A Pattern-based Approach to Parallel Execution of Sequential Code on Multicore Systems

Submitted by:
Ghulam Mustafa  2008-PhD-CS-01

Supervised by: Prof. Dr. Waqar Mahmood

Department of Computer Science & Engineering
University of Engineering and Technology
Lahore
A Pattern-based Approach to Parallel Execution of Sequential Code on Multicore Systems

Submitted to the faculty of the Computer Science and Engineering Department of the University of Engineering and Technology Lahore in partial fulfillment of the requirements for the Degree of

Doctor of Philosophy

in

Computer Science.

__________________________  _________________________
Internal Examiner            External Examiner

__________________________  _________________________
Dean                        Chairman
Faculty of Electrical Engineering

Department of Computer Science & Engineering

University of Engineering and Technology

Lahore
Declaration

I declare that the work contained in this thesis is my own, except where explicitly stated otherwise. In addition this work has not been submitted to obtain another degree or professional qualification.

Signed:  
Date:  

First of all, I would like to thank Allah subhana-ho-wa-taalla for giving me the opportunity, courage, knowledge and strength to pursue this humble effort. Without His help it was really beyond my inspiration and aspirations. He created the favorable circumstances and gathered cooperative people around me to keep me dedicated for this work. Then, I would like to thank and acknowledge the people who continuously supported me throughout this academic track.

Kind guidance of my PhD supervisor, Prof. Dr. Waqar Mahmood, especially in theoretical and financial aspects of my research is unforgettable. Under his supervision, I learned how to compose and present a doctoral level research problem. I learned how to formulate and solve a large computational problem using either top down or bottom up approach. I also gained an understanding of presenting a research problem as a viable project to be get funded.

This work has never been possible without the help and practical guidance of Dr. Abdul Waheed, consultant High Performance Computing and Networking Lab (HPCNL) at Al-Khawarizmi Institute of Computer Science (KICS), UET Lahore. His experience, knowledge, wisdom and enthusiasm guided me throughout the PhD studies. Under his guidance, I learned how to develop prototypes, perform experiments and represent performance data. Encouragement of Dr. Usman Ghani during odd times is unforgettable. His motivational discussions are highly appreciable. I am especially thankful to Dr. Amir Mahmood, Dr. Ghalib A Shah, Dr. Abad Shah, Dr. Haroon Attique Babri and Dr. Muhammad Ali Maud for their continuous support and encouragement. I am also thankful to all fellows and friends for their sharing and caring behavior, and all the excitement we enjoyed together.

I must appreciate and acknowledge the patience, support and prayers of my family members, especially my father, wife and kids.
Dedicated to my mother (may her soul rest in eternal peace), father, wife & kids . . .
# Contents

<table>
<thead>
<tr>
<th>Acknowledgments</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>xi</td>
</tr>
<tr>
<td>Abstract</td>
<td>xii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Research Questions</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Scope and Aim</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Thesis Contributions</td>
<td>5</td>
</tr>
<tr>
<td>1.6 Justification for the Research</td>
<td>7</td>
</tr>
<tr>
<td>1.6.1 Application Areas</td>
<td>8</td>
</tr>
<tr>
<td>1.6.2 Future Research Areas</td>
<td>9</td>
</tr>
<tr>
<td>1.7 Thesis Overview</td>
<td>10</td>
</tr>
<tr>
<td>1.8 Future Research Directions</td>
<td>11</td>
</tr>
<tr>
<td>1.9 Published Work</td>
<td>11</td>
</tr>
<tr>
<td><strong>2 Related Work</strong></td>
<td>13</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Software Parallelization</td>
<td>15</td>
</tr>
<tr>
<td>2.2.1 Loop level Parallelization</td>
<td>16</td>
</tr>
<tr>
<td>2.2.2 Method level Parallelization</td>
<td>16</td>
</tr>
<tr>
<td>2.2.3 JVM specific Parallelization</td>
<td>17</td>
</tr>
<tr>
<td>2.3 Pattern based approaches</td>
<td>18</td>
</tr>
<tr>
<td>2.4 Loop Profiling</td>
<td>19</td>
</tr>
<tr>
<td>2.5 Code Analysis</td>
<td>20</td>
</tr>
<tr>
<td>2.6 Summary</td>
<td>20</td>
</tr>
<tr>
<td><strong>3 Method Level Features</strong></td>
<td>23</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>23</td>
</tr>
</tbody>
</table>
3.2 Qualitative Features ............................................. 23
3.3 Quantitative Features ............................................. 24
3.4 Summary .......................................................... 25

4 Loop Profiling ......................................................... 26
  4.1 Introduction ...................................................... 26
  4.2 Motivation ......................................................... 26
  4.3 Internal Representation of Loops ............................... 28
  4.4 Identification of Single Loops .................................... 30
  4.5 Loop Forest Construction ......................................... 30
  4.6 Evaluation ........................................................ 32
  4.7 Summary .......................................................... 33

5 Instruction Patterns .................................................. 35
  5.1 Introduction ...................................................... 35
  5.2 Intermediate Representation of Instructions .................. 35
  5.3 Recognition of Instruction Patterns ............................ 36
  5.4 Inter-iteration Data Dependence Patterns ...................... 39
  5.5 Summary .......................................................... 41

6 Parallelization Methodology ......................................... 43
  6.1 Introduction ...................................................... 43
    6.1.1 Problem Formulation ........................................ 43
    6.1.2 Percentage Contribution Threshold ......................... 44
  6.2 Parallelization Methodology ..................................... 46
    6.2.1 Parallelization Criteria ..................................... 47
    6.2.2 Loop Identification ........................................... 48
    6.2.3 Loop Classification ............................................ 49
  6.3 Threading Framework .............................................. 49
  6.4 Parallelization of a Loop ........................................ 50
    6.4.1 Parallelizable Loop Selection ............................... 51
    6.4.2 Loop Extraction ............................................... 53
    6.4.3 Data Dependence Analysis .................................... 54
    6.4.4 Declaration of Run_{ik} Method ............................ 55
    6.4.5 Generation of Worker_{ijk} and Manager_{ijk} Classes .... 55
    6.4.6 Definition of Run_{ijk} Method ............................. 56
    6.4.7 Loop Replacement in Hotspot ................................ 56
  6.5 Summary .......................................................... 56

7 Case Studies .......................................................... 57
  7.1 Introduction ...................................................... 57
  7.2 Experimental Setup .............................................. 57
  7.3 Code Comprehension .............................................. 58
  7.4 Parallelization of JGF Benchmarks ............................. 59
    7.4.1 Long Running JGF Benchmarks .............................. 59
7.4.2 Short Running JGF Benchmarks ........................................ 60
7.5 Results Comparison ....................................................... 62
7.6 Summary ........................................................................ 64

8 Conclusion ........................................................................ 66

A SeekBin Details .................................................................. 68
A.1 Introduction .................................................................... 68
A.2 Java Instrumentation API ................................................ 68
A.3 Java Interactive Profiler .................................................. 70
A.4 ASM bytecode engineering library .................................. 70
A.5 How SeekBin Works ........................................................ 71
  A.5.1 High Level Algorithm .................................................. 71
  A.5.2 Detailed Steps ............................................................ 72
    A.5.2.1 Structure of premain method ................................. 72
    A.5.2.2 Structure of transform method ............................... 73
    A.5.2.3 Pre-requisites ........................................................ 73
A.6 Usage ............................................................................. 73
  A.6.1 Step 1: Profile the application (one time activity) ......... 73
  A.6.2 Step 2: Attach SeekBin and run the application .......... 74
  A.6.3 Available Options ...................................................... 74

References ............................................................................ 76
List of Figures

4.1 Loop forest in (a) transform\textsubscript{\textit{internal}} method of FFT benchmark (i.e. consists a single tree) (b) runiters method of MolDyn benchmark (consists a single tree) (c) matgen method of LUFact benchmark. The forest consists of three loop trees. ............................... 27

4.2 (a) Source code, and (b) bytecode of \textit{copySqrMatrix} method. Bytecode is marked with backward jumps. ............................... 29

4.3 Basic block level control flow graph of \textit{copySqrMatrix} method. ............................... 31

4.4 Algorithm to identify single loops in Java bytecode. ............................... 31

4.5 Algorithm to identify nested loops from Java bytecode. ............................... 32

4.6 Loops in eighteen Java Grande Forum benchmarks. ............................... 33

5.1 (a) Source code of a loop taken from \textit{SORrun} method of JGF SOR Benchmark (b) Bytecode and its IR tuples. ............................... 38

5.2 Parse tree of sample loop bytecode. Each node consists of an instruction pattern ID, pattern component ID or an independent instructions. ............................... 40

6.1 Parallelization overhead versus unattended time. \( T_{PC} \) value is obtained by subtracting unattended time from 100. ............................... 46

6.2 Selection of hotspots from JGF application benchmarks using \( T_{PC} = 90\% \). ............................... 46

6.3 Work flow of proposed parallelization methodology. ............................... 48

6.4 Class diagram showing the association of threading framework classes with the class containing hotspot method. ............................... 50

6.5 Definition of \textit{Worker\_ijk} class of Threading Framework code. “dynamic data members” in square brackets varies according to code characteristics of loop \( l_{ijk} \). ............................... 51

6.6 Definition of \textit{Manager\_ijk} class of threading framework. “dynamic data members” in square brackets varies according to code characteristics of loop \( l_{ijk} \). ............................... 52

6.7 Relevant Portion of call graph of Series Benchmark. It shows that \textit{TrapezoidIntegrate} calls \textit{thefunction} and itself called by \textit{Do}. Interprocedural relationships are presented using “\( \text{|==|} \)”. For example \textit{JGFkernel} is immediate parent of \textit{Do} but sibling of \textit{JGFvalidate}. ............................... 53

6.8 Steps of loop parallelization. (a) Source code of \textit{Do} method of Series Benchmark (b) Definition of \textit{run\_ijk} (c) Loop replacement in \textit{Do} method. ............................... 54
7.1 Scalability of long running application benchmark. .......................... 60
7.2 Scalability of short running application benchmarks. ......................... 61
7.3 Overall speedup as compared to sequential code. ............................. 62
7.4 Comparison of JIT-inspired and JOMP results of JGF (a) Arith, (b) Crypt, (c) Cast, (d) Math, (e) Series, (f) Method, (g) LUFact, and (h) SparseMatmult benchmarks. .................................................. 65
List of Tables

2.1 Summary of recent related work. ..................................... 22
3.1 Qualitative Features of Methods. ................................. 24
3.2 Quantitative Features of Methods. ............................... 25
4.1 Variation in $l_{sec}$ in case of selecting hotspot methods. ......... 33
5.1 Intermediate Representation of Bytecode Instructions. ............ 37
5.2 Variables used in sample code. ................................. 39
5.3 Instruction patterns recognized in sample code. .................. 41
6.1 Features of Potential Hotspots in JGF Series Benchmark. ......... 53
6.2 Relevant Features of $Do$ Method of JGF Series. ............... 55
7.1 Parallelization Potential of JGF Benchmark Applications. ......... 58
## Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Stands For</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CMP</td>
<td>Chip Multiprocessor</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collector</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>JGF</td>
<td>Java Grande Forum</td>
</tr>
<tr>
<td>JIT</td>
<td>Just In Time</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Development Kit</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
</tr>
<tr>
<td>JRE</td>
<td>Java Runtime Environment</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>TM</td>
<td>Transactional Memory</td>
</tr>
<tr>
<td>TLS</td>
<td>Thread-Level Speculation</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>
Abstract

Innovations in uniprocessor based systems scaled majority of single-threaded applications according to Moore’s law for decades. Sequential applications containing a single thread of execution always performed better on upcoming generation of uniprocessors. Eventually around 2003, chip density and circuit complexity increased power dissipation and it became impractical to make major improvements in processor clock rate. A viable solution sought was to put more than one processor cores instead of increasing clock rate of single processor. This led to wide spread production of commodity multicore systems. To take advantage of multiple cores, single-threaded applications need to be parallelized. The real challenge is for programmers who have to exploit multiple cores in an optimal way. A common approach is to spawn multiple parallel threads using shared address space. Threads are synchronized using lock-based mutual exclusion. Although lock-based threading becomes complex after a certain level, the solution is still viable for writing new applications from scratch. However, parallelization of existing sequential code is quite challenging. Manual porting or explicit parallelization of existing code requires significant effort due to key challenges of preserving sequential dependency, load balancing and synchronization. Introduction of new parallel programming languages and paradigms provided little help in catering existing code parallelization because they are primarily designed for source level changes and understanding of existing code is a non-trivial effort. Newer release of parallel programming languages introduce more robust constructs. However in case of existing code, we need to identify the points where these constructs could be injected for parallel execution so that compiler generate code accordingly. The goal here is to instrument bare minimum code to get maximum possible speedup. Runtime parallelization is quite promising in this regard. Applications generally consume significant fraction of execution time in a small amount of repetitive code. This repetitive code is commonly known as hotspot code. Hotspots of regular applications typically possess exploitable loop level parallelism. Java virtual machine profiles running applications to exploit these hotspots for applying optimizations. To get performance boost, hotspots are translated to native code by Just-in-time
(JIT) compiler. Using similar approach, we developed a methodology to identify hotspots and exploit their parallelization potential on multicore systems. Proposed methodology selects and parallelizes each DOALL loop that is either contained in a hotspot method or calls a hotspot method. As a case study, we analyzed eighteen Java Grande Forum (JGF) benchmarks to determine parallelization potential of hotspots. Parallelization of eight application benchmarks demonstrated a speedup of up to 7.6x on an 8-core system.
Chapter 1

Introduction

This thesis deals with the exploration and exploitation of hardware parallelism available in multicore systems. Hardware parallelism potential is realized by parallelizing software applications at most appropriate granularity. In this chapter, we present brief background, motivation, scope and aim of this work. Justification and main contributions of this work are also presented briefly. After presenting the overall structure of this thesis, we discuss the future research direction.

1.1 Background

Uniprocessor based systems remained dominant for decades. Software applications were written by considering a single thread of execution. Software engineers enjoyed performance boost with every release of new processor. Around 2003, processor vendors hit the wall due to the power consumption and heat generation [53]. Eventually, it became impractical to increase the transistor density on the processor chip. A practical solution to this problem is to put more than one processor cores instead of focusing on the improvements in clock rate of single processor per chip. This led to the proliferation of shared-memory multicore processor based systems.

Efficient utilization of multiple cores is the responsibility of software applications. Sequential applications could exploit at most a single core whenever scheduled. Only parallel applications, with multiple threads of execution, can saturate a
share-memory multicore system. Software parallelization remained active only in high performance computing (HPC) arena [13]. Majority of general purpose applications were written sequentially, because the target systems were uniprocessor based. Sequential applications were able to gain performance transparently on next breed of uniprocessors. But free lunch is over now, due to emergence of multicore processor based shared-memory systems. We need to develop parallel applications to harness the performance potential of multicore systems. Similarly, there is a dire need of parallel execution of legacy sequential applications on multicore systems.

1.2 Motivation

Multicore chips in servers, desktops, laptops, embedded systems and even in mobile devices pose a challenge for application developers [29]. Individual applications need to utilize multiple cores simultaneously in order to gain performance on multicore processor based systems. Multiple cores could be exploited by parallelizing applications in a variety of ways. In case of writing new applications, a convenient approach is to design and implement parallel algorithms explicitly [35, 36, 58, 81, 94]. This is probably the most intuitive and relatively easy approach. On the other hand, parallelization of existing sequential code sometimes need complete redesign. Redesigning existing code for parallel execution is quite cumbersome and often in-effective process. Auto-parallelizing compilers [14, 27, 85, 103] do help in this regard. Majority of auto-parallelizing compilers use static analysis [49, 62] and often require source level hints. Some compilers use profiler feedback [21, 66] and heuristics [52, 97] to optimize parallelization efforts. However, dependence on source code level changes make this approach limited, especially when the source code is either not available (as in case of commercial applications) or is too complex to understand and modify.

