PRACTICAL ANALYSES AND TRANSFORMATIONS FOR FLOW-BASED PARALLELISM

BY

NICHOLAS CHUN Y CHEN

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Research Associate Professor Ralph E. Johnson, Chair and Director of Research
Professor Gul Agha
Associate Professor Darko Marinov
Doctor Michael J. Voss, Intel Corporation
Emerging applications demand new parallel abstractions. Traditional parallel abstractions such as data parallelism and task parallelism, while necessary, do not fully address the needs of today’s multi-threaded commercial applications. Emerging applications in the domains of recognition, mining and synthesis (RMS); image and video processing; data warehousing; and automatic financial trading exhibit flow-based forms of parallelism. How can we help developers parallelize their sequential applications to leverage flow-based parallelism?

To take advantage of any form of parallelism, developers face two key challenges: analysis and transformation of their code. Both are typically done manually. Unfortunately, manual analysis is error-prone: given a sufficiently large and unfamiliar code base, it is hard for developers to identify code portions that inhibit parallelism, e.g., data races. Additionally, manual transformation is tedious: after analyzing the code to parallelize, developers still have to write repetitive, boilerplate code to initiate and stop the parallel code, e.g., starting and joining threads.

This dissertation demonstrates that it is possible to reduce the manual burden of such analyses and transformation through automation. The crux of our solution is an interactive approach based on source-to-source analyses and transformations that target the constructs available in modern flow-based parallel libraries.

Evaluation on a representative set of emerging flow-based applications demonstrates that these analyses and transformations are useful and practical. Our approach can successfully parallelize applications from the aforementioned domains with good performance, while remaining fast enough to be used interactively as part of a developer’s workflow. Moreover, by engaging the developer, we are able to parallelize more applications than would have been possible through static analysis alone.
To my parents.
I have received excellent advice and guidance from many different mentors. Gul Agha, Darko Marinov and Mike Voss graciously agreed to be on my thesis committee and provided valuable feedback during my qualifying exam. Bruce Schatz was my mentor for our project on primary healthcare. Although I ultimately decided to complete a dissertation in program analysis, he has continued to provide guidance and advice throughout my time here — in addition to his intermittent e-mails encouraging me to continue another project in primary healthcare. I had the pleasure of working with Rob Rutenbar as a teaching assistant for one semester. Though it was brief, Rob’s work ethic, time management and leadership left a lasting impression on me.

My time in graduate school was much more enjoyable and interesting through the company of my friends. To my officemates of room SC 2219 — Paul Adamczyk, Munawar Hafiz, Binh Le, Yun Young Lee, Stas Negara, Jeff Overbey, Maurice Rabb, Samira Tasharofi and Mohsen Vakilian — you have my deepest thanks and appreciation. There was never a dull moment with any of you in the office. Nathan Hirtz, Rajesh Karmani, Fredrik Kjolstad and Gopal Santharaman provided invaluable feedback and technical help for various projects that I have worked on throughout the years. My mentees, Kevin Chen, Hyuk Kwon Don, Donghun Lee and Eric Reed helped with various aspects of this work. I learned a lot from mentoring them and I hope that they too benefited from my mentoring.

I would also like to thank my best friend, Tracy Chong, who started her PhD the same time I did. She was the person whom I confided in and who kept me motivated when I was feeling unproductive and frustrated. She was also the one who challenged me to do the Illinois Half Marathon, which is the best decision that one can make in graduate school to keep oneself healthy.
The academic staff at the Computer Science Department have been tremendously helpful during my time here. Holly Bagwell, Mary Beth Kelly, Rhonda McElroy and Kathy Runck have helped numerous times when I had questions about the (confusing) procedures and formalities that the university has for graduate studies. Every time I entered their offices, they always had a smile on their faces. And they always helped me resolve whatever issues and concerns I had. I would also like to thank Donna Coleman who has helped me schedule rooms for meetings, practice talks, etc. more times than I could possibly count.

Last, but certainly not the least, I would like to thank my adviser, Ralph Johnson. I worked with Ralph for both my MS and PhD, and consider him both as mentor and friend. Ralph has taught me many valuable lessons, two of which I greatly appreciate and will always cherish. His own disposition has shown me that the best way to face any circumstance is to remain calm, and not jump to hasty conclusions. His guidance and advising throughout the years have imparted the virtues and confidence of being an independent researcher.

During my time as a graduate I have received financial support from many different organizations. The Computer Science Department has graciously supported me through teaching assistantships. I truly enjoyed teaching each of those classes. I was also funded through a Department of Energy grant (DOE DE-FG02-06ER25752) as a research assistant. In 2010, my team and I were selected as finalist for the CIMIT Prize for Primary Healthcare and received monetary support for our work. During my last year here, I was also a recipient of the Ray Ozzie Fellowship.

This dissertation would not have been possible without the help and support of many people. To everyone who was involved, both directly and indirectly, I thank you sincerely.
# Table of Contents

List of Tables ............................................................................... viii

List of Figures ........................................................................... ix

Chapter 1 Introduction ................................................................. 1
  1.1 Flow-based Parallelism ......................................................... 2
  1.2 Leveraging Flow-based Parallelism ......................................... 4
     1.2.1 The Problem .............................................................. 4
     1.2.2 Our Solution .............................................................. 5
  1.3 Thesis Statement and Dissertation Overview ......................... 7

Chapter 2 The Case for Using Parallel Library Constructs ................. 9
  2.1 Methodology ....................................................................... 11
  2.2 The TBB Pipeline Construct ................................................ 12
  2.3 The Applications .................................................................. 15
     2.3.1 Ferret ....................................................................... 15
     2.3.2 Dedup ........................................................................ 16
     2.3.3 x264 ........................................................................ 18
  2.4 Threats to Validity ................................................................. 21
  2.5 Discussion .......................................................................... 21

Chapter 3 Interactive Source-to-source Transformations — A Walk-through . 23
  3.1 Parallelizing LIRe ............................................................... 23
  3.2 Annotate ............................................................................ 25
  3.3 Extract Nodes ..................................................................... 26
  3.4 Invert Loop ........................................................................ 27
  3.5 User Interface ..................................................................... 29

Chapter 4 A Brief Introduction to Pointer Analysis ......................... 32
  4.1 Introduction ......................................................................... 32
  4.2 Heap Abstraction ................................................................. 33
  4.3 Equality-based vs. Subset-based Assignments ...................... 34
  4.4 Flow Sensitivity .................................................................. 36
  4.5 Field Sensitivity .................................................................. 38
  4.6 Context Sensitivity ............................................................... 41
  4.7 A Realistic Example ............................................................. 43
  4.8 Call Graph Analysis ............................................................. 45
List of Tables

2.1 Main challenges in transformation process . . . . . . . . . . . . . . . . . 11

6.1 Speedups on benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
6.2 Description of applications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
6.3 Inspections and changes needed from the developer . . . . . . . . . . . . . 71
List of Figures

2.1 An example of a three-stage pipeline ........................................ 12
2.2 Expressing the pipeline in Figure 2.1 using TBB .......................... 13
2.3 Six-stage pipeline of ferret ....................................................... 15
2.4 Execution time for ferret ......................................................... 16
2.5 Dedup pipeline configurations .................................................. 16
2.6 Execution time for dedup .......................................................... 19
2.7 Dependencies between I, B and P frames in H.264 .......................... 19

3.1 The indexing loop for LIRe with the different nodes annotated ........... 24
3.2 Pipeline for LIRe .................................................................... 25
3.3 The closure representing Node2 and the channels connecting it .......... 27
3.4 Inverting the loop so that each node can operate in parallel. Additionally, Node1, Node2 and Node3 can operate in a data parallel manner. ........ 28
3.5 Pipeline for LIRe .................................................................... 29
3.6 The user interface for EXTRACT NODES ................................. 30
3.7 The user interface for INVERT LOOP ......................................... 31

4.1 Representing points $\rightarrow(x)$ graphically ................................... 34
4.2 Example to illustrate different approaches for handling assignment statements .................................................. 34
4.3 Points-to relations for equality-based vs. subset-based approaches .... 35
4.4 The results of a flow-sensitive pointer analysis ............................... 36
4.5 Example to illustrate the benefits of path sensitivity ....................... 37
4.6 Example to illustrate different approaches for handling fields ........ 39
4.7 Points-to relations for different approaches of field sensitivity ........ 39
4.8 Example to illustrate method calls .............................................. 41
4.9 The points-to relations for the example in Figure 4.8 ....................... 41
4.10 Example to illustrate the benefits of context sensitivity ................ 42
4.11 The points-to relations for the example in Figure 4.10 with a context insensitive analysis .................................................. 42
4.12 The points-to relations for the example in Figure 4.10 with a context sensitive analysis .................................................. 43
4.13 A realistic example of pointer analysis ....................................... 44
4.14 The points-to relations for the example in Figure 4.13 ..................... 45
4.15 The call graph for the example in Figure 4.14 ............................. 46

5.1 Overview of the analyses workflow for the EXTRACT NODES transformation ...... 48
5.2 Keeping track of dependencies between nodes ............................. 50
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Algorithm to compute the side effects for each node</td>
<td>54</td>
</tr>
<tr>
<td>5.4</td>
<td>Basic algorithm to compute the possible data races between pairs of nodes</td>
<td>55</td>
</tr>
<tr>
<td>5.5</td>
<td>The LIRe pipeline executing through time</td>
<td>55</td>
</tr>
<tr>
<td>5.6</td>
<td>Algorithm that incorporates ownership transfer to compute the possible data races between pairs of nodes</td>
<td>56</td>
</tr>
<tr>
<td>5.7</td>
<td>Procedure isTransferred(o, n_i, n_j)</td>
<td>57</td>
</tr>
<tr>
<td>5.8</td>
<td>Parallelizing the Recipe example from Chapter 4</td>
<td>58</td>
</tr>
<tr>
<td>5.9</td>
<td>The points-to relations for the example in Figure 4.13</td>
<td>59</td>
</tr>
<tr>
<td>5.10</td>
<td>Overview of the analyses workflow for the INVERT LOOP transformation</td>
<td>61</td>
</tr>
<tr>
<td>5.11</td>
<td>Algorithm to determine if a node can run in a data-parallel manner</td>
<td>62</td>
</tr>
<tr>
<td>6.1</td>
<td>Code snippet from RayTracer illustrating the use of static factory methods</td>
<td>68</td>
</tr>
<tr>
<td>6.2</td>
<td>Code snippet illustrating calls to Java’s File API</td>
<td>70</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In 2005, Herb Sutter published an influential article entitled *The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software* [Sut05]. The article warns that traditional approaches for boosting application performance are no longer sufficient. Gone are the days when developers can optimistically assume that their applications would *just run faster* on the next version of the processor without much intervention on their part. Gone are the assumptions that the compiler would *automatically transform* their applications to gain significant performance boosts on newer processors. Herb Sutter's article was a call to action for researchers and practitioners: transitioning to multiprocessor systems is eminent.

Unfortunately, transitioning to multiprocessor systems is hard. The time and effort already invested in legacy code make it infeasible for developers to completely rewrite their applications from scratch to take advantage of parallelism. Instead, a more viable approach would be to keep the existing application as-is and incrementally parallelize the most demanding parts, i.e., the bottlenecks. A key insight is that the process of parallelization can be viewed as applying a *series of transformations* — usually manual, sometimes automated — to the existing source code. Some transformations could introduce parallelism, i.e., forking off a thread while other transformations could preserve sequential correctness, i.e., protecting a variable with a lock.

Typically, applications have been parallelized using the low-level constructs of a threading library such as Pthreads. Developers transform their sequential program by introducing threading constructs to allow for concurrency and parallelism while preserving correctness. Unfortunately, programming directly with threads is complex and could have deleterious consequences on understandability, maintainability and performance (scaling
and load balancing). Researchers and practitioners agree that programming with threads is notoriously difficult and error prone [Lee06, Boe05].

Fortunately, existing work on patterns of parallel programming [MSM04, KS11, OA10] have identified useful abstractions for expressing parallelism in ways that provide good performance, understandability and maintainability. For instance, Mattson et al. [MSM04] identified two prominent algorithmic structures of parallel programs: data parallelism and task parallelism. Data parallelism invokes an operation in parallel across multiple data elements; it is naturally expressed using languages such as OpenMP [OMP], OpenCL [OCL] and Cuda [NV1]. Task parallelism invokes independent routines in parallel; it is naturally expressed using task-based languages and libraries such as Cilk [FLR98], Java’s Fork/Join [Lea00b], Intel’s Threading Building Blocks (TBB) [Intb] and Microsoft’s Task Parallel Library (TPL) [LSB09]. The prevalence of such languages and libraries offers developers many options for parallelizing applications that exhibit data and task parallelism.

1.1 Flow-based Parallelism

There are, however, many other parallel programming patterns beyond data and task parallelism. Flow-based parallelism is one such pattern that has been largely neglected until recently. Flow-based parallelism decomposes an expensive computation into a graph with nodes that communicate via message passing. Each node embodies part of the original computation and the edges between nodes represent dependencies. Data flow between nodes. The arrival of its necessary data dependencies triggers the computation in a node — we say that the node “fires”. Parallelism is achieved when nodes can operate concurrently. Flow-based parallelism encompasses various patterns including pipeline parallelism [MSM04], event-based coordination [MSM04] and wavefront pattern [KS11]. The structure of the graph is usually simple, i.e., in the case of linear pipelines but could also be more complex, i.e., in the case of wavefront computations.

The term “flow-based parallelism” is new; we coined the term after Morrison’s Flow-Based Programming: A New Approach to Application Development [Mor10]. Morrison de-
scribes flow-based programming as a “programming paradigm that defines applications as black box processes which exchange data across predefined connections by message passing, where the connections are specified externally to the process”. The term “flow” seems natural for describing such styles of parallelism.

Many applications are naturally described as flow-based applications. Emerging applications in the domains of recognition, mining and synthesis (RMS); image and video processing; data warehousing; and financial trading exhibit flow-based forms of parallelism. While it is possible to parallelize such applications using only data parallelism or task parallelism, the results are far from desirable. The resulting program is cryptic, hard to debug and maintain, and rarely performs well. Hoffman et al. performed a detailed analysis of the performance impact of different ways of combining each parallel paradigm while parallelizing the x264 video encoder and found that a combination of the three parallel patterns works best.

Using flow-based parallelism not only improves the performance of such applications but also their modularity. By partitioning the computation into nodes of a graph that communicate via message passing, we can reduce the coupling between different parts of the software, making it easier to maintain and evolve each part independently. Partitioning also serves as an essential first step toward migrating such applications to run in a more distributed setting such as a cloud-based environment, which is where most of these applications will eventually migrate toward.

It is important to note that data parallelism, task parallelism and flow-based parallelism are not mutually exclusive patterns. All three patterns can be combined and used successfully in a single application. In fact, successful parallelization usually requires a careful combination of all three patterns. Robison documents the Three Layer Cake pattern as a hierarchical approach for composing all three patterns. Data parallelism occupies the bottom layer and coordinates low-level parallelism in the form of array operations; task parallelism occupies the middle layer and coordinates fork/join task parallelism; and, finally, message passing occupies the top layer and coordinates messages between parts of the application. Recall that flow-based parallelism is a form

\footnote{For a comparison of flow-based parallelism with other message passing paradigms, refer to Chapter 7.}
of message passing, albeit a more structured and regular form.

