at time $t_i$, we use the following strategy to decide whether to invoke GC or not:
Fig. 3.2. Garbage collection $c_1$ and $c_2$ are invoked at $t_1$ and $t_2$ respectively.

- If after GC, at least two banks can be supply gated, the garbage collector should be invoked:
  \[
  b(h_i + s_i) - b(h_i + s_i - k_2(t_i - t_2)) \geq 2,
  \]
  where $b(s)$ is a function that returns the number of banks, given the total object size $s$; $h_i$ is the total object size in the heap right before the allocation; and $s_i$ is the size of the object to be allocated.

- We consider the ability to supply-gate only one bank differently, as a fast object allocation rate may require the re-activation of the supply gated unit within a short duration of time. Due to the time penalty for re-activation, it may be more beneficial to keep the power supply to the bank on if it would result in re-activation within a few allocations.

  If after the collection, one bank could be supply gated, then the space GC creates must allow the application to run for an interval no shorter than $L$:
  \[
  b(h_i + s_i) - b(h_i + s_i - k_2(t_i - t_2)) = 1 \quad \text{and} \quad b(h_i + s_i - k_2(t_i - t_2))B - (h_i + s_i - k_2(t_i - t_2)) \geq k_1 L.
  \]
  In our implementation, $L$ is measured in terms of the number of allocations. In the last formulation above, $B$ is the size of a memory bank.
As our estimation is not perfect, to limit the penalty due to misestimation, the GC should be invoked after every $B$ byte allocation if no collection happens in between.

The overhead of our adaptive algorithm is not too much. This is because our decision making requires simple calculations and $h_i$ and $a_i$ can be easily obtained by adding only a few lines to garbage collector codes. It should also be stressed that both the information collection and the decision making activities consume energy. It is true that more sophisticated heuristic algorithms may give more accurate predictions. However, such algorithms can also be expected to be more complex and to require more detailed past history information, thereby potentially incurring a more significant performance (execution time) overhead.

### 3.2 Experimental Setup

To see how our adaptive garbage collector performs, we compared it to a class of collectors called $k$-allocation collectors that call the GC (that is, the mark-and-sweep algorithm) once after every $k$ object allocations. We conducted experiments with six different values of $k$: 10, 40, 75, 100, 250, and 500. In cases where the collection frequency of the base collector is lower than the $k$ value used, we can expect that $k$-allocation collector will generate a better energy behavior. In fact, we can expect that for each application there is a $k$ value (among the $k$ values used in our experiments) that generates the best energy result. Consequently, for the best results, the GC frequency should be tuned to application behavior. However, this, in general, may not be possible as we may not have any information about the application’s heap behavior until the runtime. Consequently, our adaptive scheme tries to detect the optimal frequency at runtime.

For each application from a given set of Java applications, a successful adaptive collector should generate at least the same energy behavior as the $k$-allocation collector that performs best for that application. For some applications, we can even expect the adaptive collector to generate better results than even the best $k$-allocation collector. This may occur, for example, when there are different phases during the course of the application’s execution and each phase works best with a different garbage collection frequency. Thus, we expect the adaptive collector to be competitive with the best $k$-allocation collector from the energy perspective. In the rest of this
chapter, this compacting mark-and-sweep collector, which has been discussed in the previous chapter, is referred to as the base collector or base for short.

The memory system energy is divided into three portions: energy spent in accessing KVM code and libraries, energy spent in accessing heap data, and energy spent in accessing the runtime stack and KVM variables. We use the same simulator and energy modeling as those used in the previous chapter to evaluate the memory energy consumption. Note that the energy spent in on-chip interconnects are included in the corresponding memory components. We assume that each memory bank can be in one of three states: R/W (Read/Write), Active, and Inactive. In the R/W state, the memory bank is being read or written. It consumes full dynamic energy as well as full leakage energy. In the active state, the memory bank is active (powered on) but not being read or written. It consumes no dynamic energy but full leakage energy. Finally, in the inactive state, the bank does not contain any useful data and is supply-gated. We conservatively assume that when in the inactive state a memory bank consumes 5% of its original leakage energy (as a result of supply-gating). When a bank in the inactive state is accessed (to service a memory request), it takes 350 cycles to transition to the active. We term this time as the resynchronization time (or resynchronization penalty). It should be We also assume that the per cycle leakage energy consumed during resynchronization is the average of per cycle leakage energy consumed when the system is in the active state and that consumed when it is in the inactive state.

3.3 Experimental Results

Unless otherwise stated, in all the results reported in this section we use an $L$ value of 20 allocations and a bank size of 16KB.

3.3.1 Heap Energy

Fig. 3.3 shows the heap energy consumption for our different versions. All values shown in this graph are normalized with respect to the base collector. In addition to base and our adaptive collector (denoted as $\text{adpt}$), we report the results for a set of $k$-allocation collectors with $k$ values of 10, 40, 75, 100, 250, and 500. Above each bar is the normalized energy consumption in heap. From these results, we can make the following observations. First, as far as the $k$-allocation collector is concerned, we clearly see that different applications work best with different garbage
Fig. 3.3. Normalized energy consumption (static+dynamic) in the heap memory.
Fig. 3.4. Normalized energy consumption breakdown.
collection frequencies. Second, our adaptive garbage collection strategy reduces the heap energy of the base collector in KVM significantly; the average (over all applications) heap energy saving is 28.4%. Third, for a given benchmark, the adaptive strategy is always competitive with the best $k$-allocation collector. For example, in Dragon, the adaptive strategy comes close to the 10-allocation collector (the best $k$-allocation collector for this benchmark). Similarly, in Kvideo (resp. Kshape), our adaptive collector generates competitive results with 75-allocation (resp. 250-allocation) collector. These results clearly show that the adaptive garbage collection is very successful in optimizing heap energy.

### 3.3.2 Overall Energy

While our objective is optimizing the energy consumption in heap, changing the frequency of garbage collections might also change the energy profile in other parts of the system such as ROM, the memory banks that hold runtime stack, and the CPU datapath. For example, a more frequent collection is expected to increase the energy spent in CPU due to garbage collection. To analyze this behavior, we give in Fig. 3.4 the energy consumption in all components that we are interested in. We observe from these results that, for each code in our experimental suite, the adaptive strategy comes very close to the best $k$-allocation collector, even when one focuses on the overall energy consumption.

### 3.3.3 Execution Profiles

To better understand the energy savings due to adaptive garbage collection, we give further data illustrating where the energy benefits come from. First, we give in Table 3.1 the number of times each version invokes garbage collection. From these results we see that the adaptive strategy calls garbage collection much more frequently than the base collector. This explain why it reduces the heap energy consumption significantly. In fact, we can see from this table that even the 500-allocation collector calls garbage collection more frequently (on average) than the base collector, indicating that the base collector is very slow in invoking the collector. As mentioned earlier however, calling garbage collection more frequently incurs an extra runtime overhead. Therefore, the garbage collector should not be invoked unless it is necessary (i.e., unless it is
expected to save energy by powering off bank(s)). We see from Table 3.1 that the adaptive strategy calls collection less frequently than the $k$-allocation strategy when $k$ is 10, 40, 75, and 100. Therefore, its runtime overhead is less than these collectors. Considering that the adaptive strategy is competitive with all these $k$-allocation collectors, one can see that the adaptive strategy achieves the same energy behavior as these allocators with much less performance penalty. As compared to the remaining versions, it calls collection more frequently on the average, but one application (Kwml) is responsible for this average.

Fig. 3.5 shows the heap footprints of different versions for three applications: Calculator, Dragon, and ManyBalls. In these graphs, the x-axis corresponds to the number of allocations made during the course of execution. The y-axis, on the other hand, represents the occupied memory space in heap (by alive or dead objects). Each horizontal line on the y-axis corresponds to a bank boundary, starting from the bottom with the first bank. In these graphs, for sake of clarify, we show only a 75-allocation collector, the base collector and our adaptive collector. We can clearly observe the impact of changing the frequency of collections on the number of active power banks required. Let us first focus on Calculator. In this application, the base collector does not invoke any collection. We observe that the adaptive collector invokes collection less frequently than the 75-allocation collector (this is also true for all studied $k$-allocation collectors except the 500-allocation collector). When using the adaptive collector, the GC is

<table>
<thead>
<tr>
<th>Application</th>
<th>10</th>
<th>40</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>base</th>
<th>adpt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculator</td>
<td>104</td>
<td>24</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Crypto</td>
<td>595</td>
<td>147</td>
<td>78</td>
<td>58</td>
<td>23</td>
<td>11</td>
<td>3</td>
<td>26</td>
</tr>
<tr>
<td>Dragon</td>
<td>84</td>
<td>15</td>
<td>8</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>MathFP</td>
<td>825</td>
<td>205</td>
<td>109</td>
<td>81</td>
<td>32</td>
<td>16</td>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td>ManyBalls</td>
<td>214</td>
<td>48</td>
<td>25</td>
<td>19</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Kshape</td>
<td>2280</td>
<td>593</td>
<td>338</td>
<td>266</td>
<td>136</td>
<td>93</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td>Kvideo</td>
<td>102</td>
<td>23</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Kwml</td>
<td>4936</td>
<td>1099</td>
<td>571</td>
<td>425</td>
<td>167</td>
<td>83</td>
<td>19</td>
<td>285</td>
</tr>
<tr>
<td>Scheduler</td>
<td>973</td>
<td>240</td>
<td>127</td>
<td>95</td>
<td>37</td>
<td>18</td>
<td>2</td>
<td>28</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td>1063.7</td>
<td>266</td>
<td>142.2</td>
<td>107.5</td>
<td>45.6</td>
<td>25.2</td>
<td>8.6</td>
<td>53.8</td>
</tr>
</tbody>
</table>

Table 3.1. The number of GC activations for each benchmark.
invoked just in time before the new bank is activated. Therefore, it’s energy savings are competitive with the other \( k \)-allocation collectors that invoke GC more frequently. It should also be noted that the 500-allocation collector (not shown in figure) activates collections a bit late (after the second bank is accessed). In Dragon, on the other hand, we observe a different behavior. This application experiences frequent object allocations, putting a pressure on the heap memory. Therefore, the best energy results are obtained by calling the GC frequently. Consequently, the adaptive strategy and 10-allocation collector (not shown in figure) achieve the best result. Finally, in ManyBalls, the adaptive collector strikes a balance between very frequent and very slow garbage collections.

### 3.3.4 Performance Behavior

It is also important to evaluate the performance impact of our adaptive garbage collection strategy. Table 3.2 gives, for each version, the percentage increase in execution cycles with respect to the base collector. We see that the average (across all benchmarks) execution time increase due to our adaptive strategy is only 4.20\%, which is less than the increases in execution times when \( k \)-allocation strategies with \( k = 10, 40, 75, \) and 100 are employed. Considering the fact that our strategy is competitive with these allocators as far as the energy behavior is concerned, the performance overhead of the adaptive scheme is acceptable. As an example, in Dragon, our approach is competitive with the 10-allocation collector as far as the energy consumption is concerned. However, the 10-allocation collector increases the execution time of this application by 14.53\%, a much larger figure than 0.68\%, the corresponding increase in the adaptive collector.

### 3.3.5 Bank Size Variations

Recall that so far all experiments have been performed using a heap size of 128KB and a bank size of 16KB. In this subsection, we change the bank size (by keeping the heap size fixed at 128KB) and measure the sensitivity of our results to the bank size variation. Fig. 3.6 gives the normalized heap energy consumptions for two of our benchmarks: Kvideo and Dragon. Our experiments with other benchmarks showed similar trends; so, they are not reported here in detail. The results given in Fig. 3.6 are values normalized with respect to the heap energy.
Fig. 3.5. Execution footprints of Calculator, Dragon, and ManyBalls under different collection frequencies.
<table>
<thead>
<tr>
<th>Application</th>
<th>10</th>
<th>40</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>adpt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculator</td>
<td>92.58</td>
<td>21.71</td>
<td>11.78</td>
<td>8.01</td>
<td>3.00</td>
<td>1.88</td>
<td>2.86</td>
</tr>
<tr>
<td>Crypto</td>
<td>80.57</td>
<td>19.89</td>
<td>10.34</td>
<td>7.73</td>
<td>2.73</td>
<td>1.08</td>
<td>3.29</td>
</tr>
<tr>
<td>Dragon</td>
<td>14.53</td>
<td>1.08</td>
<td>0.54</td>
<td>0.40</td>
<td>0.07</td>
<td>0.03</td>
<td>0.68</td>
</tr>
<tr>
<td>MathFP</td>
<td>47.82</td>
<td>11.05</td>
<td>5.34</td>
<td>3.64</td>
<td>0.65</td>
<td>-1.21</td>
<td>0.60</td>
</tr>
<tr>
<td>ManyBalls</td>
<td>25.71</td>
<td>6.41</td>
<td>8.52</td>
<td>2.56</td>
<td>1.19</td>
<td>0.66</td>
<td>6.93</td>
</tr>
<tr>
<td>Kshape</td>
<td>40.83</td>
<td>31.97</td>
<td>14.58</td>
<td>9.74</td>
<td>1.47</td>
<td>-1.79</td>
<td>2.16</td>
</tr>
<tr>
<td>Kvideo</td>
<td>38.20</td>
<td>8.59</td>
<td>4.11</td>
<td>3.11</td>
<td>1.39</td>
<td>0.59</td>
<td>1.68</td>
</tr>
<tr>
<td>Kwml</td>
<td>92.60</td>
<td>42.63</td>
<td>20.48</td>
<td>13.98</td>
<td>3.64</td>
<td>2.22</td>
<td>9.64</td>
</tr>
<tr>
<td>Scheduler</td>
<td>70.98</td>
<td>20.53</td>
<td>16.83</td>
<td>9.13</td>
<td>10.80</td>
<td>0.27</td>
<td>9.96</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td>55.98</td>
<td>18.21</td>
<td>10.28</td>
<td>6.48</td>
<td>2.77</td>
<td>0.41</td>
<td>4.20</td>
</tr>
</tbody>
</table>

Table 3.2. Percentage increase in execution cycles (with respect to the base collector).

consumption of the base allocator with 16KB bank size and 128KB heap size. We could not run
Dragon with 4KB bank size due to the large inter-bank objects allocated by this benchmark
(currently, our implementation cannot allocate objects larger than bank size). We observe two
trends from this graph. First, the effectiveness of our adaptive collection strategy as well as that
of $k$-allocation collectors increase with the reduced bank size. This is because a smaller bank
size gives more opportunity to GC to turn off heap memory regions in a finer-grain manner. Con-
sequently, more heap memory space can be turned off. Second, our adaptive strategy continues
to follow the best $k$-allocation collector even when different bank sizes are used; that is, it can
be used with different bank sizes.

### 3.3.6 Heap Size Variations

In this subsection, we fix the bank size at 16KB (our default value) and report experimental results obtained when different heap sizes are used. We focus on Kvideo only; but, the results observed with other benchmarks are similar. The heap sizes used in this set of experiments are 64KB, 96KB, 128KB (our default), and 160KB. We observe from the results shown in Fig. 3.7 that our adaptive strategy continues to follow the best $k$-allocation collector and that its performance is not very sensitive to the heap size variation.
Fig. 3.6. Results with different bank sizes: Top: Kvideo, Bottom: Dragon.
3.4 Related Work on Garbage Collection and Memory Energy Optimization

Automatic garbage collection has been an active research area for the last two decades. The current approaches to garbage collection focus on locality-aware garbage collection (e.g., [106] and [44]), concurrent and hardware-assisted garbage collection (e.g., [66]), and garbage collection for Java (e.g., [25]) among others. A comprehensive discussion of different garbage collection mechanisms can be found in [73]. All these techniques are geared towards improving performance rather than energy consumption. We showed in this chapter that for an energy-aware collection, different GC parameters should be tuned. [55] analyzed four different memory management policies from the performance as well as energy perspectives. Our work differs from theirs in that we focus on a banked-memory architecture, and try to characterize and optimize energy impact of different garbage collection strategies when a leakage control mechanism is employed.

As Java is becoming a popular programming language for both high-end and low-end systems, researchers are focusing on different aspects of Java-based systems, including Just-in-Time compilation [86, 130, 45], garbage collection [119, 25], heap allocation behavior of Java codes [54], and synchronization optimization. Most of the Java-specific optimizations proposed
so far focus on improving performance whereas we target improving energy consumption without unduly increasing execution time.

Recently, energy optimization has become a topic of interest in the software community. Prior research [37, 78, 123] shows that program transformations can be very effective in reducing memory energy of array-dominated embedded applications. Lebeck et al. [85] and Delaluz et al. [53] specifically focused on banked-memory architectures and suggested, respectively, operating system based and compiler/hardware based optimization strategies for reducing dynamic power. Our work differs from these in that we specifically target embedded Java environments and focus mainly on exploiting leakage control mechanisms for reducing energy. We also illustrate how garbage collector can be tuned to maximize the effectiveness of leakage control. Flinn et al. [58] quantifies the energy consumption of a pocket computer when running Java virtual machine. Vijaykrishnan et al. [125] characterized the energy behavior of a high-performance Java virtual machine. In contrast to these, our work targets a banked-memory architecture and adaptively tunes garbage collector for energy optimization. Finally, numerous papers attempt to optimize energy consumption at the circuit and architectural levels. In particular, the leakage optimization circuit employed here tries to reduce leakage current and is similar to that used in [132, 79]. We employ a design that is a simple enhancement of existing voltage down converters present in current memory designs. Further, the circuit with the differential feedback stage helps to respond to load variations faster during normal operation.

3.5 Concluding Remarks

Unnecessary leakage energy can be expended in maintaining the state of garbage objects until they are recognized to be unreachable using a collector. This wasted leakage energy is proportional to the duration between when an object becomes garbage and the time when it is recognized to become garbage. While invoking the GC more frequently can reduce this wasted leakage that needs to be balanced with the energy cost of invoking the GC itself. In this chapter, we show the design of an adaptive GC strategy that tailors its invocation frequency to account for these tradeoffs based on the object allocation and garbage creation frequencies. We implemented this adaptive GC within a commercial JVM and showed that it is effective in reducing the overall system energy.
Chapter 4

Reducing Energy Consumption by Memory Compression

4.1 Introduction

In this chapter, we use a system-on-a-chip (SoC) with two-level memory hierarchy where a software-managed memory known as scratch pad memory (SPM) is used between the memory and the processor core. The SPM is associated with less per access dynamic energy cost due to its smaller size. Hence, confining most accesses to the smaller SPM instead of large main memory reduces the overall dynamic energy. However, the increased memory space due to the two-level memory hierarchy can increase the overall leakage energy of the system. Various compression schemes have been widely used to reduce the memory requirements. In this work, we use compression to reduce the size of the required memory. Specifically, we store the code of the embedded JVM system and the associated library classes in a compressed form in the memory. Thus, the effective number of active transistors used for storage and the associated leakage are reduced. We employ a mechanism that turns off power supply to the unused portions of the memory to control leakage. Whenever the compressed code or classes are required by the processor core, a mapping structure stored in a reserved part of the SPM serves to locate the required block of data in the compressed store. Then, the block of data after decompression is brought into the SPM. Thus, the use of scratch pad memory in conjunction with a compressed memory store targets the reduction of both dynamic and leakage energy of a system. The focus of this chapter is on investigating the influence of different parameters on the design of such a system. The issues addressed in this work are listed below:

- Storing compressed code or data has an associated decompression cost from both the energy and performance aspects. To obtain any energy savings, the energy overhead of decompression must be smaller than the leakage energy savings obtained through storage of compressed code. The underlying compression algorithm and the corresponding implementation of the decompression critically impact the energy and performance overhead.
We explore this idea using a specific hardware compression scheme and also experiment with different decompression overheads to account for a range of possible implementations from customized hardware to software.

- The size of the compressed block influences both the compression ratio and the overhead involved in indexing the compressed data. A larger granularity of compression, typically, provides a better compression ratio. In turn, this provides an ability to turn off power supply to more unused memory blocks, thereby providing larger leakage energy savings. Also, it reduces the mapping overhead for indexing into the compressed store. However, a larger block also occupies a larger space in the SPM and increases the storage pressure. This can lead to more frequent conflicts in the scratch pad memory resulting in more frequent decompressions.

- Entities (native functions, Java method bytecodes and constant pools, etc) in Java virtual machine are not equally used. So different portions are decompressed different number of times during runtime. Based on their hotness, we determine whether compression is beneficial (or not) and selectively compress the beneficial portions of the code. This technique reduces the overall system energy by up to 10%.

- A longer reuse is essential to amortize the additional energy expended in decompressing when transferring from the memory to the scratch pad. Based on the traces of the applications, we identify native functions in the virtual machine and Java methods in the classes library that are frequently used together. Clustering items that are frequently used together improves reuse and thus saves energy.

### 4.2 SoC Architecture

A system-on-a-chip (SoC) is an integrated circuit that contains an entire electronic system in a single sliver of silicon. A typical SoC contains a library of components designed in-house as well as some cores from chipless design houses also known as intellectual property. In this work, we focus on an SoC-based system that executes KVM applications. Fig. 4.1(a) depicts the high level (logical) view of the relevant parts of our SoC architecture. This architecture has a CPU core, a scratch-pad memory (SPM), and two main memory modules. The processor in our
SoC is a microSPARC-IIep embedded core. This core is a 100MHz, 32-bit five-stage pipelined RISC that implements the SPARC architecture V8 specification. It is primarily targeted for low-cost uniprocessor applications. Both main memory and SPM are SRAMs which are organized as blocks. Unlike conventional SPM with fixed address, each block of our SPM is dynamically mapped into a virtual address space which is as large as main memory. Each main memory block can be dynamically loaded into one SPM block. Each SPM block has a tag register indicating its virtual address. The tag registers are set by SPM manager. When the CPU generates an address, the high-order bits of this address are compared with each tag in parallel. If one of the tags generates a match, the corresponding SPM block is selected and low-order bits of the address are used to access the contents of the block. If no tag match occurs, then the “Hit” signal line (shown in Fig. 4.1(b)) is disabled and an interrupt is generated. The corresponding interrupt service routine activates the SPM manager which brings the faulted block from main memory to the SPM. In case no free SPM block is available, a timer-based block replacement policy is used. Specifically, for each SPM block, there is a timer and an access bit. Whenever the block is accessed, its access bit is set and its timer is reset. When a block is not accessed for a certain period of time, the timer goes off and the access bit is reset. When a block replacement is to be performed, the SPM Manager always tries to select a block whose access bit is reset. If no such block exists, the manager selects a block in a round-robin fashion. The main memory is composed of two parts: one part which contains the KVM code and class libraries and the other part which contains all writable data including heap and C stack as well as application code.

It should be mentioned that a number of parameters in this architecture are tunable. For example, the capacities of SPM and main memory can be modified. Also, by playing with the width of the timers associated with each block, we can modify the behavior of the block replacement policy. Finally, the SPM block size can be changed. Note that changing the block size affects the block replacement rate as well as the overhead (per block) when a replacement occurs.

Our SPM is more flexible than cache in that SPM has its own address space and is managed by software. We can customize the management policy according to specific applications. For example, we can bypass SPM when accessing infrequently used data, or we can pin some frequently accessed data in the SPM, or we can use it as a conventional SPM by fixing the values
of the tag registers. In our experiments, we found that using 2-way associate 32KB instruction and data caches both with line sizes of 32 bytes consumed 11% more energy than an equivalent 64KB SPM configuration.

4.3 Compressing KVM Code and Class Libraries

As noted earlier, leakage energy consumption of SRAM blocks is proportional to their size as well as the duration of time that they are powered on. In this work, we try to reduce the number of active (powered on) memory blocks by storing read-only data, including KVM binary codes and Java class libraries, in compressed form. To avoid incurring the cost of runtime compression, writable data are stored in original form. The high-level view of our architecture with the decompression support is shown in Fig. 4.2(a). When a data item belonging to a compressed memory block is requested by the processor, the whole block is decompressed by the decompressor and is then written into the SPM. The advantage of this strategy is that since data in read-only memory is in the compressed form, it occupies fewer memory blocks. This, in turn, reduces the leakage energy consumption in read-only part of the main memory system. Note that the amount of this saving is determined by the compression rate of the algorithm used. The drawback is that decompressing a block (at runtime) incurs both time and energy penalty. The
Fig. 4.2. (a) High-level view of the SoC memory architecture with decompressor. (b) Details for the modified X-Match decompressor.

The magnitude of this penalty depends on how frequently decompression is required and how much time/energy it takes to decompress a block of data. The number of decompressions is directly related to the number of misses in the SPM (which is a characteristic of application behavior and SPM configuration). The time/energy expended in decompressing the block depends on the decompression method used and can be reduced by an efficient implementation.

The compression/decompression algorithm to be used in such a framework should have the following important characteristics: (1) good compression ratio for small blocks (typically less than 4KB); (2) fast decompression; and (3) low energy consumption in decompression. Since compression is performed offline, its speed and energy overhead are not constrained. The first characteristic is desirable because the potential leakage energy savings in memory are directly related to the number of memory blocks that need to be powered on. Note that a compressed memory block needs to be decompressed before it can be stored in the SPM. Consequently, a decompression overhead is incurred in every load to SPM and, in order for this scheme to be effective, we should spend very little time and energy during decompression.

Kjelso et al. [80] presented a dictionary-based compression/decompression algorithm called X-Match. This algorithm maintains a dictionary of data previously seen, and attempts to match the current data element (to be compressed) with an entry in the dictionary. If such a match occurs, the said data element is replaced with a short code word indicating the location of data
in the dictionary. Data elements that do not generate a match are transmitted in full (literally), prefixed by a single bit. Each data element is exactly 4 bytes in width and is referred to as a tuple. A full match occurs when all characters in the incoming tuple fully match a dictionary entry. A partial match occurs when at least two of the characters in the incoming tuple match exactly a dictionary entry; the characters that do not match are transmitted literally. The coding function for a match encodes three separate fields: (1) match location, (2) match type indicating which characters from the incoming tuple matched the dictionary entry, and (3) any characters from the incoming tuple which did not match the dictionary entry at the match location (i.e., those transmitted without encoding). The decompressor of X-Match is implemented as a 3-staged pipeline. Compared to the baseline scheme (Fig. 4.1), the performance degradation introduced by the decompressor is negligible (2 cycles' delay for each block loading).

In the original X-Match algorithm, the dictionary is maintained using a move-to-front strategy, whereby the current tuple is placed at the front of the dictionary and other tuples move down by one location. If the dictionary becomes full, the tuple occupying the last location is simply discarded. The move-to-front operation is implemented with content addressable memory, which is expensive from the energy consumption perspective. In our implementation, we replaced the move-to-front strategy with a simple round-robin strategy, i.e., the new tuple is always appended to the end of current dictionary entries. When the dictionary is full, the replacement pointer is moved to the first dictionary entry and the entry becomes the one that will be replaced next time. The elimination of the move-to-front strategy may cause a slight degradation in the compression ratio, but the implementation is simpler and energy-efficient. We also separate the literal bytes from the control bits (i.e., prefixes, match types, and dictionary locations), which allows the control bits and literal bytes to be fed into the decompressor as separate streams. We refer to this modified algorithm as the modified X-Match algorithm in the rest of the chapter. The hardware block diagram of the modified X-Match decompressor is shown in Fig. 4.2(b).

While this modified X-Match implementation is used in our evaluation, the idea of trading additional decompression energy with reduced memory leakage energy is applicable using other compression schemes such as Lempel-Ziv and Huffman.
4.4 Simulation Methodology

4.4.1 Energy Model

The energy numbers reported in this chapter are obtained by a simulator implemented on the SPARC simulation tool-set, Shade [48], augmented with energy models. The simulator takes as input the KVM system executing a Java application and computes performance as well as energy data. The current implementation runs on SPARC systems and simulates the SPARC V8 instruction set of our target processor. Our simulator tracks energy consumption in the processor core, SPM, and main memory blocks. The energy consumed in the processor core is estimated by counting (dynamically) the number of instructions of each type and multiplying the count by the base energy consumption of the corresponding instruction. The base energy consumptions of the different instruction types are obtained using a customized and validated version of our in-house cycle accurate energy simulator [123]. The simulator is configured to model a five-stage pipeline similar to that of the target microSPARC-IIep architecture.

The energy consumption in SPM and main memory is divided into two components: dynamic energy and leakage energy. In computing per access dynamic energy consumptions for SPM and main memory, we used the CACTI tool Version 2.0 [108] assuming a 0.10 micron technology. In computing the leakage energy, we assumed that the leakage energy per cycle of the entire main memory is equal to the dynamic energy consumed per access. This assumption tries to capture the anticipated importance of leakage energy in the future. It should be stressed that leakage becomes the dominant part of energy consumption for 0.10 micron (and finer) technologies for the typical internal junction temperatures in a chip [38]. Note that, as opposed to dynamic energy which is expended only when an access occurs, leakage energy is spent as long as memory is powered on.

In computing the overall energy consumption in main memory and SPM, we assumed that a memory block (or an SPM block) can be in one of three states (modes) at any given time: R/W, active, or inactive. In the R/W (read/write) mode, memory is being read or written and consumes full dynamic energy as well as full leakage energy. In the active state, on the other hand, the memory is powered on but not being accessed. In this state, it consumes no dynamic energy but full leakage energy. Finally, the memory modules that are not needed by
the system are not powered on, i.e., in the inactive state, consequently, no energy consumption at all. Obviously, one would want to place as many memory blocks as possible to the inactive state so that the energy consumption can be minimized. One way of achieving this is to reduce the amount of data stored in memory, which can be achieved using compression.

### 4.4.2 Base Configuration and Energy Distribution

Table 4.1 gives the simulation parameters used in our base configuration. Table 4.2 shows (in columns two through five) the energy consumptions (in micro-joules) for our applications executing on base configuration without decompression. The energy consumption is divided into four components: dynamic energy in SPM, leakage energy in SPM, dynamic energy in main memory, and leakage energy in main memory. The contribution of the processor energy to the overall (main memory + SPM + processor) energy is around 10% and is not much affected by decompression. Consequently, we focus only on main memory and SPM energies. A memory block that contains no valid information throughout the application execution is turned off so that it does not consume any leakage energy. We see from these results that the memory leakage energy consumption (shown in the third column) constitutes a large percentage of the memory system (main memory + SPM) energy budget (61.74% on the average) and is a suitable target for optimization. The sixth column in Table 4.2 gives percentage of energy consumption due to read-only part of the memory. We see that, on the average, the read-only part of the memory is responsible for 62.42% of the overall memory energy consumption. Finally, Table 4.3 gives the number of SPM misses and the number of execution cycles (in millions) for each application.

### 4.5 Results

In this section, we present data showing the effectiveness of our strategy in saving energy and also measure the sensitivity of our strategy to different parameters such as SPM capacity, block size, and cost of decompression. All energy numbers reported here are values normalized to the energy consumption in the base case without any decompression (Table 4.2). Also, when a simulation parameter is modified, the remaining parameters maintain their original values given in Table 4.1.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPM capacity</td>
<td>64KB</td>
</tr>
<tr>
<td></td>
<td>4KB SPM management</td>
</tr>
<tr>
<td></td>
<td>40KB for read-only data</td>
</tr>
<tr>
<td></td>
<td>20KB for writable data</td>
</tr>
<tr>
<td>SPM block size</td>
<td>1KB for read-only data</td>
</tr>
<tr>
<td></td>
<td>512 bytes for writable data</td>
</tr>
<tr>
<td>Main memory capacity</td>
<td>512KB</td>
</tr>
<tr>
<td>SPM access time</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Main memory access time</td>
<td>3 cycles</td>
</tr>
<tr>
<td>SPM dynamic energy/read</td>
<td>0.5216 nJ</td>
</tr>
<tr>
<td>SPM dynamic energy/write</td>
<td>0.6259 nJ</td>
</tr>
<tr>
<td>Main memory dynamic energy/read</td>
<td>1.334 nJ</td>
</tr>
<tr>
<td>Main memory dynamic energy/write</td>
<td>1.601 nJ</td>
</tr>
<tr>
<td>Main memory leakage energy/byte/cycle</td>
<td>$2.54 \times 10^{-6}$ nJ</td>
</tr>
<tr>
<td>SPM leakage energy/byte/cycle</td>
<td>$2.54 \times 10^{-6}$ nJ</td>
</tr>
<tr>
<td>SPM access bit reset time</td>
<td>6000 cycles for read-only</td>
</tr>
<tr>
<td></td>
<td>4000 cycles for r/w clean</td>
</tr>
<tr>
<td></td>
<td>8000 cycles for r/w dirty</td>
</tr>
</tbody>
</table>

Table 4.1. Simulation parameters and their values for our base configuration.

The top part of Fig. 4.3 gives the normalized energy consumptions in read-only portion of the main memory and the SPM. It can be observed from this figure that the energy saving is 20.9% on the average. The bottom part of Fig. 4.3, on the other hand, shows the overall (normalized) energy consumption in main memory and SPM, including the energy expended during decompression. We see that most of the applications achieve an overall energy saving of 10% (an average of 7% across all applications). In two applications (Calculator and Scheduler), the decompression overhead (energy) plays a larger role and the overall energy consumption becomes worse than the original case. We also experimented with a 50% reduction in leakage energy per main memory cell to account for design variations that permit the slower main memory cells to operate using a higher threshold voltage. In this case, the overall memory system energy saving across all applications is 5.4% on the average.

In general, there are two application-related factors that determine the effectiveness of our energy saving strategy: (1) the overall running time of the application, (2) the number of SPM misses. Since the major energy gain in our strategy comes from the memory leakage
<table>
<thead>
<tr>
<th>Application</th>
<th>Memory Energy (nJ)</th>
<th>SPM Energy (nJ)</th>
<th>Read-Only Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dynamic</td>
<td>Leakage</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Calculator</td>
<td>1.81</td>
<td>7.46</td>
<td>3.08</td>
</tr>
<tr>
<td>Crypto</td>
<td>7.30</td>
<td>137.94</td>
<td>54.97</td>
</tr>
<tr>
<td>Dragon</td>
<td>1.10</td>
<td>68.70</td>
<td>27.18</td>
</tr>
<tr>
<td>Elite</td>
<td>1.10</td>
<td>64.42</td>
<td>25.49</td>
</tr>
<tr>
<td>MathFP</td>
<td>2.42</td>
<td>104.96</td>
<td>41.70</td>
</tr>
<tr>
<td>ManyBalls</td>
<td>5.90</td>
<td>73.08</td>
<td>29.22</td>
</tr>
<tr>
<td>Missiles</td>
<td>2.51</td>
<td>60.34</td>
<td>23.96</td>
</tr>
<tr>
<td>KShape</td>
<td>16.18</td>
<td>201.90</td>
<td>80.93</td>
</tr>
<tr>
<td>KVideo</td>
<td>2.21</td>
<td>12.69</td>
<td>5.17</td>
</tr>
<tr>
<td>KWML</td>
<td>121.56</td>
<td>584.96</td>
<td>240.78</td>
</tr>
<tr>
<td>Scheduler</td>
<td>32.96</td>
<td>140.51</td>
<td>57.83</td>
</tr>
<tr>
<td>StarCruiser</td>
<td>3.22</td>
<td>48.96</td>
<td>19.58</td>
</tr>
</tbody>
</table>

Table 4.2. Energy consumptions for our applications under the base configuration.

energy, the longer the application runs, we can expect more energy benefits. Recall that each SPM misses invokes a decompression. Therefore, the number of SPM misses is an important factor in determining the energy spent in decompression during the course of execution. The reasons that Calculator and Scheduler do not get benefit from our strategy are different.

In Calculator, the execution time is rather short (only 5.59 million cycles) and the energy spent on decompression does not pay off. On the other hand, although the execution time of Scheduler is not short (105.5 million cycles), it suffers from a high number of SPM misses (a total of 96033).

Since our strategy focuses on energy savings in the read-only part of the main memory, in the rest of this section, we mainly present results pertaining only this part and the SPM (unless otherwise stated). However, to evaluate the impact of decompression, we also show the energy consumed during the decompression process.

4.5.1 Sensitivity to the Decompression Cost

To see how a more efficient or a less efficient implementation of the X-Match decompressor would impact our results, we conducted another set of experiments. Assuming a decompression rate of 4 bytes/cycle and that the energy consumed in each stage is equal to one SPM access, we determined that the energy consumption for decompressing one word (4 bytes)
Fig. 4.3. Normalized energy consumption in read-only memory and SPM (top) and in overall memory and SPM (bottom).
Table 4.3. Execution cycles and SPM misses for our applications under the base configuration.

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of SPM Misses</th>
<th>Number of Cycles (10^6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculator</td>
<td>5258</td>
<td>5.59</td>
</tr>
<tr>
<td>Crypto</td>
<td>19848</td>
<td>103.58</td>
</tr>
<tr>
<td>Dragon</td>
<td>3064</td>
<td>51.58</td>
</tr>
<tr>
<td>Elite</td>
<td>3206</td>
<td>48.37</td>
</tr>
<tr>
<td>MathFP</td>
<td>6735</td>
<td>78.81</td>
</tr>
<tr>
<td>ManyBalls</td>
<td>11455</td>
<td>54.87</td>
</tr>
<tr>
<td>Missiles</td>
<td>7288</td>
<td>45.30</td>
</tr>
<tr>
<td>KShape</td>
<td>45321</td>
<td>151.61</td>
</tr>
<tr>
<td>KVideo</td>
<td>6396</td>
<td>9.53</td>
</tr>
<tr>
<td>KWML</td>
<td>339395</td>
<td>439.26</td>
</tr>
<tr>
<td>Scheduler</td>
<td>96035</td>
<td>105.50</td>
</tr>
<tr>
<td>StarCruiser</td>
<td>9356</td>
<td>36.76</td>
</tr>
</tbody>
</table>

is equal to three SPM accesses. We normalize this energy cost to 1 and experiment with its multiple as well as its fractions. The results shown in Fig. 4.4 are average values (over all benchmarks) and illustrate that the relative cost of decompression can change the entire picture. For example, with a relative cost of 2, the energy consumption exceeds that of the original case. In contrast, with a relative cost of 0.25, the energy consumption is around 80%, even including the decompression energy. These results clearly indicate the importance of efficient implementation of decompression.