Ideally, a parallelization effort should focus only on that part of code that is actually executed, under a certain configuration on a particular target system. In real sense, a runtime system is able to meet all these requirements because it could have access to the dynamic state of running applications. Runtime systems
parallelize applications either speculatively [44–47, 63, 104] or non-speculatively [3, 7, 9, 17, 54, 99, 100]. In speculative parallelization, parallel tasks are assumed independent and run by using either thread level speculation (TLS) [101] or transactional memory (TM) [39]. Results are not committed if runtime system detects dependence violation(s) by some task(s). Dependence violations are resolved by squashing and re-running some of the parallel tasks. Speculative parallelization is a best effort approach. Sequential code is parallelized if possible, or executed sequentially otherwise. However, logging memory accesses to support commit/rollback mechanism, puts extra load on memory. Similarly, task squashing and re-execution incurs computational overhead. Some limitations of speculative parallelization are discussed in [23]. To avoid the limitations of speculative parallelization, we advocate the adoption of just-in-time (JIT) parallelization approaches by using dynamic compilers. A dynamic compiler is typically used to compile interpretable intermediate code to native code, while the program is running. It selects code hotspots that are most frequently executing code regions like method or loop bodies. Hotspots are compiled once and generated code is cached to gain performance by avoiding code generation on subsequent invocations (or iterations).

1.3 Research Questions

Parallelization of existing sequential code poses various challenges that need to be described precisely. In this thesis we tried to address the following research questions (RQ):

- **RQ 1:** How existing sequential code could utilize multiple processor cores available in state-of-the-art multicore systems?

  **Answer:** Parallelization enables sequential applications to exploit multiple cores. It involves extraction of parallel tasks form sequential code and spawning multiple threads or processes to execute parallel tasks on available cores simultaneously. This thesis presents a parallelization methodology to achieve this goal.
• RQ 2: How to parallelize existing code?

**Answer:** Repetitive code portions like method calls and loop iterations are typically targeted for parallel task extraction. Method level parallelization exploits coarse grain parallelism and loop level parallelization exploits fine grain parallelism. Both approaches have their own pros and cons. however, in both cases dependences between parallel tasks should be resolved to preserve sequential semantic. Focus of this thesis is on loop level parallelization and it is justified in section 1.6.

• RQ 3: Which parallelization paradigm is more suitable? method level or loop level?

**Answer:** It depends on parallelism granularity of existing code. One approach to determine it is by profiling the running application. For example, if a method consume significant amount of execution time when called 1-2 times and have some loops in it, then loop level parallelization is more suitable, as elaborated in Chapter 6 of this thesis.

• RQ 4: How to reduce the effort and overhead of parallelization?

**Answer:** Pick few most time consuming methods as hotspots. Use different characteristics of hotspot methods to find most suitable parallelization candidates. To address this question, a parallelization criteria and catalogs of method level qualitative and quantitative features are presented in this thesis.

• RQ 5: How to resolve dependences between parallel tasks?

**Answer:** This thesis deals with data dependences using instruction patterns, as elaborated in Chapter 5.

• RQ 6: How to manage life cycle of parallel tasks?

**Answer:** A threading framework is presented in this thesis for workload partitioning and life cycle management of parallel tasks.
1.4 Scope and Aim

A loop level parallelization methodology for sequential Java code is presented in this thesis. The idea is inspired by the hotspot selection mechanism of JIT compiler of Java virtual machine (JVM). Proposed methodology tries to parallelize each DOALL loop that is either defined in a hotspot method or calls a hotspot method. We developed a tool that hooks to JVM class loader for profiling and parsing Java classes at runtime. In current state, code transformation is done explicitly. The aim is to integrate this methodology into an open source JIT compiler front end.

1.5 Thesis Contributions

Five major contributions are described in this section along with the background (or motivation) for each contribution.

Contribution 1: Method level qualitative and quantitative features

Background. Nano-patterns [95] are binary properties of Java methods that could be used for characterization and classification of method code. Singer et al. [95] extracted a catalog of 17 nano-patterns using a static tool. The tool extracted nano-patterns statically from a class file that is given as a command line argument.

Contribution. We adapted and extended the concept of nano-patterns in terms of qualitative features. Qualitative features are extracted using SeekBin [73] that hooks to JVM class loader and parse classes just before loading in the JVM. We use profiler feedback to select hotspot methods for feature extraction. Qualitative features are boolean variables that represent the presence/absence of different characteristics of Java methods. We present a catalog of 32 qualitative features. In addition, we also extract 15 quantitative features to quantify code characteristics of the method, like number of loops, number of instructions and so on.

Contribution 2: Bytecode level loop profiling mechanism

Background. Profiling is typically done at method level to do inter-procedural analysis and runtime information gathering. In this work, we needed an integrated bytecode level loop profiling mechanism to gather customized information about
each loop execution in the vicinity of hotspot methods.

**Contribution.** We implemented loop profiling mechanism to generate a loop forest of a method. In loop forest of a method, the nested loop is represented as a multi-node tree, where each node of the tree is a loop. Single loops are represented essentially as single-node trees. Loop profiling is done during parsing phase. Loop information is not only used to initialize relevant qualitative and quantitative features but also used for loop unrolling and workload distribution during parallelization.

**Contribution 3: Bytecode instruction pattern composition scheme**

**Background.** For a parallelization effort, it is mandatory to preserve sequential semantic of the existing code to ensure the correctness. Data and control dependences between parallel tasks could potentially compromise the sequential semantic. In executable code, multiple instructions are typically executed in a specific order to perform a certain high level action. For example in case of Java bytecode, reading a double precision floating-point array element requires at least three instructions ALOAD, ILOAD, DALOAD to be executed in this specific order. ALOAD instruction loads the address of array on the stack, ILOAD loads the array index and DALOAD then gets the value stored at that index. To resolve dependences, we need to identify all such fundamental instruction patterns and their high level composition.

**Contribution.** We present an instruction pattern composition scheme for Java bytecode instruction-set. Patterns are composed by using an intermediate representation (IR) of bytecode instructions.

**Contribution 4: Threading framework**

**Background.** Multi threading support is available in almost all high level languages (including Java). Modern thread pool based implementations of Java are quite restrictive and involve number of classes to be generated. In our case, using primitive threading constructs is more robust and reliable approach. We need a threading framework that have minimal number of classes to be generated at runtime.
Contribution. We developed a threading framework consisting of only two classes. The idea of our framework is inspired by javar [11]. The main difference is that javar uses five classes to parallelize a loop at source level. We need to generate only two classes directly in bytecode because their composition depends on the target loop structure.

Contribution 5: Parallelization Methodology

Background. JIT compilation technology [8] was introduced to optimize Java code execution. JVM profiles running applications to select most frequently called code regions as hotspots. Backend of JIT compiler dynamically translates hotspot code to native code. Translated code of hotspot is cached to reuse on next invocation (or iteration).

Contribution. We developed a parallelization methodology to parallelize DOALL loops, provided that the loop either lies in a hotspot method or a hotspot method is called in the loop body. The methodology is suitable for integration in the front end of JIT compiler that deals with bytecode.

1.6 Justification for the Research

Java is one of the main stream languages, with a robust managed runtime environment. Use of JVM for runtime parallelization is quite beneficial because of its multithreaded design, JIT compilers and garbage collectors [29]. Although .NET framework is a good competitor of Java but it is a proprietary solution. We picked JVM because it is open source. A wider community could take benefit of this research by keeping it open source. Our ultimate goal is to make use of JIT compiler front-end for parallelization. Front-end of JIT compiler takes Java bytecode as input, applies high level optimizations and feed transformed bytecode to the back-end [75]. Back-end is responsible for low level optimization and native code translation for target hardware. We have two reasons for selecting loop level parallelization in this context. First, we observed that by setting a threshold on application’s execution time, we are left with only a few hotspot methods [73]. For example, in JGF Crypt benchmark [67], we observed that a single method is consuming $\sim 90\%$ time of the application. Situation like this leaves almost no
room for method level parallelization even on dual core system. This observation particularly holds for recursive methods. For example, if a method is calculating factorial of a number, it makes a single recursive call on each invocation. This method cannot be parallelized because recursive calls are inherently sequential. Each invocation waits for the results of subsequent recursive call. However, \( n \)-way recursive methods could be parallelized but their parallelization potential is limited by the value of \( n \). For example, if a method is calculating Fibonacci numbers, each invocation makes two recursive calls i.e. \( n = 2 \). In this case, two recursive calls could be executed in parallel. On a multicore system with more than 2 cores, parallelizing recursive calls could exploit at most 2 cores only. Hence, \( n \)-way recursive methods could exploit at most \( n \) cores of a multicore system, if parallelized using method level parallelization.

Second reason for selecting loop level parallelization is that modifications applied at loop level remains local to the method and does not change method signature. Modification in method signature means to modify each call site of the method. In this case, we will be dealing with whole application and taking no benefit of hotspot selection. Same argument applies to method level parallelization that exploit coarse grain parallelism using inter-procedural analysis of entire application. Regarding DOALL loops, our current focus is on a single goal: achieve whatever parallelism can be realized from sequential hotspots without any effort on the part of exploring hidden parallelism.

### 1.6.1 Application Areas

Apart from parallelization of sequential Java code, this work could be used in various other research domains. For example, qualitative and quantitative features could be used for classification and information theocratic software engineering. Some of potential application areas are briefly described below.

- **Parallelization of existing code**: This work facilitates the parallelization of general purpose regular Java applications. DOALL loops are typically found in compute intensive workloads like science and engineering applications. JVM based long running applications in different domains (e.g. big
data analytics, security applications, data compression), contain repetitive compute intensive tasks. Such applications will take benefit of this research because JVM finds more chances to warm up code and select hotspots in long running applications. Similarly, enterprise applications and cloud computing workloads would also take benefit of this work.

- **Code Analysis:** Instruction patterns scheme and method level features has a usage in the domain of reverse engineering and code analysis. It could be used to re-engineer or re-modularize Java code. Pattern composition scheme could be adapted for other instruction set architectures. One potential use of the scheme is in the area of malicious code detection.

- **Logging and Tracing:** Seekbin tool [73] is able to instrument Java classes. One possible application is to generate runtime logs. Bytecode parser of SeekBin could be employed in trace-based software engineering techniques.

### 1.6.2 Future Research Areas

- Use of dynamic compilers for parallelization: Integration to a JIT compiler front end would support runtime parallelization.

- Parallelization of irregular Java applications: Parallelization of DOACROSS loops that are typically found in irregular applications. Use of both JIT compiler and garbage collection will help in this case.

- Use of statistical methods: Catalogs of qualitative and quantitative features will help in composing statistical patterns. Parallelization efforts could be augmented by using machine learning based techniques to analyze parallelization potential.

- Reverse engineering: Compiled Java class parsing could be used for reverse engineering of Java code for different purposes like architecture recovery and code understanding.
1.7 Thesis Overview

Overall organization of the thesis is given below.

- **Chapter 1: Introduction** – Background and motivations of this research work is presented in chapter 1. Scope, aim and justification for this work is described briefly.

- **Chapter 2: Related Work** – This chapter provides the existing work related to the topics of this thesis i.e. software parallelization, code analysis, pattern based approaches.

- **Chapter 3: Method Level Features** – Brief description of each qualitative and quantitative feature along with its potential usage is given in this chapter.

- **Chapter 4: Loop Profiling** – This chapter gives the details of bytecode level loop profiling mechanism. Algorithms and data structures used in this context, are elaborated.

- **Chapter 5: Bytecode Instruction Patterns** – Intermediate representation of bytecode instruction set and its use in instruction pattern composition is discussed in this chapter.

- **Chapter 6: Parallelization Methodology** – Loop level parallelization approach is elaborated in this chapter. It contains details of threading framework, profiling and parsing phases, overall work flow and modification steps.

- **Chapter 7: Case Studies** – Experimental setup, results and case studies are discussed in this chapter.

- **Chapter 8: Conclusions** – Thesis is concluded in this chapter by providing a summary and future work.

- **Appendix A: SeekBin Tool** – Details of Seekbin design and usage are given in this appendix.


1.8 Future Research Directions

In future, parallelization methodology and other contributions discussed in this thesis will be integrated to the front end of JIT compiler of an open source JVM. This is a work in progress under a project, funded by Higher Education Commission (HEC) of Pakistan. After this project, we might move on to the other research areas discussed in section 1.6.2.

1.9 Published Work

The thesis is based on some of publications listed below.


Chapter 2

Related Work

2.1 Introduction

Software parallelization is a decades old area of research [98] but it was consigned only to high performance computing community. Majority of general purpose code was written for uniprocessor based systems using sequential execution model. Emergence of multi-core systems resurged the interest in parallel software development. Huge amount of sequential legacy code demands automated and semi-automated techniques to be developed because explicit parallelization of sequential code is a nightmare. A lot of literature is being produced in this area of research and this thesis is one of them. Review of closely related work is presented in this chapter. It is not a summery of all literature available on the topic but it covers only the subjects directly related to this work.

Background. Since long, applications have been designed and developed for sequential execution on uniprocessor based systems. Advent of multicore systems changed the computing scenario and sequential execution became a legacy [29]. To exploit multiple cores simultaneously, it is necessary to parallelize existing sequential code so that it can spawn at least as many parallel tasks as there are cores in the system. Parallelization of sequential code is done either manually or automatically [29]. Advantage of manual parallelization (i.e. parallel programming) is that the programmer has full control over parallel execution of the code. However, it
becomes impractical for complex applications especially when only static analysis is used. The worst case is the scenario when legacy software architecture has been evolved but the documentation is not updated. Similarly, proprietary code is often delivered in binary format and modification of binary code is relatively a difficult task.

To gain maximum possible speedup with bare minimum modifications, the challenging task is to identify most promising portions of code. Once the most beneficial code portion is identified, explicit or implicit parallel programming constructs, like pthreads [30] or OpenMP [25] pragmas could be used to parallelize the code. Similar approach is required for automatic parallelization by using an auto parallelizing compiler. Parallelizable code is annotated and the compiler implicitly parallelizes the annotated code portions. For example, code annotated with OpenMP pragmas could be parallelized by using GCC compiler with -fopenmp command line argument. Auto parallelizing compilers work fine for regular applications like numerical or scientific applications. Majority of irregular applications requires deep analysis of control and data dependences and auto-parallelizing compilers are not intelligent enough to do this analysis. Also, the source level compilers generally work offline and uses static analysis for code optimization. Offline techniques are limited because it could not deal with the applications that use dynamically linking libraries (dlls). Similarly, dynamic state of running code depends on configuration parameters and user input, which could not be inferred at compile time.

Limitations of source level parallelization techniques advocates the use of runtime systems like virtual machines (VMs), speculative and transactional frameworks for parallelization of sequential executable code. User-space virtual machines are typically designed to produce portable user code. Code written in VM-based languages (like Java, Ruby, Groovy, Scala, Smalltalk etc.) is emulated on a virtual architecture and often uses dynamic (i.e. JIT) compilers to translate interpretable code into native code to gain performance. Use of dynamic compilers for parallelization of sequential code is a more viable solution as compared to the source level compiler. This thesis is an attempt to devise a loop level parallelization methodology that could be integrated in the front-end of JIT compiler of Java
virtual machine. To the best of our knowledge, we did not find any effort on
the use of JIT compiler for loop level parallelization especially using method level
qualitative/quantitative features, instruction patterns and profiler feedback. Loop
level parallelization exploits fine-grained parallelism potential of the application
as compared to method level parallelization which exploits coarse-grained parallel-
ism. justification for the use of loop level parallelization in this context is given
in section 1.6.

Chapter Structure. Related work is organized in different sections. Software
parallelization especially loop level, method level, and JVM specific parallelization
is given in Section 2.2. Pattern based approaches are discussed in section 2.3.
Previous work on loop profiling and code analysis is presented in section 2.4 and
2.5, respectively. The chapter ends with a summery of recent related work that is
presented as a comparison table (using different parameters).