1.2 Leveraging Flow-based Parallelism

As with data and task parallelism, a developer can take advantage of flow-based parallelism by parallelizing her sequential application entirely using low-level threading constructs. Mattson et al. describe an example of implementing pipeline parallelism, a popular form of flow-based parallelism, using threads and custom thread-safe queues for exchanging data between stages [MSM04]. While this works, it is less than ideal. Having to manage the spawning/joining of threads and maintain custom thread-safe queues makes the code unnecessarily verbose and error-prone.

We argue that a compelling way to leverage flow-based parallelism is to transform existing sequential code to use the constructs of a parallel library such as Groovy’s GPars [GPA], Intel’s TBB Flow Graph [Intb] or Microsoft’s TPL Dataflow [TPL]. Such libraries abstract away many of the common patterns of flow-based parallelism, resulting in code that is more succinct and understandable.

We examined the constructs in an older version of Intel’s TBB and found them expressive enough to address many of the common idioms found in applications (see Chapter 2). We found using such constructs to be more advantageous compared to direct parallelizing with low-level threads. Firstly, programs written using parallel constructs were more succinct and easier to evolve because the library provides useful constructs for adding/removing nodes or edges to a computation graph. Secondly, in some cases, code parallelized using library constructs were faster than code parallelized directly with low-level threads because the library has better support for managing and coordinating threads through a thread pool.

1.2.1 The Problem

However, to reap the benefits of such library constructs, developers must first use them. To transform their code to use such constructs, developers face two overwhelming tasks: analysis and transformation of their sequential code. First, a developer has to partition
the original sequential code into a computation flow graph with nodes and edges. She must scrutinize each node to ensure that no data races exist. Given a large and unfamiliar code base, this is hard even for experts. Second, once she is convinced that there no data races exist, she still has to meticulously write repetitive, boilerplate code to create the nodes and link the edges between nodes. Given a large computation graph, there could be many nodes and edges, and manually writing code to describe them is tedious. How can we reduce the manual burden of analyzing and transforming sequential programs?

The typical approach is to build a specialized parallelizing compiler. Parallelizing compilers attempt to relieve the burden of analyzing and transforming the code in an entirely automated manner, without involving the developer. While appealing, even the most advanced static analysis in the compiler are frequently confounded by common programming idioms that abound in typical object-oriented programs, forcing the compiler to be overly pessimistic and precluding many parallelization opportunities. Kim et al. [KKL10a] tested two commercial C/C++ compilers on the OmpSCR benchmarks [DRSGE05] and found that, at best, the compilers could only automatically parallelize 4 out of 13 loops although loop parallelism had been studied since the first parallelizing compilers [Wol96]. They attributed the reasons for failure to pointer based accesses, complex control flow and insufficient cost-benefit analysis.

Even state of the art static analysis for object-oriented program [SCD+13, NAW06, VRCG+99] have trouble reasoning about many best practice idioms. For instance, in the applications we evaluated, we found that the use of static factory methods, logging constructs and, in general, the use of fairly standard library APIs confuse static analysis and force it to be safe but overly conservative. Common techniques for increasing the precision of static analysis (at the expense of memory and running time), e.g., increasing $k$ for k-object sensitivity [MRR05] or $n$ for call-string-sensitivity [SP81], do not help.

1.2.2 Our Solution

While it is difficult for static analysis to reason about such idioms, it is easier for a human. Thus, we propose a practical approach that combines static analysis with human
interaction. Our approach actively engages the developer, i.e., the domain expert in the parallelization process. We adapt ideas from human automation and divide the tasks between developer and tool such that each performs what each excels at [She00]. The developer performs the high-level reasoning task that tools have problems with — identifying which parts of the original computation to partition — whereas the tool handles the low-level analysis task that developers find tedious — analyzing if the partitions have data races and generating the parallel code.

This division of labor fits nicely into Fitts’ Law of Task Allocation, i.e., the Men-Are-Better-At Machines-Are-Better-At list [Fit51] for automation. This approach has been applied in the context of creating usable interactive refactoring tools, a form of source-to-source transformation tools [MCSW07]. Existing work by Dig et al., Kjolstad et al. and Wloka et al. on source-to-source transformation tools for concurrency and parallelism also suggest that such an approach is beneficial [DME09, DTR09, KDAS11, WST09].

We show that this combination is effective. It enables us to parallelize many more applications than would have been possible through static analysis alone.

We implemented this interactive approach in our tool, JFlow. JFlow statically analyzes the code and reports back to the developer. As a tool, JFlow has high utility and is useful in many scenarios. When the analyses are successful, the tool performs the transformation, generating source code that targets the constructs of a parallel library. Even when the analyses fail, the tool is useful — it pinpoints and reports the problems. This allows the developer to either fix the problems and retry, or proceed with the transformation directly if she deems the reported problems innocuous (e.g., as determined by her suite of test cases). JFlow can even be used exploratorily: if the developer wishes to inspect for data races, she could use the tool to analyze the code, view the results and forego the transformation step. She could even use JFlow as a code generator, ignoring all analysis results and merely using it to generate code that she will later tweak by hand. Our tool operates at the source code level. Thus, all issues reported can be easily inspected and understood by the developer. Additionally, the transformations can be inspected and tweaked by the developer, as desired.
1.3 Thesis Statement and Dissertation Overview

Interactive analyses and transformation tools that target the constructs of a parallel library are useful and practical ways to help developers parallelize their flow-based applications.

While there are many different forms of flow-based parallelism, this dissertation concentrates on the one that we have found to be most prevalent: pipeline parallelism. The approach and implementation that we describe can be generalized to other forms of flow-based parallelism with some minimal extensions.

This dissertation argues and provides evidence to support the following three claims:

**Library Constructs** We claim that leveraging the constructs available in parallel libraries is a compelling way to exploit flow-based parallelism. Using these constructs provides comparable performance while increasing understandability and maintainability compared to using low-level threads.

Chapter 2 describes our case study on three emerging applications from the PARSEC benchmark [Bie11] and our experiences using the constructs from Intel's TBB to parallelize those applications. It presents a performance analysis and a discussion of the advantages and disadvantages of using such constructs to support our claim.

**Analyses & Transformations** We claim that it is possible to automate some of the manual analyses and transformations that developers have to perform. The ideas for the transformations come from our case studies parallelizing applications from the PARSEC benchmarks.

First, it is possible to design and implement static analyses that are sufficiently precise and fast to assist developers as they parallelize their code. Second, it is possible to transform the sequential code to use the constructs of a parallel library, such that the resulting code is both readable and maintainable. Chapter 3 presents
a walk-through of how such analyses and transformations work and the interaction required from the developer. We briefly introduce the key concepts underlying pointer analysis in Chapter 4. Chapter 5 then builds upon those key concepts and details our analyses and transformations.

**Interaction** We claim that engaging the developer, i.e., the domain expert in the transformation process is an effective way to address the limitations of static analysis. Because our transformations are source-to-source, the developer can more easily inspect and understand the analysis and generated code.

Chapter 6 describes the evaluation of our approach. We parallelize seven applications: four from a previously known benchmark and three from a suite of large open source projects. Our evaluation on these applications demonstrates that JFlow, with some interaction from the developer, can successfully parallelize applications from the aforementioned domains with good performance (offering up to 3.45x speedup on a 4-core machine) and is fast enough to be used interactively as part of a developer’s workflow.

Related work for flow-based parallelism and our analyses and transformations are discussed in Chapter 7. Finally, Chapter 8 concludes and discusses some of the limitations of our work and proposes possible directions for future work.
Chapter 2

The Case for Using Parallel Library Constructs

This chapter presents the results of our case study comparing the use of parallel library constructs to low-level threading constructs on three applications. The purpose of this case study is to understand the advantages and disadvantages of using parallel library constructs. Prior work on parallelization have frequently focused on using low-level constructs such as Pthreads or POSIX Processes [RVDB10, TCA07, TF10]. We wanted to investigate if parallel library constructs could be a better choice.

This case study compared the performance and expressivity of both approaches. Our results suggest that using parallel library constructs provides comparable performance, while increasing understandability and maintainability compared to low-level threading constructs. During our manual process of parallelizing these applications, we observed that we were repeatedly performing similar transformations. This observation gave rise to the core idea of this dissertation: it would be both possible and advantageous to automate some of these well-structured transformations for flow-based parallelism.

The results presented in this chapter was joint work with Eric Reed. It was originally published in the Transitioning to Multicore workshop [RCJ11].

Parallel libraries such as Intel's TBB provide higher levels of abstraction than threads for parallel programming. Work remains, however, to determine how straightforward it is to use these libraries to express various forms of flow-based parallelism. This case study focuses on a particular pattern: pipeline parallelism. We attempted to transform three representative pipeline applications — content-based image retrieval, compression and video encoding — to use the pipeline constructs in TBB. We successfully converted
two of the three applications.

The main question we asked while conducting this case study was: can we use these parallel libraries to express the kinds of pipeline parallelism that exist in programs parallelized using Pthreads? If so, then these libraries are attractive alternatives to Pthreads. On the other hand, if these libraries are insufficient, then we need to ask what's missing? We document the challenges and lessons learned from converting three representative pipeline applications to use the pipeline constructs in TBB. Though we have used TBB, the general lessons learned are not limited to it; parts of the solutions are specific to TBB but the challenges documented serve as valuable lessons for both library developers who are considering what constructs to build and also application programmers who are deciding what constructs to use.

Version 4.0 of Intel's TBB was the first version to introduce general constructs aimed directly at flow-based parallelism. At the time that this case study was conducted, the flow graph was still a community preview feature and not part of the official TBB release. Thus, our results are based on the smaller set of flow-based constructs in TBB 3.0 that was targeted at pipeline parallelism.

In our case study, each application had a corresponding version parallelized using Pthreads. We closely mimicked the parallelism strategies used in the Pthreads versions to provide a fair and useful comparison. We successfully transformed the content-based image retrieval (ferret) and compression (dedup) applications but had difficulties with the video encoder (x264). Below, we briefly summarize the lessons learned; Section 2.3 goes into greater details.

Expressivity The initial transformation of the sequential code to TBB pipelines took the most time because we had to manually resolve and analyze the dependencies between stages. Because TBB's pipeline also enforces certain restrictions on structure and control flow, we had to manually apply some non-obvious transformations. It is likely that developers would have to apply similar transformations while working on complex sequential code. Table 2.1 summarizes those transformations; Section 2.3 discusses them in detail.
<table>
<thead>
<tr>
<th>Application</th>
<th>Challenges</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ferret</td>
<td>Recursive pipeline stage</td>
<td>Replace recursion with a stack</td>
</tr>
<tr>
<td>Dedup</td>
<td>Single input, multiple outputs</td>
<td>Nested pipelines</td>
</tr>
<tr>
<td></td>
<td>Stage bypassing</td>
<td>Enforce single path</td>
</tr>
<tr>
<td>x264</td>
<td>Backward and forward dependencies</td>
<td><em>Not expressible using TBB pipeline</em></td>
</tr>
</tbody>
</table>

Table 2.1: Main challenges in transformation process

After performing the transformations, we see a reduction in the amount of boilerplate code (setting up mutexes, semaphores, etc) that needs to be written compared to the Pthreads counterparts. Overall, the stages of the pipeline were made more explicit and it was easier to add new pipeline stages.

**Performance** The converted *ferret* program performed on par with its Pthreads counterpart in terms of running time. The converted *dedup* application ran up to 2.13 times faster than its counterpart after the conversion. In both converted applications, the memory requirements were comparable. Overall, when it was possible to successfully convert the applications, TBB presents an comparable alternative to Pthreads in terms of performance while also increasing maintainability.

### 2.1 Methodology

The three applications we studied were from the Princeton Application Repository for Shared-Memory Computers (PARSEC) [Bie11], a benchmark suite for shared-memory multithreaded programs. PARSEC is unique because it is application-driven; it aims to capture *emerging workloads* that are missing from typical high-performance computing benchmarks. PARSEC already includes parallel versions of its applications parallelized using Pthreads. Thus, PARSEC is an ideal research testbed to answer the following research questions:

**Expressivity** Are the pipeline constructs in TBB 3.0 sufficient to *faithfully* express the
patterns of pipeline parallelism present in the Pthreads versions of the applications? Does using these constructs improve the understandability and maintainability of the application?

**Performance** What are the performance impacts, if any, from transforming the applications to use the new TBB pipeline constructs? Are these performance impacts severe enough to deter developers from using TBB?

We measured the performance of each benchmark (original Pthreads version and TBB version) on a machine with four Intel Xeon L755 (1.87GHz) processors with 64GB of memory. Each processor has eight cores capable of simultaneous multi-threading with two threads each. In total, the machine is capable of up to 64 hardware threads. The operating system is CentOS release 5.5 running the 2.6.18-194.26 Linux Kernel. We used both GCC 4.1.2 and Intel ICC 11.1.

PARSEC provides the `parsecmgmt` tool for building and running its applications. Additionally, it also provides several input sets for each application. To make it possible for others to repeat our experiments, we made our modifications compliant with `parsecmgmt` and use it to run performance benchmarks using the `native` input set, the largest input set that closely resembles the typical input sets for each application. Our modifications are available from [http://vazexqi.github.com/ParsecPipelineParallelism](http://vazexqi.github.com/ParsecPipelineParallelism).

### 2.2 The TBB Pipeline Construct

![Figure 2.1: An example of a three-stage pipeline](image)

This section introduces the constructs available in TBB 3.0 to support pipeline parallelism. In TBB, a pipeline is composed of a series of filters. Each filter takes an input token, processes it and produces an output token\(^1\). The first filter in the pipeline does not require an input token; similarly the last filter does not produce an output token. To

\(^1\)The terms *filter* and *token* are used in the official TBB documentation; we use them here to be consistent.
implement the pipeline shown in Figure 2.1, requires creating three Filter objects and composing them together in a Pipeline object as shown in Figure 2.2. Filter objects can be serial or parallel. Only one token can be working at a time in a serial filter — this enforces a way to process tokens in order. Multiple tokens can be working at a time in a parallel filter — this provides a way to execute tokens in an out-of-order manner in parallel to improve throughput. Tokens in different filters may run simultaneously.

#include "tbb/pipeline.h"
class Filter1: public tbb::filter {
  // generate tokens
  void* operator()(void* token);
};
class Filter2: public tbb::filter {
  // process tokens and output tokens
  void* operator()(void* token);
};
class Filter3: public tbb::filter {
  // process tokens
  void* operator()(void* token);
};

// Create the pipeline
tbb::pipeline ThreeStagePipeline;
ThreeStagePipeline.add_filter(new Filter1());
ThreeStagePipeline.add_filter(new Filter2());
ThreeStagePipeline.add_filter(new Filter3());
// Run the pipeline
ThreeStagePipeline.run();

Figure 2.2: Expressing the pipeline in Figure 2.1 using TBB

Transforming an existing program to use pipeline parallelism in TBB always requires the same three steps: (i) identify the stages of a pipeline and convert them into serial or parallel Filter objects; (ii) identify the tokens that pass through each Filter object and override operator() to process them; (iii) construct a Pipeline object and call its run() method.