4.5.2 Sensitivity to the SPM Size

Fig. 4.5 shows the impact of SPM capacity (size) on energy savings. As before, the values shown are averages computed over all benchmark codes in our experimental suite. We observe from this figure that, if the SPM size is too small, frequent SPM misses make energy consumption very high. But, we also see that a very large SPM also degrades energy behavior. There are two factors that together create this behavior. First, a larger SPM itself consumes more dynamic and leakage energy (compared to a smaller SPM). Second, for each application, there is an SPM capacity that captures the working set. Increasing the SPM size beyond this capacity does not reduce the number of misses further. So, this stability in the number of misses, combined with the first factor, leads to an increase in energy consumption. In an embedded system
design, the maximum possible SPM size is determined by the chip budget. Our experimentation indicates that the best SPM size depends on the application at hand. So, embedded system designers should select a suitable SPM size considering the applications in question as well as the impact of SPM size on energy consumption.

4.5.3 Sensitivity to the Block Size

In this set of experiments, we tried to measure the sensitivity of our energy savings to the block size used. Recall that our default block size was 1KB. The results shown in Fig. 4.6 indicate that, given the SPM capacity, the size of each block has a great impact on the energy consumption. For most compression algorithms in the literature, a larger block size has, in general, a better compression ratio (Fig. 4.7). In addition, given the same SPM size, smaller block size increases block number, which complicates the mapping mechanism. However, it should be noted that, a very large block size might increase both SPM miss rate and miss penalty (decompression cost). That is exactly the behavior we observed during our experiments. As shown in Fig. 4.6, a block size 0.5KB generated better results than our default 1KB blocks. In
contrast, increasing the block size to 2KB increased the original energy consumption by more than a factor of two.

4.5.4 Selective Compression

Compression reduces the memory size and thus leakage energy, but it also introduces decompression overhead. The idea behind selective compression is that only the blocks for which compression brings net benefit are stored in a compressed format. The other blocks are in an uncompressed format. Specifically, a block $B$ is stored in compressed format only if $\text{load}(B) \times \text{cost}(B) < L \times T \times \text{size}(B) \times (1 - r(B))$, where $\text{load}(B)$ is the number that block $B$ is loaded from main memory, $\text{cost}(B)$ is decompression cost of $B$, $L$ is the leakage energy per byte per cycle, $T$ is the overall execution time in cycles, $\text{size}(B)$ is the size of block $B$, and $r(B)$ is the compression rate of block $B$.

In this work, we evaluated two selection strategies: application-customized selection and global selection. The application-customized strategy is based on the traces of a specific application; that is, for each application, the blocks to be stored in the compressed form are determined considering only the KVM behavior of that application. Consequently, this strategy
may achieve the optimal energy consumption of the particular application. However, since this optimization is completely for one application, it may degrade others. In other words, the KVM blocks that need to be compressed for the best behavior of one application may not be suitable candidates (for compression) for other applications. Our second strategy, global selection, is to select blocks for compression according to the traces of a group of applications. Each application has a specific weight based on the frequency of its usage. Tuning the weights with a group of carefully selected representative applications may bring general improvements on a variety of applications. The results shown in Fig. 4.8 indicate that the application-customized strategy generates the best results.
Fig. 4.8. Comparison of different compression strategies. The three groups of bars from left to right correspond to our default strategy, an application-customized selective strategy, and a global selective strategy.

4.5.5 Clustering

Compared to a cache line, our SPM block has larger granularity, and thus is more sensitive to the applications’ spatial locality. Putting items that are frequently used together into the same block can reduce accesses to main memory, and thus decompression energy. We exploit such clustering in both KVM binary and Java Class Library. Clustering in KVM binary is performed at the native function and Java method granularity. Using profiling information, we first build the weighted call graph, in which each node represents a native function in KVM binary or a Java method in the Java class library. The weight of each edge in this graph indicates how frequently each pair of native functions or Java methods are called in a row. Then a greedy algorithm is applied to clustering the functions and methods according to the weights of edges. Similar to selective compression, clustering can be performed in either application-customized or global manner. In this work, we report only the results of global clustering in Fig. 4.9. For
most applications, clustering saves energy. However, since clustering is performed according to
the common memory access patterns of all applications, the energy behaviors of some applica-
tions that have exceptional memory access patterns, such as MathFP, may degrade.

### 4.6 Concluding Remarks

Storing compressed code or data has an associated decompression cost from both the
energy and performance aspects. However, compression itself helps to reduce the portion of the
memory to be powered on and the consequent leakage energy of the memory system. Our exper-
iments with a set of embedded applications using a commercial embedded JVM and a specific
compression scheme show that the proposed technique is effective in reducing system energy.
We expect our findings to be applicable to other compression algorithms and implementations
as well.
Chapter 5

Offloading Computation/Compilation

5.1 Introduction

Wireless communication technology has been improved dramatically in the recent years. Using today’s 3G technology, we are able to achieve up to 2Mbps data rate through wireless connection. It is expected that, in 4G technology, we are going to have wireless connection with up to 100Mbps data rate and tight security for low cost. Using this high-speed and reliable wireless connection, one can offload computation tasks of mobile devices (clients) to the resource-rich servers to reduce the energy consumption of the clients. Specifically, the client sends the parameters of a task to the server; the server executes the task and sends back the results. The energy of the client is saved if the energy for sending the parameters and receiving the results is lower than the energy for executing the the task locally on the client. A lot of research [59, 91, 90, 82] has already been carried out to exploit such tradeoffs to reduce the overall energy consumption. As a specific example, Flinn and Satyanarayanan [59] have proposed an environment where applications dynamically modify their behaviors to conserve energy by lowering data fidelity and adjusting the partition of computation tasks between client and server.

The results of these prior efforts are also relevant to Java-based mobile systems. However, due to its bytecode nature, Java has its unique aspects. To achieve the cross platform compatibility, Java applications are presented in platform independent bytecode. The bytecode is executed by Java Virtual Machine (JVM) either by interpreter or by Just-In-Time (JIT) compiler. Executing Java bytecode by interpreting consumes more energy than executing the equivalent native code. JIT compiler reduces the energy consumption for code execution by compiling bytecode into native code that can be executed directly by the processor. Further, JIT compiler can even optimize the compiled code so that it can be executed more efficiently. However, compilation and optimization have their own overheads. Generally speaking, native codes generated
with higher level optimizations are more energy efficient, but performing such optimizations usually consumes more energy.

While off-loading computational load to resource-rich servers sounds promising, there are several practical issues that need to be addressed. Maybe the most pressing of these issues is the question of whether offloading would preserve the original execution semantics. Fig 5.1(a) depicts a Java runtime environment. The application is composed of a set of software components. They interact with each other through pre-defined interfaces. Each interface defines a set of public methods and the protocols on the usage of these methods. The implementation of these methods, however, is concealed from other components. Ideally, given a well-defined interface, altering the implementation of its methods does not affect the behavior of the application. For example, in Figure 5.1(b), component C is replaced by a proxy having the identical interface. When a method belonging to its interface is invoked, based on some policy, the proxy can forward the invocation (the method name and the parameters) to either the local or remote implementation. The proxy may also provide interface to component C, through which the actual implementation is able to access the interface of the other components. This interface is not necessary for the components that do not rely on other components.

Similarly, by encapsulating the JIT compiler as a component, we can use the proxy to enable remote execution of the JIT compiler in Fig 5.1(b). Specifically, the client can pass to the server the optimization level and the name of the method to be compiled. The server compiles the bytecode of the method into native code and returns the compiled code. Since the application running on the client is initially downloaded from the server, the server has the bytecode of the method, and the client does not need to send the bytecode of the “to be compiled” method. It should noted that native code, unlike Java bytecode, cannot be verified. So a signature/authentication mechanism is necessary for remote compilation (further discussion on this topic is beyond the scope of this thesis).

Therefore, based on the discussion above, by using proxies, we can dynamically offload part of execution and compilation to the server safely, which might be particularly meaningful for the applications where energy cost of compilation, execution and offloading can vary based on the operating conditions and user-supplied input. When making offloading decision, information about the code and data as well as external factors such as communication channel
conditions are required. The information about the energy complexity of the computation can be obtained through user-specified or compiler-directed annotation. This information can also be estimated by profiling. The compilation (as well as optimization) costs are static and can be obtained through off-line profiling. Wireless channel conditions, on the other hand, need to be detected at the time when the offloading decision is being made. A careful balance needs to be struck between the additional energy needed for the required communication for the offloaded computation and the energy required for the computation when offloading is not used. However, it should also be noted that not all the components of a given Java application are suitable for execution offloading. The following are some guidelines for determining if a component should be considered for remote execution: (1) a component that accesses local I/O devices should not be offloaded; (2) a stateful component should not be offloaded unless its state (i.e., the contents of its active data entities) can be migrated with low cost; (3) a component whose either execution or offloading costs is completely unpredictable is not suitable for dynamic offloading; and finally, (4) a component that frequently invokes the methods of other components should not be offloaded unless all its relevant components can also be offloaded.

In this chapter, we evaluate opportunities for saving energy in a Java-enabled mobile client by cooperating with a resource-rich server during compilation and execution of an application. Specifically, we evaluate remote and local compilation/execution opportunities using various Java applications and the LaTTe Java Virtual Machine [131]. Our results indicate the critical need for dynamically deciding whether to execute locally (on the client) or remotely (on the server) as the energy saving potential and the decisions vary based on both the application characteristics and external conditions. Based on this observation, we propose an adaptive strategy that dynamically chooses between remote and local execution based on user inputs and wireless channel conditions. This adaptive strategy is shown to provide significant energy savings as compared to static schemes that execute methods only locally or remotely. Next, we augment our adaptive execution strategy that performs compilation locally with an adaptive compilation approach that chooses between local or remote compilation dynamically. We demonstrate that our adaptive compilation strategy provides additional energy savings for the applications in our experimental suite. Finally, we show the potential for further energy savings by executing some
Fig. 5.1. An execution environment based on composable Java components.
Java methods on dynamically configured datapaths and by employing compression selectively on the data transmitted to the client.

5.2 Target Platform and Benchmarks

The target remote server platform is a SPARC workstation clocked at 750MHz. The target client platform (simulated) is a mobile PDA (see Fig. 5.2) that has the ability to communicate with the server. We use Wide-band Code-Division Multiple-Access (W-CDMA) as the wireless communication technology. It is widely used for implementation of third-generation (3G) cellular systems. The major components of the system include the processor core, off-chip main memory, and the communication chip set. We do not model the energy consumption of other system components, such as input/output devices, as operations using these devices need to be performed locally and hence, their energy consumption is largely unaffected by our approach. Also, unlike high-end PCs and workstations, the embedded devices targeted by this work typically have limited I/O components. They have no hard-disk driver, no CD driver, etc. Their LCDs are small, and the back-light can dim-down to save energy.

The processor core of the target client platform is based on the microSPARC-IIep embedded processor. This core is a 100MHz, 32-bit five-stage pipelined RISC architecture that implements the SPARC v8 specification. It is primarily intended for low-cost embedded applications. It has an on-chip 8KB direct-mapped data cache and a 16KB instruction cache. The off-chip main memory is assumed to be a 32MB DRAM module. To obtain the detailed energy profiles of the processor and memory system, we customized an energy simulator and analyzer using the Shade \[48\] (SPARC instruction set simulator) tool-set, and simulated LaTTe \[131\] JVM executing a Java code. Our simulator tracks the energy consumptions in the processor core (datapath), on-chip caches, off-chip DRAM module and the wireless communication components. The energy consumed in the processor core is estimated by counting (dynamically) the number of instructions of each type and multiplying the count by the base energy consumption of the corresponding instruction. The energy consumptions of the different instruction types are obtained using a customized version of the SimplePower energy simulator \[123\], and are shown in Table 5.1. The simulator is configured to model a five-stage pipeline similar to that of the microSPARC-IIep architecture. The DRAM energy cost is obtained from data sheets \[16\].
The components which are related to communication are represented inside the dashed box on the left part of Fig 5.2 (shown on the left are the transmitter components and on the right the receiver components). The transmit amplifier is a high-power amplifier for W-CDMA. The driver amplifier is a linear variable gain amplifier that maintains linearity over a specified power range. The modulator used is a monolithic integrated quadrature modulator that converts baseband signals to high-frequency signals. The Analog-to-Digital Converter (ADC) and Digital-to-Analog Converter (DAC) provide an interface between analog signals and digital components for wireless applications. The Intermediate Frequency (IF) demodulator with Automatic Gain Controller (AGC) is used for quadrature demodulation and IF AGC amplification. It converts IF signals to baseband signals. The down-converter with Linearizer Amplifier (LNA)/Mixer is used for converting Radio Frequency (RF) signals to IF signals and for compensating the received signals. Finally, the Voltage-Controlled Oscillator (VCO) provides the reference clock for both the modulator and demodulator. The transmitter and receiver components are in a power-down state when they are not used.

The communication components of our system support an effective data rate of 2.3Mbps, and can operate with four different power control settings for transmitting data. The power consumption numbers of the transmitter power amplifier vary from a Class 1 setting for poor channel condition (power = 5.88W) to a Class 4 setting for the best (optimal) channel condition (power = 0.37W). This adaptive power setting is useful because mobile wireless channels exhibit variations that change with time and the spatial location of a mobile node. A number of factors contribute to this phenomenon. Multipath fading that arises due to the reception of multiple copies of a transmitted signal that are randomly attenuated and delayed can cause rapid changes in the received signal strength. Also, when multiple mobile clients are co-located, multi-user interference can further contribute to the average bit error rate. To combat these effects, channel-adaptive communication techniques such as power control are employed. This in turn means that a fairly accurate and fast channel condition estimation mechanism is necessary. One such mechanism that is employed by wireless standards such as the IS-95 CDMA system is the usage of a pilot channel [61]. Here, pilot CDMA signals are periodically transmitted by a base station to provide a reference for all mobile nodes. A mobile station processes the pilot signal, and chooses the strongest signal among the multiple copies of the transmitted signal to arrive
<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>4.814 nJ</td>
</tr>
<tr>
<td>Store</td>
<td>4.479 nJ</td>
</tr>
<tr>
<td>Branch</td>
<td>2.868 nJ</td>
</tr>
<tr>
<td>ALU(Simple)</td>
<td>2.846 nJ</td>
</tr>
<tr>
<td>ALU(Complex)</td>
<td>3.726 nJ</td>
</tr>
<tr>
<td>Nop</td>
<td>2.644 nJ</td>
</tr>
<tr>
<td>Main Memory</td>
<td>4.94 nJ per access</td>
</tr>
</tbody>
</table>

Table 5.1. Energy consumption values for processor core and memory.

at an accurate estimation of its time delay, phase, and magnitude. These parameters are tracked over time to help the mobile client decide on the power-setting for its transmitter. In our simulation infrastructure, we model such tracking by varying the channel state using user-supplied distributions.

The energy consumption due to communication is evaluated by modeling the individual components of the W-CDMA chip set. The power consumptions of the individual components, obtained from data sheets [15, 2], are shown on the right part of Fig 5.2. The energy cost of communication is evaluated by using the number of bits transmitted/received, the power values of the corresponding components used, and the data rate.

We further optimize the energy consumption during remote execution by placing the processor, memory and the receiver into a power-down state (mode). In the power-down state, the processor still consumes leakage (static) energy which is assumed to be 10% of the normal power consumption in our target design. The estimate of the time for executing a method remotely at the server is used by the client to determine the duration of its power-down state. When the server is done with its computation, it checks the “mobile status table” (that also contains the estimated execution times) to see if the mobile client is awake. This can be accomplished as follows. The server computes the difference between the time the request was made by the client and the time when the object for that client is ready. If this difference is less than the estimated power-down duration (for the client), the server knows that the client will still be in power-down mode, and queues the data for that client until it wakes up. In case the server-side computation is delayed, we incur the penalty of early re-activation of the client from the power-down state.
Fig. 5.2. Top: System architecture. Bottom: Power consumption values for communication components. Rx and Tx refer to receiver and transmitter components, respectively.
<table>
<thead>
<tr>
<th>Application</th>
<th>Brief Description</th>
<th>Size Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>fe (Function-Evaluator)</td>
<td>Given a function $f$, a range $x$, and a step size, calculates the integral of $f(x)$ in this range</td>
<td>Step size and range</td>
</tr>
<tr>
<td>pf (Path-Finder)</td>
<td>Given a map and a source location (node), finds the shortest path tree with the source location as root</td>
<td>Number of nodes and number of edges</td>
</tr>
<tr>
<td>mf (Median-Filter)</td>
<td>Given an image (in PGM format) and the size of the window, generates a new image by applying median filtering</td>
<td>Image size and filter window size</td>
</tr>
<tr>
<td>hpf (High-Pass-Filter)</td>
<td>Given an image and a threshold, returns the image after filtering out all frequencies below the threshold</td>
<td>Image size and threshold frequency</td>
</tr>
<tr>
<td>ed (Edge-Detector)</td>
<td>Given an image, detects its edges by using Canny's algorithm</td>
<td>Image size</td>
</tr>
<tr>
<td>sort (Sorting)</td>
<td>Sorts a given set of array elements using Quicksort</td>
<td>Array size</td>
</tr>
<tr>
<td>jess (Jess)</td>
<td>An expert system shell from SpecJVM98 benchmark suite</td>
<td>Number of rules</td>
</tr>
<tr>
<td>db (Db)</td>
<td>A database query system from SpecJVM98 benchmark suite</td>
<td>Size of database and length of query expression</td>
</tr>
</tbody>
</table>

Table 5.2. Description of our benchmarks.

Table 5.2 lists the Java applications used in this study. None of these applications make any native method calls. Median-Filter, High-Pass-Filter, and Edge-Detector are codes that are used very frequently in embedded image and video processing. Function-Evaluator and Path-Finder are applications available in many hand-held devices and digital assistants. Sorting is a frequently-used utility package in many application domains. Our last two benchmarks (jess and db) are from SpecJVM98. As these two are benchmarks for high-performance JVMs, to make them behave more like typical applications running on embedded systems, their smallest input size (s1 dataset) was used. To make offloading possible, some necessary modifications have also been made to their codes. Their core logic, however, is carefully retained. Note that rest of the SpecJVM98 benchmarks are not representative codes targeted by our execution environment. In the rest of this chapter, we use these eight benchmark codes to evaluate different execution and compilation strategies.
5.3 Analysis of Execution and Compilation Strategies

In our approach, we attempt to partition the activities during Java code execution across the mobile client and the resource-rich server. The goal of this partitioning is to reduce the energy consumed by the mobile client in executing the application. In particular, we focus on two important aspects of Java code execution, namely, dynamic compilation and method execution.

To use our techniques discussed in this chapter, the applications need to be modified or rewritten, if they have already been deployed. However, energy optimization should be design goal as opposed to an afterthought. To achieve best energy savings, one needs to consider remote execution at the early stage of software development, instead of modifying the already existing code.

In our implementation, the local/remote decision logic is incorporated in the proxy of each components that are considered suitable for remote execution. The proxy can be either defined by the programmer at development time or automatically created using profile data when the application is deployed on the server for the client to download. When the interface method of the proxy is invoked, the proxy evaluates and compares the local execution and offloading costs for the invocation. If the method is determined to be executed locally, the proxy also determines the optimization level for the methods of the local implementation of the component. Each method of the local implementation whose current optimization level is lower than its desirable level is then compiled to the right level before its first invocation. If the method should be executed remotely, the proxy forwards the name of the method and the parameters to the server. The state of the component is also migrated if necessary. When the remote execution completes, the server sends back the results and the new state of the component (if necessary). During the remote execution, the server may call back the proxy remotely if it needs to access the other components whose state cannot be migrated together with the current component. Our prototype has been implemented using two SPARC workstations, one acting as the server and the other simulates the mobile client. We execute the mobile client using Shade simulation framework to calculate the energy consumption.

In this chapter, seven different execution strategies are considered. In the first strategy, denoted as Remote (R), all components are executed remotely at the server. In the second execution strategy, called Interpreter (I), all methods are executed locally (i.e., on the client) in the
bytecode form. Note that this strategy incurs no compilation or communication energy. In the next three execution strategies (called Local 1 (L1), Local 2 (L2), and Local 3 (L3)), the methods are compiled with different degrees of optimizations and executed on the client (in the form of native code). Local 1 performs no special optimizations in compiling the code, and just translates the bytecode to native form before the first execution. Local 2 performs some well-known optimizations during the compilation; these include common sub-expression elimination, loop invariant code motion, strength reduction, and redundancy elimination. Local 3 performs virtual method inlining [34, 131] in addition to the optimizations performed by Local 2. These five strategies are all static as they fix the execution strategy for each method in a given application. Besides these static strategies, we evaluate two adaptive strategies: Adaptive Execution/Local Compilation (AL) and Adaptive Execution/Adaptive Compilation (AA). The AL strategy determines, for each components that can be offloaded, the best execution strategy (Remote, Local 1, Local 2, Local 3, or Interpreter) dynamically just before its execution. In addition to local/remote execution modes, AA tries to further optimize the client’s energy consumption by exploiting the tradeoffs between local/remote compilation. All adaptive strategy results reported in this chapter include the overhead for the dynamic decision making (the energy consumed by the proxy).

Table 5.3 gives a summary of the static and dynamic (adaptive) strategies evaluated in this work.

5.3.1 Analysis of Static Strategies

In this subsection, we investigate the tradeoffs between the different static strategies. For the experiments in this subsection, for the native code strategies (Local 1, Local 2, and Local 3), we perform compilation on the mobile client. The energy numbers presented in this subsection include the energy cost of loading and initializing the compiler classes. Fig 5.3 on page 83 shows the energy consumption of the static strategies (R, I, L1, L2, and L3) for all our benchmarks. All energy values are normalized with respect to that of L1. For the bar denoting remote execution (R), the additional energies required when channel condition is poor is shown using stacked bars over the Class 4 operation (the best channel condition). For each benchmark, we selected two different values for the size parameters (see Table 5.2). It can be observed that the optimal static strategy varies depending on the input parameter size, and current channel condition. As an example, for a small image size (64x64), remote execution (R) is the preferable strategy for hpf
Table 5.3. Summary of the static and dynamic (adaptive) strategies. The third column indicates where and how the compilation is performed; the fourth column shows where and in what form the execution takes place; and the last two columns give the contents of the communication involved (if any).

when the channel condition is Class 4, 3, or 2. But, when the channel condition degrades to Class 1, the interpreter strategy (I) becomes the best choice. On the other hand, when the image size is increased to 512x512, the best strategy becomes L2. Similar differences in optimal strategies can be observed for the other benchmarks as well. In fact, these results motivate the need for dynamically determining the execution strategy. In particular, these experimental results show that, depending on the input parameters and channel conditions, different execution strategies might be preferable for the same Java method.

5.3.2 Analysis of the Adaptive Execution Strategy

In this subsection, we present an adaptive approach that chooses the most appropriate execution strategy for each method each time it is invoked. Specifically, when the client invokes an interface method of a component that can be offloaded, the proxy intercepts the invocation. Based on the input parameters and current wireless channel condition, the proxy estimates the offloading and the optimal local execution costs to determine whether to execute the method
Fig. 5.3. Energy consumption of eight benchmarks with static execution strategies. The energies are normalized with respect to L1. For each benchmark, left five bars: small input size, right five bars: large input size. The stacked bars labeled “R” indicate the remote execution energies under Class 4, Class 3, Class 2, and Class 1 channel conditions.
locally or remotely. If the method is to be executed locally, the client also needs to select a desir-
able optimization. Since compilation, if necessary, is always performed locally in this strategy, we call it *Adaptive Execution/Local Compilation (AL)*.

In order to make the remote/local execution decision, the client needs to estimate the remote execution energy, local compilation energy, and local execution energy. Since, given a specific platform, a method and an optimization level, the compilation cost is constant, in our prototype implementation, the local compilation energy values (for each method of each off-loadable component and each optimization level) are obtained by profiling; these values are then incorporated into the proxy’s class file as static final variables. To make this strategy platform independent, we specify a scaling factor for each platform. These variables are then referred to when the proxy is making decision.

We employ a curve fitting based technique to estimate the energy cost of executing Java methods locally. As an example, Fig 5.4 shows the curves fitted for estimating the execution energy of two interface methods using Local1. It should be noted that this cost includes the energy consumed not only by the interface methods themselves but also by the internal (private) methods of the components that are called by the interface methods. To verify the accuracy of these curves, the points computed from these curves were compared with 20 other data points (for each application) from actual simulations. We found that our curve fitting based energy estimation is within 2% of the simulation result. This input parameter based energy estimation is observed to work well in all the methods that consume a significant part of execution time and energy in our benchmark suite. Our approach, however, has a limitation in that it may not work well for methods whose execution costs cannot be estimated from their parameters. To estimate the remote execution energy, the client uses the channel condition, the sizes of the input objects (which are known at the time of method invocation), and the estimated sizes of the output objects. The formulation obtained from curve fitting is then encoded in the proxy. Since the cost estimation is performed at runtime and introduces overhead, the calculation performed for the estimation should not be too complex. For many methods, the energy cost can be predicted based on their parameters pretty accurately with simple calculations. In our experiments, the energy consumed by the proxy is less than 3% of the energy consumed by the actual implementation of the component.
Fig. 5.4. Curve fitting for execution energy estimation using Local1 (obtained using Curve-Expert 1.3[8]). The y-axis represents the execution energy and x-axis is the input size. Left: Path-Finder (the corresponding fitted equation is \( y = 0.578 - 0.172x + 0.044x^2 - 0.0009x^3 + 6e-5x^4 \)). Right: Function-Evaluator (the corresponding fitted curve is \( y = 0.0043x^{1.098} \)).

Once the local compilation and execution (Interpreter, Local1, Local2, and Local3), and remote execution energies are estimated, AL uses the following strategy to decide how and where to execute the method. Let us first define the notation required to explain our strategy:

\[
\begin{align*}
    e(m, s) & : \text{Estimated energy for bytecode interpretation} \\
    E_o(m, s) & : \text{Estimated energy for local execution} \\
    E'_o(m) & : \text{Estimated energy for local compilation} \\
    E''(m, s, p) & : \text{Estimated energy for remote execution}
\end{align*}
\]

The subscript \( o \) in this notation refers to the optimization level considered; \( m \) denotes the method in question; \( p \) is the communication power required by the current channel condition; and \( s \) is the parameter(s) of the method that determines the computation complexity of local execution and the sizes of sent and received data for remote execution (in the rest of this chapter, we refer to this parameter as the “size parameter”). In the following discussion, without loss of generality, we assume three optimization levels: \( o_1 \) (most primitive, corresponds to Local1), \( o_2 \),
and \( o_3 \) (most aggressive, corresponds to Local3). Suppose that a method has been executed \( k \) times using the current level of optimization. AL optimistically assumes that this method will be executed \( k \) more times in the remaining portion of the execution (a similar strategy has also been employed in [31]). We predict the future parameter size and communication power based on the weighted average of current and past values. Specifically, at the \( k^{th} \) invocation, we use the following formulations to estimate the future parameter size and communication power (\( s_k \) and \( p_k \) are current parameter size and communication power, respectively):

\[
\overline{s}_k = u_1 \overline{s}_{k-1} + (1 - u_1) s_k;
\]

\[
\overline{p}_k = u_2 \overline{p}_{k-1} + (1 - u_2) p_k;
\]

\( 0 \leq u_1, u_2 \leq 1 \).

In this formulation, \( \overline{s}_k \) is the expected future parameter size and \( \overline{p}_k \) is the expected future communication power after the \( k^{th} \) method invocation. \( u_1 \) and \( u_2 \) are used to appropriately weight the current and past-history values. According to our experiments, setting both \( u_1 \) and \( u_2 \) to 0.7 yields satisfactory results. Before each method invocation, a decision is made as to whether it is beneficial from the energy perspective to employ a more aggressive optimization. For example, after \( k \) times of bytecode executions, our JVM checks which of the following energy values is minimum:

\[
E_I = ke(m, \overline{s}_k)
\]

\[
E_R = kE''(m, \overline{s}_k, \overline{p}_k)
\]

\[
E_{L1} = E'_{o_1}(m) + kE_{o_1}(m, \overline{s}_k)
\]

\[
E_{L2} = E'_{o_2}(m) + kE_{o_2}(m, \overline{s}_k)
\]

\[
E_{L3} = E'_{o_3}(m) + kE_{o_3}(m, \overline{s}_k)
\]

Here, \( E_I, E_R, E_{L1}, E_{L2} \) and \( E_{L3} \) are the expected energies that will be consumed if all the remaining invocations of this method (i.e., an estimated total of \( k \) invocations) are executed using Interpreter, Remote, Local1, Local2 and Local3 modes, respectively. The alternative that gives the minimum energy is chosen as the preferred mode of execution. If either the bytecode or remote execution is preferred, no compilation is performed; otherwise, the compilation is
Fig. 5.5. Average of normalized energy consumptions. Left eight bars: channel condition is predominantly good and one input size dominates. Middle eight bars: channel condition is predominantly poor and one input size dominates. Right eight bars: both channel condition and size parameters are uniformly distributed. All values are normalized with respect to L1.

performed (locally) before execution. If a particular compiled form is already available from previous compilation, the corresponding $E'_o(m)$ term is omitted when evaluating the alternatives. As discussed earlier, if remote execution is favored, the input object is serialized and transmitted to the server. The client is then powered down for a specified interval based on the estimated execution time on the server. It then wakes up to receive the result of the computation from the server. The server also serializes and sends the output object to the client. However, in a mobile environment, there could be loss of connection to the server for prolonged time durations. When the result is not obtained within a predefined time threshold, the connectivity to server is considered lost, and a local execution begins.

In order to evaluate the adaptive execution strategy, we compare its energy behavior with the five static strategies. Each benchmark is executed by choosing three different situations having different channel condition and input distribution. The distributions have been carefully selected to mimic these three situations: (i) the channel condition is predominantly good and one input size dominates; (ii) the channel condition is predominantly poor and one input size dominates; and (iii) both channel condition and size parameters are uniformly distributed. Executing
<table>
<thead>
<tr>
<th>Scenario</th>
<th>App</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fe</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>85</td>
<td>5</td>
<td>10</td>
<td>70</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>fe</td>
<td>85</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>70</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>fe</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>mf</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>85</td>
<td>5</td>
<td>10</td>
<td>70</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>mf</td>
<td>85</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>70</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>mf</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>pf</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>85</td>
<td>5</td>
<td>10</td>
<td>70</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>pf</td>
<td>85</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>70</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>pf</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>ed</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>85</td>
<td>5</td>
<td>60</td>
<td>30</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>ed</td>
<td>85</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>60</td>
<td>30</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>hpf</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>85</td>
<td>5</td>
<td>60</td>
<td>30</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>hpf</td>
<td>85</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>60</td>
<td>30</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>sort</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>85</td>
<td>5</td>
<td>10</td>
<td>70</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>17</td>
<td>sort</td>
<td>85</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>70</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>18</td>
<td>sort</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>19</td>
<td>jess</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>85</td>
<td>10</td>
<td>80</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>jess</td>
<td>85</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>10</td>
<td>80</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>jess</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>db</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>85</td>
<td>5</td>
<td>10</td>
<td>70</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>23</td>
<td>db</td>
<td>85</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>70</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>24</td>
<td>db</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 5.4. Distribution of channel conditions and size parameter values in 24 different scenarios. C1 (Class 1) through C4 (Class 4) denote the wireless channel conditions, whereas S1 through S5 represent the values of size parameters (S5 indicates the largest computational complexity if the method is executed locally, and usually, the largest communication data size if the method is executed remotely).
Fig. 5.6. Energy consumptions of different execution/compilation strategies (Scenario 1 ∼ 24), normalized with respect to that of L1.
each of the eight applications under these three situations contributes to 24 scenarios. Each scenario characterizes an application, different channel condition, and input distributions as shown in Table 5.4. The third through sixth columns in Table 5.4 give the distribution that controls the generation of channel conditions. For example, a value such as 10% under column C3 indicates that in this scenario the possibility that channel condition C3 will occur is 10%. The seventh through eleventh columns in the figure give the distribution that controls the generation of size parameters. For example, a value such as 10% under column S4 indicates that in this scenario the possibility that the size parameter happens to be S4 is 10%. For each scenario, an application is executed 300 times with inputs and channel conditions selected to meet the required distribution.

Fig 5.5 shows the energy consumption of different execution strategies, normalized with respect to L1. Note that these values are averaged over all eight benchmarks. The individual benchmark results are provided in Fig. 5.6. We observe that under all the three situations (i, ii, iii), the adaptive strategy AL consumes less energy than the static strategies. Compared to static strategies, AL introduces overheads, but, since the calculation performed to evaluate the computation costs and make decisions is simple, these overheads are too small (less than 1%) to highlight in the graph. We observe from Fig 5.5 that AL outperforms all static strategies in all three situations. Specifically, it consumes 25%, 10%, and 22% less overall energy than the best static strategy (L2) in situations i, ii and iii, respectively. These results emphasize the importance of dynamic adaptive execution.

Our adaptive approach also has an influence on the performance of the code. This performance impact is dependent on the relative performance of the mobile client and the server as well as the bandwidth limitations of the wireless channel. When using a 750MHz SPARC server and a 2.3Mbps wireless channel, we find that performance improvements (over local client execution) vary between 2.5 times speedup and 10 times speedup based on input sizes whenever remote execution is preferred. However, it must be observed that remote execution could be detrimental to performance if the communication time dominates the computation time for the offloaded components. Such cases also tend to be detrimental for energy consumption.
5.3.3 Analysis of Adaptive Compilation Strategy

So far, we have only considered local compilation. By encapsulating the compiler as a component and using a proxy, we can also offload the compilation to the server. Specifically, the compiler proxy intercepts the compilation requests, and forwards to the server the optimization level and the name of the method. Since the application is initially downloaded from the server, we assume that the server has all the class files of the application, and that the client does not need to send the bytecode. The server compiles the method, and sends back the compiled code. The compiled method is typically larger (5-8 times) than the bytecodes of the method and incurs a larger communication cost than transmitting the methods as bytecodes. We do account for this larger code size of compiled methods into account (however, note that, the size of selectively compiled codes does not significantly impact the overall memory requirements of the JVM).

For the environments where security is critical, the signature of the server is also sent together with the compiled code. The proxy authenticates the signature (if necessary), and then invokes the linker (which is shared by both local and remote compiler) to link the compiled code. It should be noted that, although the native code generated by the compiler is platform-dependent, the compiler itself can be written in Java (e.g., the compiler used in Jikes RVM [31]), and can thus run on any Java platform. In fact, to boost their products, mobile phone manufacturers may provide such compilers. In this scenario, the remote compilation server will be able to extend the types of clients it supports by downloading the compilers from the mobile phone manufacturers’ web-site.

Table 5.5 provides the (compilation) energy consumed when a client either compiles methods of an application locally or offloads the compilation tasks to the server. We again consider three levels of compiler optimizations. For the remote compilation, the channel condition influences the energy cost as can be seen in the last four columns. We also observe that as the degree of optimization increases, the energy expended in local compilation increases. However, for remote compilation, there are cases where a more aggressive optimization reduces the code size and consequently results in less communication energy than that of a less aggressive optimization. For example, in sort, in going from Level2 to Level3 optimization, the remote compilation energy reduces. Table 5.5 also shows that in many cases, remote compilation consumes less energy than local compilation with the same optimization level (e.g., db). For applications
where a significant portion of the energy is consumed in compiling the bytecodes to the native code (due to short overall execution time, due to small number of method invocations, or due to frequent recompilations which is required to adapt to changing external conditions and/or values of runtime variables [31]), it may be very beneficial to exploit the energy tradeoffs between local and remote compilation. Note that the sizes of compiled versions of methods can differ depending on the optimization level. Optimizations such as method inlining, object inlining, and code duplication [34] can increase the size of the compiled code, whereas providing performance gains at execution time. Such optimizations, in general, provide opportunities for tradeoffs between the size of the compiled code and the performance gains may be critical in a mobile environment.

By considering the remote compilation, we now enhance our adaptive strategy presented earlier. This enhanced strategy is referred to as AA (Adaptive execution/Adaptive compilation). In this strategy, when a decision is made to compile the method, the client computes the energy needed to offload the compilation task, and compares it with the cost of generating this version locally from the bytecodes. The channel condition is used to estimate the energy cost for transmitting the method name and receiving the compiled code.

We observe from Fig 5.5 that AA saves more energy than AL. The extra savings of AA come from two sources. First, AA reduces compilation cost (energy) by selecting the best compilation alternative from the energy perspective. Second, since the cost of receiving a more optimized version can be less than the cost of local compilation to a less aggressive version, AA can also reduce execution energy. For example, in scenario 13, we observed that the highest optimization used in AL is Local2, whereas in AA it is Local3.

5.3.4 Using Reconfigurable Datapath to Optimize Local Execution

An additional level of optimization can be considered for optimizing the energy consumed when executing the Java methods on the client. Specifically, here, we consider mapping compute-intensive methods onto a dynamically reconfigurable datapaths. Such dynamically reconfigurable datapaths are being employed increasingly in embedded processors [20]. Instead of
Table 5.5. Local and remote compilation energies. For each application, all values are normalized with respect to the energy consumed when local compilation with optimization Level1 is employed.

dynamically compiling the Java method to hardware (as in [36]), we use a pre-compiled configuration bit-stream that configures the reconfigurable datapath associated with that Java method. This bit-stream is downloaded as an annotation in the class-file.