2.2 Software Parallelization

Parallelization of sequential code is one of the oldest areas of computer science
[98] and it is still hot [6, 29], thanks to the emergence of multicore and many-core
systems. Parallelization could be divided into two broad categories: domain spe-
cific and general purpose. Domain specific parallelization involves domain experts
and use domain knowledge during parallelization process. For example, Page [57]
is a parallelization framework for genome analysis applications. Similarly, Kiefer
et al. [51] parallelizes a real-time audio application. On the other hand, general
purpose parallelization efforts exploit programming language constructs like loops
or methods. Loop level parallelization exploits fine grain parallelism. Exploiting
fine grain parallelism is more beneficial but it requires more effort as compared to
course grain parallelization.
Chapter 2. Related Work

2.2.1 Loop level Parallelization

Loop level parallelism could be exploited in a variety of ways. For example, DO-cyclical [105] is a cyclic multi-threading (CMT) parallelization approach. It employs a priority-based scheme to gain performance by minimizing cross-core communication latency of loop execution. Decoupled software pipeline (DSWP) [80] is a technique to parallelize loops by extracting non-speculative decoupled threads from binary code. Java-DSWP (jDSWP) [64] is the Java version of DSWP that works at source-level. Hallou et al. [37] provides a mechanism to automatically convert loop vectorization from an older to newer SIMD extensions of CPU. They demonstrate the conversion of various loops compiled for x86 SSE to work on AVX. However, the tool is specific to x86 architecture and not suitable for portable code produced by Java compiler. Abdullah, M. [1] patented a just in time optimizer to execute code from guest binaries but its main focus is not on single application performance optimization. Schulte et al. [92] patented a tracing just-in-time compiler system for automatic parallelization. Input program could be in any language provided that the program is compiled into an intermediate format. A Loop is considered hot (i.e. parallelization candidate) if its number of iterations exceed certain threshold value. Our work differ in such a way that we use percentage contribution threshold on the time consumed by a hotspot method. Parallelization candidate loop might be within this hotspot or the hotspot is called in the loop body. Uniqueness of our approach is the use of method level features in parallelizable loop selection criteria and instruction patterns for dependence analysis. Limitation of our parallelization methodology is that it currently parallelizes only DOALL loops. However, the approach is quite extensible and other use cases can easily be integrated. For example, method level features could be used to employ machine learning techniques for intelligent loop classification.

2.2.2 Method level Parallelization

Majority of speculative parallelization efforts employ method level parallelization to exploit coarse grain parallelism [4, 18, 20, 22, 38, 55, 83, 84, 88]. Kumar et al. [56] proposes a speculative dynamic vectorizer that can assist compiler based
static vectorization. The algorithm speculatively deals with ambiguous memory accesses to expose vectorization potential. It demonstrates the performance benefits of using static+dynamic (combined) vectorization approach. However, it is not intended for Java code and works for the code produced by GCC compiler collection. Despite the advancements in dynamic instrumentation technologies, supporting runtime parallelization is still limited [31], even by using the method level parallelization. In our case, method level parallelization is not beneficial as explained in section 1.6.

2.2.3 JVM specific Parallelization

Bytecode level parallelization has been tried since the inception of Java language [10]. However, due to lack of instrumentation and on-the-fly class modification APIs, the effort relied on static modifications of single class at a time without considering profiler feedback. Currently, JIT parallelization is being revisited, thanks to the proliferation of multicore/manycore systems and advancements in virtualization technologies [41, 60, 61, 77–79]. E. sterlund and W. Lwe [77–79] exploit JVM’s garbage collector to support JIT parallelization but the scope of this effort is limited. It only deals with the parallelization of pure methods. A pure method is one that does not change state [79]. A. Leung, et al. [60, 61] proposed auto-parallelizing extensions for Java JIT compiler so that it could find parallelizable loops and compile them for parallel execution on multicore CPU and general purpose graphic processing unit (GPGPU). However, code generation depends on RapidMind [70] and GPU hardware. Huang et al. [42] studied the effect of number of cores on JIT compilation policies. The investigation is carried out by implementing new configurations for Oracle Hotspot JVM. The outcome of this investigation is a new compilation policy, called throttling compilation. The policy stops code interpretation in case of longer compilation queue. However, this effort does not target parallelization of sequential code. Noll et al. [74] proposed Java language extension to facilitate compiler in parallelization related optimizations. Aparapi [33] provides a programming model to write Java code for heterogeneous systems. Albert et al. [2] presented an auto-vectorization mechanism for Aparapi Java program. It uses source-to-source transformation and takes benefit of SIMD
instructions (if available). Aparapi-UCores [93] is another open source effort to produce OpenCL code for execution on heterogeneous cores. HJ-OpenCL [40] accelerates Java code by using parallelization constructs of Habanero-Java (HJ) language [19, 43] and generating OpenCL code. Yang Yu et al. [106] presents the results of porting OpenJDK on Xeon Phi using HotSpot VM as kernel execution engine. It evaluates multithreaded Java Grande Forum benchmark suite to quantify performance and scalability issues of JVM on Xeon Phi. It implements a semiautomatic vectorization capability to achieve a speedup of 3.4x on 60 Phi cores, as compared to Xeon CPU. Our work aligns with [106] but we do not deal with code acceleration using custom accelerators. Ango et al. [5] proposed a JVM based framework that works at bytecode level using transactional memory. However, it employs method level parallelization.

We did not find any JVM specific parallelization approach that target parallelizable loops as per our selection criteria. The reason is that we restrict to most beneficial loops by using method level features extracted from real code. Theoretical approaches are often based on certain assumptions that might overlook the corner cases that appear during actual execution of the code.

2.3 Pattern based approaches

A Study [50] was carried out on 135 parallel open source implementations in major object oriented languages (C++, Java, and C#). The study reveals that almost similar code patterns (i.e. parallel constructs) are used regardless of the language. For example, AutoFuture [68] is an approach that select and re-engineer parallelizable source code regions for execution on multicore. The approach uses some recurring parallelization patterns but it does not consider runtime information to aid in selection process. Jimborean et al. [48] proposed a framework to speculatively parallelize scientific loop nests by using algorithmic skeletons. However, the framework works with LLVM [59] compiler and dependent on x86-64 runtime system. To facilitate optimistic parallel execution, Cledat et al. [24] presented a trace-driven approach to data-access pattern processing for regular and irregular data structures. Fathy et al. [32] presented a framework for ranking applications
according to the availability of special instruction patterns in its code. This effort aligns with our work only in instruction pattern recognition activity. Invariant-induced Pattern based Loop Specialization (IPLS) [76] is an automated technique to recognize recurring patterns across multiple iterations of hot loops. Though not directly related to parallelization, the technique highlights the usage of patterns in loop optimization. We are using instruction patterns to classify DOALL loops. Our main contribution is an instruction pattern composition scheme which could be used for various other reverse engineering activities. It lays a foundation for more robust bytecode rewriting activities including the implementation of different parallelization paradigms.

2.4 Loop Profiling

In literature, we found several efforts on loop profiling. Moseley T. et al. [71, 72] worked on loop profiling using multiple techniques. They first presented LoopProf [71] that profiles unmodified binaries to collect loop related information. LoopProf implements a stack-based algorithm using Intels dynamic instrumentation API called pin [65]. Moseley T. et al. extended their work in [72] to determine execution times of loops using both instrumentation based and sampling based approaches. Other similar efforts on dynamic loop monitoring and determination of loop nests and data dependences could be found in [89–91]. However, these efforts are architecture-specific and could not deal with Java class file format. De Alba M. R. et al. [28] observed dynamic behavior of loops to predict execution path on each iteration and potential loop exit. However, its primary focus is to speculate multiple issues of loop iterations to take advantage of wide-issue architectures. Suresh D. C. et al. [96] presented a collection of profiling tools for combined function and loop profiling. However, the tools depend upon either instruction set simulator or compile-time instrumentation (of GCC). Dash S. K. et al. [26] used loop profiling to estimate instruction cache hit rate. The technique uses one-time loop profiling of weighted control flow graph of application. There is another loop profiling technique [82] that instruments loop containing code so that it could emit profiling information on execution.
Although all these efforts are related to our work but none of these efforts focus on intermediate code that is going to be consumed by a JIT compiler. Our integrated loop profiling mechanism extracts loop information along with the other class parsing activities like method level feature extraction and IR generation.

2.5 Code Analysis

Uroboros [102] is a tool to instrument static stripped binaries. It could disassemble and reassemble executables automatically. But it works on ELF binaries instead of Java class file format and it does not focus explicitly on parallelism exploration. Molitorisz et al. [69] presents an automatic dynamic profiler and an empirical study of about one million lines of code to expose five use cases of parallelization potential in object-oriented data structures. PADRONE [86] is another analysis platform for dynamic binary optimization. Similar to SeekBin [73], it hooks to running applications for profiling and optimization of binary code. However, PADRONE is not platform independent and focuses only on x86 binaries.

We presents SeekBin [73] tool that hooks class loader of standard JVM and emits method level features and intermediate representation of bytecode instructions. It implements a loops profiling mechanism and is capable of instrumenting Java classes at runtime.

2.6 Summary

Software parallelization is currently an active area of research due to the availability of multicore systems as commodity hardware. A lot of literature available on parallel execution of sequential code on multicore systems. This section summarizes most recent related articles published during 2010-2016. Articles are compared on the basis of certain parameters as shown in the comparison Table 2.1. We found that research on software parallelization is progressing in multiple directions. Researchers are developing language extensions, techniques and frameworks. Parallelism is being exploited at one granularity level or other and even at multiple levels. To highlight the contributions of this thesis, articles are also reviewed to determine if they use method level features, instruction patterns and
profiler feedback. Majority of them are not pattern based approaches and have additional requirements for their proper working (Table 2.1). Similarly, some exploits method level parallelism or work at source level. In case of runtime parallelization, some uses speculation or architecture specific binaries.

Our approach stands out due to the use of method level features and instruction patterns. It incurs no overhead due to the chance of mis-speculation, or due to the complex loop transformations because it deals with parallelization of DOALL loops only. Selection of loop level parallelization has been justified in section 1.6. Our approach works with standard JVM and does not require any additional hardware or software component.
### Table 2.1: Summary of recent related work.

<table>
<thead>
<tr>
<th>Article</th>
<th>Outcome</th>
<th>Granularity</th>
<th>Uses method Level Features</th>
<th>Uses Instruction Patterns</th>
<th>Code Level</th>
<th>Profiler Feedback</th>
<th>Runtime Parallelization</th>
<th>Uses speculation</th>
<th>Add. Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cave 2011</td>
<td>Language extension</td>
<td>Both</td>
<td>NO</td>
<td>NO</td>
<td>Java Source code</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>HJ compiler</td>
</tr>
<tr>
<td>Cledat 2012</td>
<td>Tracing framework</td>
<td>Loop level</td>
<td>NO</td>
<td>NO</td>
<td>C/C++ source code</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>LLVM</td>
</tr>
<tr>
<td>Saad 2012</td>
<td>Virtual Machine</td>
<td>Both</td>
<td>NO</td>
<td>YES</td>
<td>Java bytecode</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>Hydram</td>
</tr>
<tr>
<td>Osterlund 2012</td>
<td>GC framework</td>
<td>Method level</td>
<td>NO</td>
<td>NO</td>
<td>Java bytecode</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>Pure methods</td>
</tr>
<tr>
<td>Noll 2012</td>
<td>Language extension</td>
<td>Method level</td>
<td>NO</td>
<td>NO</td>
<td>Java Source code</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>JikesRVM</td>
</tr>
<tr>
<td>Molitorius 2012</td>
<td>Technique</td>
<td>Method level</td>
<td>NO</td>
<td>YES</td>
<td>Java Source code</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Loureiro 2013</td>
<td>Language + API</td>
<td>Loop level</td>
<td>NO</td>
<td>NO</td>
<td>Java Source code</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Hayashi 2013</td>
<td>Virtual Machine</td>
<td>Loop level</td>
<td>NO</td>
<td>NO</td>
<td>Java Source code</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>HJ compiler, OpenCL supported Device</td>
</tr>
<tr>
<td>Elder 2013</td>
<td>Study</td>
<td>Loop level</td>
<td>NO</td>
<td>YES</td>
<td>Binary code</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>ISA simulator</td>
</tr>
<tr>
<td>Leung 2013</td>
<td>Compiler extension</td>
<td>Loop level</td>
<td>YES</td>
<td>NO</td>
<td>Java Byte code</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>X86-64 runtime system, source level pragma</td>
</tr>
<tr>
<td>Jimborean 2014</td>
<td>Framework</td>
<td>Loop level</td>
<td>YES</td>
<td>Binary code</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>LLVM compiler, x86-64 runtime system, source level pragma</td>
<td></td>
</tr>
<tr>
<td>Imam 2014</td>
<td>API</td>
<td>Both</td>
<td>YES (as Abstract Execution Metrics)</td>
<td>NO</td>
<td>Java Source code</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>Java 8, Habanero-Java (HJ) compiler</td>
</tr>
<tr>
<td>Albert 2014</td>
<td>API + tool</td>
<td>Loop level</td>
<td>NO</td>
<td>NO</td>
<td>Java Source code</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>Aparapi, SIMD instructions</td>
</tr>
<tr>
<td>Kiefer 2015</td>
<td>Case study</td>
<td>Method level</td>
<td>NO</td>
<td>NO</td>
<td>Not mentioned</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>RESCON simulation tool</td>
</tr>
<tr>
<td>Segal 2015</td>
<td>Framework</td>
<td>Loop level</td>
<td>NO</td>
<td>NO</td>
<td>Java Source code</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>Aparapi, openCL supported Device</td>
</tr>
<tr>
<td>Hallou 2015</td>
<td>Technique</td>
<td>Loop level</td>
<td>YES</td>
<td>Binary code</td>
<td>YES (static)</td>
<td>YES</td>
<td>NO</td>
<td>x86 SIMD extensions, SSE, AVX</td>
<td></td>
</tr>
<tr>
<td>Anjo 2016</td>
<td>Framework</td>
<td>Method level</td>
<td>NO</td>
<td>NO</td>
<td>Java byte code</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>Speculation and recovery hardware</td>
</tr>
<tr>
<td>Kumar 2016</td>
<td>Algorithm</td>
<td>Loop level</td>
<td>NO</td>
<td>NO</td>
<td>Binary code</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Mustafa 2016</td>
<td>Technique + tool</td>
<td>Loop level</td>
<td>YES</td>
<td>YES</td>
<td>Java byte code</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 3

Method Level Features

3.1 Introduction

Feature is a distinctive characteristic of an entity. Features are used as a measure- ment to represent data. In this thesis, we present catalogs of method level qualitative and quantitative features. Method level features are particularly significant in the area of code comprehension and understanding. Code understanding is necessary for reverse engineering and re-engineering activities like architecture recovery, restructuring, parallelization and optimization.

3.2 Qualitative Features

Qualitative features are binary variables to represent different characteristics of the method. Each qualitative feature indicates the presence (or absence) of a specific characteristic of method, as described in Table 3.1. For example, if a method does not contain loops then the feature \textit{LOOPY}, in Table 3.1, will be equal to zero (and one otherwise). The idea of qualitative features is inspired by Nano-patterns that were proposed to characterize and classify Java methods [95]. Catalog of qualitative features is constructed by extended catalog of Nano-patterns from 17 to 32, and giving them more compact and descriptive names. In a previous work [73], we used qualitative (i.e. binary) features to analyze speculative parallelization potential of Java applications. We showed that binary features are very important decisive factors for runtime qualitative analysis of parallelization
potential of methods. Qualitative features in Table 3.1 are related to the method signature and body. The features are generic in nature and could be used in any software reverse/re-engineering activity.

Table 3.1: Qualitative Features of Methods.