The pipeline constructs take care of most of the bookkeeping that happens underneath. In contrast, implementing a Pthreads version would require two additional boilerplate steps: (i) create a BlockingQueue in between each filter to hold tokens that might arrive earlier due to load imbalance between stages (ii) wrap each token with a sequence number in the event to identify which tokens to process in order when necessary.
TBB's pipeline execution model is based on the work by MacDonald et al. [MSS]. The typical way to parallelize a pipeline is to dedicate a thread to each stage. However, this is wasteful. As the pipeline is ramping up (starting), its later stages are left idle with nothing to work on; similarly, as the pipeline is ramping down (finishing), the earlier stages are left idle with nothing to work on. Instead, MacDonald et al., proposed an approach that recasts the pipeline as a master/slave structure.

The key idea of their approach is based on the State design pattern [GHJV94]. Their approach transforms each item to be processed as a stateful object. Each stage of the original pipeline is a state that the stateful object can be in. Objects keep track of which state they are in. As customary of the State design pattern, operating on an object will invoke the right operation (based on its current state) and transition it to the next state. Since each object keeps track of its state, this eliminates the need to dedicate a thread to each stage. Threads are free to operate on any stateful object, with the object itself keeping track of what operation to perform.

This tasking model alleviates the ramp-up (resp. ramp-down) problem as the pipeline starts (resp. ends) while also providing better load-balancing by allocating more tasks to different filters dynamically. TBB's pipeline also preferentially carries a token as deep into the pipeline as possible before switching to a different task; this improves memory performance as the token is more likely to remain in cache for each filter. Since these optimizations are built into the constructs, the developer is freed from having to manage any of these issues. In contrast, developers using Pthreads might have to implement these optimizations by hand to improve performance.

This section described the advantages that TBB offers in terms of reducing boilerplate code and its built-in mechanisms for improving performance. The next section examines whether these built-in constructs and mechanisms actually help or hinder developers as they try to express parallelism in the PARSEC applications.
2.3 The Applications

Ferret, dedup and x264 exhibit pipeline parallelism in different forms and provide a good sample of the cases that developers might encounter. We describe the applications in increasing order of complexity.

2.3.1 Ferret

\[\text{Segment} \rightarrow \text{Extract} \rightarrow \text{Query} \rightarrow \text{Rank} \rightarrow \text{Output}\]

*Figure 2.3: Six-stage pipeline of ferret*

Ferret (10,765 SLOC) is a content-based image search application [LJW+]. Given an input image, it segments the image, extracts relevant features, queries the database for candidate images, ranks the candidates based on similarity and outputs the results. These six stages are shown in Figure 2.3. The input and output stages are serial; the four middle stages can run in parallel.

The Pthreads version uses **oversubscription**: specifying the program to run with \(x\) threads would create \(x\) threads for each of the parallel stages. BlockingQueues configured for a maximum of 20 items were used to pass tokens between stages. The files `ferret-parallel.c` (437 SLOC) and `tpool.c` (92 SLOC) set up and coordinated the parallelism using Pthreads.

Mapping ferret to TBB’s pipeline was relatively straightforward: each stage was transformed into a Filter object and marked as serial or parallel. The main challenge was the input stage: it used recursion to obtain a list of images from a root directory; TBB Filter objects are not permitted to recursively call themselves. We solved this by replacing recursion with a stack object. This problem with recursive calls seems common and would need to be handled in a similar manner for other applications.

Our TBB implementation, `ferret-tbb.cpp` reduced the lines of code to 376 SLOC (a difference of 153 compared to the Pthreads version) by eliminating the boilerplate code that needs to be written to set up the blocking queues and thread pools. It made each stage
of the pipeline more explicit and facilitated adding new stages, as necessary. Figure 2.4 shows that the TBB versions performed on par with the Pthreads version for the native test input of 3,500 image queries. The scalability of pipelines is limited by the serial I/O stages; in ferret, performance does not scale beyond 20 threads.

2.3.2 Dedup

Figure 2.5: Dedup pipeline configurations
Dedup (5,968 SLOC) is a compression kernel that uses the “de-duplication” method [Bie11]. Given a data stream to compress, it splits the data into smaller blocks; splits the blocks into smaller segments; computes and checks the hash for each segment; compresses the segments, if necessary; organizes the segments and blocks in their proper order; and, finally, writes the compressed stream. Figure 2.5a shows the configuration for the Pthreads implementation.

Two artifacts make dedup challenging to parallelize:

**Single input, multiple outputs** The SplitBlocks stage takes a block and splits it into smaller segments; it takes a single input token and produces multiple output tokens. However, Filter objects in TBB can only take a single input and produce a single output. To mimic the Pthreads version, we had to resort to nested pipelines as shown in Figure 2.5b. The inner pipeline deals with segments while the outer pipeline deals with blocks. We needed to add the ReassembleBlocks stage to reassemble segments into blocks before passing the tokens to the outer pipeline as the two pipelines operate on different types of data granularity. Figure 1 shows the code snippet for implementing nested pipelines. Lines 19 – 37 show the construction and execution of the inner pipeline.

**Stage bypassing** The CheckHash stage can either proceed to the Compress or WriteOutput stage depending on its result. In TBB, it is not possible to bypass a stage; instead, all tokens will proceed through Compress stage. An additional flag would need to be added in the token to signal whether it needs to be compressed.

At first glance, having to use nested pipelines and not being able to bypass stages seem counterintuitive and detrimental to performance. Nested pipelines create many more temporary objects in memory; not being able to by-pass stages requires redundant processing. However, our experiment shows that the increased parallelism more than compensates for the overhead. Figure 2.6 shows the execution times of both versions with the native test input of compressing an ISO file of 672 MB. The TBB version compiled using GCC consistently outperforms the other versions. The scalability of pipelines is limited by the serial I/O stages; in dedup, performance does not scale beyond 32 threads.
Nested pipelines, however, required more code to express and could be more difficult to understand. In the Pthread version, the files encoder.c (148 SLOC) and queue.c (81 SLOC) set up and coordinated the parallelism. In the TBB version, a single file, encoder-tbb (418 SLOC) was used.

2.3.3 x264

x264 (29,324 SLOC) is an HD video encoder for the H.264/MPEG-4 standard [x26]. The encoder predicts the contents of a frame from previously encoded reference frames. A frame can reference frames that occur before or after itself in play order. Parts of frames,
called macroblocks, do not necessarily use the same reference frames.

There are three types of macroblock: intra blocks (I-blocks), predicted blocks (P-blocks), and bipredicted blocked (B-blocks). I-blocks do not reference other frames. P-blocks reference only one frame. B-blocks reference a frame before and a frame after itself [Ric10]. An I-frame consists entirely of I-blocks. A P-frame contains at least one P-block, but no B-blocks. A B-frame contains at least one B-block. Circular frame dependencies are not allowed, so dependencies define a partial ordering on frames. Once a frame has been encoded, it is available for its dependents in a global buffer. Before a frame can be encoded, all of its dependencies must be in the buffer. Figure 2.7 shows a valid configuration of frames and the dependencies between frames.

The x264 implementation in PARSEC assigns a Pthread to each frame. Each frame
has a condition variable associated with it, which is used to broadcast readiness to dependents. By waiting on the condition variables of all its dependencies, a frame ensures it will block until its dependencies are ready. Just before a frame enters the encoding process, its type (I, P, or B) and dependencies are decided in a way that avoids potential deadlocks. In effect, the pipeline is a dynamically constructed directed acyclic graph where each frame is a stage [Bie11]. x264 was originally categorized as pipeline parallelism in [Bie11]; after examining the code, we believe that the categorization is inaccurate. x264, as implemented, exemplifies the wavefront pattern.

The difficulty in constructing a TBB pipeline implementation of x264 is enforcing frame dependencies. TBB pipelines structures cannot be changed while running, so stages must be constructed from the tasks involved in encoding, unlike the Pthreads implementation. In a pure TBB implementation, if multiple frames are in the pipeline then there is no guarantee that frames near the end of the pipeline will complete before frames near the start require them. Restricting the pipeline so that frames are processed one at a time forces the guarantee, but prevents any parallelism.

A mixed TBB/Pthreads implementation could use Pthread condition variables in the same manner as the existing Pthread implementation. However, when a task in the pipeline waits on a condition variable the entire thread will block. This prevents the usual TBB automatic load balancing between threads and requires oversubscription, which TBB tries to avoid, to achieve significant parallelism. In this implementation, TBB is little more than a wrapper around Pthreads that provides automatic queue management, but at a high runtime overhead due to the task scheduler.

Implementing x264 in TBB is not impossible, but the TBB pipeline structure is not suitable. An implementation using the newly introduced TBB 4.0 flow graph interface, a wavefront [DAN+11], or pure TBB tasks would be more feasible. x264 is a complex application with many approaches for parallelization; for a survey of the different partitioning strategies for parallelizing x264, refer to [HAD10].
2.4 Threats to Validity

As with any case study, there are certain limitations and threats to validity. First, our sample size of three applications is small. It is hard to predict how our results will generalize. We could have inadvertently selected a biased set of applications. We try to mitigate this bias by studying applications from an external benchmark, i.e., PARSEC. The original curators of the PARSEC benchmark focused on selecting a diverse set of applications from emerging domains [BKSL08]. The PARSEC benchmark has been constantly evolving and has been used by both practitioners and researchers worldwide [Pri].

Second, the 3-step process described in Section 2.2 was mainly based on the experiences of two developers parallelizing the applications (Eric Reed and myself). How can we generalize this process for other developers? While it is true that this 3-step process was gleaned from our experiences, we also consulted books on Intel’s TBB Pipeline construct [Rei07] and books describing the pipeline pattern [BMR+96, MSM04, OA10]. These external sources lead us to conclude that this is a natural way to apply the library constructs. Moreover, evaluation of our analyses and transformations that were implemented mimicking this 3-step process shows that it works well.

Finally, this case study concentrated on the constructs available in Intel’s TBB implementation. How would the results generalize to other parallel library implementations? We looked at two other popular flow-based libraries, Groovy’s GPars and Microsoft’s TPL Dataflow. We found that all three libraries support very similar constructs, e.g., nodes and channels (see Chapter 3).

2.5 Discussion

As highlighted in Section 2.2, there are always three similar steps involved in parallelizing an application: (i) partition into filters, (ii) identify tokens that pass through each filter and (iii) connect all filters into a pipeline.

The first step, i.e., partitioning, relies on the developer’s domain knowledge to choose a sensible partitioning scheme. It is unlikely that this step could be automated in a satis-
fiable manner. There are too many design decisions that rely on the developer's knowledge. The other two steps, however, are more structured and are suitable candidates for automation.

In our own experiences, identifying the tokens that need to be passed through each filter was the most tedious and error-prone task. We had to manually determine the dependencies between each filter and scrutinize each filter to ensure that there were no data races between different filters. It was easy to miss a dependency, causing a runtime error — the error was not caught during compile time because this version of TBB used void* pointers for tokens. It was also easy to introduce a data race when filters inadvertently accessed global variables, e.g., bookkeeping variables such as counters.

Automated tools that addressed these challenges would be a useful addition to a developer's toolkit. Tools that help identify the dependencies between filters would help prevent missing dependencies. Tools that check and warn about potential data races between filters would help prevent undefined behavior during run time.

In addition, having tools to automatically transform the sequential code to use the constructs of a parallel library would help reduce the time-consuming process of writing boilerplate code. As shown in Figure 2.2, even when using parallel library constructs, there is a non-trivial amount of boilerplate code that still needs to be written: creating a subclass of tbb::filter, overriding the operator() method and adding the filters to the pipeline.

Given the peculiarities of each application that we have encountered, it is unlikely that a single tool could ever fully automatically transform a complex sequential program to use pipeline parallelism. Pipeline parallelism is much more advanced than loop transformation and would require deeper analysis and more sophisticated tools. A more pragmatic solution, instead, would be to automate the main steps and rely on the developer for the application-specific steps (see Table 2.1).

The remaining chapters in this dissertation show how we can build tools to automate the main steps and reduce the burden of manual analysis and transformation.
Chapter 3

Interactive Source-to-source Transformations — A Walk-through

The previous chapter compared the performance and expressivity of parallel library constructs to their low-level threading counterparts. Additionally, it raised the idea of automating some of the repetitious transformation steps. We took those ideas and incorporated them into our source-to-source transformation tool, JFlow. This chapter presents a walk-through of how a developer would interact with JFlow.

While the previous chapter used C++ and Intel’s TBB, this and subsequent chapters use Java and Groovy’s GPars library. The decision to use Java instead of C++ was motivated by the availability of many analysis and transformation tools for Java that we could leverage. Specifically, we use the T.J. Watson Analysis Libraries (WALA) [201c] for analysis and the Eclipse Java Development Tools (JDT) [201b] for source-to-source transformation support. The core ideas that we present, however, are general and are not tied down to a particular language or library.

3.1 Parallelizing LIRE

Consider LIRe (16K SLOC), an open source Java content-based image retrieval (CBIR) system [LC08]. A CBIR system takes a query image, extracts features from it and, based on those features, retrieves similar images from a database of candidate images. A CBIR system relies on the features embedded in the contents of the query image itself rather than its metadata such as keywords, tags, date, location, etc. Typical features that are extracted using computer vision techniques include color histograms, shapes and textures. These features are represented in compact numerical forms, e.g., matrices that allow for
faster, though still computationally-intensive, comparisons for similarity.

In LIRe, the main bottlenecks to performance are the indexing and retrieval stages. This example focuses on the indexing stage. In the indexing stage, a set of images is analyzed to build the database of candidate images. During the analysis, features are extracted from the images and stored in the database. Because there isn’t a single feature that works across all images, multiple features are usually extracted and stored in the database. The bottleneck in the indexing stage comes from the number of features extracted and the number of images that need to be analyzed (typically in the hundreds of thousands for a decent size database).

Indexing is a recurring activity that benefits from parallelization. During the development phase of a CBIR system, different combination of features might be attempted and multiple databases might be created to evaluate the best set of features. Then, during the deployment phase, new images are constantly added to the database of candidate images to improve its retrieval results. Both of these phases rely heavily on indexing.

```java
for (String imagePath : FileUtils.getAllImages(new File(IMAGES_DIRECTORY))) {
    BufferedImage bufferedImage = ImageIO.read(new FileInputStream(imagePath));
    Document docJPEG = JPEGExtractor.createDocument(bufferedImage, imagePath);
    // End Node1

    Document docTamura = tamuraExtractor.createDocument(
        docJPEG, bufferedImage, imagePath);
    // End Node2

    Document docColor = autoColorCorrelogramExtractor.createDocument(
        docTamura, bufferedImage, imagePath);
    // End Node3

    Document docFCTH = FCTHExtractor.createDocument(
        docColor, bufferedImage, imagePath);
    indexWriter.addDocument(docFCTH);
    // End Node4
}
```

Figure 3.1: The indexing loop for LIRe with the different nodes annotated

Figure 3.1 shows the indexing loop for LIRe with four feature extractors: JPEG, Tamura [TMY78], Auto Color Correlogram [HKM+97], and Fuzzy Color and Texture Histogram (FCTH) [CB08]. Consider a developer who decides to parallelize this loop
using flow-based parallelism. She decides to partition each feature extractor into its own node. How could she make use of JFlow to parallelize her code?