Executing frequently used Java methods in a reconfigurable datapath can improve performance and reduce energy consumption in method execution, but reconfiguration also introduces both performance and energy overheads. Recently, Huang and Malik [69] have proposed the use of a co-processor with a dynamically reconfigurable datapath (see Fig 5.7) for performance enhancement in embedded systems. In this subsection, we extend our design space by making use of such a reconfigurable co-processor for energy savings.

We experiment with the reconfigurable datapath using two applications: `fe` and `mf`. We chose the method `sin()` in `fe` and method `median()` used in `mf` (which finds the median

<table>
<thead>
<tr>
<th>App</th>
<th>Opt Level</th>
<th>Local Compilation</th>
<th>Remote Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>C1</td>
</tr>
<tr>
<td>fe</td>
<td>L1</td>
<td>100.0</td>
<td>286.3</td>
</tr>
<tr>
<td>fe</td>
<td>L2</td>
<td>172.1</td>
<td>286.3</td>
</tr>
<tr>
<td>fe</td>
<td>L3</td>
<td>363.3</td>
<td>317.5</td>
</tr>
<tr>
<td>ed</td>
<td>L1</td>
<td>100.0</td>
<td>80.3</td>
</tr>
<tr>
<td>ed</td>
<td>L2</td>
<td>209.2</td>
<td>77.1</td>
</tr>
<tr>
<td>ed</td>
<td>L3</td>
<td>230.4</td>
<td>77.1</td>
</tr>
<tr>
<td>pf</td>
<td>L1</td>
<td>100.0</td>
<td>124.8</td>
</tr>
<tr>
<td>pf</td>
<td>L2</td>
<td>163.0</td>
<td>124.8</td>
</tr>
<tr>
<td>pf</td>
<td>L3</td>
<td>174.9</td>
<td>124.8</td>
</tr>
<tr>
<td>hpf</td>
<td>L1</td>
<td>100.0</td>
<td>80.1</td>
</tr>
<tr>
<td>hpf</td>
<td>L2</td>
<td>309.9</td>
<td>47.5</td>
</tr>
<tr>
<td>hpf</td>
<td>L3</td>
<td>320.8</td>
<td>47.5</td>
</tr>
<tr>
<td>mf</td>
<td>L1</td>
<td>100.0</td>
<td>92.3</td>
</tr>
<tr>
<td>mf</td>
<td>L2</td>
<td>187.5</td>
<td>92.3</td>
</tr>
<tr>
<td>mf</td>
<td>L3</td>
<td>238.0</td>
<td>96.9</td>
</tr>
<tr>
<td>sort</td>
<td>L1</td>
<td>100.0</td>
<td>244.0</td>
</tr>
<tr>
<td>sort</td>
<td>L2</td>
<td>185.6</td>
<td>244.0</td>
</tr>
<tr>
<td>sort</td>
<td>L3</td>
<td>205.2</td>
<td>217.4</td>
</tr>
<tr>
<td>db</td>
<td>L1</td>
<td>100.0</td>
<td>120.5</td>
</tr>
<tr>
<td>db</td>
<td>L2</td>
<td>144.1</td>
<td>113.1</td>
</tr>
<tr>
<td>db</td>
<td>L3</td>
<td>213.8</td>
<td>120.5</td>
</tr>
</tbody>
</table>
value from the filter window) to implement on the reconfigurable datapath. The choice of the execution strategy for the remainder of the code is orthogonal to the reconfigurable datapath implementation. Our target configurable datapath contains 6 functional units, each containing an adder, an integer multiplier, a shifter and a logic unit. The functional units are connected using a crossbar and have in-built registers for supporting pipelining. The functionality and connectivity are controlled by the configuration bits. For the method to be executed in the reconfigurable datapath, we first build the data-flow graph and then map each node to a functional unit. Note that this mapping involves both scheduling and resource binding. We implement them using the techniques proposed in [97]. While currently we perform scheduling and binding by hand, it is possible to automate them within the JVM. Thus, the energy cost of executing a method in the reconfigurable datapath is estimated from the number and types of operations performed and from the actual interconnections. We obtained the energy numbers for functional units from SimplePower and for the crossbar from [56]. Note that energy savings are obtained when executing methods on this reconfigurable datapath as it avoids many instruction fetches, instruction decoding and register accesses. However, there is a small reconfiguration energy for setting the 5Kbit configuration bit-stream. This reconfiguration energy/time is much smaller to map a method onto the reconfigurable co-processor as compared to a traditional FPGA configuration.

Fig 5.8 shows the impact of the reconfigurable datapath on both static (L3) and dynamic (AL and AA) strategies. The distribution of channel condition and input sizes is the same as specified in Table 5.4. We observe that the use of the reconfigurable datapath reduces the energy consumption of both local and adaptive strategies. Specifically, for the static strategy L3, the reconfigurable datapath reduces the overall energy consumption by around 50% in fe and 70% in mf. However, for dynamic strategies, the impact is relatively smaller since a significant portion of energy is still consumed in remote execution. We also note that the use of the reconfigurable datapath shifts the optimal energy case for some of the methods from remote execution to local execution. Specifically, when we move from AL to AL+ in mf, the percentages of methods executed remotely drop from 90% to 2%, 2% to 0% and 42% to 1% in situations i, ii, and iii (refer to Section 5.3.2), respectively. There are several interesting tradeoffs in the use of reconfigurable datapaths such as memory bandwidth limitation and mapping optimization that are planned as a part of our future work.
5.3.5 Using Data Compression to Optimize of Remote Execution

While the previous section focused on optimizing the local computation energy, there are also opportunities for reducing the cost of communication energy incurred during remote execution. In this section, we consider one such optimization for reducing communication energy. Specifically, to reduce the overall communication energy, we compress the data communicated between the client and the remote server. However, this also incurs some additional energy (due to compression and decompression).

To implement data compression, we introduced a compression tier on both the client and server sides. Data is compressed into GZIP format (using java.util.zip package) before being sent over the wireless network. The received data is decompressed before being passed to the applications. The compression and decompression can be overlapped with sending and receiving operations, respectively; thus, they are not expected to incur much performance penalty. In fact, we can even expect some performance improvements due to fewer data packets to be sent/received (as a result of compression). However, the energy cost of compression and decompression should be accounted for. Fig 5.9 shows the tradeoffs between compression/decompression and communication energy costs. For the scheme without compression (the left bar for each benchmark), the overhead is the energy consumed in running the Java serialization and communication protocol. For the scheme with compression (the right bar for each benchmark), the overhead also
Fig. 5.8. Impact of reconfigurable datapath on energy consumptions of different execution/compilation strategies. L3+ is the static strategy Local3 enhanced by the reconfigurable datapath (L3+ is normalized with respect to L3). AL+ and AA+ are the adaptive strategies AL and AA, respectively, enhanced by the reconfigurable datapath. AL+, AA and AA+ are normalized with respect to AL. Left: $\varepsilon \in$ (from left to right: situations i, ii, and iii). Right: $m \notin$ (from left to right: situations i, ii, and iii).

includes the compression and decompression energies. For either small or large input sizes, when the channel condition is very good (e.g., always Class 4), compression does not necessarily save energy. (Note that introduction of an energy-efficient hardware decompression circuit may change these observations). But, when the channel condition is bad (e.g., always Class 1), the compression overhead usually pays off. The benchmarks $\varepsilon \in$ and $\notin$ are exceptions due to the small volumes of sent and received data even without compression. Whether to compress or not can be dynamically determined at runtime, and this may affect the selection of execution modes. The details are omitted due to lack of space.

5.4 Related Work

There have been prior attempts to exploit the interaction between mobile clients and resource-rich servers for energy savings [91, 82, 112, 101]. Rudenko et al. [112] performed a series of remote process executions to show the effectiveness of remote executions. They measured the impact of the input size on the energy saved using remote execution. In contrast to our approach that makes execution choices at a component granularity, their approach executes the entire application remotely. Othman and Hailes [101] performed simulations to show
that battery life can be extended by up to 21% using migration at the process level. Kremer et al. [82] proposed a framework that analyzes the entire program in order to identify candidate code fragments for remote or local execution. Li et al. [91] developed an elegant program partitioning scheme that uses a task mapping algorithm to statically classify tasks as server and client tasks. This scheme uses profiling information on computation time and data sharing at the procedure level to optimize energy consumption. In a more recent work [90], they used a maximum-flow/minimum-cut algorithm to optimize the partition/allocation problem as opposed to the branch-and-bound policy used in [91]. In our approach, we focus on a Java-based environment, and dynamically decide whether to execute locally or remotely based on input sizes, computational complexity, and channel conditions. Further, in our approach, an additional challenge is to determine the form of local execution that would provide the most energy-efficient solution.

In order to address the tradeoffs between the slow speed of interpreted bytecode execution and the memory/performance overheads of dynamic compilation, Turbo and Quicksilver [57, 114] use pre-compiled binaries. More specifically, they pre-compile bytecode into native code, and place the generated code in the device’s ROM image. This is a good approach for
applications shipped with the device, but is problematic for applications shipped independent of the device [57].

Remote compilation is employed in [11] and [102] to avoid the memory overheads of a JIT compiler. Whenever JCod determines that a method should be compiled, it sends it to a compilation server on the local network. The compilation server replies by sending the native code back to JCod, which installs it within the VM. From that time on, the native code is used, resulting in a gain in speed only for the part of the application for which it is worthwhile. In contrast to JCod, our optimization is performed to conserve energy. Palm et al. [102] shows that offloading JIT compilation to a resource rich server saves energy for all SpecJVM98 benchmarks. In comparison, our technique attempts to dynamically select between local and remote compilation based on the method and channel conditions. Further, we augment energy savings through remote execution modes. Teodorescu and Pandey [120] proposed a Java-based ubiquitous computation environment that makes use of remote JIT compilation. Their work tries to reduce the memory requirement on the client side and does not focus on energy tradeoffs.

Besides offloading execution/compilation, there exist many energy optimization techniques for mobile devices. Cignetti et al. [46] quantify the power savings of different strategies such as CPU sleep modes, LCD shutdown in a PDA. Lebeck et al. [85] propose energy aware page allocation strategies that can achieve over 80% Energy · Delay improvements over a traditional full-power memory system with random page replacement. Chen et al. [43] propose garbage collector controlled memory bank turnoff strategies to conserve energy in Java enabled mobile environments. Zeng et al. [133] propose the Currency Model that unifies energy accounting over diverse hardware components and enables fair allocation of available energy among applications. Their particular goal is to extend battery lifetime by limiting the average discharge rate and to share this limited resource among competing tasks according to user preferences. Our execution/compilation offloading scheme is agnostic to these energy saving techniques and can be combined with them to further reduce energy consumption. For example, a method may allocate many objects during its execution; the compiler may create many intermediate data structures when compiling a method. By offloading the execution or compilation, most of these objects and intermediate data structures do not need to be allocated in the memory of the client, which create more opportunities to put memory banks in sleep mode.
5.5 Concluding Remarks

An important issue when executing a Java application on mobile devices is its energy consumption. Our work emphasizes that the choice of compilation/execution strategy for the machine-neutral Java bytecodes critically impacts the energy consumed by the device.

In particular, the conclusions from this work can be summarized as follows. First, we observe that interpreted Java execution is generally more costly in terms of energy as compared to execution using compiled code. However, the compilation itself involves an energy cost and requires additional memory footprint for storing the compiled code. Hence, one can employ remote compilation which can reduce both the energy and memory overheads. However, remote compilation also incurs the energy cost of receiving the compiled code from the server. Thus, if energy is the constraint of focus, we can dynamically decide between compiling locally and remotely. Mobile systems with larger memories are beginning to emerge that make such trade-offs for supporting local or dynamic compilation useful. Another technique for energy saving that we present is remote execution of methods. We dynamically evaluate the trade-offs between computational and communication cost based on input parameters and channel conditions in deciding between local and remote executions. The results from the comparison of our dynamic approach with static execution approaches justify the need for adaptive strategies to obtain the best energy behavior.
Part II

Memory Footprint Reduction
Chapter 6

Heap Compression for Reducing Memory Footprint

6.1 Introduction

Many mobile devices have stringent memory requirements as memory has severe implications on the cost, form factor, and energy consumption of the device. The budget for the memory has a significant impact on the overall cost of a device. The goal of this work is to enable the execution of Java applications using a smaller heap than that possible using current embedded JVMs. The memory requirement of a Java application is mainly shaped by the heap space required for executing the application. Existing techniques to reduce the heap space requirement include reducing the number of classes to be loaded (when classes are managed in the heap), using memory efficient data structures [122], early retirement of objects through last use analysis [117], and tuning the garbage collector (e.g., using Mark-Compact collector instead of Mark-Sweep collector) [124]. In this work, we explore the use of compression and lazy allocation combined with object partitioning as means to reduce the heap space required for executing a Java application. Reducing heap footprint can minimize the amount of active memory maintained throughout the execution and can enable the reuse of this space by another concurrent application or enable energy savings by powering down the unused memory portions. Furthermore, heap footprint reduction can also result in a smaller maximum heap size required to execute the application successfully. As a consequence, it may be possible to increase the number of applications that execute without out-of-memory exception for a given heap size.

Compression has been a potent technique for reducing memory requirements in different contexts [60, 47, 62, 87, 129]. In this chapter, we present a technique that compresses objects in the heap when the current execution cannot complete normally within the given heap size. Specifically, we tune the garbage collector (GC) already present in the embedded JVM to support compression. Normally, the GC is invoked to reclaim the space occupied by garbage, i.e., the objects that are no longer needed by the application. Mark-Sweep (MS) and Mark-Compact
(MC) are two garbage collection algorithms incorporated within Sun’s embedded JVM (called KVM [84]), which we use in this work as a reference point. The MS collector has two phases [74, 127]: mark phase and sweep phase. During the mark phase, the collector traverses the reference tree and marks all the objects that are reachable from the roots. In the following sweep phase, the collector scans the whole heap and puts all unmarked objects into a free-table. Since this collector does not move objects after the collection, live objects and free blocks are interleaved with each other. After several allocations and collections, the heap may become so fragmented that each free block is too small to serve any object allocation request. The total size of the free blocks, however, can still be larger than the requested size. This is called the “fragmentation problem” [74, 127], due to which an application running with the MS collector usually needs larger heap space than its footprint.1

The MC collector addresses the fragmentation problem by supporting compaction. It also has two phases [74, 127]: mark phase and compact phase. The mark phase is the same as that of the MS collector. During the compact phase, the MC collector slides all the marked objects to one end of the heap (this operation is called the “compaction”). The free blocks, on the other hand, slide to the other end of the heap and are combined into one large free area. Since the MC collector moves objects, it needs to update each reference to an object that has been moved. The MC collector allows an application to run properly with the heap space no smaller than its footprint, which is determined by the behavior of the application.

In this chapter, we propose a set of memory management strategies to reduce heap footprint of embedded Java applications that execute under severe memory constraints. Our first contribution is a new garbage collector, referred to as the Mark-Compact-Compress (MCC) collector, that allows an application to run with a heap smaller than its footprint. The proposed collector works in two phases: mark phase and compact-compress phase. In the mark phase, the collector not only marks the live objects, but also counts their sizes. Based on the total size of the live objects, it calculates the total size of the free space. If this size is larger than the object to be allocated, the MCC collector compacts the heap like a normal MC collector. On the other hand, if the size of the free space is smaller than that of the object to be allocated, the collector compresses all the live objects to increase available heap space. This introduces the overhead

1In this chapter, we define the “footprint” of an application as the maximum total size of the live objects that are in the heap simultaneously.
of compression and subsequent decompression when accessing a compressed object. However, since the peak heap demand occurs only during a very short period of execution, the compression overhead is not incurred frequently. Further, due to the locality of object accesses, the cost of decompression is amortized over multiple object accesses. Finally, many objects (or parts of them) are not even accessed after they are compressed since they have reached their last use (note that such objects may still be live from the collector’s perspective and cannot be marked as garbage), and consequently, they are collected before any decompression.

In addition to employing compression, we also consider a heap management strategy and associated garbage collector, called MCL (Mark-Compact-Lazy Allocate), based on “lazy allocation” of “object portions”. This new collector operates like the MC collector, but takes advantage of the observation that many Java applications create large objects, of which only a small portion is actually used. For example, a program may allocate a character array as the buffer when reading the user’s input. Since the length of the input data is unknown at the programming time, programmers typically allocate a buffer large enough to hold the longest possible input data that can be expected from the user. However, in most cases, the actual input data may be short and the space in the buffer may be wasted. To reduce the heap memory requirements for such programs, we break down each large array object into a set of smaller subobjects. Each subobject is “lazily allocated” upon its first write access. Therefore, the subobjects that do not contain any element actually used by the program are not heap-allocated at all, thereby saving heap space. It should be noted that, in our implementation, both breaking down large objects into subobjects and lazy allocation of subobjects are internal to JVM and transparent to the Java application being executed. In addition, we also combine MCC and MCL, and present MCCL (Mark-Compact-Compress-Lazy Allocate), which outperforms both MCC and MCL.

We implemented the proposed heap management strategies employing compression and lazy allocation using KVM [84], and compared them to two garbage collectors (MS and MC) currently used in KVM. Our experimental evaluation using a set of ten Java applications suitable for handheld devices and a zero-removal compression technique [110] shows that one of the proposed collectors reduces the peak heap demand by 35% on the average (ranging from 16% to 54%) over the MC collector. The consequent performance degradation due to compression and decompression was observed to be less than 2% on the average, over the MC collector.
using the same heap space. In addition, we show how our results change when we eliminate handle-based object accesses and present a garbage collector (called MCCL+) based on this. The heap management techniques proposed in this chapter are very general and can be applied to other Java virtual machines and garbage collectors and can make use of different compression algorithms where available.

6.2 Implementation Details

In this section, we present the details of our base implementation, which includes support for indirect object references, object compression and decompression, breaking down large objects into subobjects, and lazy allocation. Later in the chapter we also present our enhanced implementation which eliminates object handles.

6.2.1 Indirect Object References

To facilitate compression/decompression of objects, in our base implementation, references to Java objects are implemented using handles (see Fig. 6.1). Specifically, each object has a handle in the handle pool. Each handle has two components: a pointer to instance data and a pointer to class data in the class area. An object reference is actually a native pointer to a handle in the handle pool. An advantage of this scheme is that, when an object is moved, it eliminates the necessity of updating every reference to this object, which may be scattered in different locations in the runtime data area. The main drawback is that each access to an object’s instance data requires dereferencing two pointers.

In our implementation, allocation of a Java object involves two steps: (1) allocate a handle in the handle pool; (2) allocate the heap space for the data of the object. When the handle pool is used up, the GC is invoked. The handles of the objects that have been collected are returned to the handle pool. If the free space in the handle pool is smaller than a given threshold \( T \) bytes after the collection, the handle pool is expanded by \( C \) bytes. The expansion of the handle pool is to avoid frequent garbage collections due to small handle pool size. The threshold, on the other hand, is set to prevent the handle pool from growing too fast. Based on our experience, \( C \) and \( T \) are set to 1/64 and 1/32 of the heap size, respectively. Since each
Fig. 6.1. Referencing an object through a handle. In this implementation, the allocation of an object involves two steps: (1) allocate a handle in the handle pool; (2) allocate the heap space for the data of the object. In the MC collector, the handle pool is a part of the permanent space (which is not garbage collected).

handle has 8 bytes, the total size of the handle pool is bounded by $8M + T + C$, where $M$ is the maximum number of live objects in the heap.

Since the handles cannot be moved, the handle pool is considered as part of the permanent space. The permanent space is the memory space that contains the data whose lifetimes last until the application terminates. The permanent space is never garbage collected in this study. This space expands in 2KB chunks (the default value in KVM) when it is used up. Expansion of the permanent space involves garbage collection and, if necessary, compacting (or compressing) live objects to one end of the heap, as shown in Fig. 6.2. It should be noted that, in the implementation of the MS collector, there exists no separate permanent space since this collector does not compact the heap. All the permanent data are allocated in the heap and scanned by the collector, although they are never collected.

6.2.2 Compression

MCC collector compresses objects when compaction cannot provide enough space for the new object. In principle, our approach can work with any compression/decompression algorithm. However, for the best results, a compression/decompression algorithm should satisfy
Fig. 6.2. Expansion of the permanent space. (a) Before garbage collection. (b) After garbage collection — the live objects are compacted into one end of the heap, and the free blocks slide are combined into one large free area at the other end of the heap. (c) Expanding the permanent heap space.

three requirements: (1) the compressor should have a good compression ratio; (2) both compression and decompression should be fast; and (3) neither the compressor nor the decompressor should use a large working area. In this chapter, we used a “zero removal” compression scheme, which is based on the observation that a large portion of memory locations manipulated by an application contains only zeroes [110]. The uncompressed and compressed object formats are shown in Figures 6.3(a) and 6.3(b), respectively. In Fig. 6.3(b), the first eight bytes of each object (i.e., the object header) are not compressed and the compressed object contains a bitmap and an array of non-zero bytes. Each bit in the bitmap corresponds to a byte of the object’s data in the uncompressed format. A 0-bit indicates that the corresponding byte is all zero and this byte is not stored in the compressed format. A 1-bit, on the other hand, indicates a non-zero byte and that this byte is kept in the array of non-zero bytes in the compressed format. Bits 16 through 23 of the first word in the header are not used in the uncompressed format. In the compressed format, however, they contain the first eight bits of the bitmap. For an object whose size is larger than eight bytes, the extra bits of the bitmap are stored right after the object’s “original size” field. Following the bitmap is the array of non-zero bytes of the object.

When our collector decides to compress objects, it scans the whole heap and compresses each object that has not been compressed so far. In our current implementation, only the object instances and arrays that are created by the Java application are compressed. The data structures

---

2“Compression ratio” is the ratio between the size of the original data and the size of the compressed data.
Fig. 6.3. (a) Format of an uncompressed object. (b) Format of a compressed object. In both formats, the first 8 bytes (headers) are used by garbage collector for management purpose. Each bit in the bitmap in (b) corresponds to a byte of the object’s data in the uncompressed format.

Fig. 6.4. Compressing the heap. source: the next object to be compressed (or to be compacted, for the internal data structures). target: the first free location in the heap. (a) Initial state. (b) After compressing O1. (c) After compressing O1, O2, and O3.

internal to the implementation of JVM remain uncompressed. In other words, for them, the compression works in the same way as compaction. Fig. 6.4 illustrates the heap compression process. We maintain two pointers: source and target. Source points to the next object to be compressed (or to be compacted, for the internal data structures), and target points to the first free location in the heap.

Fig. 6.5 shows the algorithm for compressing an object. An example application of this algorithm is illustrated in Fig. 6.6. Step 0 shows the state right before an object is to be compressed. Pointer target points to the first free location; pointer source points to the first
byte of the object to be compressed. At step 1, pointer $t$ is initialized to the location where the compressed object will be stored; pointer $s$ is initialized to the first byte to be compressed; and pointer $p$ is initialized to the first byte where the non-zero byte array will be stored. The bitmap size is calculated using the size of the object. At step 2, the first bit of the bitmap is set to 1 as the first byte of the object (which is pointed to by $s$ at step 1) is non-zero. Since $s < p$, the byte pointed to by $s$ is temporarily stored in the buffer. Since this byte is non-zero, $p$ is increased by 1. At step 3, $s$ is now pointing to a zero byte. Therefore, at the next step, the third bit of the bitmap is set to 0 and $p$ is not increased. At step 6, $s$ catches $p$, and from this point on, no more bytes are placed into the buffer. Instead, all the non-zero bytes are copied to the location pointed to by $p$. At the last step, all the bytes in the buffer are copied to their destination locations. In our implementation, we used unmarked objects (whose addresses should be above that of the object being compressed) to hold the buffer. Since the buffer is accessed sequentially, the space allocated for the buffer does not need to be contiguous. Several unmarked objects can be chained up using links in case that the buffer becomes too large to be accommodated in a single unmarked object. It should be noted that, for most of the objects, the distance between the pointers source and target (Fig. 6.4) is so large that even the initial value of $p$ is smaller than $s$. Therefore, the buffer is rarely used during the compression process. A rare case occurs when the compressed size of the object is larger than its original uncompressed size. This does not happen frequently (on average, only 0.3% of the objects in our applications caused such an expansion) and does not impact the overall memory consumption significantly. However, it still needs to be addressed since there may be no space for the object to expand. In our implementation, the compressor checks if the pointer $p$ exceeds the starting address of the next object whenever $p$ is increased. If this happens, the compressor stops compression and uses the data in the buffer and the bitmap to recover the object.

6.2.3 Decompression

If a compressed object needs to be accessed, the decompressor is invoked. Whenever an object is accessed, the virtual machine checks if the object has been compressed. For this purpose, we use the highest order bit of the pointer to the instance data of each handle as a flag. For an uncompressed object, this bit is zero since current JVMs for memory constrained devices
byte heap[];
void compress(int source, int target)
{
    s = source; t = target;
    p = t + \left\lceil \text{length of the object}/8 \right\rceil;
    while(s - source < \text{length of the object}) {
        if(h[s]==0) {
            set the current bit in heap[t] to 0;
        } else {
            set the current bit in heap[t] to 1;
            if(p\leq s)
                heap[p]=heap[s];
            else
                append heap[s] to the buffer;
            p++;
        }
        s++;
        increase t if necessary
    }
    copy all the bytes from the buffer to the locations beginning from t;
}

Fig. 6.5. Zero-removal compression algorithm. This algorithm compresses a single object. To show the core logic of our algorithm, implementation details are omitted.

do not use an address space larger than 2GB. When an object is compressed, we set this bit to 1. To access an object, the virtual machine first loads the pointer to the instance data from the handle into a register (the cost for this instruction is captured in our experimental results). After that, the virtual machine checks if the first bit of the register is zero. If the bit is zero, the object does not need to be decompressed. Otherwise, the object needs to be decompressed first. This checking requires two instructions: one comparison and one branch. Note that the comparison instruction uses the contents of the register and does not involve any extra main memory access, and that the branch instruction is highly predictable (since most objects are not compressed). Therefore, one can expect the overhead associated with checking the compression status of objects to be small.

The decompression process is illustrated in Fig. 6.7 and our decompression algorithm is given in Figure 6.8. The decompressor first allocates a free block that is large enough to hold the object to be accessed in uncompressed format. If we fail to allocate a free block successfully,
Fig. 6.6. Example compression. X: “don’t care” bit. For illustrative purposes, each byte is assumed to have only 2 bits. Pointer target points to the first free location; pointer source points to the first byte of the object to be compressed. At step 1, pointer t is initialized to the location where the compressed object will be stored; pointer s is initialized to the first byte to be compressed; and pointer p is initialized to the first byte where the non-zero byte array will be stored.
Fig. 6.7. Decompressing object O1. (a) Before decompression. (b) After decompression. The decompressor first allocates a free block and then decompresses the object into the block. It also updates the pointer in the handle.

Decompression also happens during the mark phase of garbage collection. In the mark phase, the collector traverses the reference tree. When the collector visits a compressed object, it first checks the object’s class data to see if this object contains any reference fields. If this is the case, the collector decompresses the object to retrieve the contents of the reference fields. Note that the decompression in the mark phase is different from the decompression that happens during the application execution in that the former does not keep the data of the decompressed object. Specifically, when the collector encounters a compressed object that has at least one reference field, the collector scans the object and decompresses it field by field. When each field is decompressed, the collector first checks to see if this field contains a reference and, if so, marks the referenced object. The decompressed field is discarded immediately. Therefore, the decompression in the mark phase does not involve any allocation.

6.2.4 Breaking Down Large Objects

It is not efficient to decompress a whole object when only a few fields of the object are accessed. This is particularly true when the object in question is very large. Decompressing the
byte heap[];
void decompress(int source, int target) {
    b = source; // set b to the first byte of the bitmap
    s = source + \(\lceil\text{length of the object}/8\rceil\); // set s to the first non-zero byte
    t = target; // set t to the first byte of the target location
    while(t - target < \text{length of the object}) {
        if(current bit is 0)
            heap[t++] = 0;
        else
            heap[t++] = heap[s++];
        increase b if necessary
    }
}

Fig. 6.8. Decompression algorithm. This algorithm is invoked at object access time and decompresses a single object. To show the core logic of our algorithm, implementation details are omitted.

whole object not only increases memory requirements, but also slows down the application due to longer decompression time. To address this problem, we propose to break large objects into smaller “subobjects”, as shown in Fig. 6.9. Specifically, an object whose size is larger than a given threshold (1.5KB in this work) is broken down into a set of smaller subobjects (each with a maximum size of 1KB). Each subobject is compressed and decompressed independently. It should be noted that, since Java object instances are not likely to be larger than 1KB, in this work, only Java arrays are considered to be broken down into smaller portions.

In our implementation, upon accessing an element of a Java array, the virtual machine first checks if this array has been broken down. If this is the case, the virtual machine uses the index of the element and the size of the corresponding subobjects to calculate the index of the subobject that contains the element and the intra-subobject offset for the element. If the subobject is in compressed format, JVM also invokes the decompressor to decompress the subobject.

6.2.5 Lazy Allocation

Our observation is that many Java applications do not access different portions of objects uniformly. That is, some fields are accessed much more frequently than the others. As a result,
heap memory can be saved if different portions of a given object are allocated in an on-demand basis. In other words, it may be beneficial if we do not allocate a portion of the object unless that portion is actually accessed. Shaham et al. [117] studied a similar strategy at the whole object level; that is, no heap space is allocated unless the object is used. Our lazy allocation strategy differs from [117] in two important aspects. First, instead of whole objects, we consider different portions of objects; that is, our approach is finer granular. Second, unlike the approach in [117], our approach is entirely transparent to the application execution. In order not to introduce too much runtime overhead, we applied lazy allocation only to the large arrays. In our implementation, when the bytecode “NEWARRAY” is encountered, the allocator checks if the size of this array is larger than a pre-set threshold. If this is not the case, the array creation proceeds as usual. Otherwise (i.e., if this array needs to be broken-down into subarrays), the allocator allocates a main object for this array (see Figure 6.9). The subobjects (subarrays), however, are not allocated at this time. Instead, the pointers to the subobjects are set to null. When an element of such an array is later accessed, the virtual machine checks to see if the pointer to the subobject that contains that array element is null. If the pointer is null and the current access is write, JVM allocates the heap space for the subobject, sets each element in this subobject to the uninitialized value (as defined in JVM specification [93]), and then updates the value of the element that is

---

Fig. 6.9. Breaking down a large array into subarrays. In this way, each subarray (subobject) can be allocated independently.
Table 6.1. Garbage collection strategies. The first two of these strategies (MS and MC) are the default Mark-Sweep and Mark-Compact collectors, whereas the remaining ones are our strategies.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Compact</th>
<th>Compress</th>
<th>Reference</th>
<th>Break Down Large Objects</th>
<th>Lazy Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS Mark-Sweep</td>
<td>No</td>
<td>No</td>
<td>Direct</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MC Mark-Compact</td>
<td>Yes</td>
<td>No</td>
<td>Direct</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MCL Mark-Compact-Lazy Allocate</td>
<td>Yes</td>
<td>No</td>
<td>Direct</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MCC Mark-Compact-Compress</td>
<td>Yes</td>
<td>Yes</td>
<td>Handle</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MCCL Mark-Compact-Compress-Lazy Allocate</td>
<td>Yes</td>
<td>Yes</td>
<td>Handle</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MCCL+ Mark-Compact-Compress-Lazy Allocate</td>
<td>Yes</td>
<td>Yes</td>
<td>Direct</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

being accessed. In other words, each subobject is lazily created upon its first write access. A null pointer indicates that the elements in the subobject have not been initialized by the application since the array has been created. Therefore, for each read access to a null subobject, JVM returns a pre-defined uninitialized value according to the type of the element.

### 6.3 Benchmarks and Experimental Setup

In this chapter, we experimented with six different garbage collection strategies listed in Table 6.1. The first two of these strategies (MS and MC) are the default Mark-Sweep and Mark-Compact collectors, which are currently employed in Sun’s KVM [84, 13], whereas the remaining ones are our strategies. These strategies differ from each other in how they reference an object (direct or handle based), whether they break down large objects into smaller subobjects, or whether they employ lazy allocation. In the next section (Section 6.4), we conduct an experimental evaluation of MCC, MCL, and MCCL. The detailed discussion of MCCL+ will be presented later in Section 6.6. *The important point to emphasize here is that all our compression based collectors use compression only in object allocation and only when it is not possible to continue execution without performing compression (as explained in Section 6.1). In other
words, we evaluate a need based object compression strategy. Later in Section 6.5 we evaluate a more aggressive object compression strategy as well.

To evaluate different garbage collection strategies listed in Table 6.1, we used a set of ten Java applications as benchmarks. These benchmarks represent typical applications running on handheld devices where memory budgets are very limited. Brief descriptions of our benchmarks are given in Table 6.2. As can be seen, our benchmark suite includes utility applications as well as game programs.

Table 6.3 presents memory allocation data for these Java benchmarks. For each benchmark, the second column of this table gives the total numbers of Java objects allocated by the benchmark throughout its execution (including both Java objects and arrays), and the third column gives the average and maximum object sizes. The fourth column shows the maximum size of the objects that are live in the heap simultaneously, and the next column shows average heap occupancy (i.e., the percentage of heap that is occupied by live objects at a given time), when each benchmark is executed using the minimum heap size that allows it to run without an out-of-memory exception. We can see that, even with the minimum heap size that allows the benchmark to run, on the average (across all benchmarks), only 65.57% of the heap is occupied at a given time. Finally, the sixth column gives the overall execution time and the last column shows the GC execution time (to obtain the numbers in these last two columns, we executed each benchmark with the minimum heap size that allows it to run without an out-of-memory exception).

### 6.4 Experimental Results

#### 6.4.1 Reduction in Heap Space Demand

We expect our new garbage collection strategies to be useful in two aspects. First, we expect our strategies to reduce the minimum heap size necessary to execute an application without out-of-memory exception. Second, our strategies reduce the heap occupancy. That is, at a given time, our approach reduces the heap memory requirements of the application being executed. In this subsection, we provide experimental data to quantify these two benefits.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auction</td>
<td>Client for online auction</td>
<td>Comes with MIDP 2.0 reference implementation [13]</td>
</tr>
<tr>
<td>Calculator</td>
<td>Numeric calculator</td>
<td><a href="http://www.spruce.jp/freemidlets/">www.spruce.jp/freemidlets/</a></td>
</tr>
<tr>
<td>JBrowser</td>
<td>WAP browser</td>
<td><a href="http://www.jataayusoft.com/">www.jataayusoft.com/</a></td>
</tr>
<tr>
<td>JpegView</td>
<td>JPEG image renderer</td>
<td><a href="http://www.jshape.com/midp/index.html">www.jshape.com/midp/index.html</a></td>
</tr>
<tr>
<td>ManyBalls</td>
<td>Multithreaded game</td>
<td>Comes with MIDP 2.0 reference implementation [13]</td>
</tr>
<tr>
<td>MDoom</td>
<td>3D shooting game</td>
<td><a href="http://www.jshape.com/midp/index.html">www.jshape.com/midp/index.html</a></td>
</tr>
<tr>
<td>PhotoAlbum</td>
<td>Digital photo album</td>
<td>Comes with MIDP 2.0 reference implementation [13]</td>
</tr>
<tr>
<td>Scheduler</td>
<td>Personal monthly scheduler</td>
<td>holycow.tripod.co.jp/cooldownboy/</td>
</tr>
<tr>
<td>Sfmap</td>
<td>Interactive digital map</td>
<td><a href="http://www.jshape.com/midp/index.html">www.jshape.com/midp/index.html</a></td>
</tr>
<tr>
<td>Snake</td>
<td>Game</td>
<td>Comes with MIDP 2.0 reference implementation [13]</td>
</tr>
</tbody>
</table>

Table 6.2. Java benchmarks used in our experiments. The second column gives a brief description, and the third column shows where the benchmark can be found.