<table>
<thead>
<tr>
<th>ID</th>
<th>Feature</th>
<th>If value is 1 then the method...</th>
</tr>
</thead>
<tbody>
<tr>
<td>f0</td>
<td>NO_ARGS</td>
<td>Takes no arguments</td>
</tr>
<tr>
<td>f1</td>
<td>VALUE_ONLY_ARGS</td>
<td>Takes only pass-by-value arguments</td>
</tr>
<tr>
<td>f2</td>
<td>REF_ONLY_ARGS</td>
<td>Takes only pass-by-reference arguments</td>
</tr>
<tr>
<td>f3</td>
<td>MIXED_ARGS</td>
<td>Takes any argument</td>
</tr>
<tr>
<td>f4</td>
<td>ARRAY_ARGS</td>
<td>Takes one or more array arguments</td>
</tr>
<tr>
<td>f5</td>
<td>NO_RET</td>
<td>Returns void</td>
</tr>
<tr>
<td>f6</td>
<td>VALUE_RET</td>
<td>Returns primitive value</td>
</tr>
<tr>
<td>f7</td>
<td>REF_RET</td>
<td>Returns reference value</td>
</tr>
<tr>
<td>f8</td>
<td>STATIC</td>
<td>is static</td>
</tr>
<tr>
<td>f9</td>
<td>RECUR</td>
<td>is recursive i.e. calls itself</td>
</tr>
<tr>
<td>f10</td>
<td>LOOPY</td>
<td>code contains at least one loop</td>
</tr>
<tr>
<td>f11</td>
<td>NESTED_LOOPY</td>
<td>contains nested loops</td>
</tr>
<tr>
<td>f12</td>
<td>EXCEPT</td>
<td>may throw an unhandled exception</td>
</tr>
<tr>
<td>f13</td>
<td>LEAF</td>
<td>does not contain any call site</td>
</tr>
<tr>
<td>f14</td>
<td>OBJ_C</td>
<td>creates new objects</td>
</tr>
<tr>
<td>f15</td>
<td>FIELD_R</td>
<td>reads value of a class field</td>
</tr>
<tr>
<td>f16</td>
<td>FIELD_W</td>
<td>writes value to a class field</td>
</tr>
<tr>
<td>f17</td>
<td>TYPE_M</td>
<td>uses type casts or instanceof operator</td>
</tr>
<tr>
<td>f18</td>
<td>NO_BR</td>
<td>contains straight line code</td>
</tr>
<tr>
<td>f19</td>
<td>LOCAL_R</td>
<td>reads value(s) of local variable(s)</td>
</tr>
<tr>
<td>f20</td>
<td>LOCAL_W</td>
<td>writes value(s) of local variable(s)</td>
</tr>
<tr>
<td>f21</td>
<td>ARRAY_C</td>
<td>creates a new array</td>
</tr>
<tr>
<td>f22</td>
<td>MDARRAY_C</td>
<td>creates a new multidimensional array</td>
</tr>
<tr>
<td>f23</td>
<td>ARRAY_R</td>
<td>reads value(s) from array(s)</td>
</tr>
<tr>
<td>f24</td>
<td>ARRAY_W</td>
<td>writes value(s) to array(s)</td>
</tr>
<tr>
<td>f25</td>
<td>THIS_R</td>
<td>reads field value(s) of the object on which it is called</td>
</tr>
<tr>
<td>f26</td>
<td>THIS_W</td>
<td>writes field value(s) of the object on which it is called</td>
</tr>
<tr>
<td>f27</td>
<td>OTHER_R</td>
<td>reads value(s) of field(s) of other object(s)</td>
</tr>
<tr>
<td>f28</td>
<td>OTHER_W</td>
<td>writes to field(s) of other object(s)</td>
</tr>
<tr>
<td>f29</td>
<td>SFIELD_R</td>
<td>reads value(s) of static fields</td>
</tr>
<tr>
<td>f30</td>
<td>SFIELD_W</td>
<td>writes value(s) to static field(s)</td>
</tr>
<tr>
<td>f31</td>
<td>SAMENAME</td>
<td>calls some overloaded method of same name</td>
</tr>
</tbody>
</table>

3.3 Quantitative Features

Presence of a particular characteristic of method potentially necessitates the quantification of that characteristic. For example, if a method contains loops (i.e.
Table 3.2: Quantitative Features of Methods.

<table>
<thead>
<tr>
<th>ID</th>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f32</td>
<td>FIELDS</td>
<td>No. of fields touched in the method</td>
</tr>
<tr>
<td>f33</td>
<td>SFIELDS</td>
<td>No. of static fields touched in the method</td>
</tr>
<tr>
<td>f34</td>
<td>CALLS</td>
<td>No. of method calls in the method</td>
</tr>
<tr>
<td>f35</td>
<td>JUMPS</td>
<td>No. of jumps (jump instructions) in the method</td>
</tr>
<tr>
<td>f36</td>
<td>BRANCHES</td>
<td>No. of forward jumps (branches) in the method</td>
</tr>
<tr>
<td>f37</td>
<td>SINGLELOOPS</td>
<td>No. of single loops in the method</td>
</tr>
<tr>
<td>f38</td>
<td>NESTEDLOOPS</td>
<td>No. of nested loops in the method</td>
</tr>
<tr>
<td>f39</td>
<td>ICOUNT</td>
<td>No. of instructions in the method</td>
</tr>
<tr>
<td>f40</td>
<td>LOOPICOUNT</td>
<td>No. of instructions in the loop bodies</td>
</tr>
<tr>
<td>f41</td>
<td>STACKMAX</td>
<td>Maximum stack slots</td>
</tr>
<tr>
<td>f42</td>
<td>LOCALMAX</td>
<td>No. of local variables</td>
</tr>
<tr>
<td>f43</td>
<td>ARGS</td>
<td>No. of arguments of the method</td>
</tr>
<tr>
<td>f44</td>
<td>TIME</td>
<td>Time consumed by the method</td>
</tr>
<tr>
<td>f45</td>
<td>PC</td>
<td>Percentage Contribution of the method</td>
</tr>
<tr>
<td>f46</td>
<td>CC</td>
<td>Call Count of the method</td>
</tr>
</tbody>
</table>

LOOPY = 1), we need to determine the number of single/nested loops. For this, we will observe the quantitative features SINGLELOOPS and NESTEDLOOPS (see Table 3.2). In Table 3.2, 15 quantitative features are cataloged to represent static/dynamic characteristics of a method. Static and dynamic characteristics are gathered by parsing classes at load time and profiling the running application, respectively. In this thesis, only those features are used that are particularly relevant to loop level parallelization.

3.4 Summary

Qualitative features are helpful in boolean decision making on the basis of particular characteristics of a method. Similarly, quantitative features represent quantity of different characteristics of Java methods. Use of both features in loop level parallelization is elaborated in subsequent chapters.
Chapter 4

Loop Profiling

4.1 Introduction

Loop level parallelization necessitates the complete knowledge of loop organization in a hotspot method. Just identifying the body of outer loop is not enough to exploit loop level parallelism. We need to know the exact number of single and nested loops. Inner loops could have different distributions in each loop nest. The challenge is to classify each loop nest accurately, instead of classifying some inner loops as separate single loops. The main contribution of this work is a graph-theoretic algorithm to generate a loop tree against each nested loop. It produces a loop forest for each method, provided that the method contains more than one single and/or nested loop. Data of eighteen Java Grade Forum (JGF) application benchmarks is collected and analyzed. The results show that all loops in these applications are classified correctly.

4.2 Motivation

First intuitive step in loop level parallelization is to determine the exact number of single and nested loops in the code. These numbers not only (indirectly) indicate the parallelization potential of target application but also provide an estimate of work to be done by runtime system in analyzing loops for parallelizability. Finding single loops is trivial but the determination of nested loops is quite difficult. First issue is the correct demarcation of loop boundaries. Second, we need to determine
Figure 4.1: Loop forest in (a) `transform_internal` method of FFT benchmark (i.e. consists a single tree) (b) `runites` method of MolDyn benchmark (consists a single tree) (c) `matgen` method of LUFact benchmark. The forest consists of three loop trees.

the exact loop tree formed by a loop nest. Figure 4.1 (a) shows the loops in method `transform_internal` of JGF FFT benchmark. It consists of a single nested loop. The nest level of the loop is 3 which is indicated by the height of loop tree, shown on right side. Loop tree is constructed by taking outer most loop as root node and nodes with incoming arrow-head are inner loops of pointing node. There is a pitfall in Figure 4.1 (a). Loop3 starts after loop2 and both are inner loops of loop1. Simple offset comparison based algorithms could treat them as two separate single loops within another loop, reporting them as two separate nested loops, which is not true. True structure could be depicted by representing it as a tree structure. Some other examples of loop trees are shown in Figure 4.1 (b) and (c). As a whole, Figure 4.1 (c) represents a loop forest with 2 multi-node and 1 single-node trees. Similarly, trees in Figure 4.1 (a) and (b) could also be represented as loop forest with a single tree each.
4.3 Internal Representation of Loops

Profilers typically work on binary code (either interpretable or compiled). To elaborate internal representation of loops, we use a trivial Java code example. Figure 4.2 (a) shows the source code of copySqrMatrix method which copies content of one square matrix to other using a level-2 nested loop. Figure 4.2 (b) shows the bytecode of copySqrMatrix method and its control flow graph (CFG) is shown in Figure 4.3. Using first instruction of the method as base (at offset 0), we represent each single loop as a quadruple $<\text{Offset}, \text{Target}, \text{Index}, \text{Stride}>$, where,

- $\text{Offset}$ = Offset of single loop conditional
- $\text{Target}$ = Offset of target label
- $\text{Index}$ = Variable acting as loop index
- $\text{Stride}$ = Step size of loop iterations

$\text{Index}$ and $\text{Stride}$ are determined from IINC bytecode instruction. $\text{Index}$ could also be determined from one of $x$LOAD instruction(s) preceding the loop conditional. Suffix $x$ with LOAD represents the data type of $\text{Index}$ like ILOAD for integer load is shown in Figure 4.2 (b). Figure 4.2 (b) elaborates the quadruple for loop1, which is $<32, 5, 4, 1>$. Similarly, quadruple for each single loop is recorded. Once the data of all single loops is available, we can construct a loop tree for each nested loop. Intuitively, for two loops $\text{loop}_i$ and $\text{loop}_j$, if $\text{Offset}_i > \text{Offset}_j$ and $\text{Target}_i < \text{Target}_j$, then $\text{loop}_j$ lies inside $\text{loop}_i$. For example, quadruple for loop2 in Figure 4.2 is $<26, 10, 5, 1>$ and it is inner loop of loop1 because $32 > 26$ and $5 < 10$. Traversing nodes of tree, we can represent nested loops as a quintuple $<\text{Offset}, \text{Target}, \text{Nest-Level}, \text{Index-Vector}, \text{Stride-Vector}>$ where,

- $\text{Offset}$ = Offset of outer most loop
- $\text{Target}$ = Offset of outer most target label
- $\text{Nest-Level}$ = Height of loop tree
- $\text{Index-Vector}$ = Indexes of all loops in loop nest
- $\text{Stride-Vector}$ = Step sizes of all loop in loop nest
void copySqrMatrix(double[][] src, double[][] des, int sz)
{
    for (int J = 0; J < sz; J++)
        for (int I = 0; I < sz; I++)
            des[I][J] = src[I][J];
}

(a)

(b)

Figure 4.2: (a) Source code, and (b) bytecode of copySqrMatrix method.
Bytecode is marked with backward jumps.
4.4 Identification of Single Loops

As first step, we consider each loop as independent single loop and ignore the fact that some loops might be part of a loop nest. We can easily determine all such single loops by tracking backward jumps. A backward jump is one whose target has already been visited [95] either in terms of target label or a memory address. Bytecode uses labels because exact memory addresses are not known in intermediate code. If target label of a backward jump lies in one of dominator blocks of the block containing jump instruction, then it is a loop. In control flow graph, a block \( d \) dominates block \( v \) (i.e. \( d \text{ DOM} v \)), if all paths from entry block to \( v \) include \( d \). Also, \( \text{DOM} (v) \) denotes a set of all nodes that dominate \( v \), including \( v \) itself.

Single loop detection algorithm is shown in Figure 4.4. Let \( S_i \) be the instruction stream of a method. During interpretation, each visited label \( l \) is added to a list of visited labels \( L_v \). For each branch instruction \( b \), if the branches target label \( l_b \) has already been visited then \( b \) represents a backward jump. Let \( \text{Block}_A \) and \( \text{Block}_B \) are two basic blocks, (i.e. nodes in CFG). If \( b \in \text{Block}_A \) and \( l_b \in \text{Block}_B \) and \( \text{Block}_B \in \text{DOM} (\text{Block}_A) \), then \( b \) is a loop conditional. Prepare quadruple \(<\text{Offset}, \text{Target}, \text{Index}, \text{Stride}>\) against \( b \) and add to the list of single loops \( L_{loop} \).

4.5 Loop Forest Construction

Inner loops of a loop nest may appear in a variety of ways (as shown in Figure 4.1). Once we get a list of single loops \( L_{loop} \) using algorithm of Figure 4.4, we can determine nested loops by using algorithm shown in Figure 4.5. The algorithm constructs a loop forest as described below.

Considering each single loop \( l_s \in L_{loop} \) as a node, loop tree \( T_i \) is constructed against each nested loop and added to a loop forest \( F_i \). Depending upon the availability of loops, \( F_i \) could possibly be (1) empty (2) containing single-node tree(s) only (3) containing multi-node tree(s), or (4) containing a mixture of single-node and multi-node trees. Overall algorithm is shown in Figure 4.5. At start the loop forest \( F_i \) is empty and a tree \( T_i \) is constructed using the first loop of \( L_{loop} \) as root.
Figure 4.3: Basic block level control flow graph of copySqrMatrix method.

Figure 4.4: Algorithm to identify single loops in Java bytecode.
node. Subsequent loops from $L_{loop}$ are either added to an existing tree or cause the generation of new tree(s). An existing tree is re-adjusted if an outer loop comes after some inner loop(s) so that outer most loop is always the root node.

### 4.6 Evaluation

Loop profiling mechanism is tested on eighteen Java Grade Forum (JGF) application benchmarks. All loops in these applications are classified correctly, as shown in Figure 4.6.

A high level estimate of code executed by loops could be obtained by counting instructions in method and loop bodies of an application. Static code covered by loops is an indirect measure of loop level parallelism. Quantitative features $ICOUNT$ and $LOOPICOUNT$ (in Table 3.2) give the number of instructions in a method body and its loop bodies, respectively. If $I_m$ is the instruction count of method bodies and $I_l$ is the instruction count of loop bodies of an application,
Figure 4.6: Loops in eighteen Java Grande Forum benchmarks.

then

\[ l_{scce} = \left( \frac{I_l}{I_m} \right) \times 100 \]

Where, \( l_{scce} \) is percentage static code covered by the loops of an application. Three application benchmarks are chosen from Java Grande Forum benchmark suite to show the loop level code coverage in hotspot methods. Table 4.1 shows the variation in \( l_{scce} \) in case of selecting hotspot methods instead of whole application code. It signifies the availability of potential loop level parallelism in hotspot methods.

<table>
<thead>
<tr>
<th>Application</th>
<th>Crypt</th>
<th>LUFact</th>
<th>SOR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Called Methods</td>
<td>Hotspots</td>
<td>Called Methods</td>
</tr>
<tr>
<td>No. of Methods</td>
<td>30</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>Overall Call Count</td>
<td>48</td>
<td>2</td>
<td>253043</td>
</tr>
<tr>
<td>No. of Single loops</td>
<td>8</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>No. of Nested loops</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>( I_m )</td>
<td>1966</td>
<td>396</td>
<td>2169</td>
</tr>
<tr>
<td>( I_l )</td>
<td>778</td>
<td>374</td>
<td>820</td>
</tr>
<tr>
<td>( l_{scce} )</td>
<td>40.6%</td>
<td>35.5%</td>
<td>57.8%</td>
</tr>
</tbody>
</table>

4.7 Summary

Algorithms to profile loops in compiled Java code helps in identifying loop level parallelization potential of sequential Java application. These algorithms are particularly important for just-in-time systems to exploit loop level parallelism during
dynamic binary translation. Like any loop level parallelization effort, JIT compiler needs to know exact number of single and nested loops in the identified hotspot. Workload distribution and thread invocation incur a lot of overhead and prior knowledge of these numbers help in estimating the amount of work to be done by runtime system and expected speedup. Various aspects of JIT parallelization and loop profiling data are elaborated including the static code covered by loops.
Chapter 5

Instruction Patterns

5.1 Introduction

Identification and cataloging instruction patterns of an instruction set architecture helps in workload classification by using pattern matching. Bytecode instruction pattern composition scheme is presented in this chapter. Instruction patterns are used to identify parallelization candidate DOALL loops. Number of instruction patterns depends on the size of instruction set and we should process bare minimum patterns to reduce runtime overhead. One possible way is to work on a higher level representation of instruction patterns so that we have to process only one pattern against all possible data types and arithmetic/logical operations.