Three steps:

- Annotate
- EXTRACT NODES
- INVERT LOOP

3.2 Annotate

First, she would need to annotate her desired partitioning; we use simple comments for now. The flow graph for her partition partition, i.e., a pipeline, is shown on the right. While there have been prior work [VRDB10, Sar91] on using the program dependence graph [FOW87] to perform automatic partition, our own experience shows that it does not work as well for the kinds of modern object-oriented programs that we are interested in. Complex heap data dependencies in the program dependence graph lead to very fine-grained partitions that have high communication costs when parallelized. We found it more effective to rely on the developer to provide a suitable partition as a starting point, as done by Thies et al. [TCA07] and Jenista et al. [JED11].

After annotating her code, the developer invokes JFlow. JFlow currently supports two refactorings. The ideas for both refactorings, EXTRACT NODE and INVERT LOOP, were conceived based on our past experiences parallelizing the applications discussed in Chapter 2.
3.3 Extract Nodes

The first refactoring that JFlow performs is **Extract Nodes**. This refactoring attempts to create nodes from the developer’s annotations and links each node with other nodes through their data dependencies. In its analysis step, it statically checks for *inter*-node data races (i.e., data races *between* nodes) and warns the developer of any. In its transformation step, it creates the nodes and the edges between them using the underlying library constructs.

The underlying construct to represent a node is a closure. The body of the closure contains the annotated statements for each node. Closures as a built-in language construct are supported in C++11 and C#. In the current version of Java, closures are expressed using anonymous inner classes. The underlying construct to represent an edge is a channel. A channel is a strongly-typed queue data structure. All three parallel libraries, i.e., GPars, TBB and TPL Dataflow, follow this same scheme of using closures and channels. Thus, our approach is very applicable to all three and we can easily target the constructs in each of them.

Figure 3.3 shows the generated code for Node2 after the **Extract Nodes** refactoring. Node2 is expressed as a `DataflowMessagingRunnable` anonymous inner class (lines 15 – 27). It consumes values from `channel1` using `getVal()` (line 27) and produces values into `channel2` using `bind(...)` (line 25). The values in `channel1` are provided by Node1. The channels are expressed as `DataflowQueues`. Typically, each data dependency is communicated via one dedicated channel. However, this leads to a proliferation of channels — making the code hard to read — and also comes with an associated performance cost — additional bookkeeping for each channel. We address both these issues by bundling data dependencies together (lines 1 – 5).

Observe that there is a non-trivial amount of boilerplate code that needs to be written. Imagine how tedious it would be for a developer to do this by hand. While some amount of verbosity stems from the lack of a direct syntax for closures in Java, the rest are actually required as part of using the underlying parallel library. Note that the generated code is not parallel yet. **Extract Nodes** is an intermediate refactoring step. We
class Bundle {
    BufferedImage bufferedImage;
    Document docColor, docJPEG, docTamura;
    String imagePath;
}

final DataflowQueue<Bundle> channel0 = new DataflowQueue<Bundle>();
final DataflowQueue<Bundle> channel1 = new DataflowQueue<Bundle>();
...
for (String imagePath : FileUtils.getAllImages(new File(IMAGES_DIRECTORY))) {
    Bundle b = new Bundle();
    b.imagePath = imagePath;
    channel0.bind(b);
...
    new DataflowMessagingRunnable() {
        @Override
        protected void doRun(Object... args) {
            Bundle b = ((Bundle) args[0]);
            BufferedImage bufferedImage = b.bufferedImage;
            Document docJPEG = b.docJPEG;
            String imagePath = b.imagePath;
            Document docTamura = tamuraExtractor.createDocument(
                docJPEG, bufferedImage, imagePath);
            b.docTamura = docTamura;
            channel2.bind(b);
        }
    }.call(channel1.getVal());
...
}

Figure 3.3: The closure representing Node2 and the channels connecting it

offer the developer the opportunity to explore the code and make simple tweaks as desired. For instance, perhaps she would like to make use of other refactorings such as RENAME VARIABLE to rename the generic channels names from channel10 to something more intention-revealing.

3.4 Invert Loop

The second refactoring that JFlow supports is INVERT LOOP. INVERT LOOP locates all the DataflowMessagingRunnable objects in the loop and moves them into a computation flow graph. Intuitively, it inverts the nodes in the loop into a new construct, a FlowGraph object, allowing the nodes to run in parallel. Internally, a FlowGraph object maintains its own thread pool and coordinates the execution of node.

Figure 3.4 shows the code after the INVERT LOOP refactoring. Line 1 creates a new FlowGraph object, fGraph. Lines 3 – 16 register the DataflowMessagingRunnable for Node2 with the FlowGraph object through the fGraph.operator(...) construct.
FlowGraph fGraph = new FlowGraph();
...

fGraph.operator(Arrays.asList(channel1), Arrays.asList(channel2), 4,
new DataflowMessagingRunnable() {
  @Override
  protected void doRun(Object... args) {
    Bundle b = ((Bundle) args[0]);
    BufferedImage bufferedImage = b.bufferedImage;
    Document docJPEG = b.docJPEG;
    String imagePath = b.imagePath;
    Document docTamura = tamuraExtractor.createDocument(
        docJPEG, bufferedImage, imagePath);
    b.docTamura = docTamura;
    channel2.bind(b);
  }
});

for (String imagePath : FileUtils.getAllImages(new File(IMAGES_DIRECTORY))) {
  Bundle b = new Bundle();
  b.imagePath = imagePath;
  channel0.bind(b);
}

// Wait for all computation in the FlowGraph to complete
// and then continue execution
fGraph.waitForAll();

Figure 3.4: Inverting the loop so that each node can operate in parallel. Additionally, 
Node1, Node2 and Node3 can operate in a data parallel manner.

The FlowGraph object, at this time of writing, is not part of the official GPars release. 
It is a construct that we have added to better support flow-based parallelism. Without 
the FlowGraph construct, a developer would have to manually keep track of each node 
and manually manage its starting and termination.

For each node, JFlow checks if it is possible to run the node in a data parallel manner 
without any intra-node data races (i.e., data races within nodes). If so, it informs the 
user of this possibility. The user then needs to decide if this is suitable for her particular 
application. Operating in a data parallel manner improves throughput but does not 
preserve the ordering of processed items. For instance, in the LIRe example, the images 
will arrive out-of-order if they are processed in a data parallel manner.\(^1\) Allowing this is 
a decision that the developer needs to make. It is impossible to statically infer this from 
the sequential code alone. The sequential code is over-constrained and always imposes an 
ordering even when one is not necessary [Kno09]. If we strictly enforce the sequential

\(^1\)TBB supports in-order processing by tagging items. Each item is tagged with its arrival order. The 
receiving node keeps track of the order and buffers items until it receives an item with the right order. This 
is convenient but incurs some overhead.
ordering, we miss many parallelization opportunities.

In the LIRe example, JFlow informs the user that Node1, Node2 and Node3 can operate in a data parallel manner. She decides that it is permissible for items to be processed out-of-order. The pipeline for LIRe is shown on the right (see Figure 3.5) with some of its nodes operating in a data parallel manner. The third parameter to operator on line 3, represents the degree of parallelism to use. By default, JFlow uses the number of cores on the machine. The user can tune this by hand using an external profiler, or an auto-tuning technique [SQKP10].

The original loop serves as a generator that will provide the initial data to Node1. In the figure, the for loop will generate the imagePath variable that is used in the first and subsequent nodes. After the FlowGraph object finishes its execution, it resumes serial execution with the rest of the code (Line 20).

3.5 User Interface

The previous sections presented a general walk-through of how an interactive source-to-source transformation tool would work. This section presents the user interface of the tool as it is currently implemented using the source-to-source transformation utilities provided by the Eclipse IDE. Figure 3.6 shows the interface for the Extract Nodes refactoring, whereas Figure 3.7 shows the interface for the Invert Loop refactoring. Both interfaces are based on a modal dialog that show a preview of the changes that are about to be made in the source. The top panel of the dialog categorizes the changes to be made while the bottom panel shows the actual changes in source code. The developer can decide to continue or to abort at any time. If she continues, she will still have the opportunity to tweak the source code in the editor by hand.
Figure 3.6: The user interface for **Extract Nodes**
Figure 3.7: The user interface for INVERT LOOP.
Chapter 4

A Brief Introduction to Pointer Analysis

Our approach makes heavy use of pointer analysis. This chapter provides the background necessary for the next chapter. It presents a brief introduction to pointer analysis and discusses the trade-offs between precision and scalability. Pointer analysis is a vast topic and this chapter does not try to be a complete survey of pointer analysis; for that the reader can consult [HP00, Lho03, Lho06, Sri07].

4.1 Introduction

Accurate information about the behavior of pointers is an important prerequisite for many analyses, including our own analyses for flow-based parallelism. Pointer analysis is a compile-time analysis that attempts to determine the set of objects\(^1\) that a variable can point to. In general, the exact runtime values of each pointer in a program are undecidable [Lan92, Ram94]. Thus, many approximation techniques have been proposed, with varying degrees of precision and scalability. Because these techniques tend to be conservative, we sometimes say that they over-approximate the possible objects that a variable may point to during any execution of the program.

Consider the simple statement \(x = \text{new Object}()\), where \(x\) is a variable. How would we represent the results of pointer analysis for the variable? Traditionally, the results of pointer analysis are represented as points-to sets. Define \(\text{points} - \text{to}(x)\) as the set of objects that the variable \(x\) may reference. In this case, the object allocated through

\(^1\)We use the term \textit{object} because we are dealing with type-safe object-oriented languages such as Java. In languages such as C/C++, which allow direct pointer manipulation, pointers can point to any memory location. In those languages, it is common to use the term “memory locations” instead of “objects”.
the call to `new Object()` is an element of `points − to(x)`.

Recall that pointer analysis is an over-approximation technique. Consider an object `o`. If `o ∈ points − to(x)`, then the variable `x` may point to `o`. It is possible that `x` never points to `o`. On the other hand, if `o ∉ points − to(x)`, that means that the variable `x` can never point to `o`.

Thus, a conservative pointer analysis could say that every variable may point to every object. This is correct but is not very useful for most analyses. The goal of pointer analysis is to be sufficiently precise for the client analysis, e.g., live variable analysis, dead assignment identification, etc. that relies on it.

4.2 Heap Abstraction

Now consider the simple loop below:

```java
for (...) 
  x = new Object();
```

During runtime, the loop may allocate many (potentially unbounded if the loop does not terminate) objects through the call to `new Object()`. How would we represent these runtime objects?

We need a model of the program’s possible runtime objects. That model is provided by a particular heap abstraction. Many different heap abstractions exist. Some are coarser

43

grain, e.g., they treat all objects of a particular type as one single object in the pointer analysis. Others are more fine-grained, e.g., they keep track of the iteration of the loop, up to some bounded number [LTNS10]. One heap abstraction that is used frequently and is sufficiently precise, is based on allocation-site. This abstraction was first formalized by Andersen [And94].

Intuitively, it provides a label (e.g., line 2 in the code listing above) for each `new` in the program. For languages such as Java, it is simple to use the index of the `new` instruction in its bytecode as the label. In this dissertation, we will refer to the allocation site by its line number and write `O_n` for the allocation site at line `n`. We say that `O_2 ∈ points − to(x)`, where `O_2` is the abstract object representing all objects that could be potentially allocated
at that site. Figure 4.1 shows how we can represent \( \text{points} - \text{to}(x) \) graphically. We will use this same graphical representation throughout the dissertation.

\[ \begin{array}{c}
\text{x} \\
\rightarrow \\
\text{O}_2
\end{array} \]

Figure 4.1: Representing \( \text{points} - \text{to}(x) \) graphically

We use the allocation-site heap abstraction in our analyses in Chapter 5. For brevity, we henceforth use the term object without qualifying it as an abstract object.

### 4.3 Equality-based vs. Subset-based Assignments

Consider the example below:

```java
1 y = new Object();
2 z = new Object();
3 x = y;
4 x = z;
```

Figure 4.2: Example to illustrate different approaches for handling assignment statements

From the previous section, we know that after line 2, \( O_1 \in \text{points} - \text{to}(y) \) and \( O_2 \in \text{points} - \text{to}(z) \). What happens after the assignment statements on lines 3 and 4? There are two common approaches to handle assignments: equality-based or subset-based.\(^2\) Both approaches model assignments as a set of constraints to be solved.

The equality-based approach was proposed by Steensgaard along with an algorithm that computes it in nearly linear time with respect to program size [Ste96]. An equality-based approach handles assignments by generating the constraint \( \text{points} - \text{to}(x) = \text{points} - \text{to}(y) \): the variable \( x \) points to all objects that \( y \) could point to and the variable \( y \) points to all objects that \( x \) could point to. Intuitively, it adds the inverse assignment \( y = x \) to the analyzed code fragment. In our code example above, it generates the constraints \( \text{points} - \text{to}(x) = \text{points} - \text{to}(y) \) (line 3) and \( \text{points} - \text{to}(x) = \text{points} - \text{to}(z) \) (line 4). Solving these constraints yields identical points-to sets for all the variables, i.e.,

\(^2\)In the pointer analysis literature, the equality-based approach is sometimes referred to as a unification-based approach or Steensgaard-style (after its inventor). Similarly, the subset-based approach is also known as an inclusion-based approach or Andersen-style (after its inventor).
\( \text{points} - \text{to}(x) = \text{points} - \text{to}(y) = \text{points} - \text{to}(z) = \{O_1, O_2\} \). Figure 4.3a shows this relation graphically, illustrating the spurious edges that occur between variables \( y \) and \( z \).

The subset-based approach was proposed by Andersen along with an algorithm that computes it in cubic time with respect to program size [And94]. A subset-based approach handles assignments by generating the constraint \( \text{points} - \text{to}(y) \subseteq \text{points} - \text{to}(x) \). This constraint is more precise than the one generated by an equality-based approach. It only updates \( x \) to point to whatever \( y \) points, and not vice-versa. In our code example above, this approach generates the constraints \( \text{points} - \text{to}(y) \subseteq \text{points} - \text{to}(x) \) (line 3) and \( \text{points} - \text{to}(z) \subseteq \text{points} - \text{to}(x) \) (line 4). Solving these constraints yields the precise results: \( \text{points} - \text{to}(y) = \{O_1\} \), \( \text{points} - \text{to}(z) = \{O_2\} \), \( \text{points} - \text{to}(x) = \{O_1, O_2\} \). Figure 4.3b shows this relation graphically.

As can be seen in the figures, the subset-based approach is more precise. It does not add spurious edges between variables. A disadvantage of the subset-based approach is its worst-case cubic time complexity. However, empirical results have shown that, in practice, it is possible to compute the points-to results in quadratic time [SF09]. Empirical results have also shown that the subset-based approach offers a measurable difference in precision compared to the equality-based approach [HP00]. In this dissertation, we use the subset-based approach.
4.4 Flow Sensitivity

The astute reader might have noticed that we have a source of imprecision in the example from Figure 4.2. After the assignment on line 4, x can only point to the object $O_2$. The previous assignment to the object $O_1$ on line 3 no longer holds on line 4. This is termed a *strong update*: it models the fact that the old value in $x$ is overwritten. This more precise model requires reasoning about the points-to relation at each program point, i.e., each program statement.