Table 6.4 gives the minimum heap sizes for each benchmark to run without out-of-memory exception using different garbage collectors. In this section, we mainly focus on MS, MC, MCL, MCC, and MCCL, and postpone the discussion of MCCL+ to a later section. The first part of this table (that is, the columns two through seven) gives the absolute heap sizes in KBs, whereas the second part gives the values (heap sizes) “normalized” with respect to that of the MC collector. Our first observation from the results in Table 6.4 is that, compared to the MC collector, the MS collector requires 47.9% more heap space on the average (i.e., across all our benchmarks). This is a direct result of the fragmentation problem. We also observe that both lazy allocation (in conjunction with breaking large objects into smaller subobjects) and object compression help reduce the applications’ heap memory requirements. More specifically, on the average, MCL and MCC brought down the heap memory requirements of our benchmarks by 9.5% and 10.8%, respectively, with respect to MC (the average reduction with respect to MS is around 40%). Combining them in MCCL results in even more heap memory space savings (21% on the average). An exception is Sfmap, where MCC requires more heap memory than MC. This is because of two main reasons. The first is that the handle pool requires extra space. The second reason is that Sfmap allocates many more large objects as compared to other benchmarks in our experimental suite. As discussed earlier, when a Java object is being decompressed, JVM has to maintain the object in both compressed and uncompressed formats until the decompression is completed. Holding a large object in both formats simultaneously increases the pressure on
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Number of Allocated Objects</th>
<th>Object Size Average (Maximum)</th>
<th>Total Size of Live Objects in the Heap Average Maximum</th>
<th>Execution Time (Seconds)</th>
<th>GC Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auction</td>
<td>5123</td>
<td>39 (8492)</td>
<td>84596B 68.94%</td>
<td>96.08</td>
<td>4.85</td>
</tr>
<tr>
<td>Calculator</td>
<td>11250</td>
<td>26 (1036)</td>
<td>39824B 67.80%</td>
<td>65.94</td>
<td>2.19</td>
</tr>
<tr>
<td>JBrowser</td>
<td>16160</td>
<td>56 (12012)</td>
<td>229432B 72.37%</td>
<td>338.37</td>
<td>21.85</td>
</tr>
<tr>
<td>JpegView</td>
<td>10199</td>
<td>44 (8972)</td>
<td>86524B 88.88%</td>
<td>417.35</td>
<td>50.03</td>
</tr>
<tr>
<td>ManyBalls</td>
<td>2090</td>
<td>35 (1036)</td>
<td>35088B 77.19%</td>
<td>461.91</td>
<td>1.28</td>
</tr>
<tr>
<td>MDoom</td>
<td>1319</td>
<td>61 (16396)</td>
<td>126408B 40.98%</td>
<td>500.56</td>
<td>1.43</td>
</tr>
<tr>
<td>PhotoAlbum</td>
<td>3864</td>
<td>83 (4260)</td>
<td>54388B 55.26%</td>
<td>66.71</td>
<td>4.30</td>
</tr>
<tr>
<td>Scheduler</td>
<td>10042</td>
<td>66 (1036)</td>
<td>35464B 76.52%</td>
<td>253.09</td>
<td>5.77</td>
</tr>
<tr>
<td>Sfmap</td>
<td>6599</td>
<td>27 (2460)</td>
<td>166224B 57.90%</td>
<td>81.37</td>
<td>3.95</td>
</tr>
<tr>
<td>Snake</td>
<td>1776</td>
<td>29 (1036)</td>
<td>40072B 70.59%</td>
<td>68.96</td>
<td>1.53</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td><strong>6842</strong></td>
<td><strong>47 (5674)</strong></td>
<td><strong>89802B 65.57%</strong></td>
<td><strong>216.03</strong></td>
<td><strong>9.72</strong></td>
</tr>
</tbody>
</table>

Table 6.3. Heap-related behavior of our benchmarks. The numbers in the second column include both Java object instances and arrays. The third column gives the average and maximum object sizes. The fourth column shows the maximum size of the objects that are live in the heap simultaneously, and the next column shows average heap occupancy when each benchmark is run using the minimum heap size that allows it to run without out-of-memory exception (increasing the heap size reduces the percentage heap occupancy). Execution times and garbage collection times are measured by running each benchmark with minimum heap size that allows the benchmark to run without out-of-memory exception.

the heap memory. Actually, this is one of the motivations to break down large Java objects into smaller ones. In fact, breaking down large objects enhances the effectiveness of object compression for this benchmark. Specifically, one can observe that although compression alone (MCC) does not seem to be very useful for Sfmap, combining it with object break-down (MCCL) brings an extra 27KB heap space saving over the MCL collector. To summarize, our compression and lazy allocation based strategies reduce the minimum heap size demand of Java applications; that is, they allow applications to execute with smaller heap sizes.

We now analyze the second benefit of our strategies, namely, reducing the heap occupancy during the course of execution. Figure 6.10 shows the total size of the live objects in the heap over time. Only four benchmarks are shown here since the trends for other benchmarks are similar to those presented here. We can observe from this figure that peak of the heap memory demand does not occur frequently and that each peak lasts for only a short period of time. One can also see that, at a given time, our collectors reduce the pressure on the heap
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MS</th>
<th>MC</th>
<th>MCL</th>
<th>MCC</th>
<th>MCCL</th>
<th>MCCL+</th>
<th>MS</th>
<th>MC</th>
<th>MCL</th>
<th>MCC</th>
<th>MCCL</th>
<th>MCCL+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auction</td>
<td>128</td>
<td>83</td>
<td>76</td>
<td>72</td>
<td>62</td>
<td>58</td>
<td>154.2</td>
<td>91.6</td>
<td>86.8</td>
<td>74.7</td>
<td>69.7</td>
<td>69.9</td>
</tr>
<tr>
<td>Calculator</td>
<td>55</td>
<td>40</td>
<td>40</td>
<td>34</td>
<td>34</td>
<td>32</td>
<td>138.5</td>
<td>100.0</td>
<td>85.0</td>
<td>85.0</td>
<td>80.0</td>
<td>80.0</td>
</tr>
<tr>
<td>JBrowser</td>
<td>260</td>
<td>226</td>
<td>196</td>
<td>195</td>
<td>164</td>
<td>157</td>
<td>115.0</td>
<td>86.7</td>
<td>86.2</td>
<td>72.6</td>
<td>69.5</td>
<td>69.5</td>
</tr>
<tr>
<td>JpegView</td>
<td>127</td>
<td>85</td>
<td>85</td>
<td>79</td>
<td>77</td>
<td>64</td>
<td>149.4</td>
<td>100.0</td>
<td>92.9</td>
<td>90.6</td>
<td>75.3</td>
<td>75.3</td>
</tr>
<tr>
<td>ManyBalls</td>
<td>57</td>
<td>35</td>
<td>35</td>
<td>31</td>
<td>31</td>
<td>29</td>
<td>162.9</td>
<td>100.0</td>
<td>88.6</td>
<td>88.6</td>
<td>82.9</td>
<td>82.9</td>
</tr>
<tr>
<td>MDoom</td>
<td>178</td>
<td>124</td>
<td>71</td>
<td>114</td>
<td>76</td>
<td>57</td>
<td>143.5</td>
<td>57.3</td>
<td>91.9</td>
<td>61.3</td>
<td>46.0</td>
<td>46.0</td>
</tr>
<tr>
<td>PhotoAlbum</td>
<td>96</td>
<td>55</td>
<td>55</td>
<td>50</td>
<td>50</td>
<td>46</td>
<td>174.5</td>
<td>100.0</td>
<td>90.9</td>
<td>90.9</td>
<td>83.6</td>
<td>83.6</td>
</tr>
<tr>
<td>Scheduler</td>
<td>56</td>
<td>37</td>
<td>36</td>
<td>32</td>
<td>32</td>
<td>31</td>
<td>151.4</td>
<td>97.3</td>
<td>86.5</td>
<td>86.5</td>
<td>83.8</td>
<td>83.8</td>
</tr>
<tr>
<td>Sfmap</td>
<td>292</td>
<td>162</td>
<td>118</td>
<td>175</td>
<td>91</td>
<td>78</td>
<td>118.5</td>
<td>72.8</td>
<td>108.0</td>
<td>56.2</td>
<td>48.1</td>
<td>48.1</td>
</tr>
<tr>
<td>Snake</td>
<td>72</td>
<td>42</td>
<td>42</td>
<td>35</td>
<td>35</td>
<td>33</td>
<td>171.4</td>
<td>100.0</td>
<td>83.3</td>
<td>83.3</td>
<td>78.6</td>
<td>78.6</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td><strong>132</strong></td>
<td><strong>90</strong></td>
<td><strong>75</strong></td>
<td><strong>81</strong></td>
<td><strong>65</strong></td>
<td><strong>59</strong></td>
<td><strong>147.9</strong></td>
<td><strong>90.5</strong></td>
<td><strong>89.2</strong></td>
<td><strong>79.0</strong></td>
<td><strong>65.6</strong></td>
<td><strong>65.6</strong></td>
</tr>
</tbody>
</table>

Table 6.4. Minimum heap sizes to execute benchmarks without an out-of-memory exception. The first part of this table (that is, the columns two through seven) gives the absolute heap sizes in KBs, whereas the second part gives the values (heap sizes) “normalized” with respect to that of the MC collector.

space as compared to the MC collector. Note that this reduction in heap occupancy might be important in different contexts. For example, in a multiprogrammed environment, a reduction in heap space can allow other applications to utilize the unused memory. Alternately, the unused memory parts can be placed into a low power operating mode [43] to reduce memory energy consumption. Quantifying the impact of heap space reduction from these two angles is in our future agenda.

### 6.4.2 Compression/Decompression Behavior

Analyzing the number of compressions and decompressions is very important as it has a direct impact on performance (execution cycles). Table 6.5 gives the number of compressions and decompressions for each benchmark running with MCCL when the entire execution is considered. Note that if the same object is compressed (decompressed) \( N \) times, we counted it as \( N \) compressions (decompressions). In these experiments, each application was run with the minimum heap size that allowed it to execute without out-of-memory exception (as shown in the sixth column of Table 6.4). Comparing the number of compressions (the second column in Table 6.5) with the total number of created objects (the second column in Table 6.3), one can observe that
only a small percentage of our objects were compressed (9.1% on the average). This is a direct result of our compression policy: we compress objects only when we really need to compress them; that is, at the peaks of the heap space demand. And, since such peaks do not occur very frequently and each peak lasts for only a short period of time, we do not perform frequent object compressions. We also observe from Table 6.5 that only a small percentage (22.36% on the average) of all compressed objects are decompressed. This is an interesting result and indicates that most compressed objects are not used subsequently by the application, although they are
still live at the moment they are compressed (from the GC’s perspective). The last three columns of this table give the number of objects that have been decompressed only \( N \) \((N > 0)\) times. We find that no object has been decompressed more than three times. We can also see from these results that most of our objects have been decompressed only once or twice. In fact, among the objects that have ever been decompressed, an overwhelming majority (82.5%) have been decompressed only once. That is, after they have been compressed, they have been decompressed only once, and subsequently (after, possibly, several accesses), they have become garbage. Thus, the numbers in these last three columns explain the low percentage values shown in the column four of Table 6.5.

It is also important to compare MCC and MCCL from the perspective of the number of object compressions and decompressions. Such a comparison is given in Table 6.6, which shows the same information as in Table 6.5, except that the applications are run here using the minimum heap size that allows the MCC collector to execute without an out-of-memory exception (in contrast, the results in Table 6.5 were obtained by running applications using the minimum heap size that allows the MCCL collector to execute without an out-of-memory exception). The results given in Table 6.6 clearly indicate the importance of lazy allocation and breaking down large objects into subobjects. Specifically, using MCCL instead of MCC results in a 59% (64%) reduction in the number of compressions (decompressions).

Since MCCL is the most effective of all the strategies evaluated so far in reducing the heap memory demand, we wanted to study its behavior more carefully. Figures 6.11 and 6.12 present the heap memory usage of our benchmarks under the MCCL collector. Each benchmark was run using three different heap sizes: (1) the minimum heap size required by MCCL (the sixth column of Table 6.4), (2) the minimum heap size required by MC (the third column of Table 6.4), and (3) 150% of the size in (2). We use heap size (2) basically to compare the behavior of MCCL with MC in the course of execution, and (3) to demonstrate the behavior of MCCL when there exist plenty heap space. Each graph shown in Fig. 6.11 and Fig. 6.12 has three curves, labeled as “Overall”, “Live,” and “Compressed.” “Overall” represents the heap usage at a given point during execution, i.e., the total size of the live objects (including both compressed and uncompressed objects) and garbage in the heap. It should be noticed that each drop in the “Overall” curve indicates an invocation of the GC. The curve labeled “Live” corresponds to
Table 6.5. The number of compressions and decompressions using MCCL. Each benchmark was run with the minimum heap size that allowed it to complete without an out-of-memory exception (see the sixth column of Table 6.4). The fourth column gives the number of decompressions as a percentage of the number of compressions. The last three columns indicate that most of the Java objects are decompressed only once or twice.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Number of Comp.</th>
<th>Number of Objects Decompressed</th>
<th>Decompressions</th>
<th>Number of Objects Decompressed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N = 3</td>
<td>N = 2</td>
<td>N = 1</td>
<td>% of Total Comp.</td>
</tr>
<tr>
<td>Auction</td>
<td>583</td>
<td>183</td>
<td>4</td>
<td>31.39%</td>
</tr>
<tr>
<td>Calculator</td>
<td>226</td>
<td>46</td>
<td>0</td>
<td>20.35%</td>
</tr>
<tr>
<td>JBrowser</td>
<td>1393</td>
<td>431</td>
<td>0</td>
<td>30.94%</td>
</tr>
<tr>
<td>JpegView</td>
<td>1725</td>
<td>433</td>
<td>0</td>
<td>25.10%</td>
</tr>
<tr>
<td>ManyBalls</td>
<td>264</td>
<td>83</td>
<td>0</td>
<td>31.44%</td>
</tr>
<tr>
<td>MDoom</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>PhotoAlbum</td>
<td>235</td>
<td>54</td>
<td>0</td>
<td>22.98%</td>
</tr>
<tr>
<td>Scheduler</td>
<td>172</td>
<td>36</td>
<td>0</td>
<td>20.93%</td>
</tr>
<tr>
<td>Sfmap</td>
<td>1431</td>
<td>92</td>
<td>0</td>
<td>6.43%</td>
</tr>
<tr>
<td>Snake</td>
<td>234</td>
<td>38</td>
<td>0</td>
<td>16.24%</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td><strong>626</strong></td>
<td><strong>140</strong></td>
<td><strong>0.4</strong></td>
<td><strong>22.36%</strong></td>
</tr>
</tbody>
</table>

The total size of live objects, including both compressed and uncompressed.\(^3\) Finally, the curve labeled “Compressed” represents the total size of all compressed objects. This curve climbs up when objects are compressed and drops when a compressed object becomes garbage or is decompressed. Figures 6.11 and 6.12 clearly indicate that compressions are performed only at peaks of heap memory demands. As discussed earlier, these peaks do not occur very frequently and each peak, when occurs, does not last very long.

\(^3\)One may expect that an increase in Live curve should always be accompanied with an increase in the Overall curve in Figures 6.11 and 6.12. However, this is not true due to the allocation of stack frames. In handheld devices, due to memory constraints, KVM does not use a separate stack space for each Java thread. Instead, it allocates stack chunks in the heap. Each stack chunk is 520 bytes and contains one or more stack frames. A stack chunk may become garbage when the corresponding method returns. A garbage stack chunk is detected immediately without invoking the GC. Garbage stack chunks are put in a free stack chunk table. When the application needs a new stack chunk, if the free stack chunk table is not empty, KVM allocates the chunk from this table. In this case, we observe an increase in the Live curve trend while the Overall curve remains unchanged. The free chunk table is emptied after each invocation of the GC since all the garbage stack chunks have been collected.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Decompressions Number of Comp.</th>
<th>Decompressions % of Total Comp.</th>
<th>Number of Objects Decompressed N times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auction</td>
<td>530</td>
<td>131</td>
<td>0</td>
</tr>
<tr>
<td>Calculator</td>
<td>226</td>
<td>46</td>
<td>5</td>
</tr>
<tr>
<td>JBrowser</td>
<td>939</td>
<td>261</td>
<td>0</td>
</tr>
<tr>
<td>JpegView</td>
<td>1926</td>
<td>867</td>
<td>228</td>
</tr>
<tr>
<td>ManyBalls</td>
<td>264</td>
<td>83</td>
<td>18</td>
</tr>
<tr>
<td>MDoom</td>
<td>365</td>
<td>207</td>
<td>14</td>
</tr>
<tr>
<td>PhotoAlbum</td>
<td>235</td>
<td>54</td>
<td>0</td>
</tr>
<tr>
<td>Scheduler</td>
<td>172</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>Sfmap</td>
<td>1429</td>
<td>90</td>
<td>0</td>
</tr>
<tr>
<td>Snake</td>
<td>234</td>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td>632</td>
<td>181</td>
<td>26</td>
</tr>
</tbody>
</table>

(a) MCC

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Decompressions Number of Comp.</th>
<th>Decompressions % of Total Comp.</th>
<th>Number of Objects Decompressed N times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auction</td>
<td>530</td>
<td>131</td>
<td>0</td>
</tr>
<tr>
<td>Calculator</td>
<td>226</td>
<td>46</td>
<td>5</td>
</tr>
<tr>
<td>JBrowser</td>
<td>939</td>
<td>261</td>
<td>0</td>
</tr>
<tr>
<td>JpegView</td>
<td>1926</td>
<td>867</td>
<td>228</td>
</tr>
<tr>
<td>ManyBalls</td>
<td>264</td>
<td>83</td>
<td>18</td>
</tr>
<tr>
<td>MDoom</td>
<td>365</td>
<td>207</td>
<td>14</td>
</tr>
<tr>
<td>PhotoAlbum</td>
<td>235</td>
<td>54</td>
<td>0</td>
</tr>
<tr>
<td>Scheduler</td>
<td>172</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>Sfmap</td>
<td>1429</td>
<td>90</td>
<td>0</td>
</tr>
<tr>
<td>Snake</td>
<td>234</td>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td>260</td>
<td>65</td>
<td>2</td>
</tr>
</tbody>
</table>

(b) MCCL

Table 6.6. The number of compressions and decompressions for (a) MCC and (b) MCCL. Each benchmark was run with the minimum heap size that allowed it to execute without out-of-memory exception using MCC (see the fifth column of Table 6.4). One can see that using lazy object allocation and breaking down large objects into smaller ones helps reduce the number of compressions/decompressions significantly over the pure compression based strategy.
Fig. 6.11. Heap memory usage of MCCL. “Overall” represents the heap usage at a given point during execution, i.e., the total size of live objects (including both compressed and uncompressed objects) and garbage in the heap. The curve labeled “Live” corresponds to the total size of live objects, including both compressed and uncompressed. The curve labeled “Compressed” represents the total size of all compressed objects.
Fig. 6.12. Heap memory usage of MCCL (continued).
Fig. 6.13. Heap memory usage of MCCL (continued).

6.4.3 Performance Impact

While compressing heap objects is beneficial from the memory usage perspective, it is also important to consider its impact on performance. Fig. 6.14 gives the runtime overheads incurred by MCCL. The overheads in this figure are given as percentages of the execution time of MC assuming an “infinite” heap space (in reality, to calculate the time with the infinite heap space, we calculated the time with the finite heap space and deducted the time taken by the GC). We normalized the performance overhead to the ideal execution time (i.e., the execution time with infinite heap space) due to two reasons. The first is that, ideally, given the same application and the same input, changing the heap size can only change the time spent within the GC; the time spent for executing the bytecodes is equal to the ideal execution time regardless of the actually size of the heap. By normalizing the overheads with respect to the ideal execution time, we are able to compare the overheads across different heap sizes; this enables us to study the impact of the heap size. Second, the MC collector cannot run with the minimum heap size that
allows MCCL to run. In this case, it is difficult to normalize the overhead with respect to the execution time of MC with the same heap size.

The overhead in Fig. 6.14 is divided into several components. “Lazy Access” represents the time overhead due to lazy allocation. “Indirect Reference” corresponds to the overhead due to using object handles. “Compression” is the time spent in compression and “Decompression” is the time spent in decompression when accessing objects. “GC Decompression” is the time spent in decompression during GC (when traversing the heap). The component denoted as “Check” represents the time spent at each object access to check whether the object is in the compressed format or not. Finally, “Other GC Time” is the time spent in collecting garbage (note that the infinite heap configuration does not use garbage collection). Our first observation from the graph in Fig. 6.14 is that, on the average, working with MCCL brings a 9.1% performance overhead as compared to the ideal heap scenario. It should be noted, however, that only 3.5% overhead is actually due to factors other than the time spent in performing garbage collection (that is, the “Other GC Time” component); and, this last component should exist (in varying magnitudes) with any limited size heap (i.e., it is not due to our compression and lazy allocation based strategy, and occurs even when we use MC with a finite heap). Therefore, the extra overhead introduced by our strategy is very low.

Figures 6.15 and 6.16 compare the performance overhead incurred by MCCL and MC when they are used with the same heap size. In Fig. 6.15, for each benchmark, we used the minimum heap size that allows both MC and MCCL to execute without giving an out-of-memory exception (see the third column Table 6.4). We can see that MCCL is about only 1.5% slower than MC on the average. Note that a performance advantage of MCCL over MC is that it can reduce the number of GC invocations. Consider, for example, JpegView in Figure 6.15. For this benchmark, with a 85KB heap, MCCL is faster than MC by 4.3%. This is because MCCL reduces the number of GC invocations since the effective heap size is increased. In the graph titled as “JpegView (85KB Heap)” in Fig. 6.11, we observe two invocations of the compressor: at 90th second and at 340th second. These two invocations reduce the application’s footprint significantly, which in turn leads to fewer GC invocations. This last observation implies the possibility of using heap compression to reduce the overall garbage collection time even when there is enough heap space (Section 6.5 evaluates such an aggressive use of object compression).
Fig. 6.14. Runtime overheads due to MCCL with the minimum heap size for MCCL. The values are given as percentages of the MC execution time under an ideal (infinite) heap memory. The numbers next to benchmark names denote the heap sizes used to run the benchmark.

Comparing the graphs in Fig. 6.16 and Fig. 6.15, we observe that, as the heap size increases, the performance degradations (with respect to the ideal heap configuration) due to MC and MCCL are both reduced. This is because, since there is enough space in the heap, the collectors are invoked less frequently.

### 6.5 Aggressive Object Compression

In our compression based strategies discussed so far, the compression is invoked only when compaction is not successful to provide sufficient free space to accommodate the object to be allocated. In other words, if the compaction is successful in providing sufficient free space, the compression is not activated. Therefore, our approach is oriented towards reducing the impact of compression/decompression on performance. In this section, we investigate the pros and cons of a more aggressive compression strategy. In this strategy, we aggressively compress objects even if just using compaction would allow the application to continue successfully. Our focus is on the MCCL since it is the one that generated the best results so far. Our new collector, denoted MCCL\( (k) \), operates with a threshold parameter, \( k \). After the mark phase (of the MCCL), the
Fig. 6.15. Runtime overheads due to MCCL and MC with the minimum heap size for MC. The values are given as percentages of the MC execution time under an ideal (infinite) heap memory. For each benchmark, the bar on the left is for MCCL, and the bar on the right is for MC. The numbers next to benchmark names denote the heap sizes used to run the benchmark.

Collector compares the size of the available free space (denoted $A$) with the size of the object (denoted $S$) and $kH$, where $H$ is the size of the entire heap. If
\[
A < S \quad \text{or} \quad A < kH,
\]
the collector performs compression; otherwise, it compacts the heap without compression. Note that MCCL(0\%) is our baseline MCCL strategy discussed so far in this chapter. It should also be noticed that the larger the $k$ parameter, the more likely that the compression will be invoked.

In the following, we analyze the impact of this new strategy from the performance and heap memory perspectives. Figure 6.17 presents the performance impact of $k$ with different heap memory sizes. Each benchmark is run using three different heap sizes. The increases in execution time of MC are also presented for comparison purpose. Obviously, MC results do not change with varying $k$. On the average, MCCL is slower than MC with the same space by around 2\%. We can observe from these results that, when $k$ is increased, with smaller heap sizes, the overall performance overhead of MCCL (which includes all the components as discussed earlier) is decreased. In contrast, when the heap size is larger, increasing the value of $k$ increases the performance overhead. This observation can be explained as follows. Let us assume that
two successive garbage collections, GC1 and GC2, are invoked at times $t_1$ and $t_2$, respectively. Let us also assume that, right after GC1 is invoked at $t_1$, the total size of the objects in the compressed format is $c$; the original uncompressed size of these objects is $a$; and the total size of the objects that are decompressed during the interval $(t_1, t_2)$ is $d$ ($d \leq a$). Note that, when an object is decompressed, the compressed version of this object becomes garbage, which remains in the heap until it is collected by the next invocation of the GC. Therefore, if $a > c + d$, the compression invoked during GC1 postpones the invocation of GC2, i.e., if GC1 had not compressed the objects, GC2 would have been invoked earlier than $t_2$. However, if $a < c + d$, the compression during GC1 actually causes GC2 to be invoked earlier, i.e., if GC1 had not compressed the objects, GC2 would have been invoked later than $t_2$. Note that $a$ is determined by the total size of the live objects in the heap, and that $c$ is determined by both $a$ and the compression ratio. Neither $a$ nor $c$ is affected by the heap size. However, when the heap size is small, each interval between two successive GC invocations is short (even with compression), which means that the number of objects accessed during $(t_1, t_2)$ is small — which also means that $d$ is small.
Therefore, for small heap sizes, compression is more likely to reduce the number of GC invocations. Similarly, for larger heap sizes, the interval \((t_1, t_2)\) is longer and \(d\) tends to be larger. As a result, compression is more likely to increase the number of GC invocations. Another factor that also influences the overall performance overhead is the tradeoff between GC cost and compression/decompression cost. A large \(k\) value usually results in higher compression/decompression costs. For a small heap size, a large \(k\) value tends to reduce the number of GC invocations. If the reduction in the GC costs is larger than the increase in the compression/decompression costs, we are likely to observe a reduction in the overall cost. On the other hand, for a large heap size, a large \(k\) value is less likely to reduce the number of GC invocations. Consequently, the overall cost is increased as the value of \(k\) is increased.

We next study the heap behavior of MCCL\((k)\) and compare it to MCCL\((0\%)\). Fig. 6.18 shows the impact of \(k\) on heap behavior with different heap memory sizes. For the minimum heap size that allows each benchmark to run without an out-of-memory exception (the graphs on the left side of Figure 6.18), we observed that the maximum value of the total size of live objects does not change as the value of \(k\) varies. However, for both heap sizes we experimented, we observed that, during most of the execution time, a large \(k\) value results in a small total size for the live objects. This is due to the fact that, with a large \(k\) value, object compression is performed more frequently, and thus, the heap contains a large number of compressed objects.

### 6.6 Eliminating Object Handles

Our base implementation explained earlier employed object handles mainly because of the difficulty associated with updating reference fields (i.e., the fields that contain references to other objects) of the compressed objects in the compact-compress phase of the collector. Specifically, updating a reference field in a compressed object may cause the size of the object to expand. Since it is not always possible to find the space for the object to expand, our base implementation solved this problem using handles. It should be noted, however, that object handles incur two problems. First, a handle incurs dereferencing overhead whenever the corresponding object is accessed. Second, handle pool occupies space in the heap memory. If the size of the handle pool is small, the application uses up the object handles quickly, which forces frequent GC invocations even if we have available heap space. Consequently, to make the best use of
Fig. 6.17. Impact of the threshold parameter $k$ on performance. The influence of varying the $k$ parameter depends on whether a large or a small heap memory is used. The increase in execution time is normalized with respect to the execution time of the MC collector with an infinite heap space. The increase in execution time of MC with finite heap sizes are also presented for comparison purpose (Obviously, MC results do not change with varying $k$).
Fig. 6.18. Impact of the threshold parameter \( k \) on heap behavior.
the heap space, we need to fine-tune the size of the handle pool according to the behavior of each application. However, access patterns of different applications may differ from each other dramatically. Further, even the same application can exhibit different heap behavior depending on the user input provided. Therefore, it is very difficult to tune the size of the handle pool successfully. In this section, we present our enhanced implementation that eliminates object handles completely. The cost of doing so is the slight degradation in the compression ratio. However, as will be discussed shortly, we still achieve heap memory savings. Note that the implementation discussed in this section does not use aggressive compression explained in the previous section. It uses our base compression strategy.

Our enhanced implementation divides each object instance into “reference zone” and “non-reference zone” as shown in Fig. 6.19. The reference zone contains only the reference fields, whereas the non-reference zone represents the remaining fields. Our enhanced implementation compresses only the non-reference zone; the fields in the reference zone remain uncompressed. We apply the same technique to arrays as well. The non-reference arrays (that is, the arrays that do not contain references) are compressed as discussed earlier. The reference arrays, on the other hand, are compressed differently, i.e., each bit in the bitmap now corresponds to an element of the array. A 0-bit indicates the corresponding element is null; a 1-bit indicates a reference that is stored in the original format. Keeping the reference fields of each object instance and non-null elements of each reference array in the uncompressed form allows the collector to update the corresponding references whenever an object is moved during the compact-compress phase (without handles).

The process of object decompression in the absence of handles is depicted in Fig. 6.20. After object O1 is decompressed, a forward pointer to the decompressed data is set in the header of the compressed object. When a reference field pointing to the old object is used, our implementation first checks whether the reference needs to be forwarded. If the reference is pointing to an object that has been decompressed, JVM needs to update the reference field to point to the location of the newly decompressed object. In the mark phase of the garbage collection, the collector also checks and updates the reference fields in a similar fashion. It should be noticed that explicitly checking whether a reference has been forwarded for each object access may incur an
overhead comparable to using handles. However, we can make use of hardware to forward references transparently. KVM uses 32 bits to represent a reference. To the best of our knowledge, no embedded system today is using more than 2GB memory, which means that the highest-order address bit of each reference is always zero. We use this bit as a flag. If a reference points to a compressed object, this bit is set to 1, otherwise, it remains 0. The system is configured in such way that accessing a non-exist memory location triggers a hardware memory protection exception. We know that the mark-compact garbage collector needs to update each reference in the heap after compaction. Therefore, we modify this procedure so that the compressor also sets the flag bit to 1 for each reference pointing to a compressed object. During the execution of bytecodes, accessing an uncompressed object does not cause any overhead. However, accessing a compressed or forwarded object triggers a memory protection exception. The exception handler then checks if this is an access to a compressed object or to a forwarded object. If it is to a forwarded object, the handler sets the flag of the reference to 0, and then, the virtual machine resumes its execution. If it is to a compressed object and decompression is necessary, the handler decompresses the object.

In the rest of this chapter, we denote the MCCL without object handles as “MCCL+” (see Table 6.1). We can observe from the results in Table 6.4 that MCCL+ outperforms MCCL for all our benchmarks. This is because of two main reasons. The first is that MCCL+ does not need the handle pool. The second reason is that MCCL+ allows the reference fields of a compressed object to be accessed without decompression. However, for a fair comparison, we need to consider its performance as well. Fig. 6.21 shows the impact of handle elimination on the performance by comparing MCCL+ with MCCL. We see that, for all benchmarks except ManyBalls and Snake, MCCL+ outperforms MCCL. For ManyBalls and Snake, MCCL+ is slower than MCCL, mainly due to frequent accesses to forwarded and compressed objects. On the average, the performance degradations due to MCCL+ and MCCL over the MC with the infinite heap are 5.7% and 9.1%, respectively.

6.7 Discussion and Future Work

Up to this point, we discussed several strategies for reducing the heap memory requirements of embedded Java applications. While our different strategies allow us to explore a large
Fig. 6.19. Division of a Java object into reference and non-reference zones. Our enhanced implementation compresses only the non-reference zone; the fields in the reference zone remain uncompressed. We apply the same technique to arrays as well.

Fig. 6.20. Decompression of object O1 when no handle is used. (a) Before decompression, O1 is in the compressed format. (b) After decompression, a forward reference to the newly decompressed O1 is set in the location that used be to the header of O1.

design space, there are still many alternative designs that can potentially be studied. In this section, we discuss several such alternatives and point out the directions for further research.

Selective Compression. In our current implementation, we compress all the live Java objects in the heap. However a more sophisticated object compression strategy can minimize the number of objects that need to be compressed by considering the size of current allocation request. Specifically, the compressor can (re-)calculate heap memory saving after compressing each object. When the total memory saving from compression plus the size of the free space
Fig. 6.21. Runtime overheads due to MCCL+. Each benchmark is run with the same heap size as in Figure 6.14. The overheads are as percentages of the overall execution time of MC with ideal (infinite) heap memory. For each benchmark, the bar on the left is for MCCL+, and the bar on the right is for MCCL (its breakdown into components is as shown in Fig. 6.14). The numbers beside the names of each benchmark are the sizes of the heaps used to run the benchmarks.

calculated in the mark phase is larger than the size current allocation request, we can stop the compression process. In addition, it is also possible to rank the objects according to their access frequencies and utilize this information within the compressor to compress only the objects that are not likely to be accessed in the near future.

**Independent Compression.** In our current implementation, compression is performed during garbage collection. An advantage of this approach is that it is easy to implement. However, it may cause longer pause for garbage collection, which is not desirable in real-time or user-interactive applications. An alternative would be using a dedicated thread to incrementally compress the objects. More specifically, the virtual machine can invoke a compression thread at regular intervals. When the compression thread is scheduled, it selectively compresses a set of objects that will not be used in the near future, and then falls back to sleep. This approach to compression avoids very long pauses due to compression by distributing compressions across the lifetime of the application being executed. The apparent drawbacks include more complicated implementation and extra synchronization overhead.
**Using Compression with Generational Collectors.** In our implementation, we used compression with a mark-compact collector. However, generational collectors [74, 127] may also employ object compression to reduce the heap memory demands. A generational collector divides the heap into two generations: the young generation and the old generation. All new objects are allocated in the young generation. When the space in the young generation is used up, a local collector is invoked to collect garbage in the young generation. After several local collections, the surviving objects in the young generation are promoted to the old generation. When the space occupied by the old generation reaches a given threshold, a global collector is invoked to collect garbage in both the generations. Compared to the local collector, the global collector is much more expensive. Our compression scheme may be incorporated into the global collector. Specifically, when the virtual machine fails to allocate space for a new object, the global collector is invoked to compress live objects to make space for the new object. It should be noted that, during the peaks of memory demands, the global collector (with compression) may be very costly. Fortunately, as has been mentioned previously, the peaks do not occur frequently and, when they occur, they do not last long. Therefore, in most of the time, the global collector (with compression) may not need to be invoked very frequently.

**Hardware-Based Implementation of Compressor and Decompressor.** In this work, we employed a software-based implementation for the compressor/decompressor. However, in principle, both the compressor and decompressor may be implemented in hardware as well. There have been several hardware compression schemes proposed in the literature (e.g., [80, 51]). Similarly, the zero removal compression can also be implemented in hardware. Obviously, hardware-based compressor/decompressor is expected to run much faster than the corresponding software-based implementation. Another benefit of the hardware-based implementation is that the compressor and decompressor can work in parallel with the main processor; this can enable us to hide the overhead by overlapping compression/decompression with application execution. For example, when the virtual machine finds that a compressed object is going to be accessed in the near future and that there exist a large amount of heap memory, it can allocate the space to hold the decompressed object and then invoke the hardware decompressor. In this mode of operation, the virtual machine does not need to wait for the decompressor to finish its work. Instead, it can continue with the application execution as long as the object in question is not
accessed immediately. When the virtual machine really needs to access the object, it is very likely that the decompressed object will be ready to be used. In other words, using a hardware-based compressor/decompressor, one can implement a “pre-decompression” scheme that can significantly reduce the overhead associated with accessing the objects.

6.8 Concluding Remarks

Reducing memory demands of Java applications is critical for embedded systems as these systems operate under memory constraints. An effort in this direction, if successful, can increase the number of Java applications that can execute in systems with low memory budget. In this chapter, we present a set of heap management strategies for reducing memory footprint. Our major conclusions can be summarized as follows:

- Our compression-based garbage collection strategy, MCC, reduces the minimum heap size required to run Java applications by 10.8%, on the average, over the MC collector. The corresponding memory saving over the MS collector is 40% across all ten benchmarks. These results are obtained by using object compression when it is absolutely necessary to continue executing the application without an out-of-memory exception.

- Our lazy allocation and object break-down technique and the garbage collection strategy based on them, MCL, also reduce the required heap sizes significantly (9.5% on the average). In addition, combining MCC and MCL under the integrated strategy MCCL increases memory savings further (21% on the average).

- In addition to reducing the minimum heap sizes to execute applications, our garbage collectors also reduce the total size of the live objects that need to be kept in the heap. This can be exploited, among other things, for reducing energy consumption of the memory system or increasing the concurrency in a multiprogrammed environment.

- The performance degradation (with respect to the MC collector that uses an infinite heap memory) caused by MCCL is affected by the actual heap size that is available to the application. Generally, the larger the heap size, the smaller the degradation. Using the minimum heap size that allows the application to run without out-of-memory exception,
the average performance degradation of MCCL (over MC with infinite heap) was found to be 9.1%. However, most of this overhead (5.6%) is due to the garbage collection activity itself, which should occur with any collector with finite heap. In fact, our results show that MCCL is less than 2% slower than MC with the same heap size.

- Our experiments with a more aggressive compression strategy (which compresses objects even if is is not strictly necessary to do so) indicate that such a strategy improves the performance of our benchmarks over the baseline MCCL by up to 9.2% (5.5% on the average) with the minimum heap size.

- Our enhanced implementation that does not use object handles, called MCCL+, improves both memory behavior and performance degradation of the base MCCL. The main reasons for memory savings are that MCCL+ does not need handle pool (which occupies a significant amount of heap space), and that the reference fields can be accessed without decompressing the corresponding object. The main reasons for performance improvements are that, in MCCL+, objects can be accessed directly without incurring dereferencing, and that MCCL+ eliminates the overheads due to the handle pool maintenance.

- Our experience with compression, lazy allocation, and object break-down suggests that these strategies can also be used in conjunction with different base collectors (e.g., generational), virtual machines, and compression algorithms.
Chapter 7

Exploiting Frequent Field Values in Java Objects

7.1 Introduction

The memory compression schemes discussed in the previous chapter treat data entities in the memory as structureless streams. To retrieve information from a compressed data entity, we first have to decompress the entire (or part of) entity, which may incur both performance and space overheads. Consequently, data compression needs to be applied judiciously so that the benefits accrued are larger than the overheads imposed.

In many Java applications, a large fraction of objects in the heap are similar to each other. In fact, a small set of values tend to appear in some fields of the heap-allocated objects much more frequently than other values. This small set of values are the frequent values for these fields (or, frequent field values for short). This similarity among objects can be exploited to reduce the heap space. For example, the schemes proposed in [96, 94, 30] compare the content of an object with a set of existing objects and replace the similar ones with a single copy in order to reduce the storage space. However, this is done mainly as a manual operation. Further, such schemes do not exploit the similarity that exists in individual fields but not across the entire object. The work presented in this chapter first analyzes the similarity in fields across a set of objects. Based on this analysis, we then propose two frequent field value based object compression schemes that exploit field-level similarity to reduce heap space requirements of Java applications without manual optimization.