5.2 Intermediate Representation of Instructions

An intermediate representation (IR) of bytecode instructions is defined to reduce the number of instruction patterns and potential pattern processing effort. Although IR is defined for bytecode, the concept is generic and applicable to any other instruction set. IR symbols are generated during parsing phase. An instruction could appear more than once in an instruction pattern. To recognize an instruction pattern of length \( p \) over an instruction set containing \( n \) instructions, we have to look for \( (n)^p \) combinations. These combinations could be reduced if we reduce \( n \) by (symbolically) representing \( n \) instructions with \( m \) symbols, where \( m < n \). For example, a subset of bytecode instructions IADD, LADD,
FADD, DADD is used to perform arithmetic addition of two integer, long-integer, floating-point, double-precision-floating-point numbers, respectively. A high level IR symbol ADD could suffice to recognize any of these four instructions by recording opcode of the instruction. Similarly, we can recognize entire instruction set using a smaller set of IR symbols. In case of Java bytecode, we could represent (about 200) bytecode instructions (i.e. \( n \approx 200 \)) with 42 symbols (i.e. \( m = 42 \)), as shown in Table 5.1. Labels are typically induced by compiler to facilitate control flow and demarcation of basic blocks. We consider LBL as part of IR symbols because labels are integral part of compiled code. As elaborated in next section, presentation of instruction patterns in terms of IR symbols increases the occurrence frequency of instruction patterns. Also, there are about five times fewer choices (i.e. \( \lceil \frac{n}{m} \rceil \)) at each position of instruction pattern if we use IR symbols instead of instruction mnemonics.

## 5.3 Recognition of Instruction Patterns

Compilers convert each source code statement to either a single instruction or an instruction pattern of target architecture. In just-in-time systems, the compiler generates a stream of instructions to be executed by the abstract machine. This machine executes instructions by translating them to the target ISA. In this section, we present our approach to instruction pattern recognition in Java bytecode (produced by Java source compiler). The concepts are quite generic and we believe that this approach could be used for any other ISA by preparing a catalog of ISA-specific fundamental instruction patterns. Each fundamental pattern consists of at least two instructions in a specific order and performs a smallest indivisible high level task e.g. “variable initialization”. Some instructions like INC or LV (in Table 5.1) could independently perform an indivisible high level task e.g. “increment/decrement”. We enumerate such instructions as independent instructions. An independent instruction will become a pattern when used with at least one other instruction, including LBL. Instruction pattern recognition is elaborated using a sample code shown in Fig. 5.1 (a). The code is an inner loop taken from SORrun method of JGF SOR benchmark [67]. Bytecode and IR tuples of the
Table 5.1: Intermediate Representation of Bytecode Instructions.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Bytecode Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Do nothing</td>
<td>NOP</td>
</tr>
<tr>
<td>LC</td>
<td>Load constant</td>
<td>ACONST_NULL, ICONST_M1, ICONST_1, ICONST_2, ICONST_3, ICONST_4, ICONST_5, LCONST_0, LCONST_1, LCONST_2, LCONST_3, FCONST_0, FCONST_1, DCONST_0, DCONST_1, BI-PUSH, SIPUSH, LDC, LDC_W, LDC2_W</td>
</tr>
<tr>
<td>LV</td>
<td>Load Value</td>
<td>ILOAD, LLOAD, FLOAD, DLOAD</td>
</tr>
<tr>
<td>LR</td>
<td>Load Reference</td>
<td>ALOAD</td>
</tr>
<tr>
<td>LVA</td>
<td>Load Value from Array</td>
<td>ALOAD, LLOAD, PALOAD, BALOAD, BALOAD, CALOAD, SALOAD</td>
</tr>
<tr>
<td>LRA</td>
<td>Load Reference Array value</td>
<td>ALOAD</td>
</tr>
<tr>
<td>SV</td>
<td>Store Value</td>
<td>ISTORE, LSTORE, FSTORE, DSTORE</td>
</tr>
<tr>
<td>SR</td>
<td>Store Reference</td>
<td>ASTORE</td>
</tr>
<tr>
<td>SVA</td>
<td>Store primitive Array Value</td>
<td>ASTORE, LSTORE, PASTORE, BASTORE, SASTORE, LSTORE, SASTORE</td>
</tr>
<tr>
<td>SRA</td>
<td>Store Reference Array value</td>
<td>AASTORE</td>
</tr>
<tr>
<td>IF</td>
<td>Pop</td>
<td>DPOP, DPOP2</td>
</tr>
<tr>
<td>DP</td>
<td>Doublecast</td>
<td>DCMPL, FCMPL, DCPMP, DCPMP, DCPMP</td>
</tr>
<tr>
<td>AO</td>
<td>Arithmetic Operation</td>
<td>IADD, LADD, FADD, DADD, ISUB, LSUB, FSUB, DSUB, IMUL, LMUL, FMIN, DMIN, IDIV, DDIV, FDIV, DDIV, IREM, LREM, FREM, DREM</td>
</tr>
<tr>
<td>LO</td>
<td>Logical Operation</td>
<td>INEG, LNEG, FNEG, DNEG, ISHL, LSHL, ISHR, LSHR, IOR, LOR, IXOR, LIOR, LIXOR</td>
</tr>
<tr>
<td>INC</td>
<td>Increment</td>
<td>IINC</td>
</tr>
<tr>
<td>CMF</td>
<td>Compare</td>
<td>LCMP, FCMP, DCMP, BCMP, BCMP, BMCMP</td>
</tr>
<tr>
<td>IP1</td>
<td>1-value IF statement</td>
<td>IFJE, IFNE, IJLT, IFGE, IFGT, IFLTE</td>
</tr>
<tr>
<td>IP2</td>
<td>2-values IF statement</td>
<td>IJCPMP, IFCPMP, IFCPMPE, IFCPMPG, IFCPMPGT, IFCPMPGE, IFCPMPGTE, IFCPMPST, IFCPMPGT, IFCPMPGE, IFCPMPGTE, IFCPMPGT, IFCPMPGE, IFCPMPGC, IFCPMPGCT</td>
</tr>
<tr>
<td>C+JR</td>
<td>Unconditional Jump</td>
<td>GOTO, JSR, RET</td>
</tr>
<tr>
<td>SW</td>
<td>Switch statement</td>
<td>TABLESWITCH, LOOKUPSWITCH</td>
</tr>
<tr>
<td>RV</td>
<td>Return Value</td>
<td>RETURN, LRETURN, PRETURN, DRETURN</td>
</tr>
<tr>
<td>RR</td>
<td>Return Reference</td>
<td>ARETURN</td>
</tr>
<tr>
<td>RY</td>
<td>Void</td>
<td>RETURN</td>
</tr>
<tr>
<td>LSF</td>
<td>Load Static Field</td>
<td>GETSTATIC</td>
</tr>
<tr>
<td>SFF</td>
<td>Store Static Field</td>
<td>PUTSTATIC</td>
</tr>
<tr>
<td>LF</td>
<td>Load Class Field</td>
<td>GETFIELD</td>
</tr>
<tr>
<td>SF</td>
<td>Store Class Field</td>
<td>PUTFIELD</td>
</tr>
<tr>
<td>INV</td>
<td>Invoke a method</td>
<td>INVOKESPECIAL, INVOKESTATIC, INVOKEVIRTUAL, INVOKEINTERFACE, INVOKEINTERFACE</td>
</tr>
<tr>
<td>NW</td>
<td>create new object</td>
<td>NEW</td>
</tr>
<tr>
<td>NVA</td>
<td>Create new value Array</td>
<td>NEWARRAY</td>
</tr>
<tr>
<td>S</td>
<td>Array Length</td>
<td>ARRAYLENGTH</td>
</tr>
<tr>
<td>XCP</td>
<td>throw exception</td>
<td>ATHROW</td>
</tr>
<tr>
<td>CCH</td>
<td>Check cast</td>
<td>CHECKCAST</td>
</tr>
<tr>
<td>IOP</td>
<td>Instance of</td>
<td>INSTANICOF</td>
</tr>
<tr>
<td>MF</td>
<td>Monitor enter</td>
<td>MONITORENTER</td>
</tr>
<tr>
<td>MX</td>
<td>Monitor exit</td>
<td>MONITOREXIT</td>
</tr>
<tr>
<td>NMA</td>
<td>create new n-D array</td>
<td>MULTIANEWARRAY</td>
</tr>
<tr>
<td>IFS</td>
<td>IF statement (compares null)</td>
<td>IFNULL, IFNONNULL</td>
</tr>
<tr>
<td>LBL</td>
<td>Label induced by compiler</td>
<td></td>
</tr>
</tbody>
</table>

code are shown in Figure 5.1 (b). IR tuple of each instruction is represented as < Symbol, Opcode, [Argument(s)] >, where Symbol is IR symbol (defined in Table 5.1), Opcode is the opcode of encountered instruction and optional Argument(s) represents zero or more arguments of the instruction. IR tuple of a label does not contain any opcode and its Argument contains string representation of actual label.

For compact representation, IR symbols of an instruction pattern are concatenated as shown in Table 5.3. For example, read operation on Gim1[j] in Figure 5.1(a) was translated into bytecode instructions at line 10, 11, 12 of Figure 5.1(b) using the local variable indexes shown in Table 5.2. This instruction pattern is represented as LR-LV-LVA by using IR symbols.
for (int j = 1; j < Nm1; j++) {
    Gi[j] = omega_over_four * (Gim1[j] + Gip1[j] + Gi[j-1] + Gi[j+1])
        + one_minus_omega * Gi[j];
}

(a)

Blah Blah Blah

(b)

Figure 5.1: (a) Source code of a loop taken from SORrun method of JGF SOR Benchmark (b) Bytecode and its IR tuples.
Table 5.2: Variables used in sample code.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Name</th>
<th>Type</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCAL</td>
<td>J</td>
<td>int</td>
<td>17</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Nm1</td>
<td>int</td>
<td>11</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Gi</td>
<td>double</td>
<td>14</td>
</tr>
<tr>
<td>LOCAL</td>
<td>omega_over_four</td>
<td>double</td>
<td>6</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Gim1</td>
<td>double</td>
<td>15</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Gip1</td>
<td>double</td>
<td>16</td>
</tr>
<tr>
<td>LOCAL</td>
<td>one_minus_omega</td>
<td>double</td>
<td>8</td>
</tr>
</tbody>
</table>

In Figure 5.1(b), all occurrences of LR-LV-LVA pattern are highlighted using dotted lines. LR-LV-LVA is a fundamental pattern because it represents a smallest high level sub-task and composed of independent instructions only. In Table 5.3, all fundamental patterns and partial pattern components (of sample code) are recognized and assigned unique IDs $P_{xy}$ and $C_{xy}$, respectively. Each $P_{xy}$ (or $C_{xy}$) represents a complete pattern (or a pattern component) $y$ having $x$ level composition. As fundamental patterns are not composed of other patterns so their composition level is zero. On the other hand, a composite pattern is composed of other (complete) patterns, pattern components and/or independent instructions. Using independent instructions and IDs of fundamental patterns and pattern components, a parse tree is generated, as shown in Figure 5.2. Parse tree construction starts from leaf nodes that are fundamental patterns, pattern components or independent instructions (at level 0). First level composition contain leaf nodes only. Second level composition contains at least one first level composite pattern, third level contains at least one 2nd level composite pattern, and so on. Each non-leaf node represents a composite pattern whose composition depends on the highest level of non-leaf node used as constituent. The root of the tree represents top level composite pattern that is entire bytecode region shown in Figure 5.1(b).

5.4 Inter-iteration Data Dependence Patterns

DOALL loops could be identified by making sure that loop body either does not contain any instruction pattern corresponding to data dependences or, if present, operate on independent locations in each iteration. We are interested in identification of read/write patterns of local variables (including formal parameters),
arrays elements and class members of both primitive and user-defined types. In Figure 5.1(b), bytecode of the loop reveals that there are only those instruction that operate on local variables. Hence, there is no manipulation of class members in the loop. Table 5.2 shows the types and compiler-assigned indexes of variables involved. Bytecode deals with these indexes, as shown in Figure 5.1(b). IINC instruction indicates that iteration counter is indexed at 17 which is “j” in source code. In Table 5.3, we can see that only one write operation is performed in each iteration and it is represented by $P_{00}$. This pattern has sixth level composition and its first component $C_{00}$ contains information about the variable involved. The IR tuples of $C_{00}$ are at line 6-8. It contains index 14 which is “double[] Gi” and 17 which is iteration counter. Write operation of Gi depends on three read operations of Gi, one of which is performed in same iteration so harmless. There are inter-iteration data dependences due to other two reads in an iteration $j$ because they are performed in immediately previous and next iterations $j-1$ and $j+1$, respectively. In Table 5.3, $P_{03}$ and $P_{04}$ represent a skewed array-read instruction patterns while $P_{02}$ represent an array-read in same iteration. The patterns
Table 5.3: Instruction patterns recognized in sample code.

<table>
<thead>
<tr>
<th>ID</th>
<th>Instruction Pattern</th>
<th>Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{00}$</td>
<td>LBL-LC-SV</td>
<td>Fundamental Pattern</td>
</tr>
<tr>
<td>$P_{01}$</td>
<td>LBL-GJR</td>
<td>Fundamental Pattern</td>
</tr>
<tr>
<td>$P_{02}$</td>
<td>LR-LV-LVA</td>
<td>Fundamental Pattern</td>
</tr>
<tr>
<td>$P_{03}$</td>
<td>LR-LV-LC-AO-LVA</td>
<td>Fundamental Pattern</td>
</tr>
<tr>
<td>$P_{04}$</td>
<td>LBL-LR-LV-LC-AO-LVA</td>
<td>Fundamental Pattern</td>
</tr>
<tr>
<td>$P_{05}$</td>
<td>LBL-INC</td>
<td>Fundamental Pattern</td>
</tr>
<tr>
<td>$P_{06}$</td>
<td>LBL-LV-LV-IF2</td>
<td>Fundamental Pattern</td>
</tr>
<tr>
<td>$C_{00}$</td>
<td>LBL-LR-LV</td>
<td>Pattern Component</td>
</tr>
<tr>
<td>$C_{01}$</td>
<td>LBL-SVA</td>
<td>Pattern Component</td>
</tr>
<tr>
<td>$P_{10}$</td>
<td>LR-LV-LVA-LR-LV-LVA-AO</td>
<td>$P_{02} - P_{03}$-AO</td>
</tr>
<tr>
<td>$P_{11}$</td>
<td>LV-LR-LV-LVA-AO</td>
<td>LV-$P_{02}$-AO</td>
</tr>
<tr>
<td>$P_{20}$</td>
<td>LR-LV-LVA-LR-LV-LVA-AO-LR-LV-LC-AO-LVA-AO</td>
<td>$P_{10} - P_{03}$-AO</td>
</tr>
<tr>
<td>$P_{30}$</td>
<td>LR-LV-LVA-LR-LV-LVA-AO-LR-LV-LC-AO-LVA-AO-LBL-LR-LV-LC-AO-LVA-AO</td>
<td>$P_{20} - P_{04}$-AO</td>
</tr>
</tbody>
</table>

for reading Gi[$j-1$] and Gi[$j+1$] are $P_{03}$ at line 17-21 and $P_{04}$ at line 23-28, respectively. Due to inter-iteration data dependences, the loop in Figure 5.1 is not DOALL. Proposed methodology moves on to next available loop instead of trying dependence resolution because it parallelizes DOALL loops only.

5.5 Summary

This chapter started with intermediate representation of bytecode instructions and its usage for instruction pattern composition. Use of instruction patterns for
inter-iteration data dependence analysis is elaborated using an example. In next chapter, parallelization methodology performs data dependence analysis using instruction patterns and select DOALL loops for parallelization.
Chapter 6

Parallelization Methodology

6.1 Introduction

This chapter presents the methodology for loop level parallelization of application hotspots. This methodology is inspired by just-in-time compilation. Hotspots are identified using profiler feedback. Parallelization candidate loops are selected using method level features and instruction patterns. Selected loops are parallelized using the threading framework described in this chapter.