We call such an approach a *flow sensitive* pointer analysis. It computes the points-to relation at each program point. Figure 4.4 shows the points-to relation at each program point for the statements in Figure 4.2.

```plaintext
y = new Object();
z = new Object();
x = y
x = z
```

![Diagram of points-to relations](image)

Figure 4.4: The results of a flow-sensitive pointer analysis

Flow sensitive analyses take into account the control flow of the program. They more closely approximate how the program will actually execute. However, they are also more expensive. As shown in Figure 4.4, the analyses need to store the points-to relations at each program point. This becomes prohibitively expensive for large programs.
Thus, for scalability, most approaches for pointer analysis are flow insensitive. They assume that statements within a method can execute (i) in any order and (ii) any number of times. Such analyses typically use a worklist approach to solve the points-to constraints (see Section 4.3) and iterate until a fixed-point is reached [Lho03]. Hasti and Horwitz have shown that using single static assignment (SSA) as an intermediate representation could provide some of the precision of a flow sensitive approach [HH98]. We take advantage of this in our own analyses. Our analyses builds upon the T.J. Watson Analysis Libraries, which uses SSA as its intermediate representation [201c].

Though flow sensitivity could theoretically offer better precision, in practice, for most clients that rely on pointer analysis, the increase in precision is not significant. Hind et al. conducted an empirical study measuring the precision offered by flow sensitivity for four clients: mod/ref analysis\(^3\), live variable analysis, reaching definition analysis and interprocedural constant propagation [HP00]. They found that the use of flow sensitivity does not provide significant gains in precision; although it consumed more resources and took longer to run. These results were also confirmed in a different study by Ryder et al., where they compared the precision and scalability of flow sensitive and flow insensitive mod/ref analysis [RLS+01]. Ryder et al. found that a flow sensitive approach was about 20% more precise than a flow insensitive approach; however, it also took about an order of magnitude longer to run. Therefore, we use a flow insensitive pointer analysis in this dissertation. Unless otherwise noted, all future examples in this dissertation assumes a flow insensitive approach.

Flow sensitivity, nonetheless, plays an important role for other clients. For instance, in detecting resource leaks, flow sensitivity is essential. Without flow sensitivity, the analysis becomes overly conservative and raises too many false positive [FYD+08].

```java
if (...) 
  a = x;
else 
  b = x;
```

Figure 4.5: Example to illustrate the benefits of path sensitivity

\(^3\)Our static data race detection relies on mod/ref analysis. See Chapter 5.
Further precision might be possible through a \textit{path sensitive} approach. A path sensitive approach distinguishes between different control flow paths. In Figure 4.5, our current flow insensitive analysis would imprecisely conclude that \(a\) and \(b\) may be aliased at the end of the example. Our analysis does not consider the condition of the \texttt{if} statement, and assumes that both the statements on line 2 and 4 could execute, even though they could not in an actual execution of the program. A path sensitive approach would keep the effects of the \texttt{if} branch separate from the \texttt{else} branch. Like flow sensitivity, path sensitivity is expensive. The precision gained from path sensitivity might not justify its cost for some of the client analyses.

4.5 Field Sensitivity

In the examples so far, we have only considered points-to sets of local variables. However, in object-oriented programs, we have objects with fields that can point to other objects. How does pointer analysis handle fields? There are three main approaches (in order of precision): field insensitive approach, field-based approach and field sensitive approach.

The most conservative (and imprecise) approach is to ignore fields, i.e., field insensitive. The analysis does not distinguish between fields of an object and, instead, treats the entire object as a black-box.

When the analysis takes fields into account, there are two approaches. A field-based approach uses a single points-to set for each field of a declared type (a class in object-oriented programs). A field sensitive approach, on the other hand, creates a separate points-to set for each field of each object. Thus, a field sensitive approach could distinguish between the same field of two different objects, whereas a field based approach could not.

We illustrate these concepts with the simple program shown in Figure 4.6.

Figure 4.7a shows the points-to relation for a field insensitive approach. The assignment \(p.f1 = x\) creates an edge from \(O_6\) to \(O_8\) but does not keep track of the particular field. If we add the statement \(z = p.f2\) to the end of Figure 4.6, the pointer analysis for a field insensitive approach will say that (i) \(O_8 \in \text{points } to(p.f2)\), al-
though \( f_2 \) was never assigned (since it does not distinguish different fields), and (ii) that \( O_8 \in \text{points} – \text{to}(z) \) after the assignment. Field insensitive approaches can be very fast – processing up to a million lines of C code in less than a second — and very parsimonious with memory – consuming only 10 MB of memory [HT01]. However, ignoring fields is usually too imprecise for most client analyses and, thus, this approach is not used as often.

One particular area where a field insensitive approach is used frequently is for array accesses. Most analyses frameworks for Java are index insensitive; they do not distin-
guish between the individual elements of the array [201a, NAW06, SCD¹3, VRG⁺99].

Keeping track of array indices in modern object-oriented programs is more challenging compared to languages such as Fortran with statically declared arrays. Arrays are usually dynamically allocated and it is hard to keep track of their respective dimensions.

Figure 4.7b shows the points-to relation for a field-based approach. In this case, two pseudo variables, i.e., \( A.f1 \) and \( A.f2 \) have been created to represent the fields of objects of the class \( A \). The relation shows that there is at least one object of type \( A \), whose \( f1 \) field points to object \( O_8 \). On the other hand, field \( f2 \) of objects of type \( A \) has never been assigned. If we add the statement \( z = r.f1 \) to the end of Figure 4.6, a field-based pointer analysis will conclude that \( O_8 \in z \) although \( r.f1 \) was never assigned (since it cannot distinguish which particular object had its \( f1 \) field assigned).

Finally, Figure 4.4 shows the points-to relation for a field sensitive approach. In this case, the edge from \( O_6 \) to \( O_8 \) has been labeled with the particular field, \( f1 \). This is more precise compared to Figure 4.7b because we know that the \( f1 \) field of object \( O_6 \) points to object \( O_8 \). In the field-based approach, we only know that some object of type \( A \) points to object \( O_8 \) through its \( f1 \) field.

In our analyses, we use a field sensitive approach. Though more expensive, the precision it offers is necessary to be able to identify the fields and the particular objects that are involved in a potential data race. A field insensitive or field-based approach is too conservative and would make pinpointing the potential data race difficult.

Earlier in this chapter, we mention that pointer analysis is a compile-time analysis that attempts to determine the set of objects that a variable can point to. To be more precise, pointer analysis approximates the set of objects that a variable or an object field reference can point to. Formally, we express this as \( \text{points-to} : \{\text{Variables} \cup (\text{Object} \times \text{Field})\} \rightarrow \mathcal{P}(\text{Object}) \), where \( \mathcal{P}(\text{Object}) \) represents the power set of the set of objects in the program.
4.6 Context Sensitivity

The examples we have presented so far have only considered intraprocedural pointer analysis, i.e., we have not discussed method calls. This section discusses interprocedural pointer analysis and how we can handle them.

```java
1 Object identity(Object o) {
2     return o;
3 }
4
5 a = new Object();
6 b = new Object();
7 c = identity(a);
```

Consider the example shown in Figure 4.8. We have a simple function, `identity`, that returns the value of its parameter `o`. For the method invocation on line 7, we model parameter passing via the assignment `o = a` and the return value using another assignment `return_identity = o`. Then we model the assignment to the variable `c` using `c = return_identity`. Figure 4.9 shows the points-to relation for this example.

```
Figure 4.8: Example to illustrate method calls
```

```
Figure 4.9: The points-to relations for the example in Figure 4.8
```

Consider the example shown in Figure 4.10. The only difference from the previous example is the additional call to the method `identity` on line 8. With a context insensitive approach, we do not distinguish between the two different invocations — there is only one single context. We model parameter passing using the assignments `o = a` (for the call on line 7) and `o = b` (for the call on line 8). We model the return values using the assignments `c = return_identity` and `d = return_identity`.

```
Figure 4.10: Example to illustrate method calls
```

```
Figure 4.9: The points-to relations for the example in Figure 4.8
```
Figure 4.10: Example to illustrate the benefits of context sensitivity

Figure 4.11 shows the points-to relations for the example in Figure 4.10. Without distinguishing the different method invocations, we end up with the imprecise result that $points \rightarrow (c) = points \rightarrow (d) = \{O_5, O_6\}$.

What is needed is a way to distinguish the different method invocations. Such an analysis is called a context-sensitive analysis. There are two approaches for distinguishing contexts: cloning-based and summary-based. A cloning-based approach clones the method that is being invoked, generating a new context for each clone. A summary-based approach models method calls as a graph-reachability problem and uses a tabulation algorithm to solve the constraints. The summary-based approach is more precise but less scalable [SFB07]. In this dissertation, we consider only cloning-based approaches. Reps et al. discuss the summary-based approach in their seminal paper “Precise Interprocedural Dataflow Analysis via Graph Reachability” [RHS95].

Figure 4.12 shows the points-to relation for the example in Figure 4.10 with a context sensitive analysis. We distinguish the calls to `identity` with two different contexts, `Context1` and `Context2`. This creates two different variables assignments for the pa-
rameter \( o \), i.e., \([\text{Context1}]o = a\) and \([\text{Context2}]o = b\). Similarly, we have two different variables to model the return values. By distinguishing the different method invocations, we obtain the more precise result that \( \text{points} \to \text{to}(c) = O_5 \) and \( \text{points} \to \text{to}(d) = O_6 \).

![Diagram of points-to relations](image)

Figure 4.12: The points-to relations for the example in Figure 4.10 with a context sensitive analysis

There are many different approaches for deciding on how to clone a method. Two frequently used approaches are call-string [SP81] and object sensitivity [MRR05]. The call-string approach represents invocation contexts using a string of \( n \) enclosing call sites. The parameter \( n \) is necessary to bound the number of contexts that can be generated. In contrast, object sensitivity uses the receiver object at an instance method invocation (the implicit this parameter for object-oriented programs), to distinguish different calling contexts. As with the call-string approach, it is typical to bound the number of contexts generated. K-object sensitivity keeps track of a sequence of, at most, \( k \), object allocation sites representing the receiver object, i.e., objects are named by the sequence \( o_1, o_2, \ldots, o_k \).

It is possible to combine different approaches for context sensitivity to tune for both precision and scalability. Smaragdakis et al. have formalized this and conducted an empirical study measuring the precision and scalability of combining different context sensitive approaches. Our own analyses (see Chapter 5) also combines multiple approaches for scalability reasons.

### 4.7 A Realistic Example

We present a realistic example showing how all the different ideas discussed come together. Figure 4.13 models a simple recipe book. Each recipe has a list of instructions and photos. The pointer analysis we use is subset-based, flow insensitive, field sensitive and context sensitive using k-object sensitivity.
Figure 4.14 shows the points-to relation for the variables and object reference fields from the example program. We use \([O_xO_y]\) to denote the context for k-object sensitivity. We use \(k = 2\). For instance, the object labeled \([O_5O_9]O_{23}\) refers to the object allocated on line 23 (new \(\text{Object[]}\)) in the context of the method invoked on the object allocated on line 9 (new \(\text{List()}\)), in the context of the method invoked on the object allocated on line 5 (new \(\text{Recipe()}\)). This allows us to distinguish this \(\text{Object[]}\) array from the one allocated in context \([O_5O_{10}]\). Without context sensitivity, our analysis will not be able to distinguish between the two \(\text{Object[]}\) arrays allocated in the two different calls to the \(\text{List}\) constructor. Context-sensitivity also allows us to distinguish the local variable \(\text{temp}\) in the two separate calls to the \(\text{List}\) constructor.

```java
1 class Recipe {
2     List instructions, photos;
3
4     static void main(String[] args) {
5         Recipe r = new Recipe();
6     }
7     
8     Recipe() {
9         instructions = new List();
10         photos = new List();
11     }
12     for(int i = 0; i < MAX; i++) {
13         Instruction instr = new Instruction();
14         instructions.elements[i] = instr;
15     }
16 }
17 }
18 }
19 class List {
20     Object[] elements;
21     
22     List() {
23         Object[] temp = new Object[MAX];
24         elements = temp;
25     }
26 }
```

Figure 4.13: A realistic example of pointer analysis

Lines 13 – 14 of Figure 4.13, populate the instruction array with new \(\text{Instruction}\) objects. As shown in Figure 4.14, we do not distinguish between the different indices of the array. Instead, we treat all all access through the single index \([*]\). Our analysis is index-insensitive for arrays.
4.8 Call Graph Analysis

One topic that we have not discussed is method resolution. How do we determine which methods can be invoked at a particular call site? For object-oriented languages, it is not always possible to statically determine which method is invoked. The use of polymorphism and inheritance complicates the analyses because the method to be invoked depends on the dynamic type, not the statically declared type.

Approximating the calling relations between methods in a program is the role of call graph analysis. A call graph consists of vertices that model methods in particular contexts. The edges represent calling relations: an edge from a vertex \( V_1 \) to the vertex \( V_2 \) means that there is a call site in \( V_1 \) that could possibly invoke \( V_2 \) during program execution. Figure 4.15 shows the call graph for the example in Figure 4.14. Observe how the call graph is context sensitive: there are two different vertices to model the two
different invocations of the constructor of `List`. The initial node, called the `entrypoint`, is usually the call to the `main()` method.

A conservative call graph analysis would use the statically declared type of a variable to determine what methods it can invoke. Conservatively, it will examine the inheritance hierarchy and determine that any method overridden by a subclass could be invoked. This approach is usually called an `ahead-of-time` call graph. There are many variations of this approach. The basic `Class Hierarchy Analysis` only considers the statically declared type of a variable and its class hierarchy. The improved `Rapid Type Analysis` checks that a subclass must have been instantiated at least once in the program (there must be at least one call to `new` for that subclass) for its overridden methods to be possible choices. Tip and Palsberg described various refinements to this approach and reported empirical results on their precision [TP00].

A more precise method would be to rely on pointer analysis to determine the possible dynamic types of a variable and use that to enumerate the possible target methods. However, pointer analysis is dependent on call graph analysis for method resolution. Thus, the two approaches have to proceed simultaneously. The call graph that results is called an `on-the-fly` call graph. Grove and Chambers detail how such an approach works [GC01]. Our analyses in Chapter 5 rely on this on-the-fly approach for precision.
Chapter 5

The Core of Our Approach — Analyses and Transformations

This section details the analyses we perform to determine if the Extract Nodes and Invert Loop transformations can be performed safely. We use the T.J. Watson Analysis Libraries (WALA) [201c] for the analyses and the Eclipse Java Development Tools (JDT) [201b] for its source-to-source transformation support.

The heart of our analyses is a novel approach for detecting inter and intra node data-races. Our approach combines k-object sensitivity with ownership transfer inference — an approach that works particularly well for flow-based parallelism with their regular flows of coarse-grained data.

5.1 Extract Nodes

5.1.1 Analyses

The analyses for the Extract Nodes transformation check if the annotated nodes can be safely partitioned into a pipeline. They check that there are no abnormal control flow, no loop-carried dependencies and no inter-node data races. These are the preconditions for the transformation to be performed [Rob99]. Figure 5.1 presents an overview of the analyses.

We require that the developer annotate the original program to partition the statements in the original loop into N nodes. While there have been prior work [VRDB10, Sar91] on using the program dependence graph [FOW87] to perform automatic partition, our own experience shows that it does not work as well for the kinds of modern object-oriented programs that we are interested in. Complex heap data dependencies in the
program dependence graph lead to very fine-grained partitions that have high communication costs when parallelized. We found it more effective to rely on the developer to provide a suitable partition as a starting point, as done by Thies et al. [TCA07] and Jenista et al. [JED11].