7.2 Frequent Field Value Characterization

This section characterizes the frequent field values in Java applications, and identifies opportunities for exploiting the results of this characterization for reducing the heap memory space required to store object instances in embedded Java environments. Our optimization targets only object instances, not arrays.
7.2.1 Experimental Setup

We use the SpecJVM98 benchmark suite [17] to study the existence of frequent field values. This benchmark suite consists of eight Java programs. These benchmark programs can be run using three different inputs, which are named as s1, s10, and s100. We present our field value characterization results for s1 and s10. Since the frequent field value characteristics with these two input sets are quite similar (as will be shown shortly), we present the results of our two proposed schemes using the s1 data set only, and perform a sensitivity analysis with s10.

We use an instrumented JVM based on Kaffe VM 1.1.4 [12] to collect the execution trace of each benchmark. The traces include detailed information about each object allocation and access. Our trace-based simulator simulates the execution of each benchmark, and provides information about the memory savings and performance overheads. The important characteristics of our applications are given in Table 7.1. The third column of this table gives the number of classes loaded, the fourth column shows the number of object instance creations, and the fifth column gives the average size of an object instance for each application. The sixth column shows the execution cycles obtained by executing our applications using Sun JDK 1.4 (with Hotspot execution engine client version [21]) on a Solaris system with SPARC V9 microprocessor. The execution cycles are obtained through the performance counters available in the microprocessor. These cycles are referred as the base results in the rest of this chapter. We later quantify how much overhead the different schemes we evaluate incur over these base execution cycles. Finally, the last two columns give the maximum heap occupancy without and with arrays. In this work, the “maximum heap occupancy” is defined as the maximum total size of the live objects (and arrays) at any given point during execution. Note that this value determines the minimum heap size that the application requires to run without an out-of-memory exception.

7.2.2 Existence of Frequent Field Values

To quantify the extent of the frequent field values existing in Java applications, for each value $v$ that may appear in the $j^{th}$ field of class $C_i$, we maintain an occurrence counter $K_{i,j,v}$. At every 1KB of memory allocations, we scan the entire heap and, for each scanned instance $o$ of class $C_i$ where $o.f_j = v$, we increase the counter $K_{i,j,v}$ by 1. Note that a value that remains in a particular field of an object for longer than the sampling interval may be observed multiple
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Number of Classes</th>
<th>Number of Instances</th>
<th>Average Size of Instances</th>
<th>Execution Cycles (10^6)</th>
<th>Max. Heap Occupancy Instance</th>
<th>Array + Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>LZW-based compression</td>
<td>199</td>
<td>3101</td>
<td>24B</td>
<td>59665.7</td>
<td>54KB</td>
<td>10503KB</td>
</tr>
<tr>
<td>jess</td>
<td>An expert shell system</td>
<td>352</td>
<td>32005</td>
<td>24B</td>
<td>2019.6</td>
<td>156KB</td>
<td>374KB</td>
</tr>
<tr>
<td>raytrace</td>
<td>Single-threaded ray-tracer</td>
<td>213</td>
<td>236555</td>
<td>20B</td>
<td>6107.9</td>
<td>2395KB</td>
<td>3549KB</td>
</tr>
<tr>
<td>db</td>
<td>Database query</td>
<td>199</td>
<td>4545</td>
<td>24B</td>
<td>607.7</td>
<td>66KB</td>
<td>216KB</td>
</tr>
<tr>
<td>javac</td>
<td>Java compiler</td>
<td>352</td>
<td>21564</td>
<td>23B</td>
<td>1827.5</td>
<td>283KB</td>
<td>643KB</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>MPEG audio decoder</td>
<td>238</td>
<td>4779</td>
<td>22B</td>
<td>7545.7</td>
<td>75KB</td>
<td>314KB</td>
</tr>
<tr>
<td>mtrt</td>
<td>Multi-threaded ray-tracer</td>
<td>213</td>
<td>236492</td>
<td>20B</td>
<td>6171.3</td>
<td>2395KB</td>
<td>3548KB</td>
</tr>
<tr>
<td>jack</td>
<td>Java parser generator</td>
<td>246</td>
<td>234161</td>
<td>27B</td>
<td>6675.5</td>
<td>317KB</td>
<td>782KB</td>
</tr>
</tbody>
</table>

Table 7.1. Important characteristics of the benchmarks.

The number of occurrences is determined by both the number of objects that contain the particular value and the duration of time that this value remains in each object. For the $j^{th}$ field of class $C_i$, let us assume that:

$$K_{i,j,v}^{(1)} \geq K_{i,j,v}^{(2)} \geq \ldots \geq K_{i,j,v}^{(n)}$$

where

$$\{v_{i,j}^{(1)}, v_{i,j}^{(2)}, \ldots, v_{i,j}^{(n)}\}$$

is the set of $n$ values that appear in the $j^{th}$ field of an instance of class $C_i$. Particularly, $v_{i,j}^{(1)}$ is the most frequent value for the $j^{th}$ field of class $C_i$. For a given program, we define the distribution of the $k^{th}$ frequent value as:

$$D_k = \frac{\sum_{i,j} K_{i,j,v}^{(k)}}{\sum_{i,j,v} K_{i,j,v}}.$$ 

Fig. 7.1 shows the distribution of the occurrences of the top five frequent values for each benchmark for the s1 and s10 input sets. In this figure, “1st”, “2nd”, “3rd”, “4th”, and “5th” represent $D_1$, $D_2$, $D_3$, $D_4$, and $D_5$, respectively. One can observe from these results that, for most of the benchmarks, the most frequent value accounts for about 90% of the total number of occurrences. Only in benchmarks raytrace and mtrt do we observe that the most frequent value accounts for only about 50% of the total. This implies that, for such applications, one may need to go beyond
the most frequent values (if we are to compress a significant fraction of the objects in the heap). Among all the frequent field values, the value zero (or null for reference fields) is of particular interest because zero fields can be easily eliminated, thereby saving memory space. The bar-chart in Fig. 7.2 gives the breakdown of zero and non-zero most frequent values for each benchmark in the SpecJVM98 suite. We see from these results that non-zero values account for 55% of the most frequent values on the average. This means that a scheme that tries to exploit frequent values should accommodate for both zero and non-zero values.

While these results are encouraging from the perspective of potential memory space optimizations, one might also be interested in understanding why such frequent field values exist. To answer this, we studied the application codes in the SpecJVM98 benchmark suite, and found that frequent field values can occur due to many different reasons. As an example, in the benchmark raytrace, we found that the instances of class spec.benchmarks.raytrace.OctNode are observed 24,917,836 times during profiling. Each instance of this class represents a region of three-dimensional space that contains a certain number of three-dimensional objects. An instance of this class has five fields, two of which, ObjList and NumObj, are of particular interest. ObjList is a pointer to the header of a link table of the three-dimensional objects in this region, and NumObj is the number of the objects. Since objects are typically distributed in the three-dimensional space sparsely, many regions in the space are actually empty. We observed that, for 36% of the observed times of the instances of this class, these two fields contain null and zero, respectively. And, throughout its execution, the application creates 3,583 instances of this class, up to 2,995 of which can co-exist in the heap at a given time. This example shows how frequent field values can exist in a typical Java application.

### 7.2.3 Opportunities for Storage Optimizations

An important question now is how one can exploit these results for memory space optimization. There are at least two ways to achieve this. The first method is based on providing feedback to the programmer. More specifically, using a suitable interface, the field value characteristics discussed above can be presented to the application programmer. The programmer in turn may rewrite/restructure the application code based on these characteristics. For example, if, for a given class, a subset of the fields always have the same value, the application programmer
can consider making these fields \textit{static}, and consequently make such fields associated with the class rather than each object instance. There are also automatic static tools, such as JAX [121], that statically analyze the application codes to remove from the class files the fields that are not used by the application. Such tools may also statically identify the fields that all read of these fields observe the same value across all the instances and make these fields static. We refer to the techniques in this category as \textit{user-level} space optimization since they are performed at the user level, and they do not need special support from JVM. A common problem with the user-level optimization techniques is that the optimizations must be conservative, and consequently, they may not be able to catch all the space optimization opportunities. To evaluate the upper bound (i.e., the maximum potential memory savings) that could be achieved by such user-level space optimizations, one can assume that the profile represents the behavior of the application 100% accurately. Based on this assumption, we modified the fields that contain a single value across all the instances to be static, and we also removed the fields that are never accessed from the class. The heap occupancy behavior of our benchmarks with this user-level optimization will be presented later in the chapter and compared to our two schemes. However, in this work, we

Fig. 7.1. Frequent value distribution (the top five and the rest). Left: s1 input; right: s10 input.
mainly focus on exploiting the information about frequent field values within the virtual machine. That is, our schemes can reduce memory space requirements of Java applications without rewriting the application code. Note that the applications that are already optimized using user-level techniques can still benefit from our optimization schemes. In the next section, we discuss our optimization schemes in detail.

### 7.3 Our Compression Approach

A closer look at the problem of taking advantage of frequent field values reveals that, in order to do a good job, one needs two kinds of information: the fields that have small numbers of frequent values and the frequent values themselves. Our experience with different input sets (e.g., s1 and s10) indicates that, while the values themselves may change from one input set (execution) to another, the fields having frequent values do not change significantly. This is understandable since the fields with frequent values are usually shaped by the characteristics of the application, rather than the particular input set used. On the other hand, the field values themselves depend strongly on the input set used in a particular execution. As a result, one can use profiling with a typical input set to determine the fields that have small numbers of frequent
values, and this information can then be encoded within so-called field description files that could be distributed together with the class files of the application. We prefer using separate field description files instead of annotating fields in the class files because it is not possible to directly annotate the classes that belong to the class library. The virtual machine loads the field description files together with the class files to appropriately annotate the fields with frequent values. This is the approach employed by both the strategies presented in this chapter. The details of the process used to determine the fields that hold frequent values are explained in Section 7.3.1.

7.3.1 Determining the Level of Each Field

In this subsection, we explain how we use profiling results to identify the fields that are likely to hold frequent values at runtime. Specifically, based on the profiling information, we classify the object fields into three levels:

- **Level-0**: the field does not have a dominant frequent value;
- **Level-1**: the field has a non-zero (or non-null for reference fields) frequent value; and
- **Level-2**: the field has a frequent value that is zero or null.

It should be noted that this classification is performed offline. The results of the classification are stored in the field description file and will be used in the future executions of the program. Since the actual input to the application during the execution may be different from the input used during the profiling execution, the profiling results may not 100% accurately reflect the behavior of the application during the actual execution. The inaccuracy in the profiling results may affect the performance and the amount of the memory savings when using our schemes, however, it does not cause the program to run incorrectly.

Let us assume that class $C_i$ has fields $F = \{f_1, f_2, \ldots, f_n\}$, and $S_i$ is the set of subclasses of $C_i$. For a field $f_j \in F$, we define:

$$q(i, j) = \sum_{C_x \in S_i \cup \{C_i\}} K_{x,j,v}^{(1)}.$$
We select a subset of fields, $F^* \subseteq F$, such that:

$$|F^*| \min_{f_j \in F^*} q(i, j)$$

is maximized, where $|F^*|$ is the number of fields in $F^*$. Fields in $F^*$ are the candidates for the level-1 and level-2 fields. Another applicable rule for selecting candidates for the level-1 and level-2 fields will be discussed in Section 7.4.2.1. A field is considered to be a level-1 field if it belongs to $F^*$ and the most frequent value of this field is non-zero. On the other hand, a field is considered to be a level-2 field if it belongs to $F^*$ and the most frequent value of this field is zero (or null for pointer filed). In addition, field $f_i$ is a level-$k$ field in class $C_i$, it must be level-$k$ in all the subclasses of $C_i$. Therefore, if the level in the subclass conflicts with that in the super-class, the level of the super-class overrides that of the subclass. This is necessary since the type of an object may be implicitly cast into its super-class.

We illustrate the procedure of field classification with an example. Let us assume that we have three classes, namely, $C_x$, $C_y$ and $C_z$, and that both $C_y$ and $C_z$ are subclasses of $C_x$. Assume further that class $C_x$ has fields $C_x.f_1$, $C_x.f_2$ and $C_x.f_3$, and both $C_y$ and $C_z$ have five fields, three of which ($f_1$, $f_2$ and $f_3$) are inherited from $C_x$. By profiling the application using our instrumented JVM, we obtain the value of $q$ defined above for each field of each class. (see Table 7.2). We now show how we determine the level of each field of class $C_x$. Let us consider

\footnote{$F^*$ is actually the prefix of a list of the fields in the descending order of $q(i, j)$.}
the following subsets of $F$ (the field set of $C_x$):

$$F_1 = \{f_1, f_2, f_3\};$$
$$F_2 = \{f_1, f_2\};$$
$$F_3 = \{f_1\}.$$

Using the profile data in Table 7.2, we have:

$$|F_1| \min \{q(x, 1), q(x, 2), q(x, 3)\} = 3 \times 2000 = 6000;$$
$$|F_2| \min \{q(x, 1), q(x, 2)\} = 2 \times 8000 = 16000;$$
$$|F_3| \min \{q(x, 1)\} = 1 \times 10000 = 10000.$$

Since $16000 > 6000$ and $16000 > 10000$, for class $C_x$, we obtain:

$$F^*_x = F_2 = \{f_1, f_2\}.$$

Since $f_1 \in F^*_x$ and its most frequent value is 0, $f_1$ is classified as level-2. Field $f_2$, however, is level-1 as its most frequent value is non-zero. The fields that are not in $F^*_x$ are made level-0.

Similarly, for class $C_z$, we have:

$$F^*_z = \{f_1, f_2, f_3, f_4, f_5\}.$$

Note that, although we have $f_3 \in F^*_z$, this field is still classified as level-0 since it has been determined to be so in the super-class $C_x$.

As discussed earlier, after this profiling, we create field description files and attach them to class files. During application execution, the VM checks the field description files and uses the information there to decide the object formats, which is discussed in detail in the rest of this section. Also, in Section 7.4.2.1, we explain and evaluate an alternate scheme for determining the level of fields.

### 7.3.2 Scheme-1: Eliminating Level-2 Fields

This scheme removes the level-2 fields from the objects whose level-2 fields contain only zeros to save memory space. Fig. 7.3 shows the formats of an object in both uncompressed and compressed formats. An object is divided into two parts: the primary part containing level-0 and
level-1 fields, and the secondary part containing level-2 fields. Each memory block allocated in the heap is associated with a one-bit flag ($C$). If $C = 0$, the block contains the primary part of a compressed object or the secondary part of an uncompressed object. The rest of the first word (four bytes) of this block is the GC Header (i.e., GCHeader1 in Fig. 7.3), which contains information (such as the size of the block) needed by the garbage collector. If $C = 1$, the block contains the primary part of an uncompressed object. The remainder of the first word contains SPtr, a pointer to the secondary part of this object. The GC Header of the primary part of an uncompressed object is stored in the secondary part of this object (i.e., GCHeader2 in Fig. 7.3).

When a “NEW” instruction in the program creates an object, only the primary part is allocated. The secondary part is lazily allocated when the first non-zero value is written into one of the level-2 fields of this object. During garbage collection, the collector removes the secondary parts of the uncompressed objects whose level-2 fields contain only zero values.

At any point during execution, an object can be either in uncompressed or in compressed format. Checking the current format of the object at each field access incurs some performance overhead. However, it should be noted that the level of each field is statically determined before the execution starts. A JIT compiler can use this information to avoid the format checking overheads in most of the cases. An interpreter can also avoid a significant portion of this overhead by marking each “getfield” or “putfield” instruction according to the level of the field being accessed. Such a marking is performed when this instruction is first executed. For example, when a “getfield#n” instruction (loading the $n^{th}$ field from an object) is executed for the first time, we replace this instruction with a customized instruction “getfield0,1#n” (or “getfield2#n”) if the $n^{th}$ field of the object being accessed is of level-0 or 1 (or level-2). When a “getfield0,1#n” is executed by the interpreter, we can simply load the value from the field using the object reference and the offset of the field without checking the current format of the object. To execute “getfield2#n”, however, we first load into a register the first word (containing the flag $C$ and the pointer SPtr) of the primary part of the object being accessed, and then check the value of flag $C$. If $C = 0$, we know that the value in the field being accessed is zero; otherwise, we need to load the value of this field from the secondary part of this object. We mark “putfield#n” as “putfield0,1#n” or “putfield2#n” in the same manner. Similar to the case with “getfield” instructions, “putfield0,1#n” instructions do not incur performance overhead due to compression.
To execute a “putfield\_2\#n” instruction, however, we need to check the current format of the object. If the object is compressed and the value to be written is zero, we skip the write access since it is not necessary. If the object is not compressed, we write the value into the secondary part of the object. If the object is compressed and the value is not zero, we have to allocate the space for the secondary part of this object and then write the value into this part.

Compared to the original JVM implementation, our scheme incurs extra memory accesses in the following cases: (1) reading a level-2 field of an uncompressed object; (2) writing a value to a level-2 field of an uncompressed object; and (3) writing a non-zero value to a compressed object (this also involves allocating memory for the secondary part of the object being accessed). Since most level-2 fields contain zero, and most values written to level-2 fields are zero, these cases do not happen frequently. Therefore, we can expect the overall performance degradation due to our scheme to be low. Further, compared to the memory space allocated for each object in the original implementation of JVM, our scheme allocates two more words for each uncompressed object. This space overhead is amortized by the memory savings achieved by the compressed objects. That is, since most of the objects are compressed, our scheme reduces the overall heap memory requirement of an application.

Our scheme requires “putfield” and “getfield” be atomic operations. This incurs extra synchronization overheads for the JVM implementations where Java threads are mapped to native threads. Fortunately, most JVMs for embedded systems schedule Java threads by themselves without mapping Java threads to the native threads. For such JVMs, we can avoid the synchronization overheads by not preempting a Java thread when the thread is executing a “putfield” or “getfield” instruction. It should be noted that, most JVM implementations for embedded systems, such as KVM [6], schedule threads only between the boundaries of bytecode instructions, and thus never preempt a Java thread when the thread is executing an instruction.

### 7.3.3 Scheme-2: Sharing Level-1 Fields

The scheme explained in the previous subsection removes the level-2 fields from some objects to save memory space. In this section, we extend this scheme by sharing level-1 fields among multiple objects. Fig. 7.4 shows the object formats used in our level-1 field sharing scheme: uncompressed, compressed, and shared. Fig. 7.4(a) shows an uncompressed object.
This object has two parts: the primary part containing the level-0 fields, and the uncompressed secondary part containing both level-2 and level-1 fields. The first word in the primary part contains two one-bit flags ($U$ and $C$) and a pointer to the secondary part (SPtr). For an uncompressed object, we have $U = 1$ and $C = 1$, indicating that this object does not share the level-1 fields with any other objects, and that the secondary part of this object contains both level-1 and level-2 fields. It should be noted that the pointer SPtr points to the middle of the secondary part. The level-1 fields are stored in the locations with positive offsets, while the level-2 fields are stored in the locations with negative offsets. The level-2 fields of an object can be removed to save memory space if all these fields contain zeros. Fig. 7.4(b) depicts the format of a compressed object, i.e., an object with its level-2 fields removed. In this format, the secondary part of this object contains only the level-1 fields. Multiple compressed objects whose level-1 fields contain the same values can share the same secondary part (Fig. 7.4(c)) to reduce the overall memory space consumption.

The formats presented in Fig. 7.4 allow us to read and write the level-0 fields of an object in the same manner, irrespective of the current format of the object. Accessing a level-1 field of an object involves loading into a register the pointer SPtr and the flags $U$ and $C$ from the primary part. To read a level-1 field, there is no need to check the current format of the object. To write a value into a level-1 field, however, we need to check flag $U$. If $U = 1$, the secondary part of this object is not shared and we can write the value to the field. If $U = 0$, on the other hand, the secondary part of this object is shared with other objects and we have to create an unshared secondary part (compressed, containing only level-1 fields) for this object. To do this,
we allocate a memory block large enough to hold the level-1 fields of this object and copy the values of the level-1 fields to this block from the shared secondary part. After this, we write the value into the field in the newly-created secondary part. Note that, the pointer SPtr and the flag $U$ in the primary part of this object should also be updated.

To read a level-2 field of an object, we first check the state of flag $C$. If $C = 0$, this indicates that the value of this field is zero. If $C = 1$, however, we have to load the value from the secondary part of the object. To write a value to a level-2 field, we first check the state of $C$. If $C = 0$ and the value to be written is zero, we skip the write operation since it is unnecessary. If $C = 1$, we write the value to the field in the secondary part of the object. If $C = 0$ and the value to be written is non-zero, we have to create an uncompressed secondary part (containing both level-1 and level-2 fields) for this object. To do this, we allocate a memory block large enough to hold both level-1 and level-2 fields of this object, and then initialize this block by setting all the level-2 fields to zero and copying the values of the level-1 fields from the original secondary part of the object. Of course, the pointer SPtr and the flags $U$ and $C$ in the primary part of this object should be updated. After the uncompressed secondary part is created, we write the new value into the field specified by the instruction.
In our implementation, each object is created in the compressed format (Fig. 7.4(b)). During execution, it may be expanded into “uncompressed” format or further compressed into the “shared” format. A Mark-Sweep-Compact-Compress garbage collector is invoked when the free space in the heap is insufficient for creating a new object. This collector not only collects dead objects, but also compresses objects by eliminating level-2 fields. Specifically, when marking the primary part of an uncompressed live object during the mark phase, the collector also checks the values of the level-2 fields in the secondary part of this object. If all the level-2 fields of this object contain zero values, the collector splits the secondary part of this object into two blocks: the block containing the level-2 fields and the block containing the level-1 fields. The former block is not marked so that it can be swept in the following sweep phase. The latter block is marked as live since it is the compressed (but not shared) secondary part of a live object.

If the Mark-Sweep-Compact-Compress garbage collector cannot collect sufficient space for the new object, we scan the heap using an additional pass to find the compressed objects that can share their secondary parts. In our approach, only the objects of the same class can share a compressed secondary part with each other. To identify the objects that can share their compressed secondary parts, we maintain \( n \) frequent value pointers \( (p_i, i = 1, 2, ..., n) \) for each class. Further, each frequent value pointer \( p_i \) is associated with a counter \( (c_i) \). A frequent value pointer \( p_i \) of class \( C_x \) either is null or points to the secondary part of an object of class \( C_x \). We scan the heap, and, for each object \( O \) of class \( C_x \) in the heap, we compare its secondary part field-by-field against each secondary part that is pointed by a frequent value pointer of class \( C_x \). The counters associated with the frequent value pointers that point to a secondary part not identical to that of \( O \) are decreased by one, and, if counter \( c_i \) is less than a threshold \( N \), we set the corresponding frequent value pointer \( p_i \) to null. On the other hand, if there is a frequent value pointer \( p_i \) pointing to a secondary part that is identical to that of \( O \), we increase \( c_i \) by one, and let \( O \) share the secondary part pointed to by \( p_i \). If we cannot find any matches for \( O \) and there is a frequent value pointer \( p_i \) whose value is null, we let \( p_i \) point to the secondary part of \( O \), and initialize the counter associated with \( p_i \) to zero. In our experiments, we assumed that \( n = 3 \) and \( N = -3 \).
7.4 Experimental Results

In this section, we present the results from our experimental evaluation. Our presentation is in two parts. First, in Section 7.4.1, we present our baseline results. Then, in Section 7.4.2, we conduct a sensitivity analysis by modifying the parameters/strategies used in the baseline experiments. As mentioned in Section 7.2.1, we use a trace-based simulator to evaluate the memory behavior of a JVM. Our simulator maintains a heap and allocates objects in this heap as a JVM does. It invokes garbage collector/compressor when the heap space is used up. It also reads and writes the contents of object fields as captured by the trace file. Therefore, the heap memory access behavior of our simulator is very close to that of JVM.

7.4.1 Baseline Results

We present the maximum heap occupancy results in Fig. 7.5. Recall that the heap occupancy is the sum of the sizes of all the live objects at any given moment, and the maximum heap occupancy gives the minimum heap size needed to run the application without giving an out-of-memory exception.\(^2\) The y-axis in this figure represents the values normalized with respect to the maximum heap occupancy of the original JVM without any object compression. All space overheads incurred by each scheme are included in these results. User-level optimization reflects the memory saving potential of the static analysis based space optimization scheme. This scheme can reduce the sizes of some class files by removing the fields that are not used by the application. Consequently, the memory space for storing the loaded classes can be reduced. Further, if these classes are instantiated, the size of each instance can also be reduced. It should be noted that, in order to ensure the correctness of the optimized program, the static analysis based optimizations must be conservative. In Fig. 7.5, we observe that user-level optimization, Scheme-1, and Scheme-2 reduce the space for storing object instances by 7%, 26%, and 38% on average, respectively. Although we only present the characterization of the heap occupancy

\(^2\) Depending on the garbage collection algorithm used, a JVM may need larger heap memory than the maximum heap occupancy to execute a Java application without an out-of-memory exception. However, no JVM can execute a Java application without out-of-memory exception when the heap size is smaller than the maximum heap occupancy of the application. Since our schemes compress objects during garbage collection, the frequency of garbage collection invocations affects the value of the maximum heap occupancy. To find the lower bound of maximum heap occupancy for each scheme, we execute each benchmark many times with different heap sizes. The minimum heap size that allows the benchmark to execute without out-of-memory exception gives the maximum heap occupancy values presented in Fig. 7.5.
of object instances and arrays, we can still conclude based on these results that our schemes can be applied to the applications that have already been statically optimized using user-level optimizations to further reduce heap memory requirements. When we consider both arrays and instances, we observe that there is a drop in savings as compared to just instance size reduction. This is because of the large size of the arrays involved in some of these applications. In particular, none of the approaches achieve any significant savings in the benchmark compress, which is dominated by a few large arrays. Still, the average maximum heap occupancy savings achieved by the user-level optimization, Scheme-1, and Scheme-2 are 2%, 7%, and 14%, respectively.

While the savings in maximum heap occupancy are important, there are also cases where the average heap occupancy can be critical to consider. For example, this could provide more opportunities for energy savings in a multi-banked memory based system [43], by increasing the number of memory banks that can be turned off at a given time. Therefore, it is also important to consider the heap usage profile over the course of execution. Fig. 7.6 gives this profile for two representative benchmark codes, jess and raytrace, with the original JVM and the JVM with our Scheme-2. In obtaining these results, each scheme was executed using the minimum size heap with which it could complete execution. We observe that the JVM with Scheme-2 consistently

![Graph showing normalized maximum heap occupancy](image)

Fig. 7.5. Maximum heap occupancy. The y-axis represent the values, normalized with respect to the original maximum heap occupancy without any object compression.
Fig. 7.6. Heap occupancy profiles for two benchmarks with s1 input: jess and raytrace.

utilizes a smaller heap space as compared to the original JVM. The bar-chart in Fig. 7.7 shows the normalized average heap occupancy for all the benchmarks when using Scheme-2. We see that the average heap occupancy saving is about 10%, even when considering both object instances and arrays. That is, the proposed scheme reduces both maximum heap occupancy and average heap occupancy.

Because our experiments are based on simulation, it is not possible for us to obtain 100% accurate information on the execution time overheads incurred by our schemes. However, we estimate the performance overheads of our schemes by counting the number of the executed instructions and the number of memory accesses for performing the extra operations that are due to our schemes. For example, in Scheme-1, to access a level-2 field of an object, we need three extra operations: loading flag $C$ and pointer SPtr into a register, executing a conditional branch based on the value of flag $C$, and loading the value from the secondary part of the object (if $C = 1$) or returning a zero (if $C = 0$). We assume that each instruction incurred by our schemes is executed in one cycle and each extra memory access requires an extra cycle. Fig. 7.8 shows estimated performance overheads incurred by Scheme-1 and Scheme-2. The y-axis in this bar-chart gives the numbers of extra execution cycles introduced by our schemes, which are normalized with respect to the baseline execution cycles shown in the sixth column of Table 7.1.
Fig. 7.7. Average heap occupancy for Scheme-2. The y-axis are normalized with respect to the original average heap occupancy without any object compression.

(i.e., the execution cycles obtained by running the benchmarks using JDK 1.4 with Hotspot engine client version). Each bar in this figure is broken down into four parts. The first part (denoted COMPRESS) gives the time spent in compressing objects (such as checking if each level-2 field contains zero, and finding the objects that can share the same secondary part in Scheme-2), the second part (denoted EXPAND) gives the time spent in expanding objects (such as allocating memory for the secondary part of the compressed objects). The last two parts capture the extra overheads due to putfield and getfield operations. It should be noted that, the extra execution cycles for COMPRESS and EXPAND are affected by the heap size – the larger the heap size, the less frequent compressions and decompressions. To estimate the maximum overheads, we simulate each benchmark with the minimum heap size that allows the benchmark to complete its execution with the specific compression scheme. From these results, we observe that the extra execution cycles due to Scheme-1 is marginal (less than 3.7% for all the benchmarks), and the numbers of the extra execution cycles introduced by Scheme-2 are slightly greater than those of Scheme-1 for most of the applications. For benchmarks raytrace and mtrt, however, Scheme-2 incurs a performance overhead of about 8.6%. This is mainly due to the fact that Scheme-2 compares the level-1 fields of a large number of objects to find the objects that can
Fig. 7.8. Percentage increases due to Scheme-1 (S1) and Scheme-2 (S2) in execution cycles over the base results. Each scheme is simulated with the minimum heap size that allows the benchmark to complete its execution.

share level-2 fields (Fig. 7.1 confirms this observation). If we assume that each extra memory access due to our schemes costs two cycles, the average overheads for Scheme-1 and Scheme-2 are 1.8% and 4.2%, respectively. This estimation may not be accurate for high performance systems with deep-pipelined processor core and multiple-level caches. However, for low-end embedded environments at which our heap compression techniques are targeting, counting the number of memory accesses and instructions can give a reliable estimation of the performance overheads incurred by our schemes with a reasonable accuracy. For example, a widely used processor for today’s mobile phones is ARM7TDMI [3], which has a 3-stage pipeline, and no cache. The maximum frequency of this processor is 100MHz at 0.13um technology (133MHz at 0.13um). Note that the length of a cycle at this frequency is close to the access delay of typical SDRAM today. Further, for 3-stage pipeline, the branch penalty would normally be small. As a result, estimating the performance impact by counting the number of instructions and memory operations is expected to be reasonably accurate in practice.
7.4.2 Sensitivity Analysis

In this subsection, we vary some of the parameters and strategies used for obtaining the baseline results. The objective is to test the robustness of the proposed approach.

7.4.2.1 Impact of the Field Selection Scheme

Recall that our default method for classifying the fields of class $C_i$ is to select $F^* \subseteq F$ (where $F$ is the set of the fields of class $C_i$) such that the value of:

$$|F^*| \min_{f_j \in F^*} q(i, j)$$

is maximized. Let us refer to this method as the “minimum-based” scheme. In this subsection, we experiment with a “product-based” field classification scheme, in which the fields of class $C_i$ are classified by maximizing:

$$|F^*| \prod_{f_j \in F^*} \frac{q(i, j)}{N}, \text{ where } N = \sum_{f_j \in F} q(i, j).$$

The minimum-based selection scheme is based on the optimistic assumption that the most frequent values of the different fields tend to co-exist in the same object. In other words, we assume that, for class $C_i$, the set of objects where $f_1 = v_1$ is the subset of the set of objects where $f_2 = v_2$ if $q(i, 1) \leq q(i, 2)$, where $v_1$ and $v_2$ are the most frequent values for $f_1$ and $f_2$, respectively. In comparison, the product-based scheme is based on the assumption that each field assumes its most frequent value independently from the other fields of the same object. Fig. 7.9 presents the percentage increase in the maximum heap occupancy due to the product-based Scheme-2 over the heap occupancy of the minimum-based Scheme-2. One can observe that, for most of the benchmarks, the product-based scheme increases the maximum heap occupancy. This indicates that the most frequent values of the different fields do co-exist in the same object. Therefore, these results suggest that our default method seems to work better. In fact, although not presented here in detail, we also found during our experiments that the minimum-based scheme incurs less performance overhead than the product-based scheme.
7.4.2.2 Robustness of the Profiling-Based Approach

In Section 7.3, we mentioned that, while the values themselves may change from one input set (execution) to another, the fields with frequent values do not change significantly. To demonstrate this, we run our benchmarks with the s10 input; the level of each field, however, is determined using the profile information obtained from s1 input. Fig. 7.10 presents the maximum heap occupancy of each benchmark using Scheme-2. On an average, with s10, we achieve 35% reduction in the maximum heap occupancy for object instances. With arrays included, we still achieve 8% reduction in the maximum heap occupancy on the average. Recall that the corresponding values with s1 were 38% and 14%. Therefore, we can conclude that our profiling-based approach performs well across the different input sets.

7.5 Concluding Remarks

The proposed schemes take advantage of the frequent field value locality, which says that the fields of multiple objects hold the same value for a large fraction of their lifetimes. Our first scheme focuses on eliminating the space allocated for holding zeroes. Our second scheme
Fig. 7.10. Maximum heap occupancy of Scheme-2 with the s10 input. The y-axis represent the values, normalized with respect to the maximum heap occupancy of the original applications without any object compression. The level of each field is determined using the profile information obtained from the s1 input.

enhances the first one by letting multiple object instances share the same copy of the fields that contain frequent values. In addition, we also quantified the benefits that could come from a pure user-level compression strategy. The performance overhead imposed by these schemes is below 2% for most of the cases.
Field Level Analysis for Heap Space Optimization

8.1 Introduction

Drag period of an object is defined as the time interval from the last use of the object until its collection [115]. The idea behind drag time analysis and optimization is to deallocate memory space allocated to an object as soon as the object enters its drag period, i.e., reaches its last use. This can give us the potential space savings over the underlying garbage collector. In lazy object allocation, the memory space allocation is delayed until the object is actually written. In object compression, live Java objects are compressed when they are predicted to be not accessed in the near future. A common characteristic of these space-saving techniques is that they operate at an object granularity. That is, the granularity of space deallocation is an object. While these prior studies have been shown to be effective in reducing memory footprint of heap objects, one can potentially do better in terms of memory space saving by operating at a field granularity. This chapter explores this approach, and investigates two closely related issues:

- What is the potential memory space savings if we are able to manage memory allocations/deallocations at a field granularity instead of an object granularity? Can we achieve better savings than prior techniques such as drag time analysis and optimization?

- What are the difficulties associated with field-level memory space management as far as implementation is concerned? How can field-level profiling tools help the application programmer optimize the application code?

This chapter discusses these two issues with the help of two field-level profiling techniques. The first of these, called the field-level lifetime analysis, takes advantage of the observation that, for a given object instance, not all the fields have the same lifetime. More specifically, some fields start to get accessed much later than the others, and similarly, some fields reach their
last uses much earlier than the others. The field-level lifetime analysis demonstrates the potential benefits of exploiting this information. Our second analysis, referred to as the disjointness analysis, is built upon the fact that, for a given object, some fields have disjoint lifetimes, and therefore, they can potentially share the same memory space. Both these techniques analyze the execution traces of the applications to identify potential points in the application code where the programmer can revise the code to reduce memory consumption. For these two techniques, we both illustrate potential benefits (space savings) and describe implementation options based on current virtual machine support. We also point out what characteristics need to be considered by potential implementations.

8.2 Experimental Setup

To study the field level object access characterization, we instrument Kaffe VM [12] to track each object allocation and field access. Specifically, we record object ID, size and time for each allocation, and, object ID, access type (read or write), field offset and time for each field access. To identify when an object becomes unreachable, we invoke garbage collect upon each memory allocation request. Since in this study we measure the time using the accumulated allocated bytes, invoking garbage collection upon each allocation request tells us the exact time when each object becomes unreachable. All the information we gather is stored in a trace file.
Our trace-based simulator analyzes the trace file, and evaluates potential benefits of field level analysis and optimization.

Our results are based on SpecJVM98 benchmark suite (S1 input size). We omit the benchmark “compress” because it is not a typical object-oriented program in the sense that it mainly operates with a few large arrays rather than object instances. The important characteristics of the benchmarks used in this study are given in Table 7.1.

8.3 Field Access Characterization

8.3.1 Field Lifetime Analysis

A Java object instance may contain multiple fields. For a given object instance, not all the fields have the same lifetime. In this section, we study this issue in detail.

8.3.1.1 Life Cycle of Java Instances

Current implementations of JVM manage the heap memory space at the granularity of an object (instance or array). That is, an object is the minimum unit for memory allocation and deallocation. Fig. 8.1 depicts the life cycle of a Java object with five fields in detail. The life cycle of an object begins when the object is allocated. The period between the allocation and the last access is the lifetime of the object. Following the lifetime is the drag time. An object that is beyond its last access but is still reachable from the root set [72, 127] is said to be in its drag time [115]. The contents of an object in the drag time are no longer needed by the program, but the object cannot be collected by the garbage collector since it is not “officially” dead. When the last reference to an object is removed, the object becomes unreachable. An unreachable object is collected in the next invocation of garbage collection.

An instance of a Java class may have multiple fields. Fig. 8.1 also shows the life cycle of the fields. The lifetime of a field is the period between the time points when the application issues the first and last accesses to this field. If the first access is a write access (e.g., o.f1), the field is explicitly initialized with the given new value; if the first access is a read access (e.g., o.f3), the field is implicitly initialized with zero (or null for a reference field). As far as correctness is concerned, the program needs the value of a field only during the lifetime of
the field. For a typical Java program, different fields of the same instance may have different lifetimes, as illustrated in Fig. 8.1. Typically, the lifetime of a field is shorter than that of the containing instance, i.e., the instance that contains this field. The lifetime of a field usually begins later than the time when the containing instance is created and ends earlier than the time when the containing instance is last accessed. If the JVM could manage the memory space at the granularity of a field instead of an object (instance or array) and allocate the space for each field only during the field’s lifetime, one can achieve reduction in the heap memory occupancy of embedded Java applications.