6.1.1 Problem Formulation

During execution, let a Java application calls $N_m$ methods and each method $m_j$ consists of $k$ loops, where $j \geq 1$ and $k \geq 0$. Starting from first method (i.e. main in most of applications), $j-1$ other methods are typically called in hierarchical manner and inter-procedural relationships could be visualized as a call graph [87]. Call graph is a directed graph $G = \langle V, E \rangle$, where $V$ is a finite set of vertices and $E$ is a finite set of edges. Each vertex $v \in V$ represents a method invocation and each edge $e \in E$ between a vertex pair $(u, v)$ represents one or more invocations of $v$ by $u$ (i.e. $u \rightarrow v$). Static call graph is constructed by source code browsing whereas dynamic call graph is obtained by profiling the running application. Flat profile $F$ is an unstructured representation of call graph, where $|F| = N_m$. Typically, $F$ is a sorted list of methods which, in addition, contains runtime information like call count, time consumption and percentage time consumption of each method.
Chapter 6. Parallelization Methodology

Percentage time consumption of a method is actually the percentage contribution \(PC\) of method toward total execution time of application. \(PC\) is defined as,

\[
PC = \frac{\text{Self time consumed by the method}}{\text{Total time consumed by the application}} \times 100 \quad (6.1)
\]

6.1.2 Percentage Contribution Threshold

We introduced the term “percentage contribution threshold” to analyze speculative parallelization potential of applications in the SeekBin paper [73]. Percentage contribution threshold \(T_{PC}\) is the part of application run time \((\leq 100\%)\) that we want to be parallelized. For example, setting \(T_{PC} = 80\%\) for an application means that we are interested in parallelizing only those hotspots that collectively consume \(\geq 80\%\) time of the application. Purpose of percentage contribution threshold value is to select fewer most time consuming methods for further analysis of parallelization potential. By relaxing the constraint of parallelizing 100\% (execution) time of the application, the goal is to achieve maximum possible speedup with bare minimum parallelization effort. For example, if we say that we just parallelize 95\% time of the application, we might save time and effort by not parallelizing a set of low contribution methods that collectively consume only 5\% time of the application. Parallelizing low contribution methods are likely to degrade performance (even worst than that of sequential execution) because workload for each thread would be too trivial to hide the overhead of threads life cycle management. An optimal value of \(T_{PC}\) is one that is nearer to 100\% and it selects minimum number of hotspot methods. Lower values of \(T_{PC}\) select lesser number of methods but at the cost of leaving \((100 - T_{PC})\%)\) time of application unattended. On the other hand, higher values select more methods to be analyzed for parallelization potential and thus are likely to incur additional overhead. An optimal value could trade-off this situation.

To get the optimal value of \(T_{PC}\), we randomly selected a set of Java applications from Java Grande Forum benchmarks [67], NAS parallel benchmarks [34] and DaCapo benchmarks [12]. All applications were profiled to get flat profile of each application. Flat Profiles were sorted by \(PC\) value of each method, where \(PC\)
is calculated using eq. 6.1. Methods are sorted in descending order and ranked as 1, 2, ..., n, according to their PC values \( PC_1, PC_2, \ldots, PC_n \), where \( PC_1 > PC_2 > \ldots > PC_n \). Unattended time \( t_u \) is calculated by aggregating the PC values of methods selected as hotspots. In each application, \( t_u \) for \( n \) hotspots is

\[
t_u = 100 - \sum_{i=1}^{n} PC_i
\]  

(6.2)

For \( m \) applications, the mean value of \( t_u \) is

\[
(t_u)_{\text{mean}} = \frac{\sum_{j=1}^{m} (t_u)_j}{m}
\]  

(6.3)

Parallelization overhead is a linear function of number of hotspots, with the assumption that parallelization effort of each hotspot is constant. Mean values of unattended time are plotted against parallelization overhead to find the optimal value of unattended time which is 8.5%, as shown in Figure 6.1. \( T_{PC} \) value is calculated as

\[
T_{PC} = 100 - t_u
\]  

(6.4)

which is 91.5%. Hence, optimal threshold value is around 90% because we observed in majority of applications that successive \( T_{PC} \) values often select similar number of hotspot methods. We used \( T_{PC} = 90\% \) in this thesis and Figure 6.2 shows its effect on eighteen JGF benchmarks, where \( N_h \) is the number of hotspots. It is obvious from Figure 6.2 that majority of methods are shunt out because they collectively consume \( \leq 10\% \) time of the application.

After hotspots selection, we need to determine the reasons of high time consumption of hotspots for at least two reasons: (1) to determine the feasibility of hotspot code for loop level parallelization (2) amount of work to be done for parallelization of whole application. For example, we need to determine that how many loops are there, how they are organized and how many are DOALL. It is quite possible for a hotspot to have no loop and still consume a lot time due to its high call count.
These characteristics are determined using catalogs of qualitative and quantitative features of hotspots. Details of method level qualitative and quantitative features is given in Chapter 3.

### 6.2 Parallelization Methodology

Executable Java code is written in methods of one or more classes. Bytecode instructions in a method manipulates data in terms of class members and local variables. JVM loads a class on demand i.e. whenever a class member is first
Chapter 6. *Parallelization Methodology* 47

accessed by the application. Parallelization methodology works in three phases as shown in Figure 6.3. In profiling phase, an application is test-run to get its execution profile. Profiler output is fed back to JVM during actual run. Using a value of $T_{PC}$ (e.g. 90%), top $N_h$ hotspot methods are selected from the flat profile $F$, where $F$ is sorted by $PC$ in descending order. JVM class loader is hooked so that classes could be parsed and transformed at load time [73]. Each class is parsed and modified just before it is loaded by the JVM. In parsing phase, list of methods $L_m$ of a class $i$ is acquired to determine if it contains a hotspot. If a method $m_{ij}$ is hotspot, it is parsed to generate (1) list of qualitative features (2) list of quantitative features (3) list of backward jumps (4) intermediate representation (IR) tuples, and (5) instruction patterns. IR tuples and instruction patterns are elaborated in Chapter 5. A list of single loops $L_{SL}$ and a list of nested loops $L_{NL}$ is then generated using the list of backward jumps and control flow graph of the hotspot. In modification phase, a heuristic on quantitative feature "call count" ($CC$) is used to estimate the potential location of parallelizable loop(s). Heuristic says that if $CC < N_m$ and $m_{ij}$ is $LOOPY$ then potentially parallelizable loop(s) lies within $m_{ij}$ otherwise it lies within some caller of $m_{ij}$. The reason is that per invocation time consumption of $m_{ij}$ is inversely proportional to $CC$. Hence, the heuristic implies that if $CC$ is significantly large, the time consumption of $m_{ij}$ is not due to the loops in it but (potentially) it has been called within a loop of its parent method. In later case, parent of $m_{ij}$ may be the actual hotspot if it calls $m_{ij}$ in a loop. In any case, once we determine a potentially parallelizable loop, we need to check if it is DOALL, as elaborated in section 6.2.3. In modification phase, a DOALL loop $l_{ijk}$ is transformed for parallel execution, using the steps given in modification phase of Figure 6.3. Modification steps are elaborated in section 6.4 by using an example.

6.2.1 Parallelization Criteria

There are two criteria for loop level parallelization:

1. **Hotspot Selection**

   Set $T_{PC} = 90\%$ and select few most time consuming methods as hotspots, provided
that they collectively consume at least 90% time of application.

2. Loop Selection

I. If a hotspot has significantly high CC value (e.g. ≥ \(N_m\)), then go to its caller(s). In (any of) caller methods, if hotspot is called in a DOALL loop, then transform this loop for parallel execution.

II. Otherwise, if hotspot itself contain DOALL loop(s), transform the loop(s) for parallel execution.

III. In case of invalidation of (I) and (II), run unmodified sequential application.

6.2.2 Loop Identification

Hotspots are selected using percentage contribution threshold. Loops in each hotspot has been recorded in quantitative features SINGLELOOPS and NESTEDLOOPS. A single loop is recorded as a quadruple <Offset, Target, Index, Stride>

![Figure 6.3: Work flow of proposed parallelization methodology.](image-url)
and a nested loop is represented as a 5-tuple \(<\text{Offset},\text{Target},\text{Nest-Level},\text{Index-Vector},\text{Stride-Vector}>\), as elaborated in chapter 4. Offsets are taken relative to first instruction of the method. Data about all loops of a hotspot are recorded during parsing phase. Detailed discussion of loop profiling algorithms could be seen in chapter 4.

### 6.2.3 Loop Classification

By determining the number of single/nested loops in each hotspot, we are able to iterate on all loops for available parallelism. This thesis deals with directly exploitable parallelism and does not try to expose hidden parallelism using loop transformation techniques. Only loop unrolling is used to distribute workload on multiple cores. This work parallelizes each DOALL loop of arbitrary stride size that is either defined within a hotspot method or it calls a hotspot method. Potential inter-iteration data dependences in each loop are observed by using instruction pattern compositions (Chapter 5). Data dependences are analyzed by recognizing instruction patterns corresponding to read/write of local variables, arrays elements, and class members of primitive/user-defined types. A loop is classified as DOALL if its memory access patterns operate on independent memory locations in each iteration.

### 6.3 Threading Framework

A threading mechanism is required to transform selected loops for parallel execution. We designed a Java threading framework that comprises only two classes, worker and manager. According to loop characteristics, both classes are (dynamically) generated directly in bytecode. Our framework is inspired by source level JAVAR framework [11]. If \(l_{ijk}\) is a loop \(k\) defined in method \(j\) of class \(i\) then for each loop \(l_{ijk}\), two classes \(\text{Manager}_{ijk}\) and \(\text{Worker}_{ijk}\) are generated using ASM bytecode engineering library [15]. \(\text{Worker}_{ijk}\) encapsulates the entire implementation of parallel task. \(\text{Manager}_{ijk}\) is responsible for creation, orchestration and joining of workers. \(\text{Manager}_{ijk}\) contains only one static method \(\text{work}(...)\) and each candidate loop \(l_{ijk}\) is replaced with just a single call to \(\text{Manager}_{ijk}.\text{work}(...)\). Class diagram in Figure 6.4 shows the interaction of threading framework with
class $i$. In Figure 6.4, a single manager manages life cycle of $n$ worker threads. Each worker calls $run_{ijk}$ method defined in class $i$. Loop $l_{ijk}$ is replaced with a call to $Manager_{ijk}.work(...)$. In class $i$, $Manager_{ijk}.work(...)$ would be called $j \times k$ times, if there are $k$ DOALL loops in $j$ hotspot methods of the class. Figure 6.4 shows a cyclic dependency that is removed by declaring $run_{ijk}$ before generating $Worker_{ijk}$ and defining it after the generation of $Manager_{ijk}$. Definitions of $Worker_{ijk}$ and $Manager_{ijk}$ classes are shown (at source level) in Figure 6.5 and 6.6, respectively.

### 6.4 Parallelization of a Loop

Parallelization of a loop is done in various steps as explained in this section. All modifications are done on compiled code (i.e. bytecode). However, modifications steps are shown here at source level for the reader convenience.

![Class diagram](image)

**Figure 6.4:** Class diagram showing the association of threading framework classes with the class containing hotspot method.
class Worker_ijk implements Runnable
{
    /* fixed data members*/
    int ID, ncores, a, b, c, fr, to, step;

    /* dynamic data members*/
    [Target-Class tc;
    [Local-variable v1, v2 ... vn;]

    Worker_ijk([Target-Class cls,]
    int aa, int bb, int cc,
    int nc, int id,
    [Local-Variable l1, l2 ... ln])
    {
        [tc = cls;]
        ID = id;
        ncores = nc;
        a = aa; b = bb; c = cc;
        [v1 = l1; v2 = l2; ... vn = ln;]
    }

    private void partitionLoop()
    {
        //Using loop tiling
        step = c;
        int blk = (b + ncores-1)/ncores;
        fr = ID*blk;
        if(ID == 0) fr = ID*blk+1;
        to = (ID+1)*blk;
        if (to > b ) to = b;
    }

    public void run()
    {
        partitionLoop();
        [Target-Class|tc].runijk(fr, to, step
        [, v1, v2, ... vn]);
    }
}

Figure 6.5: Definition of Worker_ijk class of Threading Framework code. “dynamic data members” in square brackets varies according to code characteristics of loop l_ijk.

6.4.1 Parallelizable Loop Selection

To demonstrate step-by-step working of parallelization methodology, we show the steps of identifying and parallelizing the most suitable loop of JGF Series benchmark application [67]. This application manipulates various transcendental and trigonometric functions to calculate Fourier coefficients of function \( f(x) = (x+1)^x \). About 10,000 coefficients are computed with an interval of 0.2. Parallelization methodology starts with profiling phase in which it found that the application calls 28 methods i.e. \( N_m = 28 \). By setting \( T_{PC} = 90\% \) in parsing phase, it found 2 methods as potential hotspots. As mentioned earlier, top-ranking value of a method’s percentage contribution (\( PC \)) could be due to its high call count
public class Manager_ijk {
    static void work([Target-Class tcls,]
            int st, int en, int step
            [, Local-Variable v1, v2 ... vn])
    {
        int cores = getAvailableProcessors();
        Runnable job[] = new Runnable [cores];
        Thread th[] = new Thread [cores];
        for (int i = 1; i < job.length; i++)
        {
            job[i] = new Workerijk([tcls,] st, en, step,
                    cores, i, [, v1, v2, ... vn]);
            th[i] = new Thread(job[i]);
            th[i].start();
        }
        job[0] = new Workerijk([tcls,] st, en, step,
                    cores, 0 [, v1, v2, ... vn]);
        job[0].run();
        for (int i = 1; i < job.length; i++)
        {
            try { th[i].join(); }
            catch (InterruptedException e) {} //...
        }
    }
}

Figure 6.6: Definition of Manager_ijk class of threading framework. “dynamic data members” in square brackets varies according to code characteristics of loop $l_{ijk}$.

or compute intensive loop(s) in it or both. The reason is that $PC$ is based on self time of method and self time does not include the time consumed by other method(s) called in the method.

Features of potential hotspots in Table 6.1 shows that $CC$ value of both methods is significantly high. Method $\text{thefunction}$ does not contain any loop and $\text{TrapezoidIntegrate}$ method contains only a single loop. It means high time consumption (i.e. 99.9% collectively) of these methods is not due to loops but due to their high call count. Call graph in Figure 6.7 shows that immediate caller of both methods is $\text{Do}$. Qualitative/quantitative features of $\text{Do}$ method are shown in Table 6.2 and reveal following characteristics of $\text{Do}$:

- It is non-static and contains a single loop.
- Though contains 5 calls sites, its own call count is 1.
- It does not create any object or array, reads 2 class members and reads/writes 4 local variables.
Figure 6.7: Relevant Portion of call graph of Series Benchmark. It shows that TrapezoidIntegrate calls thefunction and itself called by Do. Inter-procedural relationships are presented using “|===|” . For example JGFkernel is immediate parent of Do but sibling of JGFvalidate.

- Although its self time consumption is 0.1%, it calls two most time consuming methods in a single loop.

Hence, it is the exploitable hotspot and its single loop is parallelization candidate loop.

6.4.2 Loop Extraction

The loop is defined at line 7-10 in source code of Do method, as shown in Figure 6.8(a). Bytecode of this loop is extracted from the method and represented in terms of IR tuples so that instruction patterns could be recognized for data dependence analysis.

Table 6.1: Features of Potential Hotspots in JGF Series Benchmark.