After the Extract Nodes transformation, the original loop in the program will serve as an iterator. It will iterate over all the elements to be processed and generates the data for the first extracted node in the pipeline. For instance in the LIRe example from Chapter 3, its top-level for loop generates new images that are fed to the first node of the pipeline.

We used a stage-based analysis. Earlier stages are less expensive to run; we use their results to determine if we need to run the later, more expensive stages or to abort the analysis. This allows us to provide prompt feedback to the developer without incurring the cost of unnecessary analysis. We detail each stage below.

Figure 5.1: Overview of the analyses workflow for the Extract Nodes transformation

Check Control Dependencies

Nodes in a pipeline can have both control and data dependencies. We handle control dependencies implicitly by constraining each node to have a single exit. The first stage of our analysis builds a control flow graph for the program and checks that each node obeys this single exit constraint. One consequence of this is that a node should not throw an exception that it does not catch (see discussion in Section 8.2.1). Other popular transformations such as the Extract Method refactoring found in modern IDEs also enforce this constraint to simplify the analysis and transformation.

If a node violates this constraint, we inform the developer and abort the rest of the
analysis. While this constraint might seem overly restrictive, it works well for the applications that we have examined. Most applications can also be transformed manually to handle the exceptions within the same node. Current flow-based libraries also do not provide much support for handling control dependencies; they focus on data dependencies and leave the managing of control dependencies to the developer.

Compute Data Dependencies

The second stage of the analysis computes the data dependencies between nodes and determines how to route the values of variables between nodes. For instance, in Figure 3.1, Node2 consumes the values of variables docJPEG, bufferedImage and imagePath from Node1; in turn, it produces the value of variable docTamura for Node3.

To compute the data dependencies between nodes, we first build a data dependence graph for the method that contains the annotated nodes.\(^1\) Let \(V\) be the set of variables and \(S\) be the set of statements in the program. A data dependence graph contains vertices that represent program statements and edges that represent data dependencies between statements. A data dependence exists between statements \(s_1, s_2 \in S\) if \(s_1\) defines a variable \(v \in V\) and \(s_2\) uses \(v\), and there is no intervening definition of \(v\) along the execution path from \(s_1\) to \(s_2\). Let all data dependencies in the enclosing method be represented as tuples of the form \(\langle s_1, v, s_2 \rangle \in D \subseteq S \times V \times S\), where \(s_1\) defines the variable \(v\) that \(s_2\) uses.

Each node, \(n \in N\) has a set of input and output data dependencies. Denote the statements in node \(n\) by the set \(S_n\). Define the set of input dependencies for a node as \(IN(n) = \{\langle s_1, v, s_2 \rangle \in D \mid s_1 \notin S_n \land s_2 \in S_n\}\). Intuitively, \(IN(n)\) denote the data dependencies coming into the current node from outside, which is why we ignore statements internal to a node (the \(s_1 \notin S_n\) clause). Similarly, define the set of output dependencies as \(OUT(n) = \{\langle s_1, v, s_2 \rangle \in D \mid s_1 \in S_n \land s_2 \notin S_n\}\). Intuitively, \(OUT(n)\) denote the data dependencies going out of the current node, again ignoring statements internal to a node.

Finally, define the function \(ContainingNode : S \rightarrow N \cup \{\text{generator}\}\) that maps

\(^1\) We leverage WALA’s use of static single assignment (SSA) form as its intermediate representation. SSA offers the same information as a data dependence graph but in a more compact representation \([CFR^*91]\).
statements to nodes. Recall that the original loop serves as a generator that provides the necessary initialization data that are not produced by any of the nodes, e.g., loop variables. Using the functions \( IN(n) \), \( OUT(n) \) and \( ContainingNode(s) \), we determine how to route variables between nodes. Figure 5.2 illustrates these functions for a simple example.

![Diagram of node connections](image)

**Figure 5.2: Keeping track of dependencies between nodes**

In the LIRe example, one of the elements of \( IN(Node2) \) is \( (S\#5, docJPEG, S\#9) \), where \( S\#X \) represents the statement at line \( X \) in Figure 3.1. \( ContainingNode(S\#5) = Node1 \) so we know that \( docJPEG \) comes from \( Node1 \).

If we discover a loop-carried dependency during the process of building the data dependence graph, we inform the developer and abort the rest of the analysis. A loop-carried dependency essentially serializes the execution since it needs to wait for the previous iteration to complete before continuing. Our current analyses and transformations do not support forwarding of values between different iterations (we do not track variables and the iteration that produces/consumes them). Future work could investigate how to keep track of iteration values to support loop-carried dependencies.

The previous stage computes the dependencies between nodes and determines how to route them between nodes. For primitive values such as integers, doubles, booleans, etc. this amounts to copying the values from one node to another. Because primitive values are copied, each node operates on its own copy and, thus, there are no data races.

For reference values, i.e., references to objects on the heap, it is not always feasible to copy the object between each node. Copying an object requires serializing all of its
field and the objects those fields point to transitively. This is an expensive operation; in a shared memory environment, it would be better to refer to such objects by reference. However, because objects are not copied, it is possible for different nodes to access (read and/or write) the same object at the same time, causing a possible data race.

The remaining two analyses for Extract Node detect potential data races between objects on the heap. The developer can then decide on how to eliminate the data race: she can either decide to copy the object or protect concurrent access through the use of locks. Our tool currently does help eliminate data races; it only detects them.

Compute side effects

This stage of the analysis computes the possible side effects to the heap for each statement. The results of this stage are used in the final stage of our analysis to determine if two nodes might have a data race. We distinguish read and write effects to objects on the heap. There is a data race between two nodes if their contained statements may access the same object on the heap and at least one of those accesses is a write.

Computing the side effects requires both call graph and pointer analysis. Call graph analysis attempts to approximate the calling relations between methods in a program. Pointer analysis is a compile-time analysis that attempts to determine the set of objects pointed to by a variable or a reference object field. The results of both analyses allow us to track read and write accesses to the heap across method calls. Call graph and pointer analysis are highly interdependent; we rely on WALA’s on-the-fly call graph construction that builds the call graph and performs pointer analysis simultaneously, yielding better precision [GC01].

Our call graph analysis employs k-object sensitivity, a notion of context sensitivity proposed by Milanova et al. [MRR05] and recently shown by Naik et al. [NAW06] to be effective for detecting data races in object-oriented programs. Object-sensitivity uses the receiver object at an instance method invocation (the implicit this parameter for object-oriented programs), to distinguish different calling contexts. Static methods that lack a receiver object have an empty context $\varepsilon$. K-object sensitivity keeps track of a sequence of, at most, $k$, object allocation sites representing the receiver object, i.e., objects are named
by the sequence $o_1, o_2, \ldots, o_k$. In general, the larger the value of $k$, the more precise the results, albeit at the expense of scalability. We use $k = 2$ in our analysis.

Our pointer analysis is a subset-based, flow insensitive, field sensitive and context sensitive analysis\(^2\) [SCD\(^+\)13]. Abstract objects are distinguished based on their allocation sites in a particular context, i.e., $k$-object sensitivity, in our case. This allows us to treat context and objects uniformly, as done in Naik et al. [NAW06]. This uniformity plays a key role in the final stage of our analysis.

$k$-object sensitivity, however, is expensive. Java programs frequently make heavy use of the classes provided through the Java Development Kit (JDK). The JDK is a complex library. The version of the JDK that comes with Java 1.6 is about 1 millions SLOC. This means that, in the worst case, analyzing a Java program could involve analyzing the entire 1 million SLOC of the JDK. Using $k$-object sensitivity to analyze both the developer’s application and the relevant parts of the JDK consumes large amounts of time and memory. When we tried this approach on one of our smaller benchmarks, the pointer analysis completed in about 10 minutes and consumed about 8 GB of memory. This is not acceptable for use in an interactive tool.

Sridharan et al. [SCD\(^+\)13] and Naik [Nai08] have observed this problem in their own analysis of modern Java programs and advocate using a demand-driven approach: increase precision where it matters for the client analysis. In our case, our analyses would benefit from having higher precision from user-defined objects but would not be adversely affected if we have lower precision for JDK objects. Therefore, we tune our analyses as such. For user-defined objects, we used $k$-object sensitivity; for JDK objects, we use a type-based context sensitivity, where we distinguish context based on the declared type of the this parameter.

The idea of type-based context sensitivity has been used in the pointer analysis community for some time; however, it was first formalized in Smaragdakis et al. [SBL11]. Like $k$-object sensitivity, their notion of type-based sensitivity allows specifying a sequence of, at most $n$, types, i.e., $t_1, t_2, \ldots, t_n$. We use $n = 1$ in our implementation. Future work could evaluate how increasing the value of $n$ might improve our precision.

\(^2\)These terms are explained in Chapter 4
Using type-based context sensitivity improves scalability at the expense of precision. Chapter 6 discusses the impact that this has when parallelizing an application.

Our side effects analysis is based on an existing interprocedural mod-ref analysis by Ryder et al. [RLS+01]. We compute the effects on instance fields, static fields and array elements. We use the results from our pointer analysis to determine what objects they may point to. For arrays, our analysis is index insensitive, i.e., it does not distinguish between individual elements of the array. Index insensitivity is less precise but more scalable; it’s the approach taken in most analysis frameworks for Java [201a, NAW06, SCD+13, VRCG+99].

More precisely, we keep track of the effects of the following heap-accessing statements, where $x$, $y$ and $i$ are local variables:

- **Instance Fields** $y = x.f$ (resp. $x.f = y$) that read (resp. write) instance field $f$ of the set of objects that $x$ may point to.

- **Static Field** $y = C.s$ (resp. $C.s = y$) that read (resp. write) to the static field $s$ of class $C$.

- **Arrays** $y = x[i]$ (resp. $x[i] = y$) that read (resp. write) an element of the set of arrays that $x$ may point to.

Define $\text{StatementRef}(s,c)$ (resp. $\text{StatementMod}(s,c)$) as the set of heap objects that statement $s$ in context $c$ of the enclosing method can read (resp. write). Define $\text{MethodRef}(c,m)$ (resp. $\text{MethodMod}(c,m)$) as the set of heap objects that can be read (resp. written) in context $c$ of method $m$. Recall from Chapter 4, that a context is a way to distinguish different invocations of the same method. A method $m$ might be invoked several times in the program, and we distinguish it based on the context $c$, in which it was invoked. As illustrated by the example in Section 4.6, distinguishing by separate contexts allows us to be more precise in our analysis.

Intuitively, $\text{MethodRef}$ and $\text{MethodMod}$ summarize all the heap accesses for a particular method, including its callees (as determined by the call graph analysis) transitively. Figure 5.3 shows the algorithm for computing the side effects for each node in our
pipeline. On line 8, we consult the call graph to determine the possible target methods for the invocation statement in the current context. We apply the algorithm to each node in our pipeline and collect its read effects in nodeRef and write effects in nodeMod. For brevity, we write $Set_1 ← Set_1 ∪ Set_2$ as $Set_1 ∪ = Set_2$.

**Input:** callgraph, currentContext, statementsInNode

**Output:** nodeRef, nodeMod

```plaintext
1: nodeRef ← ∅
2: nodeMod ← ∅
3: for all Statement s : statementsInNode do
4:   if isHeapAccessingStmt(s) then
5:     nodeRef ∪ = StatementRef(s, currentContext)
6:     nodeMod ∪ = StatementMod(s, currentContext)
7:   else if isMethodInvocationStmt(s) then
8:     methods ← possibleInvocations(callgraph, s, currentContext)
9:     for all <c, m> : methods do
10:        nodeRef ∪ = MethodRef(c,m)
11:        nodeMod ∪ = MethodMod(c,m)
12:     end for
13:   else
14:    {Statement has no effect on heap}
15:   end if
16: end for
```

Figure 5.3: Algorithm to compute the side effects for each node

Check Inter-node Data Races

The final stage of the analysis makes use of information from the previous stages to check for data race between nodes. Our analysis checks for data races between pairs of nodes. Considering pairs of nodes at a time simplifies the analysis and also makes it easier to pinpoint and report the possible data races to the developer in an understandable way.

Let $\{(n_i, n_j) | n_i, n_j ∈ N ∧ i ≠ j\}$ represent pairs of nodes in the pipeline. From the previous stage of the analysis, we computed both nodeRef and nodeMod for each node. Figure 5.4 presents our algorithm for computing possible data races.

Because $n_i$ and $n_j$ can operate in parallel, there is a possible data race if $n_i$ reads from or writes to a memory location that $n_j$ also writes to. Line 1 of Figure 5.4 computes the set of objects that could have a data race. The algorithm then examines each object and prints specialized warning messages based on the type of the object and the pairs of
Input: $nodeRef_{n_i}, nodeMod_{n_i}, nodeMod_{n_j}$

1: conflictingHeapObjects ← $(nodeRef_{n_i} \cup nodeMod_{n_i}) \cap nodeMod_{n_j}$
2: for all Object $o : conflictingHeapObjects$ do
3:     if isStaticField($o$) then
4:         warningMessageForStaticField($o, n_i, n_j$)
5:     else if isInstanceField($o$) then
6:         warningMessageForInstanceField($o, n_i, n_j$)
7:     else
8:         warningMessageForArray($o, n_i, n_j$)
9:     end if
10: end for

Figure 5.4: Basic algorithm to compute the possible data races between pairs of nodes.

While the basic algorithm shown in Figure 5.4 works (it is sound), it is too conservative. For instance, in Figure 3.1, it will warn that the object pointed to by the variable $docJPEG$ could be involved in a data race since it is being accessed (and at least one access is a write) in both $Node1$ and $Node2$. The basic algorithm fails to take advantage of a useful property inherent in flow-based applications: ownership transfer.

Figure 5.5 shows the pipeline execution of LIRe. At each time slice, $t$, at most $n$ nodes can execute in parallel, where $n$ is the number of annotated nodes in the pipeline. Each node operates on a different image. We illustrate this by using a different color to represent which image each node is operating on at each time slice. A common pattern we have observed is that each stage receives an object, operates on it, (possibly) generates new objects and passes them on to the next stage. This is a form of ownership transfer [NKA11]. A node transfers ownership of the object it was operating on to the next node.
in the pipeline.

Input: nodeRef\textsubscript{n}, nodeMod\textsubscript{n}, nodeMod\textsubscript{j}

1: conflictingHeapObjects ← (nodeRef\textsubscript{n} \cup nodeMod\textsubscript{n}) \cap nodeMod\textsubscript{j}

2: \textbf{for all} Object o : conflictingHeapObjects \textbf{do}

3: \textbf{if} isStaticField(o) \textbf{then}

4: \hspace{1em} warningMessageForStaticField(o, n\textsubscript{i}, n\textsubscript{j})

5: \textbf{else if} isTransferred(o, n\textsubscript{i}, n\textsubscript{j}) \textbf{then}

6: \hspace{1em} \{No data race since object ownership is transferred\}

7: \hspace{1em} continue with checking next object

8: \textbf{else if} isInstanceField(o) \textbf{then}

9: \hspace{1em} warningMessageForInstanceField(o, n\textsubscript{i}, n\textsubscript{j})

10: \textbf{else}

11: \hspace{1em} warningMessageForArray(o, n\textsubscript{i}, n\textsubscript{j})

12: \textbf{end if}

13: \textbf{end for}

Figure 5.6: Algorithm that incorporates ownership transfer to compute the possible data races between pairs of nodes

Figure 5.6 shows the new algorithm that incorporates ownership transfer. The main changes are on lines 5 – 7. Notice that ownership transfer only affects non-static objects. Static objects are essentially global variables and we cannot transfer their ownership since all nodes operate on the same static object. On the other hand, it is possible to transfer ownership of instance objects and arrays.