8.3.1.2 Idle Field Ratio of Java Instances

Given an instance \( o \), we define the idle field ratio of \( o \) (denoted as \( r(o) \)) as follows:

\[
r(o) = 1 - \frac{\sum_{i=1}^{n} l_i}{nL_o},
\]

where \( n \) is the number of fields in \( o \); \( l_i \) is the length of the lifetime of the \( i^{th} \) field of \( o \); and, \( L_o \) is the length of the lifetime of object \( o \). Obviously, if most instances created by an application have very low idle field ratios, managing the heap space at the field granularity is not likely to bring much benefits for that application. On the other hand, a high idle field ratio for most instances indicates that managing the heap space at the field granularity instead of the object granularity might save significant amount of heap space. Fig. 8.2 shows the Cumulative Distribution Functions (CDFs) for idle field ratios for our benchmarks. A point \((x\%, y\%)\) on the curve of a benchmark means that \( y\% \) of the instances (in terms of allocated bytes) have an idle field ratios higher than or equal to \( x\% \). We observe from this figure that, for all our benchmarks, a significant portion \((35\% \sim 65\%)\) of the instances have idle field ratio higher than 80\%, which indicates that managing the heap space based on lifetime of fields could significantly reduce the heap memory requirements.

8.3.1.3 Potential of Lifetime-Aware Space Optimization

We carried out a set of experiments to investigate the potential memory reductions that could be obtained from a field-granularity memory management policies. Specifically, we experimented with these three schemes:
Fig. 8.2. CDFs for idle field ratio. A point \((x\%, y\%)\) on the curve of a benchmark means that \(y\%\) of the instances (in terms of allocated bytes) have an idle field ratio higher than or equal to \(x\%\).
**Field Last Use (FLU)** assumes that JVM is able to perfectly identify the last access to each field of each instance and collect every last used field immediately after the field is last used. The scheme allocates each instance in the same manner as the original JVM does. The fields that are never accessed remain in the heap until the containing instances become unreachable.

**Field First Use (FFU)** allocates the space for each field of each instance upon the first access to the field. The instances are collected when they become unreachable as in the original JVM. The fields that are never accessed are not allocated.

**Field First-Last Use (FFL)** combines FLU and FFU, allocating each field upon first access and collecting each field immediately after last access. This scheme indicates the potential of field granularity heap management.

Fig. 8.3 shows the heap occupancies of our Java benchmarks using the three field level heap management schemes discussed above. As mentioned earlier, “heap occupancy” corresponds to the total size of the live objects in the heap. Note that the maximum value of heap occupancy indicates the minimum size of heap that allows the application to run without out-of-memory exception. The average value of heap occupancy over the execution time of the application, however, indicates the opportunities to save leakage energy by turning off unused memory banks in a banked memory [43] architecture. We observe from this figure that field lifetime aware heap management schemes reduce the heap occupancy over the execution of each benchmark. Figures 8.4(a) and (b) summarize respectively the maximum and the average (over time) heap occupancies achieved by the schemes described above. All the results are normalized with respect to the maximum and the average heap occupancy of the original JVM without any heap space optimization. Note that since our optimizations are focused on the instances, the allocation and deallocation of arrays are not changed. In Fig. 8.4, we observe that, on an average, FLU could reduce maximum and average heap occupancies of object instances by 63% and 64%, respectively; and FFU could reduce maximum and average heap occupancies of object instances by 23% and 26%, respectively. Note that, in FFU, the space for each field is allocated upon the first access to that field. No space is allocated for the fields that are never accessed.
Fig. 8.3. Heap occupancies for object instances. The space occupied by arrays is not included. “Original” corresponds to the original (unoptimized) heap management scheme.
There are static analysis tools, such as JAX [121, 9], that can perform memory space optimization by eliminating the fields that are not needed by the application. Our FFU profiling data gives the upper bound of heap memory savings that could be provided by a static-analysis-based field elimination optimization. Further, an ideal heap memory management system based on FFL scheme could reduce maximum and average heap occupancies of object instances by 75% and 90%, respectively. Even with arrays included, this scheme could still reduce maximum and average heap occupancies by 37% and 28% on average, respectively. These results clearly demonstrate the potential of field level heap management in saving memory space.

To study the advantage of managing heap space at a field granularity over at an object granularity, we compare the maximum and average heap occupancies of object instances when using FFL against an ideal object granularity heap manage scheme (denoted as OFL), which allocates the space for each instance upon the first access to that instance, and collects each instance immediately after that instance is last used. Fig. 8.5 compares the maximum and average heap occupancies of using FFL against that of using OFL. Each bar in these graphs represents the maximum (or average) heap occupancy of object instances when using FFL, normalized with respect to that of OFL. From this figure, we observe that, on average, FFL reduces both the maximum and average heap occupancies of OFL by about 34%.

### 8.3.1.4 Determining the Last Access Time

Implementing an ideal memory management system is a challenging task. One of the difficulties is how to identify when a field becomes last used. A possible solution is using a static code analyzer to identify the instructions that issue last accesses and insert special instructions to release the fields that have been last accessed. To guarantee correctness of the optimized code, however, such a static analyzer must be conservative. However, due to the dynamic features of Java programming language such as virtual method invocation and dynamic class loading, a conservative static analyzer may miss a lot of optimization opportunities. Another solution is to use a trace-based analyzer that could provide useful hints to the application programmer, and let the programmer rewrite the source code to improve the efficiency of heap memory utilization. Manually rewriting the code increases the burden on the programmer; however, it allows us to catch interesting optimization opportunities based on field level analysis. Section 8.4 discusses
(a) The maximum heap occupancy, normalized with respect to the JVM with original (unoptimized) memory management. The bars denoted as “average” represent the average values over all the benchmarks.

(b) The average heap occupancy, normalized with respect to the JVM with original (unoptimized) memory management. The bars denoted as “average” represent the average values over all the benchmarks.

Fig. 8.4. Normalized maximum and average heap occupancy when using object/field lifetime analysis. All the optimizations are performed on instances; the allocation and deallocation of arrays are not modified.
Fig. 8.5. Comparing the heap occupancy of FFL against OFL. Each bar represents the maximum (or average) heap occupancy of object instances using FFL, normalized with respect to that of OFL.

detailed examples on how one can rewrite the code to reduce heap memory occupancy based on field level analysis.

To study when a field becomes last used, we carried out another set of experiments and recorded the number of accesses during the lifetime of each field. Fig. 8.6 shows the corresponding CDF curves. A point (x, y%) on the curve of each benchmark means that y% of the fields become last used no later than x accesses. For most of the benchmarks, we observe that around 90% of the fields are accessed no more than 20 times during their lifetimes.

8.3.1.5 Allocation Site Based Analysis

The instances that are allocated at different allocation sites may have different access patterns, and thus have different field lifetime behavior. To investigate the memory reduction potential for the instances allocated at each allocation site, let us first define a metric called $p(o)$, “the idle space-time product for instance $o$” as:

$$p(o) = \sum_{i=1}^{n} (L_o - l_i)b = r(o)B_oL_o,$$

where $n$ is the number of fields of $o$; $L_o$ is the lifetime of instance $o$; $l_i$ is the lifetime of the $i^{th}$ field of $o$; $r(o)$ is the idle field ratio for $o$; $b$ is the size of each field; and $B_o$ is the total size of $o$.
Fig. 8.6. Percentage of last used fields as a function of the number of accesses. A point \((x, y\%)\) on the curve of a benchmark indicates that \(y\%\) of the fields become last used no later than \(x\) accesses.
(i.e., $B_o = nb$). Thus, $p(o)$ is determined by three factors: the idle field ratio, size and lifetime of instance $o$. It represents the potential for reducing heap occupancy of the instances $o$. For allocation site $s$, we define $P(s)$, called “the idle space-time product for allocation site $s$”, as follows:

$$P(s) = \sum_{o \in A_s} p(o),$$

where $A_s$ is the set of the object instances that are allocated at $s$. Obviously, $P(s)$ is determined by both the number of the instances and the potential for reducing heap occupancy of each instance allocated at site $s$.

Fig. 8.7 plots the idle space-time products for allocation sites for each benchmark. The x-axis represents the allocation sites, which are sorted in the descending order of their idle space-time products, and the y-axis is the idle space-time product of each allocation site. We see from this figure that, for all the benchmarks, there is a small number (around 10) of allocation sites that account for most of the potential space savings. This fact indicates that we may need to focus only on a small number of the allocation sites when performing field lifetime-aware heap space optimization. This is especially important when the optimizations are performed manually by the application programmer.

### 8.3.2 Disjointness Analysis

For a given object, some fields can have disjoint lifetimes, and therefore, they may be able to share the same memory space. For example, the lifetimes of fields $f_0$, $f_1$ and $f_2$ in Fig. 8.1 are disjoint from each other. The purpose of the disjointness analysis is to reveal the potential heap space savings when the fields with disjoint lifetimes share the same memory location.

#### 8.3.2.1 Using Unions in Java

Union is a data structure that allows multiple fields that are not needed simultaneously to share the same memory location [22]. Overlapping multiple fields in the same memory location reduces the memory occupancy of the program; however, it may also lead to a safety problem.
Fig. 8.7. Idle space-time product for allocation sites. X-axis: the allocation sites that are sorted in the descending order of their idle space-time products. Y-axis: the idle space-time product for each allocation site in $10^6$ Byte$^2$ (The unit of space-time product is Byte$^2$ since both the space and time are measured in bytes in this study).
Specifically, by unionizing an integer and a pointer, the programmer can easily create a pointer pointing to an arbitrary address. This is one of the major reasons why Java programming language does not support unions [76, 50]. Another potential reason for Java not supporting unions is that, if not used properly, unions can make program hard to understand, maintain, and debug. Our field level profiling results, however, show that using unions may reduce the maximum heap occupancy of our programs by up to 18%. This profiling data thus brings up an important question: can we enjoy the memory savings of the union while still keeping Java safe and easy to program? In other words, is it possible to unionize object fields in a programmer-transparent way? One possible solution is to allow the programmers to use unions in Java source code as in C/C++ while letting the Java source compiler generate conventional bytecode that does not use union. Section 8.4 presents an example (based on the source code of raytrace) showing how we manually create regular Java classes (i.e., $U$ and $C_i$’s) to implement a union structure. Note that, we can also let a compiler do the similar work automatically.

### 8.3.2.2 Memory Saving Potential of Using Unions

Fig. 8.8 presents an example that explains how we profile programs for the disjointness analysis. Since the execution trace of a benchmark may vary from one input set to another, our trace-based profiling technique does not guarantee the correctness of the program as conservative static analysis techniques do. However, our profiling results give the upper bound for the heap space savings that static analysis techniques can achieve. More importantly, our profiling technique provides useful hints that can help the programmer optimize the application code. Fig. 8.8(a) shows the CFG (control flow graph) of a Java code fragment. In this CFG, we have three allocation sites and two accessing sites. The accessing sites access the instances allocated by the allocation sites. Let us assume that the allocation site $s$ allocates instances of class $C$ as follows. Each instance of class $C$ has fields $f_1, f_2, \ldots, f_n$. For allocation site $s$, we build an interference graph $G(s) = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ and $v_i$ corresponds to $f_i$ ($i = 1, 2, \ldots, n$). Edge $(v_i, v_j) \in E$ if and only if $i \neq j$ and there exists an object $o$ allocated by $s$ such that $o.f_i$ interferes with $o.f_j$. Interference between fields $o.f_i$ and $o.f_j$ is defined as follows. Let us assume that, at time $t_1$, we write a value to $o.f_i$ and, at time $t_2 (> t_1)$, we read the value from $o.f_i$. Assume further that during period $(t_1, t_2)$, we do not perform any write
to $o.f_i$. Similarly, at times $t_3$ and $t_4$ (where $t_4 > t_3$), we write and read the value of $o.f_j$, respectively, and no write to $o.f_j$ occurs during $(t_3, t_4)$. Then, we say that $o.f_i$ interferes with $o.f_j$ if and only if $(t_1, t_2) \cap (t_3, t_4) \neq \emptyset$, i.e., the two periods intersect with each other.

By tracking each access to each object allocated by allocation site $s$, we can construct the interference graph $G(s) = (V, E)$. The nodes of $G(s)$ are then colored using the minimum number of colors such that $v_i$ and $v_j$ are assigned different colors if $(v_i, v_j) \in E$. Note that, for the instances allocated by $s$, fields whose corresponding nodes in $V$ have the same color can be unionized to share the same memory location. Fig. 8.8(b) shows the formats of the unions that are determined by coloring the interference graphs.

In some programs, a given class $C$ may have multiple allocation sites and these allocation sites may have different interference graphs. As a result, the fields of instances of class $C$ that are allocated by different allocation sites may be unionized in different ways. Therefore, when an instance ($o$) of class $C$ is accessed by site $a$, the JVM needs to check which allocation site allocated $o$ and how this allocation site unionizes the fields. Based on this, it can determine the offset of the field that is being accessed. These checks can introduce too much runtime overhead. In the following discussion, we describe a possible solution to eliminate this overhead.

Let us assume class $C$ has allocation sites $s_1, s_2, \ldots, s_k$. We construct a graph $G(C) = (V_C, E_C)$ for class $C$ as follows. Each node ($v_i$) in $V_C$ corresponds to an allocation site ($s_i$). $(v_i, v_j) \in E_C$ if and only if there exist two instances, $x$ and $y$, that are created by $s_i$ and $s_j$, respectively, and there exists an instruction (accessing site) that accesses both $x$ and $y$ (see Fig. 8.8(c) for an example). For each maximally connected component\(^2\) of $G(C)$, we construct a minimum global interference graph $H$ such that the interference graph of each node in this connected component is a subgraph of $H$. After that, for the allocation sites whose corresponding nodes are in this connected component, we use $H$ instead of their own interference graphs to determine the unionization of the fields. Note that, with this scheme, we guarantee that the fields of the instances that may be accessed by the same accessing site have the same union format (see Fig. 8.8(d)).

Fig. 8.9 gives the optimized maximum and average heap occupancies when using unions. We observe that, with arrays included, using unions reduces both maximum and average heap

\(^2\)Given a graph $G = (V, E)$ and its subgraph $S = (V', E')$ is a maximally connected component if $S$ is connected; and for all nodes $u$ such that $u \in V$ and $u \notin V'$, there is no node $v \in V'$ for which $(u, v) \in E$. 

occupancy by up to 15% and 10%, respectively. The average savings (including arrays) are 8% and 6% for maximum and average heap occupancies, respectively. Merging the interference graphs of the allocation sites that allocate instances that may be accessed by the same accessing site increases the size of the union instances. However, for most of the benchmarks we tested, the impact of these merges was not significant.

8.3.2.3 Allocation Site Based Analysis

A Java program usually has hundreds of allocation sites. The potential for heap space savings through unions, however, may vary from one allocation site to another. The first reason for this is that different allocation sites may allocate different number of object instances. The second reason is that the access patterns of the object instances that are created by different allocation sites may differ from each other significantly. Consequently, for different allocation sites, we may unionize the fields in different ways and, thus, achieve different heap space savings for each instance.

Fig. 8.10 shows the potential for reducing the allocated memory size for each allocation site when using merged unionization. The y-axis represents the reduction in the total size of the allocated memory at each allocation site, which is calculated by multiplying the total number of the instances allocated at the allocation site and the reduction in the size of each instance when unionizing fields that do not interfere with each other. The x-axis represents the allocation sites, which are sorted in the descending order of their size reductions when using unions. We observe from this figure that only a small number (less than 20) of allocation sites account for most of the reduction in the allocated memories. This observation indicates that we only need to focus on a small number of allocation sites when applying union-based memory space optimization.

8.4 Realizing Memory Savings

So far, we have studied the characteristics of SpecJVM98 benchmarks as far as field-level analysis is concerned, and discussed potential for heap memory space savings. However, exploiting this potential is a challenging task. In the following, we discuss some components that are necessary when performing field lifetime aware heap space optimizations.

\[\text{Footnote 3: Only the first 50 allocation sites with maximum potential savings for each benchmark are presented in the figure.}\]
(a) The control flow graph of a code segment. $s_1$, $s_2$, and $s_3$ are the allocation sites for class $C$, and $a_1$ and $a_2$ are the accessing sites that access field $f_3$ of the instances of class $C$.

(b) The interference graphs and the union formats for $s_1$, $s_2$, and $s_3$. Since $f_3$ has different offsets in different union formats, $a_1$ needs to determine the offset of $f_3$ at each access.

(c) Each node in $G(C)$ corresponds to an allocation site. $s_1$ and $s_2$ are adjacent to each other since $a_1$ accesses the instances allocated by both $s_1$ and $s_2$. Note that this $G(C)$ has two maximally connected components: \{s_1, s_2\} and \{s_3\}.

(d) The interference graphs of $s_1$ and $s_2$ are merged so that the unions allocated by $s_1$ and $s_2$ have the same format.

Fig. 8.8. Determining the union format for allocation sites.
Fig. 8.9. Normalized maximum and average heap occupancies when using unions. Unmerged: without merging the interference graphs of different allocation sites. Merged: merging the interference graphs of the allocation sites that allocate instances that may be accessed by the same accessing site.
Fig. 8.10. Reduction in allocated memory size when using merged unionization at the first 50 allocation sites. X-axis: allocation sites sorted in descending order of size reduction. Y-axis: reduction in the total size of the allocated memory at each allocation site, which is calculated using the formula: \((\text{original instance size} - \text{unionized instance size}) \times \text{number of allocated instances at the allocation site}\).
• A mechanism to dynamically expand/shrink the format of object instances. Using such a mechanism, we can lazy-allocate the space for each field when it is first accessed, and promptly eliminate the fields when they are last used.

• An analysis strategy to identify points in the code where a particular field of a set of instances are last accessed. Based on this analysis, we may be able to develop a static analysis tool that can modify the application code to expand/shrink object instances. Or, at least, such a mechanism would guide the programmers to systematically revise their code. Profiling results presented in this chapter may help us narrow down the scope of analysis to a few allocation sites.

• A mechanism to enable safe employment of field unions. A possible solution is associating each group of unionized fields with a flag indicating the type of value that is currently held in the unionized field. This flag is transparent to the programmer and is automatically updated when a value is written to the field. Any field access that violates the type rule (e.g., loading a pointer value from a unionized field that currently holds an integer value) would cause the JVM to throw a type violation exception [93].

• An analysis technique to identify the disjointness of fields of the same instance. Our profiling results can identify the potential allocation sites and the possible union format for each allocation site. However, the profiling results are affected by the input of the application. To guarantee the correctness of the program, we need a programmer or an automatic static analyzer to verify the profiling results and modify the code.

To demonstrate the actual memory reduction achievable by applying field lifetime/disjointness aware space optimizations, we focused on two benchmarks, jess and raytrace, as examples. Specifically, by profiling, we identified the allocation sites that create large number of potential instances, and then re-wrote the source codes of the applications to reduce heap memory occupancy of the application by taking advantage of field lifetime/disjointness aware space optimization.

Jess is an example of the set of applications that can benefit significantly from field unionization. For S1 input size, we find that this benchmark allocates 1151 instances of class spec.benchmarks\_202\_jess.jess.Value. This class is used as a container for a value of arbitrary
type. Each instance of this class contains one and only one value. However, since Java does not support union, the definition of this class has to include three fields of different types. Consequently, the fields “floatval” and “Objectval” are not used in an instance containing an integer value, the fields “intval” and “Objectval” are not used in an instance containing a float value, and the fields “intval” and “floatval” are not used in an instance containing a reference value. The most straightforward way to eliminate this memory waste is using unions so that the fields “intval”, “floatval” and “Objectval” can share the same memory location. However, since the union construct is not supported in Java, we can address this problem by defining a set of sub-classes of Value, each of which corresponds to a type of value. Each of the original constructors of class Value is now replaced with a static “makeValue” method, which takes the same parameters as the original constructor, and creates and returns an instance of the appropriate subclass. Fig. 8.11 shows how we modify the code to reduce heap memory requirement of this benchmark. By rewriting the source code, we reduced the maximum and average (over the time) heap occupancies of object instances by 15% and 13%, respectively. Fig. 8.13(a) compares the heap occupancies of the object instances of the original and optimized versions of jess.

Raytrace is another benchmark, whose heap memory requirement can be reduced by source code rewriting. Through profiling, we find that this benchmark creates 3582 instances of class spec.benchmarks._205_raytrace.OctNode. Each instance of this class has five fields (see Fig. 8.12). In each instance, the lifetimes of these five fields differ from each other significantly. In the instances where “Child” fields have long lifetimes, we observe that the lifetimes of “ObjList” and “NumObj” fields are very short. Meanwhile, in the instances where ‘ObjList” and “NumObj” fields have long lifetimes, the lifetimes of “Child” fields are very short. By analyzing the application’s source code, we make two observations. First, each instance of this class represents a region of three dimensional space. Each instance is in one of the following states: (1) containing eight sub-regions (instances of OctNode and each instance represents a smaller region of three dimensional space), (2) containing a set of three dimensional objects, and (3) containing nothing. For state (1), the subregions are stored in the field “Child”; the fields “ObjList” and “NumObj” are not used. For state (2), the objects are stored in “ObjList”; “NumObj” stores the number of the objects; “Child” is not used. For state (3), none of the fields “Child”, “ObjList” and “NumObj” is used. Second, the field “OctFaces” refers to an array with eight elements, only
//Original definition of class Value.
public class Value {
    int type;
    int intval;
    double floatval;
    Object Objectval;
    public Value(int v) {
        type = INTEGER;
        intval = v;
    }
    public Value(double v) {
        type = FLOAT;
        floatval = v;
    }
    public Value(Object v) {
        type = OBJECT;
        Objectval = v;
    }
}

//Optimized definition of class Value.
public class Value {
    int type;
    public static class IntValue extends Value {
        int intval;
        IntValue(int v) {
            type = INTEGER;
            intval = v;
        }
    }
    public static class FloatValue extends Value {
        int floatval;
        FloatValue(double v) {
            type = FLOAT;
            floatval = v;
        }
    }
    public static class ObjectValue extends Value {
        int Objectval;
        ObjectValue(Object v) {
            type = OBJECT;
            Objectval = v;
        }
    }
    public static Value makeValue(int v) {
        return new IntValue(v);
    }
    public static Value makeValue(double v) {
        return new FloatValue(v);
    }
    public static Value makeValue(Object v) {
        return new ObjectValue(v);
    }
}

Fig. 8.11. Modifying the source code of jess to reduce heap memory requirements.
two of which are actually used by the program. Based on these observations, we modify the source code of this benchmark to eliminate unnecessary objects and fields. Fig. 8.12 shows how we remove the redundant fields of class OctNode. Specifically, we define three classes, OctNode.Child, OctNode.ObjList and OctNode, for states (1), (2) and (3) respectively. Note that state (2) and (3) cannot be distinguished until the instance is actually created and initialized. We address this problem by using the “shrink” method to dynamically shrink “OctNode.ObjList” instances into “OctNode” instances when the programmer knows that the fields “ObjList” and “NumObj” are no longer needed. By rewriting the source code, we reduced the maximum and average heap occupancies of object instances by 43% and 39%, respectively. Fig. 8.13(b) compares the heap occupancies of the object instances for the original and optimized versions of raytrace.

8.5 Related Work

Embedded virtual machines are being increasingly used in many embedded and mobile environments, and commercial implementations [84, 5, 23] have now been around for some time. In addition to these, McDowell et al. [96] presented a Java environment that supports the complete Java language and all the core Java packages except AWT using as little as 1MB of RAM. TinyVM [18] is an open source Java platform for the Lego Mindstorms RCX microcontroller. TinyVM’s footprint is about 10KB in the RCX. Shaylor [118] implemented a Java JIT compiler for memory-constrained low-power devices. Their implementation requires 60KB of the ARM machine code. An example non-Java small-footprint virtual machine is Maté [89], a tiny communication-centric virtual machine designed for sensor networks. While memory footprint reduction has been one of the objectives of many of these efforts, to our knowledge, none of them has considered compressing heap objects by exploiting the frequent field values.

Many embedded systems employ data/code compression to reduce memory space requirements, power consumption, and the overall cost of the system. In the domain of code compression, instruction compression has been an active area of research for the last decade or so [87, 88]. Clausen et al. [47] proposed compressing bytecodes by factoring out common sequences. A different line of work [104] tried to reduce the space occupied by bytecode sequences using a new format to represent files. Debray and Evans [52] used profiling information
public class OctNode {  public class OctNode {
  OctNode[]   Adjacent;
  Face[]     OctFaces;
  OctNode[]   Child;
  ObjNode    ObjList;
  int        NumObj;
  public OctNode(...) {
    // construct the instance
  }
}

public static class OctNode.Child extends OctNode {
  OctNode[]   Child;
  OctNode.Child(...) {
    // construct the OctNode instance that
    // contains eight children (subregions).
  }
}

public static class OctNode.ObjList extends OctNode {
  ObjNode ObjList;
  int NumObj;
  OctNode.ObjList(...) {
    // construct the OctNode instance that
    // contains a set of 3-D objects.
  }
  
  /**
   * Method shrink() eliminates the fields ObjList
   * and NumObj from an OctNode.ObjList instance
   * that actually contains no objects.
   */
  OctNode shrink() {
    OctNode node = new OctNode();
    node.Adjacent = this.Adjacent;
    node.OctFaces = this.OctFaces;
    return node;
  }
}

public static OctNode makeOctNode(...) {
  if(this instance contains eight children) {
    return new OctNode.Child(...)  
  } else {
    if(node.NumObj==0) {
      // this instance contains nothing
      return node.shrink();
    } else
      return node;
  }
}

Fig. 8.12. Modifying the source code of raytrace to reduce heap memory requirements.
to guide code compression. The main focus of the work in [62] is to reduce leakage energy consumption by turning off memory banks saved by compressing class libraries. In addition to these studies, memory compression has also been adopted in high-end systems. For example, Rizzo [110] presented a fast algorithm for RAM compression. Similarly, Franaszek et al. [60] developed a set of algorithms and data structures for compressed-memory machines, effectively doubling the available memory capacity. The study discussed in this chapter is different from these prior studies in that it targets reducing heap space by exploiting frequent field values in Java objects.

In [40], Chen et al. proposed heap compression techniques to reduce the size of the heap memory. Their compression scheme treats each object as a structureless byte stream. Therefore, a compressed object must be decompressed before its contents can be accessed. The work described in this chapter is different from that in [40] in that our new object compression schemes are aware of the structures of objects, which allows the contents of a compressed object to be accessed without decompressing the entire object.

Marinov and O’Callahan [94] proposed Object Equality Profiling (OEP) for helping programmers discover optimization opportunities in programs. Based on profiling of objects, they partition the objects into equality sets. The objects of the same equality set can be replaced with a single representative object by rewriting the program. However, there are several limitations to applying this optimization. For example, to merge multiple objects into the representative one,
their technique requires that the objects should not be mutated, and that the program should not perform any operation that depends on the object’s identity. In addition, their approach is meant to be applied by the programmer. Our work differs from [94] in three main aspects. First, we do not have the limitations mentioned above. Second, our schemes are meant to be used within the virtual machine in a programmer-transparent fashion. Third, we can reduce the space in cases where there are some fields with the same value, but no objects are equal to each other.

Tip et al. [121] presented an application extraction tool, JAX, that reduces the size of class files, as well as memory footprint of Java programs by removal of redundant methods and fields, transformation of the class hierarchy, and renaming of packages, classes, methods, and fields. Ananian et al. [28] presented a set of techniques for reducing memory consumption of Java programs. Their optimizations include field reduction, unread and constant field elimination, static specialization, field externalization, class pointer compressions, and byte Packing. Except for field externalization, all these optimizations are compiler-based and are similar to the optimizations performed by JAX. Our schemes can be applied to the programs that are already optimized by these optimizations (except for field externalization). Field externalization uses profiling to find the fields that almost always have the same default value, and removes these fields from their enclosing class. A hash table stores the values of these fields that differ from the default value. Write accesses to these field are replaced with an insertion into the hash table (if the written value is not the default value) or a removal from the hash table (if the written value is the default value). Read accesses, on the other hand, are replaced with hash table lookups; if the object is not present in the hash table, the lookup simply returns the default value. Our scheme-1 achieves similar memory occupancy reduction effects of field externalization; however, as compared their hash-table-based approach, our scheme incurs less performance overhead. Bacon et al. [32] proposed an object model for Java programming language that can instantiate most Java objects with only a single-word object header. This model uses heuristic compression techniques to retain lock state, hash code, and garbage collection flags of the relevant objects. The model retains two-word headers, containing thin lock state, for objects that have synchronized methods. Compared to an object model with a two-word header, their mostly single-word headers saves heap space by 7% on an average.
Rojeno and Runciman [111] presented heap profiling of functional programs. Their research shows that a significant portion of the heap space is occupied by the drag or void objects. Shaham et al. [116] studied the effectiveness of garbage collection algorithms by measuring the time difference between the actual collection time of an object and the potential earliest collection time, i.e., the last use time, for that object. This time difference is called the drag time, which indicates potential space savings. Their purpose is to develop static analysis techniques that could be used together with garbage collector to enable earlier reclamation of objects. Their results can also be used to pinpoint application source code that could be rewritten in a way that allow more timely garbage collection. In [115], they present a heap-profiling tool that measures the difference between the actual collection time and the potential earliest collection time of objects for a Java application. Based on the output of this tool, they rewrote the source code of their benchmarks to save heap space. In [68], Hirzel et al. presented their experimental studies showing that the liveness can have significant impact on the ability of a garbage collector or memory leak detector to identify dead objects. However, to achieve large benefits, the liveness scheme should incorporate interprocedural analysis of global variables. Our work is different from the previous work in that we focus on the lifetime of individual fields instead of an entire object. Our profiling information can guide the programmer to redesign the formats of relevant classes to save heap space. Hertz et al. [67] presented an algorithm that can precisely trace the lifetime of the objects in the heap without frequently invoking expensive garbage collection to identify the dead time of each object.

8.6 Concluding Remarks

In many embedded systems, tight limitations are usually put on memory resources and CPU processing power. Consequently, effective use of the system’s resources presents major design challenges. One of these limited resources is memory, and performance (and in some cases, correctness) of many applications depends partly on how they utilize memory space. While the garbage collection mechanism of Java collects dead objects from time to time, reducing the memory space occupied by live objects can bring further memory savings. Motivated by this, this chapter studies two related issues:
• What are the potential benefits of managing the heap memory space at a field granularity instead of an object granularity?

• What are the challenges waiting for an implementation that targets field-level heap management?

We present the result of two analyses, namely, field-level lifetime analysis and disjointness analysis. The results of our study indicate that a significant potential for memory reduction exists. We also describe how two applications can be modified to realize some of these potential savings.
Part III

Reliability and Security
Chapter 9

Transient Error Tolerance

9.1 Introduction

Many non-trivial Java applications, such as web browsers and email processors, are popular on the small devices such as mobile phones and PDAs. In the near future, one can even see mission-critical applications (e.g., on-line transaction processing) running on mobile-phone-like devices. Following this trend, fault-tolerance will be an important issue for Java-enabled embedded environments.

Transient hardware faults in modern microprocessors are caused by several reasons. A certain class of transient hardware faults are caused by several reasons, cosmic rays to which we are exposed everyday [26]. The glitches in the voltage of power supply may also cause transient faults. As the feature size and the voltage of integrated circuits keep scaling down, the future microprocessors are becoming increasingly susceptible to transient hardware faults. Many solutions have been proposed for improving the reliability of computer systems against these faults. For example, dual-execution techniques enable us detect the transient errors in the datapath [63, 107]. However, such solutions require special hardware support, and thus, increase the overall cost of the system. Embedded systems are usually sold in huge quantities and thus tend to be more sensitive to the per device cost as compared to their high-performance counterparts. Consequently, the existing fault-tolerance solutions for high-end systems may not be attractive for low-cost embedded systems. Further, an embedded system may run a set of applications and not all of them may require fault-tolerance. Employing expensive hardware for just a few applications that need fault-tolerance may not be the best economic option.

Intuitively, a transient fault tolerant JVM can be designed as follows. First, we use two bytecode execution engine instances ($E_1$ and $E_2$) to execute the same copy of an application, and compare their states (including the state of each object in the heap\(^1\), and the state of each register)

\(^1\)The stacks of Java threads are allocated in the heap memory.
of the CPU) at certain points of the execution. A mismatch in these two states normally indicates an error. Second, from time to time, we take checkpoints by copying the current states of $E_1$ and $E_2$ to a reserved memory area so that we can roll back to the state of the previous checkpoint in case an error is detected. However, this approach has two major drawbacks. First, we need memory space to store three copies of the execution state, two copies for the two execution engine instances and one copy for the checkpoint. Consequently, compared to a JVM without transient fault tolerance, this approach increases the memory requirement of a Java application by 200%, which makes it unsuitable for memory-constrained systems. Second, comparing the states of each corresponding pair of Java objects incurs a heavy performance overhead.

This chapter focuses on embedded JVM as a case study and develop a scheme that allows $E_1$, $E_2$, and the checkpoint to share a significant portion of the objects in the heap. Our approach has the following advantages, which make it very attractive for a low-budget embedded environment. (1) By sharing objects, the memory overhead due to fault-tolerance mechanism is significantly reduced. (2) Performance overhead due to comparing the states of $E_1$ and $E_2$ is reduced since there is no need to compare the states of the objects that are shared by both the execution engine instances. (3) Our scheme does not require any special hardware support, except for the memory page protection mechanism, which is currently supported by the MMU (Memory Manage Unit) of many embedded processors. (4) Our scheme is flexible in the sense that $E_1$ and $E_2$ can be scheduled either dynamically or statically. In the dynamic scheduling approach, each execution engine instance is implemented as a system thread that is scheduled by the operating system. $E_1$ and $E_2$ synchronize with each other using the synchronization APIs provided by the operating system. This approach is suitable for multiprocessor environments where two execution engine instances can run in parallel to achieve high performance. On the other hand, the static scheduling approach uses a single system thread to run both the execution engine instances. In this case, the programmer of the JVM inserts scheduling points in the code of the execution engines to explicitly switch contexts between $E_1$ and $E_2$. This scheme is suitable for the single-processor environments or for the environments where the operating system does not support multi-threaded execution. When using the static scheduling, the synchronization overhead is reduced since, at any time, there is only one thread running.
The two execution engine instances \((E_1 \text{ and } E_2)\) synchronize with each other at checkpoints to compare their states. Our approach can detect any type of transient error that causes the states of the two execution engine instances differ from each other. To recover from the errors by rolling back, we assume that the memory system (including MMU and the main memory) is reliable, which can be achieved by using well-known error detection and correction codes.

9.2 Implementation

9.2.1 Object Model

Fig. 9.1 shows the object model for our transient fault tolerant JVM. In this model, each object has a two-word header, containing the size and ID of the object. The ID is also be used as the default hash code for the object [64]. Each object is referenced by an entry of the object table, which is indexed by the object ID. The reference fields of the objects contain the IDs of the target objects, instead of direct pointers. A drawback of such indirect object references is that we need an extra dereferencing when accessing the contents of an object, and that the object table incurs some space overhead. The main advantage is that the objects can be moved freely without updating the reference fields in other objects, and this is why we adopt this object representation in our implementation.

In our implementation, almost all the data structures, including the user created objects, the code of the Java applications, the execution stacks of the Java threads, and the other data structures internal to the implementation of the JVM, are allocated in the heap. Many objects are accessed very frequently. Accessing these objects using the indirect reference each time incurs performance/power overheads. To address this problem, we employ pre-dereference optimization to avoid the indirect reference overheads for the most frequently accessed data structures. For example, the execution stacks of Java threads are the most frequently accessed data structures in the heap. Each execution engine instance maintains a pointer to the execution stack of the thread that is currently being executed. To avoid the cost for indirect reference at each access to the execution stack, whenever a thread is scheduled, we load the actual address of the execution stack of the scheduled thread into a register, i.e., we pre-dereference the indirect pointer to the execution stack. When the memory block for the execution stack of the scheduled thread is
moved during garbage collection, we need to update the register that contains the address of the current execution stack. Similarly, we pre-dereference the pointer to the memory block that contains the bytecode of the method that is currently being executed to avoid the indirect reference overhead in fetching bytecode instructions.

9.2.2 Object Sharing

Our approach reduces the memory space overhead by allowing the two execution engine instances and the checkpoint to share as many objects as possible. In this section, we explain this sharing mechanism. Fig. 9.2 depicts the architecture for our transient fault tolerant JVM. Our JVM maintains a heap that is physically divided into pages. Each page can be either in the read-only mode or writable (read/write) mode. Writing to a read-only page triggers a memory protection exception. Each execution engine instance has its own object table. Each entry in each object table contains a pointer to an object in the heap. The heap is logically divided into three subheaps: a global subheap \( H_g \) and two local subheaps \( H_1 \) for \( E_1 \) and \( H_2 \) for \( E_2 \). The size of each subheap can change during execution. The boundary between two adjacent subheaps aligns with the page boundaries. The objects in \( H_g \) may be shared by both the execution engine instances, i.e., a pair of entries in the two object tables indexed by the same object ID may refer to the same object in \( H_g \). The pages in \( H_g \) are read only during the execution time. During global garbage collection (which will be discussed in Section 9.2.4.2) time, however, a page of
$H_g$ can be put in writable mode. During the execution, writing to an object in a $H_g$ triggers a memory protection exception. The exception handler copies the object being accessed to the free area of $H_i$ ($i = 1, 2$), assuming that it is $E_i$ that causes the exception. The corresponding entry of $E_i$’s object table is also updated so that the future accesses to this object made by $E_i$ are redirected to the new copy of this object (in $H_i$). The state of the object before the write access, however, is preserved in $H_g$. Since $H_g$ is read-only, this state cannot be modified by transient errors in the processor core. Consequently, when any error is detected, we are guaranteed to be able to roll back to the correct state. Fig. 9.3 illustrates the procedure of writing to an object $O$ in $H_g$. Note that the JVM for embedded devices can be integrated to the kernel of the embedded operating system. As a result, the overhead for exception handling is much smaller than that in high-end operating systems. The objects in $H_i$ are moved to $H_g$ during the global garbage collection.