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Feature Name</th>
<th>TrapezoidIntegrate</th>
<th>thefunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qualitative</td>
<td>STATIC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>LOOPY</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>NESTED_LOOPY</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>LEAF</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quantitative</td>
<td>PC</td>
<td>60.70%</td>
<td>39.20%</td>
</tr>
<tr>
<td></td>
<td>CC</td>
<td>19999</td>
<td>19999000</td>
</tr>
<tr>
<td></td>
<td>CALLS</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>SINGLELOOPS</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>NESTEDLOOPS</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>LOCALMAX</td>
<td>15</td>
<td>6</td>
</tr>
</tbody>
</table>
6.4.3 Data Dependence Analysis

Bytecode of Do method contains instruction patterns of read/write of local variables. Besides loop index $i$, one local variable $omega$ is defined outside the loop and used in the loop body. Local variables $omega$ and $i$ are not written in the loop body so there is no inter-iteration data dependence due to local variables. Table 6.2 shows that no static field is read/written and non-static fields are read but not written. However, arrays are read and written although source code does not show any array read. The bytecode reveals that in TestArray[$\cdot$][$\cdot$] write, TestArray[$\cdot$] is first loaded on stack and then its TestArray[$\cdot$][$i$] element is written. There is no

```java
1 void Do() {
2     double omega;
3     JGFInstrumentor.startTimer("Section2:Series:Kernel");
4     TestArray[0][0] = TrapezoidIntegrate((double)0.0,
5         (double)2.0, 1000,(double)0.0,0)/(double)2.0;
6     omega = (double) 3.1415926535897932;
7     for (int i = 1; i < array_rows; i++) {
8         TestArray[0][i] = TrapezoidIntegrate((double)0.0,
9             (double)2.0,1000,omega * (double)i,1);
10     } Manager_ijk .work(this, 1, array_rows, 1, omega);
11     JGFInstrumentor.stopTimer("Section2:Series:Kernel");
12 }
```

(a) void run_ijk (int a, int b, int c, double omega){
    for (int i = a; i < b; i = i+c) {
        TestArray[0][i] = TrapezoidIntegrate((double)0.0,
            (double)2.0,1000,omega * (double)i,1);
        TestArray[1][i] = TrapezoidIntegrate((double)0.0,
            (double)2.0,1000,omega * (double)i,2);
    }
}

(b) void Do() {
     double omega;
     JGFInstrumentor.startTimer("Section2:Series:Kernel");
     TestArray[0][0]= TrapezoidIntegrate((double)0.0,
         (double)2.0, 1000,(double)0.0,0)/(double)2.0;
     omega = (double) 3.1415926535897932;
     Manager_ijk .work(this, 1, array_rows, 1, omega);
     JGFInstrumentor.stopTimer("Section2:Series:Kernel");
}

(c) Figure 6.8: Steps of loop parallelization. (a) Source code of Do method of Series Benchmark (b) Definition of run_ijk (c) Loop replacement in Do method.
Table 6.2: Relevan... Do Method of JGF Series.

<table>
<thead>
<tr>
<th>Qualitative Features</th>
<th>Value</th>
<th>Quantitative Features</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>0</td>
<td>PC</td>
<td>0.10%</td>
</tr>
<tr>
<td>LOOPY</td>
<td>1</td>
<td>CC</td>
<td>1</td>
</tr>
<tr>
<td>NESTED_LOOPY</td>
<td>0</td>
<td>FIELDS</td>
<td>2</td>
</tr>
<tr>
<td>LEAF</td>
<td>0</td>
<td>SFIELDS</td>
<td>0</td>
</tr>
<tr>
<td>OBJ_C</td>
<td>0</td>
<td>CALLS</td>
<td>5</td>
</tr>
<tr>
<td>FIELD_R</td>
<td>1</td>
<td>SINGLELOOP</td>
<td>1</td>
</tr>
<tr>
<td>FIELD_W</td>
<td>0</td>
<td>NESTEDLOOP</td>
<td>0</td>
</tr>
<tr>
<td>LOCAL_R</td>
<td>1</td>
<td>LOCALMAX</td>
<td>4</td>
</tr>
<tr>
<td>LOCAL_W</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAY_C</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MDARRAY_C</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAY_R</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARRAY_W</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SFIELD_R</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SFIELD_W</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

data dependence due to TestArray[][] because it is independently written in each iteration and without involving a read. Hence, the loop is DOALL and we can parallelize it using the proposed methodology.

6.4.4 Declaration of Run_{ijk} Method

A method run_{ijk} is declared in the class of Do method, as shown in Figure 6.8(b), where a, b, c are loop < start, end, step > tuple for each worker thread. We cannot define run_{ijk} yet because < start, end, step > is calculated in dynamically generated partitionLoop method of Worker_{ijk} class. We just declare run_{ijk} here so that a call in Worker_{ijk} could not pop error.

6.4.5 Generation of Worker_{ijk} and Manager_{ijk} Classes

Next step is to generate and load Worker_{ijk} and Manager_{ijk} classes. We observed that all classes have to be loaded by the same class loader as that of the application. Against the source code shown in Figure 6.5 and 6.6, bytecode is generated on-the-fly by using ASM [15].
6.4.6 Definition of $Run_{ijk}$ Method

Due to cyclic dependency shown in Figure 6.4, we define $Run_{ijk}$ after generation of $Worker_{ijk}$ and $Manager_{ijk}$ code. The definition is shown in Figure 6.8(b), where on-the-fly calculation of $a$, $b$, $c$ depends on the number of workers created in $Manager_{ijk}$. In Figure 6.6(b), number of workers is equal to available processor cores in the system.

6.4.7 Loop Replacement in Hotspot

Finally, the loop in $Do$ method is replaced with a single call to $work$ method of $Manager_{ijk}$ (see line 7 of Figure 6.8(c)). Original loop and its replacement is highlighted using dotted lines in Figure 6.8(a) and (c), respectively.

6.5 Summary

This chapter presents the overall parallelization methodology, threading framework and loop modifications. As an example, JGF Series Benchmark is analyzed and its most beneficial loop is parallelized. Working elaborated in this chapter is applied to various other application benchmarks. Detailed analysis and results of parallelization are presented as case studies in the next chapter.
Chapter 7

Case Studies

7.1 Introduction

As a proof of concept, we implemented a research prototype by extending SeekBin [73]. Being a Java agent, SeekBin hooks JVM class loader to manipulate bytecode just before loading. Hotspots are determined by profiling the application during a test run. SeekBin then reads sorted flat profile to determine the classes to be manipulated. Hotspots are parsed using ASM bytecode engineering library [15] and loaded using `java.lang.instrument` API. SeekBin is used to generate qualitative/quantitative features, IR tuples, instruction pattern compositions and loop profiling. Modification steps are done explicitly to manipulate different cores of the target multicore system.

7.2 Experimental Setup

Data is collected by profiling and parsing 18 benchmark applications from Java Grande Forum benchmark suite [67] to demonstrate the potential of proposed methodology. Data is analyzed for comprehension of parallelization potential. Thirteen benchmark applications are explicitly transformed using the recipe shown in Figure 6.3. Transformed applications are run on an 8-core system comprising 2 x Quad Core Intel Xeon E5405 processor, 1333 MHz FSB, CPU Speed 2.0 GHz, L1 D Cache 32 KB, L1 I Cache 32 KB, L2 Cache 2 x (2 x 6 ) = 24 MB and 8 GB DRAM.
7.3 Code Comprehension

The purpose of code comprehension is twofold: first, we want to explore the parallelization potential of the application at hand. To avoid additional runtime overhead, it is crucial to estimate the feasibility of parallelization. We also need to decide the locality and extent of transformations that will be needed. In other words, we want to transform bare minimum amount of most promising code. Table 7.1 represents an estimate of parallelization potential of 18 benchmarks. Parallelization potential of an application depends on various (quantitative) features like number of methods called during execution \((N_m)\), each method’s call count \((CC)\), number of loops \((SINGLELOOPS, NESTEDLOOPS)\), number of instructions in loop bodies \((LOOPICOUNT)\), and dependencies among loop iterations. However, not all methods/loops are potentially feasible for parallelization and we filter them out by setting \(T_{PC} = 90\%\) (see Table 7.1). As a result, we converge to only few methods as potential hotspots. The most time consuming potential hotspot is the starting point of parallelization effort.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>(T_{PC} = 100%)</th>
<th></th>
<th>(T_{PC} = 90%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(N_m)</td>
<td>(f_{46})</td>
<td>(f_{37})</td>
</tr>
<tr>
<td>Arith</td>
<td>19</td>
<td>1805</td>
<td>1</td>
</tr>
<tr>
<td>Assign</td>
<td>21</td>
<td>1597</td>
<td>1</td>
</tr>
<tr>
<td>Cast</td>
<td>19</td>
<td>641</td>
<td>1</td>
</tr>
<tr>
<td>Create</td>
<td>29</td>
<td>1E+8</td>
<td>1</td>
</tr>
<tr>
<td>Loop</td>
<td>19</td>
<td>482</td>
<td>1</td>
</tr>
<tr>
<td>Math</td>
<td>19</td>
<td>3875</td>
<td>1</td>
</tr>
<tr>
<td>Method</td>
<td>33</td>
<td>9E+7</td>
<td>1</td>
</tr>
<tr>
<td>Serial</td>
<td>25</td>
<td>2E+6</td>
<td>9</td>
</tr>
<tr>
<td>Crypt</td>
<td>30</td>
<td>48</td>
<td>8</td>
</tr>
<tr>
<td>FFT</td>
<td>30</td>
<td>37</td>
<td>5</td>
</tr>
<tr>
<td>HeapSort</td>
<td>28</td>
<td>2E+6</td>
<td>4</td>
</tr>
<tr>
<td>LUFact</td>
<td>32</td>
<td>3E+5</td>
<td>15</td>
</tr>
<tr>
<td>Series</td>
<td>28</td>
<td>2E+7</td>
<td>2</td>
</tr>
<tr>
<td>SOR</td>
<td>26</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>SparseMatmult</td>
<td>27</td>
<td>27</td>
<td>3</td>
</tr>
<tr>
<td>MolDyn</td>
<td>35</td>
<td>4E+5</td>
<td>12</td>
</tr>
<tr>
<td>Ray’Tracer</td>
<td>68</td>
<td>4E+8</td>
<td>4</td>
</tr>
<tr>
<td>ABSearch</td>
<td>39</td>
<td>7E+7</td>
<td>18</td>
</tr>
</tbody>
</table>
7.4 Parallelization of JGF Benchmarks

Thirteen benchmark applications are explicitly transformed using the recipe of Figure 6.3. Instead of exposing hidden parallelism, this work prefers to restore sequential version if an application does not show speedup. Eight applications demonstrated the speedup shown in Figure 7.3. To demonstrate scalability, we passed “number of workers” as command line argument, instead of getting it from target system as mentioned in Manager$_{ijk}$ (see Figure 6.6(b)). Transformed applications are run on an 8-core system comprising 2 x Quad Core Intel Xeon E5405 processor, 1333 MHz FSB, CPU Speed 2.0 GHz, L1 D Cache 32 KB, L1 I Cache 32 KB, L2 Cache 2 x (2 x 6 ) = 24 MB and 8 GB DRAM. To avoid overlapping curves, data is presented as long running and short running applications in Figure 7.1 and 7.2, respectively.

7.4.1 Long Running JGF Benchmarks

Four long running benchmarks that demonstrated speedup are Series, Arith, Math and Method (see Figure 7.1). Parallelization of JGF Series benchmark has been described as the motivational example in chapter 6. JGF Series benchmark demonstrated a speedup of 6.9x on an 8-core system, as shown in Figure 7.3. The speedup is taken as single to 8-thread elapsed time ratio. Table 7.1 shows that Arith, Math and Method benchmarks call 19, 19, and 33 methods, respectively. By setting $T_{PC} = 90\%$, we get single hotspot in Arith and Math, but 7 methods in Method benchmark. There is no single loop in these benchmarks. However, Arith, Math and Method benchmarks contain 12, 30 and 8 nested loops, respectively. All loops lie in a hotspot that is called only once. In nested loops of these benchmarks, compute intensive code was found in DOALL inner loops that were parallelized. In Figure 7.3, speedup of Arith, Math and Method benchmarks is 2.7x, 1.6x and 1.4x, respectively. Although speedup of Math and Method is not quite significant on an 8-core system, the point is that it is a best effort achievement without applying rigorous dependence analysis and loop transformations. In case we are not satisfied with the speedup, we can restore to sequential execution anytime because transformations are intended to be applied to loaded instances of classes (i.e. code
on the disk will be intact). By integrating in a JIT compiler, proposed approach is expected to automatically parallelize applications wherever possible and restore sequential code otherwise.

### 7.4.2 Short Running JGF Benchmarks

Short running benchmarks that show speedup are Crypt, LUFact, SparseMatmult and Cast, as shown in Figure 7.2 and 7.3. In Crypt, out of 30 methods, only one method `cipher_idea` consumes 90% time when called twice in the application, as shown in Table 7.1. In `cipher_idea`, there is one 2-level nested loop. The loop is DOALL and its outer loop is parallelized. Crypt demonstrated a speedup of 5.8x as compared to sequential version and perfectly scale with the increasing number of threads, as shown in Figure 7.2. The result is quite encouraging because proposed methodology could transform code on-the-fly without putting burden on the programmer.

In LUFact, out of 32 methods, only 4 methods consume 90.8% time. LUFact contains 15 single and 4 nested loops. However, 4 hotspot methods contain 4 single and 1 nested loops (collectively), as shown in Table 7.1. Starting from most time consuming method `dgefa`, we found that it is called once and contains one 2-level nested loop. Method `daxpy` and `idamax` are called in inner
and outer loops of dgefa’s nested loop, respectively. Outer loop is parallelized using proposed methodology and a speedup of 1.2x achieved on 8-core system. In SparseMatmult, 27 methods are called but only two methods consumed 90.1% time (when each called once), as shown in Table 7.1. Most time consuming method test contains one single and one 2-level nested loop and second method JGInitialise contains one single loop. There is no harmful data dependences in all 3 loops. However, single loops contain trivial amount of computation as indicated by LOOPICOUNT feature. On parallelizing all 3 loops, we observed performance degradation as compared to sequential version. By parallelizing only nested loop of test, we observed scalability with increasing number of threads, as shown in Figure 7.2, and achieved a speedup of 1.4x. Cast benchmark called 19 methods and by setting $T_{PC} = 90\%$, we converge to 6 methods that collectively consume 91.7% time of application (see Table 7.1). Starting from most time consuming method JGFrun, we found 4 nested loops here and this method is called once. Only a single loop is found in one of other 5 methods i.e. in printperf. In nested loops, compute intensive code was found in inner loops that were parallelized. JGF Cast demonstrated highest speedup of 7.6x compared to other applications presented.

![Figure 7.2: Scalability of short running application benchmarks.](image)
in this thesis. Overall, the observed speedup is in the range 1.2 - 7.6x.

### 7.5 Results Comparison

OpenMP [25] is an industry standard API for parallelization of sequential code on shared memory multiprocessor systems. OpenMP provides a directive system and library functions to abstract the complexity of parallelizing C/C++ and Fortran code. JOMP [16] is a Java implementation of OpenMP specification. It consists of a Java source-to-source compiler and a runtime library to facilitate the use of OpenMP-like directives for parallelization of Java code. In this section, results of JIT-inspired methodology and JOMP implementation are compared to elaborate the merits and demerits of this work. Data is collected by executing JOMP implementations of JGF benchmarks on the same multicore system. Figure 7.4 presents a comparison of elapsed time of eight JGF benchmarks, using different threads.

In JGF Arith benchmark, elapsed time of JIT-inspired methodology scales perfectly as the No. of threads increased from 1 to 8, as shown in Figure 7.4 (a).
In case of JOMP, elapsed time of 2-threads is more than the single threaded execution, although it goes down with additional threads. In each multithreaded configuration, elapsed time of JIT results remains less than the JOMP results. This is potentially due to the large memory footprint of JOMP. JOMP makes extra copies of local variables and class fields during parallel code generation. Overall, JIT-inspired results outperform the JOMP results. In Crypt benchmark, the scalability trend of both implementations is similar and overall speedup is also comparable, as shown in Figure 7.4 (b). Comparison of JGF Cast benchmark is shown in Figure 7.4 (c). JOMP performed very poorly while casting 64-bit values to 32-bit values and vice versa. Making additional copies of variables degraded the performance of JOMP generated multithreaded code. JIT-inspired methodology outperforms JOMP by showing the expected scalability with increasing No. of threads. Figure 7.4 (d) shows that JIT-inspired results of JGF Math benchmark outperform the JOMP results. Scalability trend of JIT results is also better than JOMP. On the other hand, JOMP outperforms JIT-inspired results in case of JGF Series benchmark, as shown in Figure 7.4 (e). JOMP performed well in this case because of small memory footprint. Series benchmark contains only one variable to be duplicated by JOMP. However, scalability trend of both implementations is similar. In JGF Method benchmark, scalability trend and overall speedup of JIT-inspired methodology is better than JOMP version, as shown in Figure 7.4 (f). We can see in Figure 7.4 (f) that the scalability trend of JOMP suddenly changes for 5 onward threads. This is potentially due to caching effect because the target system contains two quad-core chips and with 5 onward threads, cores on the second chip get involved. It also emphasizes that the large memory footprint of JOMP is a critical hindrance in achieving the desired scalability.