To be able to transfer an object, a node must first own it. There are two ways that a node becomes the owner. First, a previous node (earlier in the pipeline) could have transferred ownership to it. Second, and the most common case, is that the node created the object. For instance, from Section 5.1.1, we computed that the variable docJPEG is passed from Node1 to Node2. The object pointed to by docJPEG was indeed created in Node2. Thus, it can transfer the ownership to Node3. Using this object as the root, we then calculate all other objects that could also be transferred.

A root object is an object that is allocated through a call to its constructor, i.e., through new ClassName(...) in a node in the pipeline. We call it a root because this object could create other objects internally through its own constructor. All the internal objects are rooted on this root object (see Section 4.7). Inferring how root objects are transferred is simple for a linear pipeline. A node earlier in the pipeline can transfer objects to nodes
later in the pipeline. For a more general flow graph, which is a directed acyclic graph, one would first perform a topological sort and propagate the different transferred objects along the edges of the graph in topological order.

**Input:** pointerAnalysis, o, ni, nj

**Output:** true if the object o is transferred, false otherwise

1: if j > i then
2: {j > i implies that ni is earlier in the pipeline}
3: rootObjects ← {o | o allocated in nk, where 1 <= k <= i}
4: else
5: {j < i implies that nj is earlier in the pipeline}
6: rootObjects ← {o | o allocated in nk, where 1 <= k <= j}
7: end if
8: o1, o2, . . . , ok = label(o)
9: return true if ∃r ∈ rootObjects that is part of label(o)

Figure 5.7: Procedure isTransferred(o, ni, nj)

Figure 5.7 shows the procedure `isTransferred`, which determines if object o is transferred between node ni to node nj. First we check which node, i.e., ni or nj occurs earlier in the pipeline. Only objects allocated in earlier stages can be shared between two nodes. The algorithm gathers the objects that are allocated in the first node, ni up to nj, whichever occurs earlier.

The key idea of `isTransferred` is to take advantage of the way that objects are labeled using a k-object sensitive pointer analysis. Objects are labeled through a sequence of their allocation sites, i.e., o1, o2, . . . , ok. Section 4.7 presents an example of the objects and how those objects are labeled using k-object sensitive pointer analysis. We can use this label to determine if an object is rooted at one of the objects that was transferred. If so, then by transitivity, it is also transferred. We make use of this fact on line 3 of Figure 5.7.

To better illustrate the idea of inferring ownership transfer, we revisit the example from Figure 4.13 in Chapter 4. Figure 5.8 shows its modified main method that now contains a loop that we wish to parallelize. This is a simplified example to illustrate the idea of ownership transfer, it is unlikely that any developer would parallelize this trivial loop in this manner.

There are three nodes in the pipeline. Node1 creates a new Recipe. The sec-
static void main (String[] args) {
    for (int i = 0; i < MAX; i++) {
        // Begin Node1
        Recipe r = new Recipe();
        // End Node1
        // Begin Node2
        r.updateInstructions(); // Modifies contents of instructions list
        r.updatePhotos(); // Modifies contents of photos list
        // End Node2
        // Begin Node3
        r.display(); // Reads contents of instructions and photos lists
        // End Node3
    }
}

Figure 5.8: Parallelizing the Recipe example from Chapter 4

ond node updates the contents of the fields instructions and photos. Recall that
instructions and photos are both List objects with a single field, an array of
Objects. Node2 modifies the contents of the instructions and photos fields while
Node3 reads those values. Without incorporating ownership transfer, we would impre-
cisely conclude that they could potentially have a data race on the objects referenced by
instructions and photos.

The root object that is being transferred from Node1 to Node2 and Node3 is r. This
is clear from data dependence analysis. Then using our ownership transfer algorithm,
we further infer that the contents of the List elements are also transferred. As shown
in Figure 5.9 (reproduced from Figure 4.14) the root object r has the label $O_5$. The
elements array for instructions has the label $O_5O_9O_{23}$ while the elements array
for photos has the label $O_5O_{10}O_{23}$. Both are rooted on $O_5$, the label for r. Thus, both
have also been transferred and there is no potential data race.

The idea of ownership based on root objects shares similarities with the object-as-
dominator protocol introduced by Clarke et al. in their Ownership Types type
system [CPN98]. In our case, we try to infer ownership transfer, whereas their original
paper was about enforcing it through a type system.

An important assumption of our algorithm is that the developer starts with a purely
sequential version of the application that she wishes to parallelize. If her application
already incorporates some form of parallelism (perhaps through threads), then we would

58
need to also perform a thread escape analysis to ensure that the objects in one node are not leaked to other threads. Section 8.2.2 provides more details on other aspects that need to be considered.

5.1.2 Transformations

If the previous four analyses succeed (or if the developer explicitly requests the tool to proceed), then Extract Nodes performs the transformations shown in Section 3.3. There are two main transformations: bundling the dependencies and converting the nodes to closure objects. Both these transformations relieve the burden of manual partitioning and routing the dependences between different nodes in a computation flow graph.

The transformations are performed using the source-to-source rewriter provided by Eclipse JDT. The rewriter allows us to describe the changes as Abstract Syntax Tree (AST) operations, e.g., move, insert, delete, etc to the original source code. This is a more
systematic way of implementing the transformations since it allows the developer to select and undo specific changes.

Bundling Dependencies

The purpose of this transformation is to provide a way for dependencies from one node to be forwarded to another node. For example, as shown in Figure 3.1, Node2, Node3 and Node4 depend on the bufferedImage variable from Node1. Sometimes these dependencies are simple, Nodeₙ depends on data from Nodeₙ₋₁, i.e., its direct neighbor. Sometimes these dependencies are more complicated, e.g., a node might depend on data from much earlier node. One approach to solve these data dependencies is to bundle and forward them along the pipeline.

This transformation creates a bundle data object with fields for each object that needs to be forwarded. A data object is a simple class with only fields and no methods. The fields are determined from the variables in the \textsc{IN} \((n)\) and \textsc{OUT} \((n)\) sets that we computed for each node \(n \in N\).

This transformation has the advantage that all the dependencies are bundled neatly so it is easier to reason about the code. It reduces the number of \texttt{DataflowQueue}s that need to be created and, thus, reduces the internal bookkeeping. A disadvantage though, is that it closely mimics the sequential ordering of the original program and creates a linear pipeline; it assumes that each node has to be executed in the original order and precludes different nodes from executing at the same time even if they don't have any data dependencies.

This transformation bears similarities to the \textsc{Introduce Parameter Object} refactoring [Fow99]. It is not strictly necessary but has the advantage that it can make the code more readable by reducing the number of parameters that need to be passed separately.

Introduce Closures

This transformation converts the original statements in each node into its own closure object. In our case, we move the original statements into a \texttt{DataflowMessagingRunnable}
anonymous inner class. The transformation takes care of providing each closure with its set of input and output variables (both from the bundle object). It reads the variables from the relevant fields of the bundle object; and it writes the modified variables to the relevant fields of the bundle object (see Section 3.3). The variables read from and written to in each node are determined by consulting the $IN(n)$ and $Out(n)$ sets for node $n \in N$.

5.2 Invert Loop

5.2.1 Analyses

The analyses for the Invert Loop transformation checks if any of the extracted nodes from the Extract Node transformation can be run in a data parallel manner. While the analyses for Extract Nodes check for inter-node parallelism, these analyses check for intra-node parallelism. Figure 5.10 presents an overview of the analyses. The first and second stages follow the same principles as the previous analyses; we discuss only the third stage — computing possible data races.

Check Intra-node Data Races

![Diagram of the analyses workflow for the Invert Loop transformation](image)

**Figure 5.10:** Overview of the analyses workflow for the Invert Loop transformation

Figure 5.11 shows the algorithm for computing possible data races when a node is run in a data-parallel manner. It checks all possible write effects to the heap from the selected node. Any write access to a static field will cause a data race when the node operates in parallel. However, write accesses to instance fields and arrays are safe as long as the node only writes to objects in the heap that it owns (directly or transitively)
Input: $nodeMod_n$

1: root $\leftarrow \emptyset$
2: \{Collect all objects allocated in $n$\}
3: for all Object $o : nodeMod_n$ do
4: \hspace{1em} if isOwner($o, n$) then
5: \hspace{2em} root $\leftarrow o$
6: \hspace{1em} end if
7: end for
8: 
9: for all Object $o : nodeMod_n$ do
10: \hspace{1em} if isStaticField($o$) then
11: \hspace{2em} warningMessageForStaticField($o, n$)
12: \hspace{1em} else if $\exists r \in root$ that is part of the label($o$) $\lor \isTransferred(o, n, predecessor(n))$ then
13: \hspace{2em} \{No data race on this object\}
14: \hspace{1em} continue with checking next object
15: \hspace{1em} else if isInstanceField($o$) then
16: \hspace{2em} warningMessageForInstanceField($o, n$)
17: \hspace{1em} else
18: \hspace{2em} warningMessageForArray($o, n$)
19: \hspace{1em} end if
20: end for

Figure 5.11: Algorithm to determine if a node can run in a data-parallel manner

or has been transferred to it from a predecessor node. Intuitively, a node can run in parallel if it operates only on fresh objects.

The algorithm first iterates through $NodeMod$, and collects all objects that it is a direct owner of (Lines 1 – 7) in $root$. This root set is used to check for objects that it owns directly or transitively on line 12. Line 12 also checks to see if the object that it is writing to was transferred from a previous node. If either of these conditions hold it is safe to run the node in a data parallel manner. If they do not hold, we warn the user of the potential data races, which she can inspect manually.

5.2.2 Transformation

If the analyses for Invert Loop succeeds, JFlow informs the developer of the possibility of running a node in a data-parallel manner. Recall that running a node in this manner causes its processed objects to arrive out-of-order to the next node. Thus, the developer needs to use her domain knowledge to decide if this is permissible for her application.
The main transformation for *Invert Loop* is to register the extracted closures with a new FlowGraph object. A FlowGraph object represents a computation flow graph. Internally, the FlowGraph uses a fork-join parallel model. It forks off threads to process the closures that are registered with it. When all the input dependencies for a closure is available, it schedules that closure for execution. It allows the current thread to wait until all the computation in the flow graph has been completed before resuming serial execution. This fork-join model allows us to incrementally parallelize a portion of the code (the main bottleneck) without having to change the overall design and architecture of the original application.

The FlowGraph is not part of the official GPars 1.0 release. It is a construct that we added to support flow-based parallelism. We abstracted much of the necessary boilerplate code for starting the computation and waiting for it to complete. Without the FlowGraph construct, we would have had to generate much more code.

The transformation creates a new FlowGraph object. It then registers each DataflowMessagingRunnable object extracted before with the FlowGraph using the *operator(...)* construct. The *operator(...)* construct takes three main parameters: a list of input channels, a list of output channels and the DataflowMessagingRunnable representing the computation to perform when data are available on the list of input channels. Optionally, it takes a fourth parameter, p, which represents the degree of data parallelism that this node can have. When p > 1, the node will run in a data parallel manner. A developer can easily tune the performance of the pipeline by changing the values of p.

This transformation reduces the manual burden of writing boilerplate code to register each closure and its input and output channels with a pipeline.
Chapter 6

Evaluating Our Approach

This chapter presents the evaluation of our interactive approach. We evaluated our approach along two dimensions: performance and developer interaction. Of the two, performance is easier to define and measure. We compared the execution time of the parallelized version to the serial version and found that we achieve reasonable performance on a 4-core machine.

The other dimension, developer interaction, is harder to evaluate. Our approach requires the developer to annotate the nodes in the original sequential application. The choice of annotation directly affects the interaction required. Some annotation choices might cause more warnings and, thus, require more interaction from the developer. Others might cause less warnings but not perform as well. To mitigate this effect, we selected applications where we have a reasonable understanding of the partitioning. The partitions could come directly from comments in the application, a previously parallelized version or publications about the particular application. This allows us to more reliably determine the developer interaction and effort required to parallelize the application in a way that the developer intended. We document the interaction (inspection and changes) necessary for each application that we parallelized.

The lessons learned from parallelizing these applications highlight future opportunities for improving source-to-source transformation tools for parallelization.
6.1 Methodology

We implemented the analyses and transformations discussed in the previous chapter in our interactive source-to-source transformation tool, JFlow. We evaluated JFlow on a 2.33 GHz 4-core Core 2 Quad processor with 4 GB of memory. The operating system is Ubuntu 12.04 running the 3.2.0-38-generic Linux Kernel.

When invoked interactively as an Eclipse plug-in, JFlow takes less than a minute to perform the analysis and transformations. Ideally, we would like it to be as prompt as possible but we think that one minute is a reasonable upper threshold for an interactive tool. The bulk of this time was spent initializing the call graph and pointer analysis.

After performing the transformations, we used Oracle’s Java HotSpot 64-Bit Server JVM to measure the execution times. We set the min and max heap space for the JVM to 512M. We average the running times over 6 runs. The source code for both the serial and parallel versions of the benchmarks are available at http://vazexqi.github.io/JFlow/.

Table 6.1 shows the results. The $\Delta$ SLOC Parallel column gives an estimate of the transformation effort required to use Groovy’s GPars parallel library constructs. The greater the number of lines changed, the greater the effort required from the developer. The actual parallelization effort, if done manually, is much higher since the developer has to also spend time analyzing and scrutinizing the code for data races.

The performance numbers show that applications parallelized through the constructs of a parallel library can perform reasonably well. We now discuss the necessary inspections and changes (if any) involved with parallelizing these applications using JFlow.

6.2 Benchmark Applications

The first four applications are taken directly from OoOJava [JED11], which is most similar to our work. OoOJava solves the complexity of parallelizing modern object-oriented applications through a combined static and dynamic approach with a custom runtime (see Section 7.5). JFlow, on the other hand, solves the problem by involving the devel-
<table>
<thead>
<tr>
<th>Application</th>
<th>Domain</th>
<th>SLOC Sequential</th>
<th>∆ SLOC</th>
<th>SLOC Parallel</th>
<th>Sequential (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMeans</td>
<td>Mining &amp; Synthesis</td>
<td>504</td>
<td>+73</td>
<td>12980</td>
<td>2.98x</td>
<td></td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>Financial Trading</td>
<td>991</td>
<td>+45</td>
<td>14906</td>
<td>2.82x</td>
<td></td>
</tr>
<tr>
<td>MolDyn</td>
<td>Molecular Dynamics</td>
<td>653</td>
<td>+56</td>
<td>23694</td>
<td>3.37x</td>
<td></td>
</tr>
<tr>
<td>RayTracer</td>
<td>Image processing</td>
<td>828</td>
<td>+55</td>
<td>17704</td>
<td>3.45x</td>
<td></td>
</tr>
<tr>
<td>Duke</td>
<td>Data warehousing</td>
<td>11464</td>
<td>+55</td>
<td>9502</td>
<td>1.49x</td>
<td></td>
</tr>
<tr>
<td>Jbzip2</td>
<td>Data warehousing</td>
<td>3238</td>
<td>+38</td>
<td>17785</td>
<td>2.54x</td>
<td></td>
</tr>
<tr>
<td>LIRe</td>
<td>Image processing</td>
<td>15476</td>
<td>+76</td>
<td>49226</td>
<td>3.19x</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Speedups on benchmarks

Our approach was to define the parallelization process. OoOJava ported the benchmarks over from two original sources. Kmeans comes from the STAMP benchmark suite [MCKO08]. Monte Carlo, MolDyn and RayTracer are from the Java Grande Forum benchmark suite [SBO01]. Table 6.2 briefly describes each application.