Transient errors may also cause the JVM to mistakenly put an $H_g$ page into writable mode. Our scheme can not completely prevent the failure of execution due to such errors. However, we can significantly reduce such risk by using “page mode checking” exception. Specifically, a “page mode checking” exception is triggered whenever an execution engine instance tries to place a read-only page into the writable mode. The exception handler checks if the JVM is performing a global garbage collection. Only during global garbage collections can a page of $H_g$ be put in writable mode. Any attempt to put a read-only page into writable mode in occasions other than global garbage collections indicates an error and causes the execution engine instance that triggers the “page mode checking” exception to roll back to the previous checkpoint.

### 9.2.3 Checkpointing and Rolling Back

Our transient fault tolerant JVM implicitly takes checkpoints at two occasions: (1) right after the global garbage collection, and (2) at the invocation and return points of a non-dual-executable method. The following subsections will discuss the details of taking the checkpoints.

To roll back to the previous checkpoint, we first set each entry in the object tables of both the execution engine instances to NULL. Then, we scan the objects in $H_g$ in the order of their allocation times, and use the address and the ID field of each scanned object to set up the corresponding entry in the object table. Note that the order used in this scanning is critical, i.e., if an object $O_1$
is created before $O_2$, $O_1$ must be scanned earlier than $O_2$. This is due to the possibility that an object might be duplicated multiple times, and only the latest copy contains the correct state of the object before the previous checkpoint. By scanning the objects in the order of their creation times, we guarantee that the object table entry contains the pointer to the newest version of each object. Finally, both the execution engine instances load the processor state that was stored at the previous checkpoint and resume execution.

9.2.4 Garbage Collection

JVM relies on garbage collector to manage heap memory. Letting some object to be shared by two execution engine instances requires modifications to how garbage collection is carried out. Specifically, our transient fault tolerant JVM uses two different garbage collectors: a local collector operating on local subheaps, and a global collector operating on the entire heap. When execution engine instance $E_i$ ($i = 1, 2$) uses up the memory in $H_i$, it invokes the local garbage collector to reclaim the space occupied by the garbage objects (i.e., the objects that are unreachable from the root set) in $H_i$. If the local garbage collector cannot find sufficient space for the new object, the global garbage collector will be invoked to collect garbage objects in the entire heap ($H_1$, $H_2$, and $H_g$).
(a) Initially, the entries of both object tables point to object $O$ in $H_g$.

(b) Upon $E_1$ writing to the field of $O$, $O$ is copied into $H_1$.

(c) Upon $E_2$ writing to the field of $O$, $O$ is copied into $H_2$.

Fig. 9.3. Writing to an object in $H_g$. 
9.2.4.1 Local Collector

Each execution engine instance invokes the local collector to collect its local subheap when it uses up the space in its local subheap. The two execution engine instances do not necessarily invoke local collectors simultaneously, and, during the garbage collection process, they do not synchronize with each other. The local collection is performed in two phases: a mark phase and a compact phase. In the remaining part of this section, we describe our local collector in detail. We assume that we are collecting subheap $H_i$ ($i = 1$ or 2).

Mark phase: In this phase, the collector for $E_i$ traverses the reference graph to mark the objects that are reachable from the root set along the pointers in the reference fields of each object. The root set is the set of objects that are known to be live when the garbage collector is invoked. For local collection, the root set includes the following objects:

- (1) The objects that were copied from $H_g$ by the memory protection exception handler;
- (2) The stack frames of the live threads in $H_i$;
- (3) The instances of class “Class” in $H_i$; and
- (4) Other JVM implementation-specific data structures in $H_i$.

Compact phase: In this phase, we slide the live objects that were marked in the mark phase to one end of subheap $H_i$. Consequently, we obtain a contiguous free area in the other end of $H_i$. When an object is moved, the corresponding pointer in the object table entry (see Section 9.2.1) should also be updated accordingly. Since there is no need to update the reference fields of the objects, the implementation of the compact phase is straightforward.

Errors happening during local garbage collection will cause the states of the two execution engine instances to diverge, and this divergence will be detected at the next checkpoint. Since the local garbage collector does not change the contents of the objects in $H_g$ (which are in the read only mode), it cannot damage the state of the previous checkpoint. Therefore, we can recover from the errors that happen during the local garbage collection by rolling back to the state of the previous checkpoint.

---

2 A reference graph is a directed graph, each node of which corresponds an object in the heap, and each edge of which corresponds to a reference field of the source object.

3 It is very unlikely that the same errors would occur in two local collections.
9.2.4.2 Global Collector

The global collector collects the garbage in the entire heap. Since global garbage collection affects the states of both the execution engine instances, it requires $E_1$ and $E_2$ to synchronize with each other. When designing the global garbage collector, there are two major challenges. The first one is how to detect errors during global garbage collection. Note that we have only one copy of $H_g$, and thus it is impossible to detect errors by comparing the states of two global collector instances. The second challenge is how to recover from the errors that occur during the global garbage collection. Since the global collector changes the contents $H_g$, it damages the state of the previous checkpoint. A straightforward solution is to copy the entire $H_g$ to a safe place. When an error happens, we can roll back to the previous state by copying back the state of $H_g$. However, this solution is not attractive for a memory-constrained embedded system due to its heavy memory overhead. The rest of this section presents our solution to these challenges.

Our solution requires the global garbage collector follow the steps below:

**Step 1.** Each execution engine instance ($E_i$) independently traverses the reference graph in both $H_g$ and its local subheap ($H_i$) to mark the objects reachable from the root set. We use the least significant bit of each object table entry as the mark flag. Note that, since the objects are aligned to word (4 bytes) boundaries, this bit is not used by a reference. When marking objects, we also compute the CRC checksum of the contents of all the live objects using a table-based algorithm [126].

**Step 2.** $E_1$ and $E_2$ synchronize with each other to compare the checksums computed in step 1. If the two checksums do not match, both $E_1$ and $E_2$ have to roll back to the previous checkpoint because errors might have occurred since the previous checkpoint. Otherwise, we continue with step 3.

**Step 3.** Reaching this step indicates that $E_1$ and $E_2$ have the same live objects. Now, we are going to compact live objects. Note that, at this point, the contents of $H_1$ and $H_2$ are identical. We combine $H_g$ and $H_1$ into a “collection area” and set all the pages in this area to read-only mode. We also make the object table of $E_1$ (where the live objects are marked) readable to both $E_1$ and $E_2$. From now on, both $E_1$ and $E_2$ work on the collection area and the object table of $E_1$. $H_2$ and the object table of $E_2$ are not used.
in the following steps. Setting the collection area and the object table to read-only is to prevent any error during garbage collection from damaging the state of the heap.

**Step 4.** We use cursor $S_i$ ($i = 1, 2$) to indicate the object that is currently being processed by $E_i$. $S_i$ is initialized to the beginning of the collection area. We use pointer $P$ to indicate the target page into which the live objects will be compacted. $P$ is initialized to 0, i.e., to point to the first page of the heap.

**Step 5.** We use a buffer page to temporarily store a set of live objects (Fig. 9.4(a)). The size of the buffer page is the same as the size of the pages of the heap. $E_1$ uses $S_1$ to scan the heap and copies the scanned live objects into the buffer page until the buffer page is full. $E_2$ uses $S_2$ to scan the heap and computes the checksum ($C_1$) of the scanned live objects until the total size of the scanned objects reaches the size of the buffer page. An object with ID $x$ is considered to be live if and only if the $x^{th}$ entry of the object table of $E_1$ is marked and points to the address of this object.

**Step 6.** $E_1$ sets the buffer page to read-only mode, computes the checksum ($C_2$) of the objects in this page, and then compares it against $C_1$ (Fig. 9.4(b)). If $C_1 \neq C_2$, we rewind $S_1$ and $S_2$ to their previous positions and then go to step 5.

**Step 7.** $E_1$ sets the mode of page $P$ to writable, copies the objects in the buffer page into this page, and then sets the mode of page $P$ back to read only. After setting page $P$ to read only, $E_1$ computes the checksum ($C_3$) of the objects in this page (Fig. 9.4(c)). If $C_3 \neq C_2$, there might be errors during copying. Therefore, we repeat step 7 until we have $C_3 = C_2$.

**Step 8.** We increase $P$ by one (i.e., move to the next page). If all the objects in page $P$ have been scanned by $S_1$ and $S_2$ (i.e., the end address of page $P < S1 = S2$), we continue with step 9; otherwise, we jump back to step 5.

**Step 9.** Reaching this step indicates that we have collected enough free space so that the gap between the compacted area and the uncompacted area is larger than a page. Consequently, in the following steps, we are able to compact the live objects to the target page $P$ directly, without using the buffer page. $E_1$ sets the mode of page $P$ to writable and then uses $S_1$ to scan and copy the live objects to page $P$ until $P$ is full. $E_2$ uses $S_2$ to scan the heap and
computes the checksum ($C_1$) of the scanned live objects until the total size of the scanned objects reaches the size of page $P$ (Fig. 9.4(d)).

**Step 10.** $E_1$ sets the mode of page $P$ to read-only, and then computes the checksum ($C_2$) of the objects in this page (Fig. 9.4(e)). If $C_1 \neq C_2$, we rewind $S_1$ and $S_2$ to the previous positions, and then go back to step 9.

**Step 11.** We increase $P$ by one (i.e., move to the next page). If $S_1$ and $S_2$ have not reached the end of the collected area, go to step 9 (Figures 9.4(f) and (g)).

**Step 12.** Mark page$_0$ through page$_{P-1}$ as $H_g$. The rest of the pages in the heap are equally distributed between $H_1$ and $H_2$. Make the pages in $H_1$ and $H_2$ writable to $E_1$ and $E_2$, respectively (Fig. 9.4(h)).

**Step 13.** Both $E_1$ and $E_2$ set the entries of their object tables to NULL, and then they scan $H_g$, using the addresses and the ID fields of the scanned objects to update the entries of their object tables.

**Step 14.** Both $E_1$ and $E_2$ resume execution of the application code.

From the above description, we observe that, during the global collection, no two pages can be in the writable mode simultaneously. This is to minimize the risk that a transient error damages the state of the heap. Writing to a read-only page during garbage collection indicates an error. Further, the pages in the heap are put in the writable mode one-by-one in the order of their addresses. This information can be exploited to enhance the reliability of the global collection. Specifically, when putting page$_i$ into the writable mode, the page mode checking exception handler checks if the page in the writable mode last time was page$_{i-1}$. If it is not page$_{i-1}$, we know that an error must have occurred.

### 9.2.5 Handling Non-Dual-Executable Methods

A non-dual-executable method is a method whose side effects or return value is sensitive to the time of invocation. Such a method may return a different value or affect the state of the application differently when invoked at different times. Executing a non-dual-executable
Fig. 9.4. Global garbage collection.
method on two execution engine instances may cause the states to diverge. To avoid such a
divergence, we execute such methods using only one execution engine instance and then copy
the return values and the objects that are created by the non-dual-executable methods into the
$H_g$ so that they can be accessed by the other execution engine instance as well. To invoke a
non-dual-executable method, we follow the following steps:

Step 1. Both $E_1$ and $E_2$ invoke a local collection and then compute the checksums of the live
objects in the their local subheaps.

Step 2. $E_1$ and $E_2$ synchronize with each other to compare the checksums computed in step 1.

The difference in the checksums indicates an error, and then both $E_1$ and $E_2$ must roll
back. If no error is detected, we promote the live objects in $H_1$ to $H_g$. Note that $H_1$ is
adjacent to $H_g$, therefore, a promotion can be performed by simply increasing the size
of $H_g$. After the promotion, the pages that do not belong to $H_g$ are equally distributed
between subheaps $H_1$ and $H_2$.

Step 3. At this point, both $H_1$ and $H_2$ are empty. $E_2$ pauses while $E_1$ continues with the
execution of the non-dual-executable method.

Step 4. $E_1$ returns from the method.

Step 5. $E_1$ invokes a local collection, promotes the live objects in $H_1$ into $H_g$, and distributes
the pages that do not belong to $H_g$ equally between $H_1$ and $H_2$.

Step 6. $E_2$ updates the references to the promoted objects in its object table.

Step 7. Both $E_1$ and $E_2$ resume execution.

In the above description, it should be noted that we invoke the local garbage collection
twice. The overall performance overhead due to non-dual-executable methods, however, is not
as significant as it seems to be. The first reason for this is that most of the non-dual-executable
methods are methods performing I/O operations, such as reading user input from the keyboard.
Compared to the long pauses incurred by these types of I/O operations, the additional overhead
due to two local collections is not expected to be very significant. The second reason is that
the non-dual-executable I/O methods usually create only a small number of objects (if any); and
consequently, the extra cost due to the second local collection tends to be very small. Further, these two local collections reclaim free space in the local heaps, and thus postpone the future invocation of garbage collection.

9.2.6 Scheduling Java Threads

A Java program can create multiple Java threads. In our transient fault tolerant JVM, all the Java threads are user level threads that are scheduled by the JVM; and a Java thread that is created by $E_i$ is scheduled only by $E_i$. Further, to prevent the state divergence, $E_1$ and $E_2$ schedule the Java threads in the identical order.

9.3 Experiments

9.3.1 Experimental Setup

We implemented our transient fault tolerant JVM based on KVM [84]. KVM is a JVM implementation for the embedded devices with limited memory budget. The total memory space required for loading KVM and its Java class library is about 350KB. The memory size for the heap, however is determined by the specific application being executed. The fourth column of Table 9.1 shows the minimum heap size that is required to run each of our benchmarks on the original KVM. To evaluate our approach, we execute our JVM in an embedded environment, which is simulated using the Shade [48] instruction level simulation toolkit.

Table 9.1 presents the information about the benchmarks that are used in our experiments. They represent a typical set of benchmarks that are executed in embedded Java environments. The third column in this table gives the total size of the objects that are allocated during the execution of each benchmark. The fourth column gives the maximum heap occupancy (i.e., the maximum value of the total size of the objects that are live simultaneously in the heap) for each benchmark. Finally, the last column is the maximum value of the number of objects that are live in the heap simultaneously. This value determines the minimum number of entries in the object table.
Table 9.1. The Java benchmarks for our experiments.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Description</th>
<th>Accumulated Allocation Size</th>
<th>Max. Heap Occupancy</th>
<th>Max. Number of Live Obj.</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td>Print “hello”</td>
<td>8KB</td>
<td>4KB</td>
<td>40</td>
</tr>
<tr>
<td>chess</td>
<td>Chess game</td>
<td>209,430KB</td>
<td>27KB</td>
<td>2194</td>
</tr>
<tr>
<td>crypto</td>
<td>Encryption package</td>
<td>1,528KB</td>
<td>116KB</td>
<td>1181</td>
</tr>
<tr>
<td>db</td>
<td>Database management</td>
<td>101KB</td>
<td>43KB</td>
<td>196</td>
</tr>
<tr>
<td>edge</td>
<td>Edge detection</td>
<td>241KB</td>
<td>216KB</td>
<td>268</td>
</tr>
<tr>
<td>fft2d</td>
<td>2-D Fourier filter</td>
<td>94KB</td>
<td>81KB</td>
<td>40</td>
</tr>
<tr>
<td>xml</td>
<td>XML parser</td>
<td>143KB</td>
<td>65KB</td>
<td>1026</td>
</tr>
</tbody>
</table>

**9.3.2 Experimental Results**

Our dual-execution based error detection and checkpoint based error recovery mechanism incurs both memory space and performance overheads. In this section, we quantify these overheads.

Table 9.2 gives the memory overheads incurred by our transient fault tolerance mechanism. The memory page size used in the experiments is 2KB. The second column in this table gives the original maximum heap occupancy for each benchmark, which is also shown in Table 9.1. The third and fourth columns give the absolute and normalized (with respect to the original maximum heap occupancy) values of the minimum heap size that is required by our fault tolerance JVM to execute each benchmark without an “out-of-memory” exception. In Table 9.2, we observe that, for the benchmark hello, an application whose original memory requirement is small, the memory overhead incurred by the transient fault tolerant mechanism is large (100%). This is because the size of each subheap must be rounded up to the minimum multiplier of the page size. Note that, the page size used (2KB) is large, compared to the maximum heap occupancy of this benchmark. For the applications whose original memory requirements are large, the impact of the page round-up problem reduces. We also observe that the memory overhead of the benchmark edge is larger (62%) than most of the other benchmarks. This can be explained by observing that this benchmark allocates large arrays (about 64KB each) and that we need to
Table 9.2. Minimum heap space \((H_g + H_1 + H_2)\) requirement for our transient fault tolerant JVM. The values in the fourth column are normalized with respect to those in the second column. The memory page size is 2KB.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original Max.</th>
<th>Min. Heap Size with Fault Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Heap Occupancy</td>
<td>Size</td>
</tr>
<tr>
<td>hello</td>
<td>4KB</td>
<td>8KB</td>
</tr>
<tr>
<td>chess</td>
<td>27KB</td>
<td>32KB</td>
</tr>
<tr>
<td>crypto</td>
<td>116KB</td>
<td>132KB</td>
</tr>
<tr>
<td>db</td>
<td>43KB</td>
<td>48KB</td>
</tr>
<tr>
<td>edge</td>
<td>216KB</td>
<td>350KB</td>
</tr>
<tr>
<td>fft2d</td>
<td>81KB</td>
<td>86KB</td>
</tr>
<tr>
<td>xml</td>
<td>65KB</td>
<td>88KB</td>
</tr>
</tbody>
</table>

keep up to three copies of each array in three subheaps simultaneously when the array is write-accessed. For most of our benchmarks, however, the memory overheads are within 20%, which indicates that our object sharing technique reduces the memory overheads significantly.

Fig. 9.5 shows the breakdown of the execution times when no error occurs during the entire execution time. We experiment with both single-CPU and two-CPU based embedded environments. In the two-CPU environment, \(E_1\) and \(E_2\) execute in parallel; and in the single-CPU environment, \(E_1\) and \(E_2\) are scheduled on the same CPU. We conduct experiments with two different heap sizes: the minimum heap size that allows each benchmark to execute without an “out-of-memory” exception, which is shown in the fourth column of Table 9.1 (Fig. 9.5(a)); and the heap size that is four times of the minimum heap size (Fig. 9.5(b)). The execution time of each benchmark is normalized with respect to the overall execution time of the original KVM with the same heap size. Each bar is broken into five components as explained in the caption of this figure. In Fig. 9.5, we observe that, when using single CPU, our JVM increases the execution of each benchmark by more than 100%, compared to the original KVM. The major reason for this overhead is that we execute the application twice in the single CPU. In the two-CPU environment where we can execute \(E_1\) and \(E_2\) in parallel, the performance of our transient fault tolerant JVM gets closer to the original KVM (that uses single CPU). In Fig. 9.5(a), we also observe that, when using the minimum heap size, the garbage collection in the JVM can incur significant overheads (up to 53% in the two-CPU environment) since the collector has to
be invoked very frequently. By increasing the heap size to four times of the minimum heap size, the overhead due to garbage collection is reduced to a marginal extent. Besides the garbage collection, indirect object references via the object tables incur another important performance overhead. By using the pre-dereference optimization described in Section 9.2.1, we managed to reduce this overhead to less than 10% (3% on the average) of the overall execution time.

9.4 Related Work

Transient error detection and recovery has been an active research area for several years. Ray et al. [105], and Reinhardt and Mukherjee [107] proposed architectural schemes that use the spare functional units in a superscalar processor to detect and recover from the errors in the datapath. Gomaa et al. [63] used on-chip multiprocessors for transient error detection and recovery. These approaches require hardware support that is not commonly available in low-end embedded systems. Satyanarayanan et al. [113] proposed a lightweight recoverable virtual memory system
for Unix applications with persistent data structures that must be updated in a fault-tolerant manner. Caldwell and Rennels [35] present a transient fault tolerant scheme for embedded spacecraft computing. Their schemes use multiple processors to perform the same computing and let them vote for the correct results. Chen et al. [39] reported fault injection experiments that investigate memory error susceptibility of JVMs. Chen et al. [41] analyzed the heap error behavior in embedded JVM environments. As compared to these studies, our approach detects and recovers from transient errors using dual execution. An important characteristic of our approach is that it reduces memory overheads brought by dual execution and checkpointing.

9.5 Concluding Remarks

In this chapter, we present transient fault tolerant JVM for embedded environments. Our JVM use two execution engine instances to execute the bytecode of the same application and compare the states of the two execution engine instances to detect any transient errors in the datapath that can cause the states of the two execution engine instances to differ from each other. Our JVM recovers from transient errors by rolling back to the state of the last checkpoint. The two execution engine instances and the checkpoint share the objects in the heap to reduce the memory space overhead. Our experiments with seven embedded Java applications show that the average memory overhead due to our transient error tolerant mechanism is 35%, and the average performance overhead is not too much in a two-CPU environment.
Chapter 10

Verifiable Annotations for JIT Compilation

10.1 Introduction

While JIT (Just-in-Time) compilation can improve execution of a Java application significantly as compared to an interpretation-based execution, it can also introduce considerable overheads in terms of both compilation time and memory space requirements. In addition, dynamic compilation can be a very energy-consuming activity [42]. The overheads introduced by JIT compilation are particularly problematic in memory-constrained, battery-operated embedded execution environments.

The dynamic (JIT) compilers in Java Virtual Machines (JVMs) for embedded systems are usually light-weight. They translate bytecodes into native code by scanning them in one or two passes (e.g., [12]). They can only perform a limited set of optimizations due to the lack of the resources that are necessary for a detailed analysis of the code being compiled. One promising solution to improve the quality of the native code generated by such a light-weight dynamic compiler without incurring significant overheads is bytecode annotation. Specifically, one can employ offline analyzers to perform expensive analyses. The results of these analyses are incorporated into the Java class files as annotations. At runtime, the annotations guide the compiler to generate optimized code without performing expensive analyses.

There are two types of annotations. The first type of annotations cannot cause the dynamic compiler to generate code that violates the security policy of the JVM. For example, some annotations provide hints about whether a certain optimization is beneficial to the given program. Since Java bytecode (which is verified when the class file is loaded) is safe in nature and the optimizations performed by the dynamic compiler preserve the semantics of the program being compiled, the generated native code is guaranteed not to violate the security policy of the JVM. The second type of annotations, however, can cause the dynamic compiler to generate code that violates the security policy of the JVM. For example, certain annotations provide the
dynamic compiler with hints about whether to generate code for checking the safety of certain
operations at certain points of the program. JVM specification [93] requires the JVM check the
pre-conditions of some operations at runtime, e.g., before accessing an element of a Java array,
JVM must check if the index is within the array boundaries. Good annotations can help the
dynamic compiler generate optimized code by honestly telling the compiler which safety checks
are redundant and can thus be dropped. Bad annotations, however, can cheat the compiler into
omitting the safety check codes that are not redundant. This presents a safety problem if the
annotations in this category are not checked before being used by the compiler.

In this chapter, we focus on annotations of the second type. Specifically, we concentrates
on three annotation-based optimizations: null pointer check removal, value liveness analysis,
and array bound check removal. Since these annotations can change the semantics of Java byte-
code instructions, they must be verified at runtime. We present the required data flow analyses,
annotation generation, and annotation verification algorithms for these optimizations so that they
can be applied in a reliable manner. In addition, we present an experimental evaluation of the
proposed approach for null pointer check removal and array bound check removal. While this
chapter presents results for these particular optimizations, our approach is general and can ac-
commodate other optimizations as well.

Fig. 10.1 compares the approach proposed in this chapter with the conventional JIT com-
pilation. Fig. 10.1(a) depicts a conventional JIT compiler that compiles the Java bytecode that is
produced by the Java source file compiler (javac) into native code at runtime. Fig. 10.1(b) illus-
trates JIT compilation with our verifiable annotations. The annotator generates the annotations
by offline-analyzing the bytecodes of the application. The annotation verifier verifies these an-
notations at runtime. The annotations that pass the verification can safely guide the JIT compiler
to generate optimized native code. Note that, the verification step in (b) is very important. This
is because a harmful annotation, if not verified, can compromise both reliability and safety. As
compared to the optimizing analysis performed by a conventional dynamic compiler, verifying
annotation incurs much smaller performance and memory space overheads.
10.2 Data Flow Analysis and Verification for Java Program

10.2.1 Flow Analysis Algorithms

In this section, we give the notation used in this paper. We use $\mathbb{Z}$ to denote the set of integer numbers, $J$ to denote the instruction set of JVM, and $T$ to denote the set of all possible types of values that may be used in a given Java program, including the elementary types and classes. Given a vector $\vec{v}$, we use $\vec{v}[i]$ to denote the $i^{th}$ element of $\vec{v}$, and $|\vec{v}|$ to denote the length of $\vec{v}$.

A method $M$ is denoted as $(\vec{b}, \vec{a})$ where $\vec{b}$ is the vector of the instructions and $\vec{a}$ is the vector of the types of the arguments. We write “$i \xrightarrow{M} j$” if and only if the execution of the $j^{th}$ instruction ($\vec{b}[j]$) can immediately follow that of the $i^{th}$ instruction ($\vec{b}[i]$). Based on this, we define the following sets:

- $\text{pred}(M, i) = \{j \mid j \xrightarrow{M} i\}$,
- $\text{succ}(M, i) = \{j \mid i \xrightarrow{M} j\}$,
- $R(M) = \{i \mid \vec{b}[i] \text{ is a return instruction of } M\}$.

The JVM uses stack frames to hold the states of Java method invocations. When a thread invokes a method, the JVM pushes a new stack frame onto the thread’s stack. A stack frame has
three parts: local variables, operand stack, and frame data. The frame data includes the necessary information to support method return, exception dispatch, and constant pool resolution. In this research, we focus on the first two parts of the stack frame. The state of the frame (or the frame state) at a certain point of the program is determined by the states of the local variables and stack operands. Two frame states are equivalent if and only if the states of each pair of the corresponding local variables and stack operands in the two frame states are equivalent. In this work, unless otherwise stated, the “frame state of the instruction” means the “state of the frame at the entry of the instruction”.

The frame state of each instruction of a method $M$ can be statically calculated using data flow analysis [99]. In this work, we use $\mathcal{F}_i = (\vec{L}_i, \vec{S}_i)$ to denote the frame state of the $i^{th}$ instruction of method $M$, where $\vec{L}_i[j]$ represents the state of the $j^{th}$ local variable and $\vec{S}_i[k]$ represents the state of the $k^{th}$ operand in the operand stack. $|\vec{S}_i|$ gives the depth of the operand stack at the entry of this instruction. Assuming that $F$ is the set of all frame states, the data flow analysis requires a merge function (denoted as $\sqcup$) $F \times F \rightarrow F$ and a partial order relation (denoted as $\preceq$) on $F$ such that:

$$\forall \mathcal{F}_i, \mathcal{F}_j \in F : \mathcal{F}_i, \mathcal{F}_j \preceq \mathcal{F}_i \sqcup \mathcal{F}_j.$$  \hspace{1cm} (10.1)

Almost all Java instructions read or modify the contents of the local variables and stack operands. The semantic of each instruction can be encoded as a flow function $f : J \times F \rightarrow F^1$. Given the current frame state and the instruction to be executed, this function calculates the new frame state after the execution of the instruction.

A data flow analysis can be either forward or backward. The forward flow analysis (with flow function $f^+$) of a method $M$ can be formulated as: given the entry state $\mathcal{F}^e$, determine the frame state $\mathcal{F}_i$ of each instruction $\vec{b}[i]$ such that:

$$\mathcal{F}^e \preceq \mathcal{F}_0 \text{ and } \forall 0 \leq i < |\vec{b}| : \mathcal{F}_i = \bigcup_{j \in \text{pred}(i)} f^+ (\vec{b}[j], \mathcal{F}_j).$$

The backward flow analysis (with flow function $f^-$) of a method $M$, on the other hand, can be formulated as: given the exit state $\mathcal{B}^e$, determine the frame state $\mathcal{B}_i$ of each instruction $\vec{b}[i]$ such that:

---

$^1$More strictly, we should write “$f : J \times F \rightarrow F \times F$” since we may have different frame states for different outcomes of a branch instruction.
that:

\[ \forall i \in R(M) : B^* \preceq B_i \quad \text{and} \quad \forall 0 \leq i < |\bar{b}| : B_i = \bigsqcup_{j \in \text{succ}(i)} f^-(\bar{b}[i], B_j). \]

In the discussion so far, for clarity purposes, we have not talked about the exception handling mechanism of JVM. It should be noted that our framework can handle exceptions as well. When performing data flow analysis on a method in which Java exceptions might be thrown, the first instruction in the catch block should be considered as the immediate successor of every instruction in the try block that could throw the exception. That is, if the \( i \)th instruction of method \( M \) might throw an exception \( E \) and the first instruction in the catch block of \( E \) is \( j \), the relation \( i \xrightarrow{M} j \) holds; therefore, we have \( j \in \text{succ}(i) \) and \( i \in \text{pred}(j) \). For the backward data flow analysis, the set \( R(M) \) includes not only the return instructions, but also the “athrow” instructions that are not caught in \( M \). Subroutine instructions (“jsr” and “ret”) are often used in an exception handling code. However, since we mainly focus on the embedded Java environments and the Java applications that satisfy the CLDC [7] specification do not use the these instructions, we do not consider them further in this paper.

10.2.2 Verifying the Results of Flow Analysis

Given the frame state of each instruction, verifying the result of the forward data flow analysis requires checking whether the following constraint holds: \( F^* \preceq F_0 \quad \text{and} \quad \forall i \xrightarrow{M} j : f^+(\bar{b}[i], F_i) \preceq F_j \). Similarly, verifying the result of the backward data flow analysis requires checking whether the following constraint holds: \( \forall i \in R(M) : B^* \preceq B_i \quad \text{and} \quad \forall i \xrightarrow{M} j : f^-(\bar{b}[i], B_j) \preceq B_i \). Note that, the result of the flow analysis, if not modified, is guaranteed to pass the verification. Passing the verification does not guarantee that the frame states in the result are not modified. However, with a properly-defined flow function, passing the verification guarantees that the given frame states do not violate the constraints imposed by the semantics of Java instructions, i.e., the frame states do not cause the dynamic compiler to generate incorrect or insecure code.

Compared to calculating the frame states using data flow analysis, the cost for verifying the result of data flow analysis is usually much lower in terms of execution time, power consumption, and memory space requirements. When performing verification, we do not need to
evaluate merge functions or maintain work-lists. More importantly, when verifying the result of a data flow analysis, we do not need to iterate over the same instruction multiple times until a fix point is reached (as the data flow analysis does). Instead, we linearly scan the frame state of each instruction only once to check whether the state of the current instruction matches the states of its immediate successors. In the following sections, we discuss how we verify the results of three data flow analyses that are commonly used in optimizing JIT compilers.

10.3 Array Bound Check Removal

The JVM specification [93] requires the virtual machine throw an ArrayIndexOutOfBoundsException to indicate that an array has been accessed with an illegal index, i.e., the index is either negative, or greater than or equal to the size of the array. Checking whether the index is legal at every instruction where an array is accessed can introduce a significant performance overhead. Data flow analysis is one of the techniques that can remove redundant array bound checks. Our annotation keeps the result of the offline flow analysis to guide the dynamic compiler in removing the redundant array bound checks at runtime. Since maliciously-modified annotations may cause the dynamic compiler to remove some bounds checks that are not redundant, the annotations should be verified before being used by the dynamic compiler.

We start by giving an overview of our annotation based approach to array bound check removal. First, we represent the length of each array as an abstract (symbolic) value. Using flow analysis, we calculate a value range for each variable at the entry of each instruction, and store the result of this calculation as an annotation. At dynamic compilation time, for each instruction that accesses an array element, the dynamic compiler checks the range of the value of the index variable. For example, at the instruction that accesses $a[i]$, if the annotation shows that variable $i$ is in the range $[0, v - 1]$ (where $v$ is an abstract value), we can safely remove the lower bound check (i.e., $i \geq 0$). Further, if $v$ is the abstract value for the length of array $a$, we can safely remove the upper bound check (i.e., $i < a.length$) since $i$ is guaranteed to be less than the length of $a$. 
10.3.1 Definition of Frame States

We assume that \( V = \{ v_i \mid i = 0, 1, \ldots \} \) is the set of abstract values that represent the lengths of the arrays manipulated by the Java application. An exception is that the \( v_0 \) represents the integer constant 0. In addition, we use \( +\infty \) to represent a positive integer whose value is greater than \( G \), and \( -\infty \) to represent a negative integer whose value is less than \(-G\). Here, \( G \) is the largest integer that can be used in the annotation. In our experiments, we use three bits to represent integer values in the annotation; and thus, we have \( G = 7 \). Now, we can define the state of the frame at the entry of the \( k^{th} \) instruction as:

\[
F_k = \begin{bmatrix} a_1, a_2, \ldots, a_m; x_1, x_2, \ldots, x_n \end{bmatrix},
\]

where \( m \) is the number of local variables, \( n \) is the depth of the operand stack, \([a_i, b_i]\) represents the state of the \( i^{th} \) local variable, and \([x_i, y_i]\) represents the state of the \( i^{th} \) operand in the operand stack. \([a_i, b_i]\) can have one of the following values:

- \([v_a + z_1, v_b + z_2]\) \((v_a, v_b \in V \text{ and } z_1, z_2 \in \mathbb{Z})\): an integer equal to \( x \) or a reference to an array whose length is equal to \( x \) such that \( v_a + z_1 \leq x \leq v_b + z_2 \).
- \([-\infty, +\infty]\): a value that we do not care about.
- \([-\infty, v_b + z_2]\): an integer \( x \) such that \( x \leq v_b + z_2 \).
- \([v_a + z_1, +\infty]\): an integer \( x \) such that \( v_a + z_1 \leq x \).

Besides the values listed above, the state of a stack operand, \([x_i, y_i]\) may also have a value \([*, k]\), which indicates that this stack operand is equal to the \( k^{th} \) local variable.

The frame state implicitly expresses the relations between the local variables and the stack operands. For example, “the \( i^{th} \) local variable contains a reference to an array and the length of this array is stored in the \( j^{th} \) local variable” is expressed as “\([a_i, b_i] = [v_k, v_k]\) and \([a_j, b_j] = [v_k, v_k]\)”, that is:

\[
\begin{bmatrix} \ldots, a_{i-1}, v_k, a_{i+1}, \ldots, a_{j-1}, v_k, a_{j+1}, \ldots; \ldots \end{bmatrix},
\]

“The \( i^{th} \) local variable contains an integer whose value is larger than 0 and smaller than the length of the array whose reference is stored in the \( j^{th} \) local variable” is expressed as “\([a_i, b_i] = \ldots; \ldots \]
\[0, v_k - 1\] and \([a_j, b_j] = [v_k, v_k]\]

that is:

\[
\begin{bmatrix}
... \ a_{i-1} & 0 & a_{i+1} & ... & a_{j-1} & v_k & a_{j+1} & ... & \ldots \\
... \ b_{i-1} & v_k-1 & b_{i+1} & ... & b_{j-1} & v_k & b_{j+1} & ... & \ldots 
\end{bmatrix}.
\]

“The \(i^{th}\) stack operand contains the same value as the \(j^{th}\) local variable” is expressed as “\([x_i, y_i] = [*], j]\)” that is:

\[
\begin{bmatrix}
... \ a_i & \ldots & x_{i-1} & * & x_{i+1} & \ldots \\
... \ b_i & \ldots & y_{i-1} & j & y_{i+1} & \ldots 
\end{bmatrix}.
\]

We define a partial order “\(\preceq\)” on the frame states such that \(F \preceq F’\) indicates that all the relations among the local variables and stack operands that hold in \(F’\) also hold in \(F\). The relations that hold in \(F\), however, may not necessarily hold in \(F’\). Fig. 10.2 presents the definition of a “match” function such that:

\[\text{match}(F, F') = \text{true} \iff F \preceq F'.\]

### 10.3.2 Forward Flow Analysis

We use forward flow analysis to identify the range of the values for each local variable and each stack operand at the entry of each instruction. To perform such a flow analysis, we need to define a flow function, \(f^{+}\), for each bytecode instruction. For clarity of presentation, let us focus on a reduced JVM that supports only a small subset of Java bytecode instructions [93] as follows: \{\text{iconst}, \text{i}, \text{aload}, \text{iadd}, \text{isub}, \text{astore}, \text{istore}, \text{newarray}, \text{arraylength}, \text{xaload}, \text{ifcmp}\}

- \text{iconst}: push an integer constant onto the operand stack. The semantic of this instruction can be expressed as:

\[
\frac{l_1, l_2, \ldots, l_m; s_1, s_2, \ldots, s_n}{l_1, l_2, \ldots, l_m; s_1, s_2, \ldots, s_n, s_{n+1} \left[ s_{n+1} = z \right]}.
\]

\footnote{We use \(a[[c]]\) to denote the execution semantics of bytecode instructions, where \(a\) is the state at the entry of a bytecode instruction, \(b\) is the state after execution of this instruction, and \(c\) is the constraint on the states \(a\) and \(b\).}
\[ \mathcal{F} = \left[ \begin{array}{c}
\alpha_1, \alpha_2, \ldots, \alpha_m; x_1, x_2, \ldots, x_n \\
\beta_1, \beta_2, \ldots, \beta_m; y_1, y_2, \ldots, y_n
\end{array} \right] \quad \mathcal{F}' = \left[ \begin{array}{c}
\alpha'_1, \alpha'_2, \ldots, \alpha'_m; x'_1, x'_2, \ldots, x'_n \\
\beta'_1, \beta'_2, \ldots, \beta'_m; y'_1, y'_2, \ldots, y'_n
\end{array} \right] \]

**function** match(\( \mathcal{F}, \mathcal{F}' \)) {
    for \( i := 1 \) to \( m \) {
        if(not match_range([\( \alpha_i, \beta_i \]), [\( \alpha'_i, \beta'_i \)])) { return false; }
    }
    for \( i := 1 \) to \( n \) {
        if(\( [x_i, y_i] = [*; j] \)) {
            if(\( [x'_i, y'_i] \neq [*; j] \)) { return false; }
        } else if(not match_range([\( x_i, y_i \]), [\( x'_i, y'_i \)]))
            return false;
    }
}

**function** match_range([\( A, B \]), [\( C, D \)]) {
    // return true if and only if we are sure \( [A, B] \subseteq [C, D] \)
    if(we cannot conservatively determine that \( C \leq A \))
        return false;
    if(we cannot conservatively determine that \( D \geq B \))
        return false;
    return true;
}

Fig. 10.2. The match function for the array bound check removal annotation.
where \( l_1, \ldots, l_m \) are the local variables, and \( s_1, s_2, \ldots, s_n, s_{n+1} \) are the stack operands. We define the flow function for \( \text{iconst}_z \) as:

\[
f^+ \left( \text{iconst}_z ; \begin{bmatrix} a_1, \ldots, a_m; x_1, \ldots, x_n \\ b_1, \ldots, b_m; y_1, \ldots, y_n \end{bmatrix} \right) = \begin{bmatrix} a_1, \ldots, a_m; x_1, \ldots, x_n, v_0 + z \\ b_1, \ldots, b_m; y_1, \ldots, y_n, v_0 + z \end{bmatrix},
\]

- **iload\_i, alo\_i**: load the \( i^{th} \) local variable onto the operand stack. The semantic of this instruction can be expressed as:

\[
\begin{aligned}
\frac{l_1, \ldots, l_i, \ldots, l_m; s_1, \ldots, s_n}{l_1, \ldots, l_i, \ldots, l_m; s_1, \ldots, s_{n+1} \quad [s_{n+1} = l_i]};
\end{aligned}
\]

We define \( f^+ \) for xload\_i (either iload\_i or alo\_i) as follows:

\[
f^+ \left( \text{xload}_i ; \begin{bmatrix} a_1, \ldots, a_m; x_1, \ldots, x_n \\ b_1, \ldots, b_m; y_1, \ldots, y_n \end{bmatrix} \right) = \begin{bmatrix} a_1, \ldots, a_m; x_1, \ldots, x_n, \star \\ b_1, \ldots, b_m; y_1, \ldots, y_n, i \end{bmatrix}.
\]

That is, \( f^+ \) pushes \([\star, i]\) onto the stack, indicating that the operand on top of the stack is equal to the \( i^{th} \) local variable.