As compared to JOMP, the speedup of both JGF LUFact and SparseMatmult benchmarks is not quite encouraging, as shown in Figure 7.4 (g) and (h), respectively. In case of LUFact, JOMP parallelizes a DOACROSS loop by introducing thread private variables. Parallelization of DOACROSS is currently beyond the scope of this thesis and we are achieving shown speedup only by parallelizing DOALL loops. In case of JGF SparseMatMult benchmark, JOMP achieve this
speedup by restructuring the sequential algorithm, although both implementations identify and parallelize the same nested loop that is defined in test method. Another major difference is that JIT-inspired methodology parallelized the outer loop whereas JOMP transformed inner loop. Results in Figure 7.4 (h) also emphasizes that while parallelizing nested loops, selection of inner/outer loop for parallelization is a critical decision and should be made carefully. Even though the parallelization of outer loop is typically considered as more beneficial.

7.6 Summary

This chapter demonstrates the potential of JIT-inspired loop level parallelization methodology presented in this thesis. Out of eighteen JGF benchmarks, thirteen benchmarks contain loop level parallelism. Loop level parallelism in some of benchmarks is not easily exploitable by the proposed methodology and need extra work of exposing hidden parallelism. Eight benchmarks showed the speedup and demonstrate the parallelization potential of this work. Section 7.4 discusses the speedup achieved against the base case i.e. the sequential code. Parallelizable loop selection, scalability trend and achieved speedup of each application is presented in section 7.4. Results comparison is provided in section 7.5. JIT-inspired methodology results are compared with the results of JOMP implementations. Results of both implementations were gathered by executing in the same experimental setup. Five out of eight benchmarks outperformed the JOMP implementations. Merits and demerits of both JIT-inspired methodology and JOMP are critically analyzed in terms of memory footprint, algorithmic restructuring, selection of inner/outer loop for parallelization, use of thread local data structures and effort required in each case.
Figure 7.4: Comparison of JIT-inspired and JOMP results of JGF (a) Arith, (b) Crypt, (c) Cast, (d) Math, (e) Series, (f) Method, (g) LUFact, and (h) SparseMatmull benchmarks.
Chapter 8

Conclusion

This thesis demonstrates the selection and parallelization of most promising code portions of compiled sequential Java applications. Code selection is inspired by JIT compilation mechanism of JVM. JVM profiles running applications to select hotspot code for JIT compilation. The thesis first justifies that the use of loop level parallelization is more suitable for parallelization of hotspots selected for JIT compilation. It then presents a loop level parallelization methodology that uses method level features and instruction patterns. It tries to parallelize each DOALL loop found in the vicinity of hotspot methods. Proposed methodology might not be able to parallelize each and every application, it is plausible to exploit parallelism if done with minimal effort. No complex loop transformations are involved to expose hidden parallelism and there is no mis-speculation overhead involved. It exploits parallelism wherever possible and there is no harm because intended in-memory transformations are not made permanent. In case of failure, sequential execution could be restored. However, in case of success, transformations could be made permanent at any time. The main contributions of this thesis are:

- Catalogs of method level qualitative and quantitative features that are usable in various reverse engineering and re-engineering activities.

- Compact intermediate representation of Java bytecode ISA and instruction pattern composition scheme.
• Threading framework to facilitate the parallelization of sequential code.

• A set of algorithms to facilitate parallelization of DOALL loops during JIT compilation.

Parallelization is known to be a difficult problem. With increasing number of cores per chip, it is now possible to use at least part of this compute power to analyze the runtime characteristics of an application with minimal impact on expected performance. Such approaches are particularly beneficial for complex long-running applications, which may not be simple to analyze manually. DoALL loops are one of the simplest constructs that can be extracted from general purpose code by using instruction patterns, particularly related to shared data accesses. This work is an effort to demonstrate the feasibility of parallelizing DOALL loops found in the vicinity of hotspot methods. Restricting to hotspots enhances the possibilities of parallelizing long running compute-intensive Java applications with minimal runtime overhead. Limitation of this work is that it does not attempt to parallelize DOACROSS loops if found in the vicinity of hotspot methods. Since iterations of DOACROSS loops contain data dependences, a dependence resolution framework is required to resolve dependences between potential parallel tasks extracted from DOACROSS loops.

As future work, the methodology will be integrated in JIT compiler front end of an open source JVM. This is a work in progress under a project, funded by Higher Education Commission (HEC) of Pakistan. Once done, the methodology will be augmented with dependence analysis framework to parallelize DOACROSS loops. Another potential extension of this work is to use method level features to formulate parallelizable method classification, using machine learning techniques.
Appendix A

SeekBin Details

A.1 Introduction

SeekBin [73] is an instrumentation agent for profiling, parsing and transformation of compiled Java classes. It uses Java instrumentation API, Java Interactive Profiler (JIP) (http://jiprof.sourceforge.net) and ASM bytecode engineering library [15]. Brief description of open source technologies used by SeekBin and the working of SeekBin is given in this appendix.

A.2 Java Instrumentation API

Java instrumentation API is available since JDK 1.5 as java.lang.instrument package. In this package, public interface Instrumentation provides services for developing instrumentation agents for Java classes. Instrumented classes can emit desired information about the runtime behavior of Java applications executing in JVM. For example, using this API a profiler can add code to collect entry time and exit time of each method to calculate the execution time of called methods. Instrumented applications do not deviate from its intended behavior because the changes made to the code are additive in nature. Instrumentation object can be obtained either when the JVM is launched or after that (while the JVM is already running). In former case, instrumentation object is passed to premain method of the agent class. In later case, it is passed to agentmain method. An agent can call methods on the instrumentation object by acquiring it in either way. SeekBin
uses \textit{premain} method. Like \textit{main} method in Java application, \textit{premain} is the entry point of an agent code. Java agent is simply a Java application that hooks JVM class loader (whenever needed). It is packaged as a \textit{Jar} file and may be specified as a command line argument to the JVM (using \texttt{-javaagent} option). It can also be attached to a running JVM through an implementation dependent mechanism. In either way, \textit{manifest} file of the \textit{Jar} defines an attribute \textit{Premain-Class} or \textit{Agent-Class} that refer to the agent class (i.e. the entry point of the agent code). SeekBin is implemented to be used through command line interface so we discuss only command line interface mechanism here.

Command line interface allows the agent to be attached via \texttt{-javaagent} to JVM launching command:

\begin{verbatim}
$ java -javaagent:<agent jar path>[/=agent options] -jar <application jar path> [application options]
\end{verbatim}

The \textit{manifest} file of the agent jar must contain \textit{Premain-Class} attribute that is properly set to an agent class. An agent class is one that contains the implementation of \textit{premain} method.

We can create multiple agents using \texttt{-javaagent} option more than once and each agent may use the same jar (with different options). After initializations of JVM, \textit{premain} method of each agent is called in the same order as mentioned on command line. In the end, \textit{main} method of the target application is called.

There are two signatures of \textit{premain}, as \textit{premain(String, Instrumentation)} and \textit{premain(String)}. JVM tries to execute former method if available otherwise it executes the later. The agent class may also have \textit{agentmain} method (used to attach the agent to running JVM) but it will not be invoked in case of using command line option. Agent class is loaded by the same class loader that loads the class containing \textit{main} method (i.e. system class loader). So \textit{premain} can do whatever a \textit{main} can do and works under same security and class loader rules. The string argument to \textit{premain} is a string of agent options. Agent implementer is responsible for appropriate parsing of agent options.
A.3 Java Interactive Profiler

Java Interactive profiler (JIP) is an open source Java profiler. It uses aspect-oriented approach to profile running Java application. JIP uses `java.lang.instrument` instead of JNI and works in similar way as SeekBin (i.e. using `-javaagent` option). This option allows the developer modify bytecode when a class is loaded by the class loader. JIP adds profiling aspect to each method to gather profiling data. We selected JIP to include in SeekBin because of the following salient features of JIP.

**Interactivity.** JIP can be turned on/off at run time to allow developer gather multiple measurements. Most of profilers allows only single measurement in one run.

**Portability.** JIP does not rely on JNI to exploit any native components. The only requirement is the availability of standard JDK 1.5 or later.

**Open Source.** JIP is licensed under a BSD style license that is considered as the most friendly open source license. BSD license allows the access, use and modification of source code in very less-restricted manner. However, currently we are using it in SeekBin as it is.

**Lightweight.** JIP is, on average, 10-20 times faster than hprof. When used interactively, there is almost no overhead when profiling is turned off.

**Filtering.** JIP is able to filter out classes or packages in final output. Instead of displaying a whole bunch of irrelevant data, JIP lets user filter out classes and packages that are not required in the output.

**Accurate Timing.** JIP keeps track of measuring overhead. The measuring overhead is factored out to ensure the accuracy of performance measurements.

A.4 ASM bytecode engineering library

The ASM [15] library is designed to work on compiled Java classes. ASM is one of the tools that are designed for runtime (and offline) class generation and transformation. It was designed to be as fast and as small as possible because these
are two important factors for rating runtime tools and techniques. ASM is one of the most recent and efficient addition to bytecode manipulation tools. ASM is well designed modular API that has an associated Eclipse plug-in. Its open source license allows using it in almost any way (BSD style license). The goal of the ASM library is to generate, transform and analyze compiled Java classes, represented as byte arrays (as they are stored on disk and loaded in JVM). For this purpose ASM provides tools to read/write and transform byte arrays by using higher level concepts than bytes. However, scope of the ASM library is strictly limited to reading, writing, transforming and analyzing classes. In particular the class loading process is out of scope.

A.5 How SeekBin Works

SeekBin starts from Main class, although it does not contain any main method. SeekBin being a Java agent starts by calling premain method implemented in Main class of parse package. It is indicated by Premain-Class: parse.Main entry in the manifest file of SeekBin jar.

A.5.1 High Level Algorithm

A high level work flow of SeekBin described here.

1. The execution starts from premain method. In premain method, file containing flat profile is loaded in memory, command line options are parsed and a class file transformer is registered. Flat profile of the application and options are passed to the Transformer.

2. The Transformer parses and possibly modifies every supplied class except the classes of standard library.

3. During parsing, the transformer passes the profile, options and bytecode of the class to a Method Parse Adapter.

4. Method Parse Adapter first gets the No. of candidate methods according to the value of percentage contribution threshold. For each candidate method, it then creates a new Method Parser.
5. Method Parser visits all instructions of the method for qualitative and quantitative features. It also generates IR symbol listing, opcode sequence, mnemonic listing and keeps track of forward and backward jumps.

6. Finally Method Parser populates the Method Data object (\textit{MData}).

\section*{A.5.2 Detailed Steps}

 SeekBin starts its execution by calling \textit{premain} method as mentioned above.

\subsection*{A.5.2.1 Structure of \textit{premain} method}

- \textit{Premain} method first splits \textit{agentArgs} string i.e. first argument. \textit{agentArgs} is a comma separated list of arguments passed to SeekBin. First argument is the absolute path of profiler output file, second is a set of single letter options and third optional argument is the threshold on application execution time. Generally it looks like \texttt{<profiler output>[,options][,threshold]} where “threshold” is optional and should be mentioned only when “options” include “T” option e.g. “feedback/\texttt{<appname>/profile/<appname>.txt,nicbT,90}”

- Profiler output file is loaded in an \texttt{ArrayList} object, using \texttt{loadFile} method of \texttt{FileManipulator} class. It is passed to the constructor of \texttt{ProfInfo}. Form the loaded file, \texttt{ProfInfo} object extracts profiler information e.g. entries in sorted profile. An object of \texttt{Profile} class is then created using values of different members of \texttt{ProfInfo}. \texttt{Profile} object fills an array of \texttt{NetProfileEntry}.

- A new \texttt{Opts} object is created from options substring

- Now, \texttt{premain} creates a \texttt{Parser} object. \texttt{Parser} class is an implementation of \texttt{java.lang.instrument.ClassFileTransformer} interface.

- \texttt{Parser} object is passed to \texttt{addTransformer} of \texttt{Instrumentation} (the 2nd argument of \texttt{premain})

  - \texttt{addTransformer} registers \texttt{Parser} (a \texttt{ClassFileTransformer}). Now, future class definitions will be scanned by the \texttt{Parser} except the class definitions upon which \texttt{Parser} itself depends. \texttt{Parser} is called on class
loading and redefinition but not on retransformation because `canRe-
transform` argument to `addTransformer` is not set.

- It eventually calls `transform` method. All parsing and transformations
  are done in the `transform` method. (See JDK 1.5 documentation for
  more details of `transform`).

### A.5.2.2 Structure of `transform` method

In this method, the bytecode of supplied class is analyzed and optionally trans-
formed. This is done before the class is defined by the JVM.

**Parsing the class.** A `MethodParseAdapter` object is created. `MethodParseAdapter`
extends `ClassAdapter` and implements `Opcodes` interface of ASM bytecode engi-
eering library [15]. A new constructor is defined as

```java
MethodParseAdapter(ClassVisitor cv, NetProfileEntry[] f, Opts opts);
```

Where, `f` contains sorted flat profile of the application and `opts` is the command
line options passed to the agent.

**Modifying the class.** A `ClassModifyAdapter` object is created.

### A.5.2.3 Pre-requisites

Properly installed Oracle JDK 1.5 or higher is required. Application must have
been profiled using JIP before using SeekBin because SeekBin depends on flat
profile generated by JIP and the file containing profiler output must be mentioned
as SeekBin agent options.

### A.6 Usage

SeekBin relies on profiler feedback and works in two steps:

#### A.6.1 Step 1: Profile the application (one time activity)

Let `<seekbin>` is the top level directory and `<ur_app.jar>` is jar file of your appli-
cation, Using a `.properties` file from `<seekbin>/jip1.2/profile` directory, prepare
a configuration file for your application. See documentation of JIP for available
options.
Appendix A. SeekBin Details

Simplest way is to keep default settings except setting

`file=feedback/<ur_app>/profile_<ur_app>.txt`

and add a directory named `<ur_app>` in feedback directory. Issue following command. Profile of your application should be available in feedback/<ur_app> directory.

```
$ java -javaagent:./jip1_2/profile/profile.jar -Dprofile.properties=./jip1_2/profile/<ur_app>/*<ur_app_config_file>.*properties -jar apps/<ur_app>.jar [ur_app options]
```

Example:

```
$ java -javaagent:./jip1_2/profile/profile.jar -Dprofile.properties=./jip1_2/profile/JGFS/section1/arith.properties -jar apps/JGFS.jar
```

SeekBin uses this output of JIP as feedback. This step needs not to repeat every time (unless the application is modified). Once the profiler output is available, your application could be run by attaching SeekBin with it in the way stated below.

A.6.2 Step 2: Attach SeekBin and run the application

Copy `<ur_app>.jar` to apps directory If you have multiple classes containing `main` methods, use following command to (optionally) set the intended class containing the `main` method in the jar’s manifest,

```
$ [jar ufve ./apps/<ur_app>.jar package/class_name]
```

```
$ java -javaagent:/SeekBin.jar="feedback/<ur_app>/profile_<ur_app>.txt[,Options][,threshold]" -jar apps/<ur_app>.jar [> <ur_app\_output>.txt]
```

Example:

```
$ java -javaagent:/SeekBin.jar="feedback/JGFS/section1/arith.txt,J100" -jar apps/JGFS.jar > features/coverage/section1/arith-It.txt
```

A.6.3 Available Options

Following boolean options are available to configure SeekBin:

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
</tr>
</thead>
</table>

```
Appendix A. *SeekBin Details*

b  Output bytecode

c  Output configurations

g  Output group symbol sequence (i.e., intermediate representation of candidate methods)

h  Print headers

i  Output quantitative (integer) features

I  Output loop iteration count

j  Output jumps (branches, loops)

m  Apply Transformations

n  Output qualitative (binary) features of candidate methods

o  Output opcode sequences of candidate methods

p  Output Profiling information

T  Percentage Contribution Threshold (used to select potential candidate methods)

v  Verbose Output

z  Experimental modifications
References


[7] Olivier Aumage, Denis Barthou, Christopher Haine, and Tamara Muenier. Detecting simdization opportunities through static/dynamic dependence analysis. In Euro-Par 2013: Parallel Processing Workshops, volume
References