- **istore\_i, astore\_i**: store the value on top of the operand stack to the \( i^{th} \) local variable. The semantics of this instruction can be captured as:

\[
\begin{aligned}
\frac{l_1, \ldots, l_i, \ldots, l_m; s_1, \ldots, s_n}{l_1, \ldots, l_i', \ldots, l_m; s_1, \ldots, s_{n+1} \quad [l_i' = s_{n}]}.
\end{aligned}
\]

Note that some stack operands might have state \([\star, i]\), indicating that these stack operands hold a value that is equal to the \( i^{th} \) local variable. Since istore\_i (or astore\_i) changes the value of the \( i^{th} \) local variable, it breaks the equality relation between the \( i^{th} \) local variable and those stack operands. Consequently, the stack operands with states \([\star, i]\) should be updated to reflect the original state of the \( i^{th} \) local variable. Further, if the value on top of the frame is \([\star, j]\) (\( j \neq i \)), the value written to the \( i^{th} \) local variable is \([a_j, b_j]\), instead of \([x_n, y_n]\). Therefore, we can define
\(f^+\) for \(\text{xstore}_i\) (either \(\text{istore}_i\) or \(\text{astore}_i\)) as follows:

\[
f^+\left(\text{xstore}_i, \begin{bmatrix} \ldots, a_{i-1}^i, a_{i+1}^i, \ldots, x_{k-1}^i, x_{k+1}^i, \ldots, x_n \end{bmatrix} \right) = \begin{bmatrix} \ldots, a_{i-1}^i x_n, a_{i+1}^i \ldots, x_{k-1}^i a_i x_{k+1}^i \ldots x_n \end{bmatrix}
\]

\[
f^+\left(\text{xstore}_i, \begin{bmatrix} \ldots, a_{i-1}^i a_i, a_{i+1}^i \ldots, x_{k-1}^i a_i x_{k+1}^i \ldots x_n \end{bmatrix} \right) = \begin{bmatrix} \ldots, a_{i-1}^i a_j a_{i+1}^i \ldots, x_{k-1}^i a_i x_{k+1}^i \ldots x_n \end{bmatrix}
\]

- \text{iadd, isub}: The semantics of these two instructions can be expressed as follows:

\[
\begin{align*}
l_1, \ldots, l_m; s_1, \ldots, s_{n-1}, s_n & \quad \begin{bmatrix} s'_{n-1} = s_{n-1} + s_n, \quad \text{for iadd}; \\ s'_{n-1} = s_{n-1} - s_n, \quad \text{for isub}; \end{bmatrix}
\end{align*}
\]

Given two integer values, \(A \in [x_1, x_2]\) and \(B \in [y_1, y_2]\), the ranges of the results of the addition and subtraction instructions can be computed as \(A + B \in [x_1 + y_1, x_2 + y_2]\), and \(A - B \in [x_1 - y_2, x_2 - y_1]\), respectively. Therefore, we have:

\[
f^+\left(\text{iadd}, \begin{bmatrix} a_1, \ldots, a_m; x_1, \ldots, x_{n-2}, x_{n-1}, x_n \end{bmatrix} \right) = \begin{bmatrix} a_1, \ldots, a_m; x_1, \ldots, x_{n-2}, x_{n-1} + x_n \end{bmatrix}
\]

and

\[
f^+\left(\text{isub}, \begin{bmatrix} a_1, \ldots, a_m; x_1, \ldots, x_{n-2}, x_{n-1}, x_n \end{bmatrix} \right) = \begin{bmatrix} a_1, \ldots, a_m; x_1, \ldots, x_{n-2}, x_{n-1} - y_n \end{bmatrix}
\]

- \text{arraylength}: get the length of an array:

\[
l_1, \ldots, l_m; s_1, \ldots, s_{n-1}, s_n \quad \begin{bmatrix} s_n \text{ is a reference to an array, and } s'_{n} \text{ is the length of this array} \end{bmatrix}
\]
At the entry of this instruction, a reference to this array is on the top of the operand stack. If a reference to this array is also stored in a local variable i.e., \([x_n, y_n] = [* , j]\) and length of this array is unknown, we create a new abstract (symbolic) value representing the length of this array. Therefore, we have:

\[
\begin{align*}
\mathbf{f}^+ \left( \text{arraylength}, \left[ \begin{array}{c}
\ldots, a_i, \ldots, x_{n-1}, * \\
\ldots, b_i, \ldots, y_{n-1}, i
\end{array} \right] \right) = \left[ \begin{array}{c}
\ldots, a_i \triangledown v_k, \ldots, x_{n-1}, a_i \triangledown v_k \\
\ldots, b_i \triangleright v_k, \ldots, y_{n-1}, b_i \triangleright v_k
\end{array} \right],
\end{align*}
\]

where \(v_k\) is a symbolic value that is not does not appear in

\[
\left[ \begin{array}{c}
\ldots, a_i, \ldots, x_{n-1}, * \\
\ldots, b_i, \ldots, y_{n-1}, i
\end{array} \right],
\]

and “\(\triangledown\)” and “\(\triangleright\)” are defined as:

\[
\begin{align*}
p \triangledown q &= \begin{cases} 
q, & \text{if we can conservatively determine } p < q \\
p, & \text{otherwise}
\end{cases} \\
p \triangleright q &= \begin{cases} 
q, & \text{if we can conservatively determine } p > q \\
p, & \text{otherwise}
\end{cases}
\end{align*}
\]

Particularly, since symbolic value \(v_k \in V\) represents a value greater than or equal to zero, we have \(-\infty \triangledown v_k = v_k\) and \(+\infty \triangleright v_k = v_k\). If \([a_i , b_i] = [\infty, \infty]\) (which means that the length of the array is unknown at the entry of this instruction), we have \([a_i \triangledown v_k, b_i \triangleright v_k] = [v_k, v_k]\). That is, after the execution of an arraylength instruction, we introduce a new symbolic value, \(v_k\), to the frame state to represent the length of this array if the length of this array is unknown before the execution of this instruction.

On the other hand, if the reference to this array is not stored in any local variable, there is no need to create a new abstract value. Since the length of an array cannot be less than zero, we have:

\[
\begin{align*}
\mathbf{f}^+ \left( \text{arraylength}, \left[ \begin{array}{c}
\ldots, x_{n-1}, x_n \\
\ldots, y_{n-1}, y_n
\end{array} \right] \right) = \left[ \begin{array}{c}
\ldots, x_{n-1}, v_0 \triangledown x_n \\
\ldots, y_{n-1}, y_n
\end{array} \right], \text{ where } [x_n , y_n] \neq [* , i].
\end{align*}
\]
• **newarray:** create a new array:

\[
\begin{align*}
\frac{l_1, \ldots, l_m; s_1, \ldots, s_{n-1}, s_n}{l_1, \ldots, l_m; s_1, \ldots, s_{n-1}, s_n'}
\end{align*}
\]

\(s_n\) is the length of the array, and \(s_n'\) is a reference to the created array.

If the operand on the top of the operand stack is a single point (i.e., \([y_k + z, v_k + z]\)), we use this value to represent the length of the newly-created array:

\[
f^+ \left( \text{newarray}, \left[ \ldots; x_{n-1}, v_i + z \right] \right) = \left[ \ldots; x_{n-1}, v_i + z \right].
\]

Otherwise, we create a new abstract value \((v_k)\) for the length of the newly created array. If the operand on top of the operand stack has the \([*, i],\) we should also update the content of the \(l^\text{th}\) local variable since we now know that the value of this local variable is equal to the length of this array. We can express this as follows:

\[
f^+ \left( \text{newarray}, \left[ \ldots; a_{i-1}, a_i, a_{i+1}, \ldots; x_{n-1}, * \right] \right) = \left[ \ldots; a_{i-1}, v_k, a_i, a_{i+1}, \ldots; x_{n-1}, v_k \right],
\]

and

\[
f^+ \left( \text{newarray}, \left[ \ldots; y_{n-1}, x_n \right] \right) = \left[ \ldots; y_{n-1}, v_k \right], \quad \text{where} \quad [x_n, y_n] \neq [*, i].
\]

• **xaload:** load an element of an array:

\[
\begin{align*}
\frac{l_1, \ldots, l_m; s_1, \ldots, s_{n-1}, s_n}{l_1, \ldots, l_m; s_1, \ldots, s_{n-1}'}
\end{align*}
\]

\(s_{n-1}\) is a reference to an array, \(s_n\) is the index to an array element, and \(s_n'\) is the value of the array element.

If this instruction is successfully executed, for the instructions that follow them, we know for sure that the length of the accessed array is larger than zero, and that the value of the index variable is larger than or equal to zero and smaller than the length of the array. Therefore, the states of the relevant local variables and stack operands must be updated. We have:

\[
f^+ \left( \text{xaload}, \left[ \ldots; a_i, \ldots, a_j, \ldots; x_{n-2}, *, *, \right] \right) = \left[ \ldots; a_i, \ldots, 0 \triangleright a_j, \ldots; x_{n-1}, -\infty \right],
\]

\[
\left[ \ldots, b_i, \ldots, (b_i - 1) \triangle b_j, \ldots, y_{n-1}, +\infty \right].
\]
• **if icmplt:** take branch if \( s_{n-1} < s_n \), where \( s_{n+1} \) and \( s_n \) are two integer values on the top of the operand stack. This instruction introduces constraints on the values of the local variables and stack operands. In both the taken and untaken branches, the states of the relevant local variables and stack operands should be updated accordingly. Specifically, if the branch is taken, we have:

\[
 f^+ \left( \text{if icmplt, } \left[ \ldots, a_i, \ldots, a_j, \ldots, x_{n-2}, *, * \right] \right) = \left[ \ldots, a_i, \ldots, a_j \triangle a_i + 1, \ldots, \ldots, x_{n-2} \right].
\]

On the other hand, if the branch is not taken, we have:

\[
 f^+ \left( \text{if icmplt, } \left[ \ldots, a_i, \ldots, a_j, \ldots, x_{n-2}, *, * \right] \right) = \left[ \ldots, a_i \nabla a_j, \ldots, a_j, \ldots, \ldots, x_{n-2} \right].
\]

Fig. 10.3 shows a work-list-based forward flow analysis algorithm for array bound removal. Since the merge function used in this analysis is monotonic, and the number of frame states is limited, this algorithm is guaranteed to terminate.

### 10.3.3 Annotation Generation and Verification

We use the algorithm in Fig. 10.4 to reduce the space for storing annotations. An abstract value expression \( v_i + z \) can be stored in one byte (eight bits): the first half byte (four bits) is for the name of the abstract value \( v_i + i \), and the other half byte (four bits) is for the integer value \( z \). The value “11111111” is reserved for “\(-\infty\)”. If an “11111111” appears as the lower bound of a range, it represents \(-\infty\); on the other hand, if it appears as the upper bound of range, it represents \(+\infty\). In this format, we can use at most 15 abstract values (\( v_0 \) through \( v_{14} \)) in a frame state. Since the number of array references that are simultaneously stored in the local variables and the operand stack is small for typical Java methods, we are not likely to use a large number of abstract value names. In fact, we could not find any method in our benchmarks that requires more than 14 abstract value names. The second half byte stores the two’s complement code of integer value \( z \). In this format, we can use 16 integer values: \(-7\) through \(+8\). An abstract expression that cannot be represented in this format is treated as either \(-\infty\) (for lower bound) or \(+\infty\) (for upper bound).
Input:

Method $M = (\vec{b}, \vec{a})$; $l = |\vec{b}| - 1$

Entry state: $F^*$

Output:

Flow state of each instruction: $\{F_0, F_1, ..., F_1\}$

\begin{verbatim}
procedure flow_forward(t)
    for $i := 0$ to $l$ { $F_i := F^*$;
work_list := $\{0\}$;
while(work_list $\neq \phi$) {
    foreach $i \in$ work_list {
        if($\vec{C}[i] > 0$) { $\vec{C}[i] = -1$;
work_list := work_list $\setminus \{i\}$;
    foreach $j \in$ succ($M, i$) {
        $F := f^+(\vec{b}[i], F_j)$;
        if($F_j = \epsilon$) {
            $F_j := F$;
work_list := work_list $\cup \{j\}$;
    } else if(not match($F, F_j$)) {
            $F_j := merge(F_j, F)$;
work_list := work_list $\cup \{j\}$;
    }
}
}
}

// $F = \begin{bmatrix} a_1, a_2, ..., a_m ; x_1, x_2, ..., x_n \\ b_1, b_2, ..., b_m ; y_1, y_2, ..., y_n \end{bmatrix}$;

// $F' = \begin{bmatrix} a_1', a_2', ..., a_m' ; x_1', x_2', ..., x_n' \\ b_1', b_2', ..., b_m' ; y_1', y_2', ..., y_n' \end{bmatrix}$;

function merge($F, F'$) {
    for $i := 0$ to $m$
        $[a_i'', b_i''] := merge_range([a_i, b_i], [a_i', b_i'])$;
    for $i := 0$ to $n$
        $[x_i'', y_i''] := merge_stack(i, F, F')$;
    return $F''$;
}

function merge_range($[A, B], [C, D])$ {
    if(min($A, C$) can be conservatively determined)
        $X = min(A, C)$;
    else
        $X = -\infty$;
    if(max($B, D$) can be conservatively determined)
        $Y = \max(B, D)$;
    else
        $Y = +\infty$;
    return $[X, Y]$; // $[A, B] \cup [C, D] \subseteq [X, Y]$;
}

function merge_stack($i, F, F'$) {
    if($[x_i, y_i] = [*, j]$ and $[x_i', y_i'] = [*, j]$)
        return $[*, j]$;
    if($[x_i, y_i] = [*, p]$) [$x_i, y_i] := [a_i, b_i]$;
    if($[x_i', y_i'] = [*, q]$) [$x_i', y_i'] := [a_i', b_i']$;
    return merge_range($[x_i, y_i], [x_i', y_i']$);
}

Fig. 10.3. Forward flow analysis for the array bound check removal.
\end{verbatim}
The annotation is verified using the algorithm in Fig. 10.5. The rules for removing the bounds checks at each instruction that accesses an array element are as follows. Assume that we are at the entry of the instruction that accesses \( a[i] \) and the annotations for variables \( a \) and \( i \) are \([x, y]\) and \([p, q]\), respectively. We can remove the check for the lower bound if \( p \) has the form “\( v_i + z \)” where \( z \geq 0 \). Similarly, we can remove the check for the upper bound if we can conservatively conclude \( q < x \). Specifically, if \( q = v_i + z_1 \) and \( x = v_j + z_2 \), we can remove the upper bound check if \((i = j \text{ or } i = 0) \text{ and } (z_1 < z_2)\). Note that, \( v_0 \) is the name for the integer zero and the other abstract value names (\( v_1 \) through \( v_{13} \)) represent values that are greater than or equal to zero.

The time complexity of evaluating the \( \preceq \) relation between a pair of flow states is \( O(m + n) \), where \( m \) is the number of the local variables and \( n \) is the maximum depth of the operand stack. Consequently, the time complexity of our verification algorithm is \( O((m + n)sl) \), where \( l \) is the total number of the instructions and \( s \) is the average number of immediate successors for a given instruction. The space complexity of our verification algorithm is the same as the space required to store the annotation, i.e., \( O(mk_1 + nk_2) \), where \( k_1 \) is the number of instructions whose states of local variables cannot be removed by our annotation generation algorithm (Fig. 10.4), and \( k_2 \) is the number of instructions whose states of the stack operands cannot be removed by our annotation generation algorithm. For the applications in our experimental suite, we find \( k_1 \approx 0.06l \) and \( k_2 \approx 0.05l \), i.e., both of which are much smaller than \( l \), the total number of instructions.

### 10.4 Null Pointer Check Removal

The JVM specification [93] requires JVM to throw a NullPointerException when an application attempts to use null in a case where an object is required. These cases include calling the instance method of a null object, accessing or modifying the field of a null object, taking the length of null as if it were an array, accessing or modifying the slots of null as if it were an array, and throwing null as if it were a Throwable value.

Checking for null pointer at the execution of each instruction where a reference is used incurs an overhead. An optimizing dynamic compiler can remove most redundant null checks using data flow analysis. However, as we discussed earlier, performing data flow analysis at
Input:
Bytecode of method \( M: \vec{b}; l = |\vec{b}| - 1 \)
Result of forward flow analysis: \( \{F_0, F_1, \ldots, F_l\} \)

Output:
Annotation \( \{A_0, A_1, \ldots, A_l\} \)

procedure annotate()
\[
A_0 := F_0;
\]
for \( i := 1 \) to \( l \) do
\[
\text{if}(\text{pred}(i) = \{i - 1\}) \{ \\
\quad F' := f^+(\vec{b}[i-1], F_{i-1}) \\
\quad \text{// } F_i = \begin{bmatrix} a_1, a_2, \ldots, a_m; x_{1}, x_{2}, \ldots, x_n \\ b_1, b_2, \ldots, b_m; y_{1}, y_{2}, \ldots, y_n \end{bmatrix}, \\
\quad \text{// } F_i = \begin{bmatrix} a_1, a_2, \ldots, a_m; x_{1}, x_{2}, \ldots, x_n \\ b_1, b_2, \ldots, b_m; y_{1}, y_{2}, \ldots, y_n \end{bmatrix} \\
\quad \text{if}(\exists j : [x_j, y_j'] \neq [x_j, y_j]) \{ \\
\quad \quad \text{store } [x_1, y_1], [x_2, y_2], \ldots, [x_n, y_n] \text{ in } A_i; \\
\quad \quad \text{else } \\
\quad \quad \quad \text{// there is no need to store } [x_1, y_1], [x_2, y_2], \ldots, [x_n, y_n] \text{ in } A_i; \\
\quad \quad \} \\
\quad \text{else} \\
\quad \quad \quad \text{// there is no need to store } [a_1, b_1], [a_2, b_2], \ldots, [a_m, b_m] \text{ in } A_i; \\
\quad \} \\
\text{if}(\exists j : [a'_j, b'_j] \neq [a_j, b_j]) \{ \\
\quad \text{store } [a'_1, b'_1], [a'_2, b'_2], \ldots, [a'_m, b'_m] \text{ in } A_i; \\
\quad \text{else} \\
\quad \quad \text{// there is no need to store } [a'_1, b'_1], [a'_2, b'_2], \ldots, [a'_m, b'_m] \text{ in } A_i; \\
\} \\
\} \\
\text{else} \\
\quad \text{store } [x_1, y_1], [x_2, y_2], \ldots, [x_n, y_n] \text{ in } A_i; \\
\quad \text{store } [a_1, b_1], [a_2, b_2], \ldots, [a_m, b_m] \text{ in } A_i; \\
\}
\]

Fig. 10.4. Annotation generation algorithm.
Input:
Method \( M = (\vec{b}, \vec{a}); l = |\vec{b}| - 1 \)
Annotation \( \{ A_0, A_1, ..., A_l \} \)
Entry State: \( F^* \)

Output:
Translates \( M \) into native code if the annotation passes the verification;

\begin{verbatim}
procedure translate() {
if(\( F^* \not\preceq A_0 \)) abort due to incorrect annotation;
for \( i := 0 \) to \( l \) do {
    \( F := A_i; \)
    \( F' := \Phi^{-1}(\vec{b}[0], (\vec{L}, \vec{S})); \)
    if(\( \exists j \in \text{succ}(i) : A_j \not\preceq F' \)) abort due to incorrect annotation;
    translate bytecode \( \vec{b}[i] \) into native code with respect to annotation \( A_i \);
}
}
\end{verbatim}

Fig. 10.5. Compilation with annotation verification. The compiler linearly scans the instructions of method \( M \), and translates each instruction into native code. Before translating each instruction, it verifies if the frame state of the current instruction satisfies the constraints of the flow equations.

runtime introduces both performance and memory overheads. Our annotation-based approach performs the analysis offline. The results of the offline analysis are then stored with the Java class files. At runtime, before compiling a method with annotation, the JVM verifies the annotation to make sure that it is not modified. If the annotation passes the verification, the JIT compiler can safely use the annotation to identify the instructions for which the null pointer checks are redundant. Compared to actually performing the data flow analysis at runtime, just verifying its result requires less time and smaller memory space. Our annotation-based null pointer check removal is performed in three steps: forward flow analysis, annotation generation, and annotation verification. The first two steps are performed offline, whereas the last step, which is crucial from a reliable execution viewpoint, is performed online. Since the algorithms for null check removal are similar to those for array bound removal, we omit the its details due to space limitation.\(^3\)

\(^3\)The details of the flow analysis, annotation generation, and annotation verification algorithms for the null check removal is included in our technical report, which is available at \url{http://www.cse.psu.edu/~gchen/techreport.pdf}.  

10.5 Value Liveness Analysis

Input:
Method \( M = (\vec{b}, \vec{a}); l = |\vec{b}| - 1 \)
Exit state: \( \vec{L}^* \)

Output:
Flow state of each instruction: \( \{\vec{L}_0, \vec{L}_1, ..., \vec{L}_l\} \)

procedure backward_flow() {
    foreach \( i \in R(M) \) \{ \( \vec{L}_i := \vec{L}^*; \) \}
    foreach \( i \notin R(M) \) \{ \( \vec{L}_i := \varepsilon; \) \}
    work_list := R(M);
    while(work_list \neq \emptyset) {
        foreach \( i \in \text{work_list} \) {
            work_list := work_list - \{i\};
            foreach \( j \in \text{pred}(M, i) \) {
                \( \vec{L} := f^{-1}(\vec{b}[j], \vec{L}_i); \)
                if(\( \vec{L}_j = \varepsilon \)) {
                    \( \vec{L}_j := \vec{L}; \)
                    work_list := work_list \cup \{j\};
                } else if(\( \vec{L} \not\preceq \vec{L}_j \)) {
                    \( \vec{L}_j := \vec{L}_j \sqcup \vec{L}; \)
                    work_list := work_list \cup \{j\};
                }
            }
        }
    }
}

Fig. 10.6. Backward flow algorithm for value liveness analysis.

Value liveness analysis identifies at each instruction the liveness of each value that is defined in the program. One of its main purposes is to help the dynamic compiler allocate registers more effectively, though it can be used for other purposes as well. Since determining the liveness of stack operands is trivial, our value liveness analysis focuses on local variables. It
Table 10.1. Backward flow function for the value liveness analysis.

<table>
<thead>
<tr>
<th>Operation Code of b</th>
<th>( f^- (b, L) ) where ( L = (l_1, l_2, \ldots, l_n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>xload_{i}</td>
<td>((l_1, \ldots, l_{i-1}, 0, l_{i+1}, \ldots, l_n))</td>
</tr>
<tr>
<td>xstore_{i}</td>
<td>((l_1, \ldots, l_{i-1}, 1, l_{i+1}, \ldots, l_n))</td>
</tr>
<tr>
<td>inc_{i}</td>
<td>((l_1, \ldots, l_{i-1}, 0, l_{i+1}, \ldots, l_n))</td>
</tr>
<tr>
<td>Other</td>
<td>((l_1, l_2, \ldots, l_n))</td>
</tr>
</tbody>
</table>

is performed in three steps: backward data flow analysis, annotation generation, and annotation verification. The following two subsections explain the details of these steps.

10.5.1 Backward Flow Analysis

The algorithm for backward flow analysis is given in Fig. 10.6. What we need is to define the frame states, the merge function, and the partial order relation on the frame states. Since we consider only the local variables, the state of the frame at the entry of the \( i \)th instruction has the form: \( L_i \), where \( \forall 0 \leq j < |L_i| : L_i[j] \in \{0, 1\} \). \( L_i[j] = 0 \) indicates that, at the entry of the \( i \)th instruction, the value in the \( j \)th local variable could be live and must be kept, whereas \( L_i[j] = 1 \) indicates that the value is definitely dead and can be safely discarded.

The merge function and the partial order on the frame states are defined as follows:

\[
L_x \sqcup L_y = L \iff L[i] = L_x[i] \cdot L_y[i] (\forall 0 \leq i < |L|), \quad \text{and}
\]

\[
L_x \preceq L_y \iff L_x[i] \geq L_y[i] (\forall 0 \leq i < |L|).
\]

The exit state for the analysis is \( L^* \), where \( \forall 0 \leq i < |L^*| : L^*[i] = 1 \). The backward flow function \( f^- () \) is presented in Table 10.1. With these definitions, we can use the algorithm in Fig. 10.6 to perform the analysis. Since the merge function (\( \sqcup \)) is monotonic, and the number of frame states is limited, this algorithm is guaranteed to terminate.

10.5.2 Annotation Generation and Verification

To generate an annotation for the liveness of each value based on the result of the backward flow analysis, we encode the flow state of each instruction as a bitmap, each bit of which represents the state of a local variable. Since the number of local variables is small for a typical
Input:
Method $M = (\vec{b}, \vec{a}); l = |\vec{b}| - 1$
Annotation $\vec{L}_0, \vec{L}_1, \ldots, \vec{L}_l$

Output:
Translates $M$ into native code if the annotation passes the verification;

```
procedure translate()
    for $i := 1$ to $l$ do {
        if(\vec{b}[i] reads the $k$th local variable and $\vec{L}_i[k] = 1$)
            abort due to reading a dead value;
        foreach $j \in \text{succ}(i)$
            if($f^- (\vec{b}[i], \vec{L}_j) \not\preceq \vec{L}_i$)
                abort due to violation of flow constraints;
        $\vec{T} := \bigcup_{j \in \text{succ}(i)} \vec{L}_j$;
        // $\vec{T}[k] = 0$: the $k$th local variable is still alive after the execution of bytecode $\vec{b}[i]$;
        // $\vec{T}[k] = 1$: the $k$th local variable is dead after the execution of bytecode $\vec{b}[i]$;
        translate bytecode $\vec{b}[i]$ with respect to $\vec{T}$;
    }
```

Fig. 10.7. Compilation with the value lifetime annotation.

Java method, we can expect the size of the annotation to be small. The annotation must be verified before being used by the dynamic compiler. Fig. 10.7 presents the algorithm for verifying annotations before compiling each instruction. We abort the compilation if an instruction reads a value that is annotated as “dead”, or if a flow constraint is violated.

10.6 Experiments

10.6.1 Setup

We use a set of embedded Java applications to evaluate the effectiveness of our verifiable annotation techniques. Our experimental suite contains nine Java applications for embedded devices such as PDAs and mobile phones. Table 10.2 gives brief descriptions of our applications. The third column of this table lists the total size (in bytes) of the class files that are used by each application, including the classes from both the application itself and the class library. We use an instrumented KVM [84] to count at runtime the numbers of object accesses (in the null check
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Total Size of Class Files (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>album</td>
<td>Digital photo album</td>
<td>602,787</td>
</tr>
<tr>
<td>calculator</td>
<td>Numeric calculator</td>
<td>550,271</td>
</tr>
<tr>
<td>chess</td>
<td>Chess game</td>
<td>598,491</td>
</tr>
<tr>
<td>jpegview</td>
<td>JPEG image renderer</td>
<td>554,463</td>
</tr>
<tr>
<td>mdoom</td>
<td>3D shooting game</td>
<td>553,258</td>
</tr>
<tr>
<td>pman</td>
<td>Packman game</td>
<td>558,374</td>
</tr>
<tr>
<td>pushpuzzle</td>
<td>Puzzle game</td>
<td>579,900</td>
</tr>
<tr>
<td>scheduler</td>
<td>Personal scheduler</td>
<td>614,398</td>
</tr>
<tr>
<td>sfmap</td>
<td>Digital map</td>
<td>600,720</td>
</tr>
</tbody>
</table>

Table 10.2. Our Java applications.

removal experiments) and array accesses (in the array bound check removal experiments). KVM is a JVM for the embedded systems with constrained memory budget (typically, less than one megabyte), and satisfies the CLDC specification [7].

10.6.2 Results

Table 10.3 presents the results for our verifiable annotation-guided null pointer check removal. The numbers in the second and third columns are dynamic counts (i.e., they are collected at runtime). On an average, our approach safely removes 70.6% of the total null pointer checks that otherwise would be performed at runtime. In these experiments, we annotate only the methods that actually benefit from the annotation. The average space cost (per application) for our annotations is 5,878 bytes per application (see the last column of Table 10.3), which includes the annotations for the application classes and the standard library classes that are used by the application. Compared to the total size of the class files used by the applications (see the third and sixth columns of Table 10.2), the average increase in the sizes of the class files due to our annotation is only 0.97%.

Table 10.4 presents the results for our verifiable annotation-guided array bound check removal. The numbers in the second, third and fifth columns are dynamic counts. On an average, our annotations safely remove 17.6% of the total lower bound checks and 11.8% of the total upper bound checks that would normally be performed at runtime. As before, we annotate
Table 10.3. Results for the null check removal experiments. The numbers in the second and third columns are dynamic counts.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original Number of Null Checks</th>
<th>Removed Null Checks Count</th>
<th>Size of Annotation (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>album</td>
<td>798,490</td>
<td>577,499</td>
<td>6,292</td>
</tr>
<tr>
<td>calculator</td>
<td>853,768</td>
<td>611,964</td>
<td>5,324</td>
</tr>
<tr>
<td>chess</td>
<td>6,880,863</td>
<td>4,105,570</td>
<td>6,071</td>
</tr>
<tr>
<td>jpegview</td>
<td>6,907,180</td>
<td>4,568,569</td>
<td>5,777</td>
</tr>
<tr>
<td>mdoom</td>
<td>22,834,931</td>
<td>18,048,498</td>
<td>5,547</td>
</tr>
<tr>
<td>pman</td>
<td>1,629,226</td>
<td>1,149,095</td>
<td>5,574</td>
</tr>
<tr>
<td>pushpuzzle</td>
<td>1,130,278</td>
<td>844,513</td>
<td>5,790</td>
</tr>
<tr>
<td>scheduler</td>
<td>995,146</td>
<td>703,006</td>
<td>6,583</td>
</tr>
<tr>
<td>sfmap</td>
<td>1,247,404</td>
<td>878,894</td>
<td>5,948</td>
</tr>
<tr>
<td>Average</td>
<td>4,808,587</td>
<td>3,498,623</td>
<td>5,878</td>
</tr>
</tbody>
</table>

only the methods that actually benefit from the annotation. Therefore, we do not introduce any overhead to the methods from which the bounds checks cannot be removed by our annotation. The average space overhead due to our annotations is 6,563 bytes per application (see the last column), which includes the annotations for the application classes and the standard library classes that are used by the application. Compared to the total size of the class files used by the applications (see the third and sixth columns of Table 10.2), the average increase in the sizes of the class files due to our annotation is about 1.0%.

10.7 Related Work

The idea of using annotations to speed up a dynamic compiler has been studied by several research groups. Krintz and Calder [83] proposed an annotation based framework that can reduce the compilation overhead of Java programs. They implemented four types of annotations: those providing static analysis information, those enabling optimization reuse, those enabling selective optimization, and those enabling optimization filtering. In their framework, the annotations are not verified. Therefore, in execution environments where security is of concern, the annotations that provide static analysis information cannot be safely used.
Table 10.4. Results for the array bound check removal experiments. The numbers in the second, third and fifth columns are dynamic counts.

Azevedo et al. [70] presented a Java bytecode annotation framework to support dynamic compiler optimization. They annotated null checks elimination, array bound check elimination, virtual register assignment, and memory disambiguation. However, their annotations are not safe and not verified; therefore, their annotations cannot be used in environments where security is required. Their annotations increase the sizes of Java class files by $33\% \sim 100\%$. Pominville et al. [103] also presented a framework for supporting the optimization of Java programs using annotations. Their annotations targeted at eliminating array bound and null pointer checks. These annotations speed up the Java applications by up to 35\% and increase the size of class files by $8\% \sim 16\%$. Again, their annotations are not safe and not verified. Jones and Kamin [71] proposed a scheme that annotates Java class files with virtual register assignment information. In a technical report [65], Haldar talked about annotating the Java class files with results of offline data flow analysis. These annotations are verified at runtime to make sure the annotations satisfy the underlying data flow equations. He reported results from two data flow analyses: liveness of local variables and null check elimination. His annotations increase the sizes of the class files by $7\% \sim 9\%$ and the verification of these annotations is 2\sim 11 times faster than their generation. Since he does not present technical details of his approach, a thorough comparison is not possible.
Our work can also be considered as a specific case of the more general concept of proof-carrying code (PCC) [100], adapted to embedded computing domain. PCC is a mechanism by which a host system can determine with certainty that it is safe to execute a program supplied by an untrusted source. For this purpose, the untrusted code producer augments the code with a safety proof, indicating that the code does not violate a previously defined safety property. The host can then verify the proof easily and quickly to validate the proof without using cryptography or consulting any external agents. Morrisett et al. [98] demonstrated that proof carrying code can be automatically generated from high-level programming languages. Colby et al. [49] implemented a certifying compiler for Java programming language, which compiles Java bytecodes into the x86 assembly language. Their compiler also generates the proof for type safety of the assembly code, which can be verified by a small proof-checker.

Another work that is close to our verifiable annotation is the light-weight bytecode verification [81], which is used in KVM [7, 84]. The JVM specification requires the Java bytecodes to be verified before execution. Verifying bytecodes involve expensive flow analysis, which is not suitable for memory space constrained embedded systems. The light-weight bytecode verification approach offloads the expensive flow analysis to an offline pre-verifier. The results of the verification are incorporated into the class files as “stackmaps”, which can then be verified quickly by the embedded JVM. Although the stackmaps are not targeted at optimization, they can be considered as a kind of annotation that can be verified online at a low cost. Amme et al. [27] presented SafeTSA, an intermediate representation for mobile code, which allows the dynamic compiler to safely use the results of offline optimizations. Code in SafeTSA can be easily verified for type safety. However, the format of SafeTSA is based on a syntax tree and static single assignment form, which makes SafeTSA not suitable for interpretation based execution. Bodik el al [33] presented a light-weight algorithm for elimination of array bound checks on demand. It works by adding a few edges to the SSA data flow graph and performing a traversal of the graph. Adl-Tabatabai et al. [24] implemented a Just-in-Time Java compiler, which can eliminate redundant array bound checks. However, their approach can be applied only locally to each extended basic block and only constant operands can be used.
10.8 Concluding Remarks

Exploiting annotations is an effective approach for reducing the runtime overheads caused by JIT compilation. Therefore, they are particularly suitable for embedded environments. In this chapter, we present a framework that generates annotations that can be verified with a low cost during execution. At runtime, the verifier can detect modifications in the annotations, if any, that may cause the JIT compiler to generate incorrect or insecure code. In this framework, we implement the annotations that can guide the dynamic compiler to eliminate redundant null pointer checks and array bound checks, and can support optimizations based on liveness analysis. We also present techniques that reduce the size of annotations, which is very important for memory-constrained embedded systems. Our experimental evaluation with a set of embedded Java applications shows promising results.
References


[19] US mobile devices to 2006: A land of opportunity. Based on an extensive research program conducted by Datamonitor on mobile device markets in the US.


[90] Z. Li, C. Wang, and Z. Li. Task allocation for distributed multimedia processing on wirelessly networked handheld devices. In 16th International Parallel and Distributed Processing Symposium, April 2002.


Vita

Guangyu Chen obtained his B.S. degree in Computer Science at University of Science and Technology of China. He entered Computer Science and Engineering Department of The Pennsylvania State University in Fall 2000. His research is focused on Java Virtual Machine for resource constrained systems.