We selected these applications from OoOJava because, like our approach, it also relies on the developer to provide the partitioning annotations. We followed the partitioning annotations from OoOJava. We used these applications to compare the effectiveness of our approach in analyzing and transforming sequential code to parallel code. Note that we do not directly compare execution times as the runtime of OoOJava is sufficiently different from the approach that we are taking.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMeans</td>
<td>Uses the k-means clustering algorithm to cluster 65536 objects of 32-</td>
</tr>
<tr>
<td></td>
<td>dimensional space into 40 clusters.</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>Financial simulation, using Monte Carlo techniques (repeated random</td>
</tr>
<tr>
<td></td>
<td>sampling) to price products derived from the price of an underlying</td>
</tr>
<tr>
<td></td>
<td>asset. The simulation was run using a sample of 300 generated time</td>
</tr>
<tr>
<td></td>
<td>series.</td>
</tr>
<tr>
<td>MolDyn</td>
<td>N-body simulation modeling particles interacting under a Lennard-Jones</td>
</tr>
<tr>
<td></td>
<td>potential model [Jon24]. The simulation was run with 300 particles.</td>
</tr>
<tr>
<td>RayTracer</td>
<td>Renders a scene with 64 spheres by tracing rays from light sources.</td>
</tr>
</tbody>
</table>

Table 6.2: Description of applications
For each benchmark, we describe

1. The main bottleneck

2. The partitioning strategy that was used

3. Inter-node data races reported, if any

4. Intra-node data races reported, if any

The bottleneck for KMeans is the computation of clusters for each of the 65536 objects. The main loop for this computation is partitioned into two nodes. After the initial annotation for the nodes, JFlow was able to parallelize KMeans successfully without any user intervention. It correctly reported that there were no inter-node data races. The first node computes the cluster for a chunk of the 65536 objects. This node can run in a data parallel manner and JFlow confirms that this is possible. The second node updates global values that keep track of the cluster that each object belongs to. This node cannot run in a data parallel manner since it updates global variables. JFlow confirms that there were indeed writes to global variables.

The bottleneck for Monte Carlo is the main loop that runs multiple Monte Carlo simulations. We partitioned this loop into two nodes. The first node runs the simulation for a chunk of the 300 samples. The second node updates the estimated pricing for the products.

The first node can be run in a data parallel manner. However, JFlow reported several data races from the use of `System.out.println` and `Math.util.random` constructs from the JDK because of the way we treat objects originating from the JDK. As mentioned in Section 5.1.1, we use a selective form of context-sensitivity. For scalability, we tune the pointer analysis to have greater precision for user-defined objects and lower precision for objects from the JDK. Thus, there is some imprecision in distinguishing objects from the JDK. Nonetheless, it was not difficult to inspect the reported data races since they were all easily identifiable as logging constructs.

The second node for the Monte Carlo application cannot be run in a data parallel manner and JFlow confirms this. This second node is responsibility for updating some
global values about the price estimated so far.

The bottleneck for MolDyn comes from its pair-wise computation of forces between the different particles. The main loop for this computation is partitioned into two nodes. JFlow reported an inter-node data race on a global 3D array accessed in both nodes, stemming from our lack of index-sensitivity on arrays. Upon closer inspection, we found that the nodes accessed different portions of the array and was safe to run in parallel.

The first node for Moldyn computes the forces between particles. This node can be run in a data parallel manner. However, our tool reported a spurious intra-node data race because our analysis is index-insensitive for arrays. After we confirmed that the warning was spurious, we allowed this node to run in a data parallel manner. The second node updates the computed forces and moves the particles for the next simulation cycle. This node updates global variables so it cannot be run in a data parallel manner. JFlow confirms that the there were indeed writes to global variables.

Finally, for RayTracer, the main bottleneck is the computation of the pixel color rendered by each ray as it traces through the objects (64 spheres) in the scene. This loop is partitioned into two nodes. The first node calculates the pixel color rendered by each ray in a row (there are \( y \) rows with \( x \) columns each). The second node accumulates the color into the rendered image. JFlow correctly reported that there are no inter-node data races.

The first node can be operated in a data parallel manner. However, JFlow reported spurious data races stemming from one statement: a method call. This single statement made use of multiple objects allocated using static factory methods. Figure 6.1 shows a code snippet illustrating one of those static factory methods.

```
/**
 * adds: Returns a new vector such as
 * new = sA + B
 */

public static Vec adds(double s, Vec a, Vec b) {
    return new Vec(s * a.x + b.x, s * a.y + b.y, s * a.z + b.z);
}
```

Figure 6.1: Code snippet from RayTracer illustrating the use of static factory methods

Recall that for our k-object sensitive analysis, static methods are all lumped under
one context, $\epsilon$, which reduces the precision of the analysis. To complicate matters, this method was also recursive in nature. K-object sensitivity can handle recursion by bounding the context that it creates, but the results will be more conservative and less precise.

However, because all the spurious data races only involved a single statement and the recursive algorithm is a standard way to perform ray tracing, it was not difficult for a developer to inspect these by hand and proceed with the transformation. Situations such as this highlight the importance of engaging the developer in the process. A developer who is familiar with the code or the core algorithm, can quickly inspect the code to determine if the code can be parallelized, while even precise static analysis for handling method calls have trouble with recursive code [RHS95].

6.3 Large Applications

The next three applications are what we term large applications; they are about an order of magnitude larger in terms of lines of code. We chose these applications because they are similar to the applications parallelized using pipeline parallelism from the PARSEC benchmark (see Chapter 2). PARSEC comprises a representative set of emerging applications and we are interested to see how well JFlow can handle them. We used the parallel versions of the PARSEC benchmarks to guide our partitioning annotations.

Duke is a data-deduplication engine. Our application processed a 700MB file looking for pairs of duplicates and links them together. Duke has a parallel version with threads. Its parallel version used a single node with data parallelism. We followed the same parallelization scheme and achieved similar speedups as its threaded version, i.e., 1.49x. This shows that JFlow is able to handle how a developer might parallelize it by hand. In fact, it detected two data races that was present in the original parallelization by the author (writes to two separate counters without proper synchronization). The speedup is low because Duke is primarily I/O bound rather than CPU bound.

Jbzip2 is a bzip2 compression/decompression library. Our application decompressed and compressed 1000 files; this represents a typical workflow where one would decompress a file, read and manipulate its contents and then compress it again. The main
The challenge here was the spurious data races reported from statements that used Java's File API. These statements were the cause of both the inter and intra-node data races reported by JFlow. JFlow isolated these data races to particular statements that used the File API. Once we identified the relevant statements, it was easy for a developer to manually inspect by hand and proceed. Figure 6.2 shows two lines (out of six) that were involved in compressing the contents of a file. The main source of confusion in static analysis was the wrapping of each stream (a key characteristic of the Decorator design pattern). While it is hard for static analysis, any developer can quickly inspect these lines and conclude that the streams all operate on different files and, thus, cannot have a data race.

```java
InputStream fileInputStream = new BufferedInputStream(new FileInputStream(inputFile));
// This 'wraps' fileInputStream
BZip2InputStream compressedInputStream = new BZip2InputStream(fileInputStream, false);
```

Figure 6.2: Code snippet illustrating calls to Java's File API

Section 3 described LIRe in detail. Here we focus on the challenges to parallelization. The original version of LIRe used reflection to instantiate its classes based on class names. For instance, to instantiate some of its feature extractors, it used the Class.forName(x).newInstance() construct. This form of dynamic class creation, which is common in dependency-injection frameworks, poses great difficulties for static analysis. We manually transformed the code so that it instantiates the class directly via a new statement to the specific class. Like most modern applications, LIRe makes use of the highly reflective java.util.logging API for error reporting. Specifically it used the log4j API; we treated such calls as being thread safe and ignored data races through those calls.

JFlow correctly reported that there were no inter-node data races. It also correctly reported that Node2 and Node3 could run in a data parallel manner. However, it reported spurious data races that prevent Node1 from running in a data parallel manner even though it is possible. We traced the spurious data race to the statement ImageIO.read(...). As with Jbzip2, JFlow has trouble distinguishing that ImageIO
produces new images each time it is invoked. Since this was a single statement, it is easy for a developer to inspect and conclude that the warning was spurious.

Both Duke and LIRe use the Lucene search engine library for its indexing and querying. By default, JFlow does not analyze the source code for external libraries such as Lucene. We justify this decision in favor of scalability and understandability. First, external libraries are huge; moreover, many projects make use of more than one external library. Analyzing these libraries will increase the analysis time significantly. Second, even if we do analyze external libraries, data races reported that involve the internal data structures will be hard to understand by a developer who is merely using it as a black-box library. Her efforts would be better rewarded from reading the APIs and documentation for those external libraries with regards to their thread safety. When we ignore calls to external libraries, we issue a warning to the developer so that she knows that some of the information might be missing. If she chooses, she can instruct JFlow to analyze these libraries, albeit at the expense of time and memory.

6.4 Discussion

<table>
<thead>
<tr>
<th>Application</th>
<th>Inspections &amp; changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kmeans</td>
<td>None</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>Inspect intra-node data race on System.out.println and Math.util.random constructs</td>
</tr>
<tr>
<td>MolDyn</td>
<td>Inspect two statements accessing a global 3D array</td>
</tr>
<tr>
<td>RayTracer</td>
<td>Inspect one recursive method call using objects created through static factory methods</td>
</tr>
<tr>
<td>Duke</td>
<td>Inspect two statements accessing Lucene API</td>
</tr>
<tr>
<td>Jbzip2</td>
<td>Inspect six statements using File API</td>
</tr>
<tr>
<td></td>
<td>Convert reflective construction to use new</td>
</tr>
<tr>
<td>Lire</td>
<td>Inspect data races from logging constructs</td>
</tr>
<tr>
<td></td>
<td>Inspect four statements accessing Lucene API</td>
</tr>
</tbody>
</table>

Table 6.3: Inspections and changes needed from the developer

Table 6.3 summarizes the inspections and changes necessary from the developer. We
believe that the effort required from the developer is reasonable. If a developer were to do this entirely manually, it would have required more effort to analyze and transform. The feedback from JFlow pinpoints the potential problems and guides her parallelization efforts. Our sample of seven applications from a diverse set of emerging domains leads us to believe that the inspections and changes required are typical of modern applications. We discuss possible areas for improvements in the following paragraphs.

We have seen that the lack of index-sensitivity for arrays causes spurious data races to be reported for MolDyn. Future work could focus on adding index-sensitivity to our analysis. One approach would be to encode each array access as a constraint and use a constraint solver to reason about the possible accesses. This very precise approach was proposed by Dillig et al. and was used to verify access properties not only for array accesses but also for general containers such as lists, vectors, etc. [DDA11].

Another common pattern we observed is that most of the applications manipulate data through File APIs. Currently, we do not model how the File APIs work. We treat them as normal method calls without any special semantics. Future work could focus on better modeling these File APIs (understanding that they wrap different streams) and providing better semantics to avoid spurious data races.

Though we have only seen one instance of static factory method usage in our suite of applications, it is frequently used in real world code. In Effective Java, Bloch advocates to “consider static factory methods instead of constructors” [Blo08]. There are many variations of static factory methods: some substitute for a constructor and always return a new object; others return cached instances of a shared object. Our current approach of handling static methods is insufficient to distinguish these variations. Future work might take advantage of developer annotations on static factory methods as hints of how to interpret them and improve the precision of our analysis.
Chapter 7

Related Work

This chapter surveys related work in five broad areas. First, we contrast flow-based parallelism to other related paradigms in parallel programming. Then, we discuss the roles and contributions of patterns in parallel programming. We then discuss alternative libraries and frameworks that exist besides Groovy’s GPars, Intel’s TBB and Microsoft’s TPL Dataflow; these alternative libraries and frameworks discussed might be suitable targets for future transformations. Next, we survey existing work on interactive transformation tools. Finally, we discuss related work on inferring side effects, a key idea behind our analyses for race detection.

7.1 Flow-based Parallelism

Flow-based parallelism is an algorithmic structure that organizes parallelism based on the flow of data. The term “flow-based parallelism” was inspired from Morrison's *Flow-Based Programming: A New Approach to Application Development* [Mor10]. Morrison’s original work on flow-based applications focused mostly on the architecture and design of systems using flow-based programming, i.e., decomposing a system into black boxes that communicate via message passing [Mor10]. While he did acknowledge that flow-based programming lends itself well to parallelism, his implementation [Mor] focused mostly on modularity and not on parallel performance. This differs from streaming languages such as Brook [BFH⁺], StreamIt [TKA] and StreamC/KernelC [KRD⁺03], where performance was the primary focus. By building upon well-designed parallel libraries, our work attempts to bring both performance and modularity to a parallelized application.
The ideas behind flow-based parallelism are not new; these ideas have existed in various forms throughout the history of computing. The earliest influences come from the dataflow programming community. For the interested reader, Johnston et al. provide a detailed history of the evolution of the hardware and software for dataflow programming in [JHM04]. Similar to flow-based programming, a program in the dataflow execution model is represented by a directed graph; the nodes of the graph represent operations and the edges between nodes represent data dependencies [DK82]. Dataflow programming typically relies on dedicated dataflow languages. Lucid [WA85] and Id [AGP78] are examples of early dataflow languages while LabView [Nat] and Prograph [MP85] are examples of current dataflow languages that are in use. Most dataflow languages tend to be restrictive, favoring functional styles without side effects [Ack82]. These restrictions allow for easier reasoning of the system but also require that existing programs be rewritten to conform to those restrictions. Our work on flow-based parallelism favors a more relaxed approach that does not mandate a functional style of programming. This approach is more practical and allows existing programs with side-effects to take advantage of flow-based parallelism as long as the side-effects are well isolated.

Many message passing programming styles for parallel programming exist. The most prominent one is MPI [Mes09], which is used for many high-performance and distributed scientific applications. While we could have adhered to the MPI specification for flow-based parallelism, we found the constructs of MPI too low-level for our needs. The MPI specifications focus on providing low-level support for packing and unpacking data into array forms, which works well for scientific applications but does not work as well for object-oriented flow-based applications.

Communicating Sequential Processes (CSP) and the Actor model are two other prominent message passing concurrent models. In CSP, processes communicate by exchanging messages synchronously through channels [Hoa85]. In the Actor model, actors (processes) communicate asynchronously by sending messages to other actors; sent messages are buffered in the receiving actor's mailbox [Agh86]. Both models are general and can support many different styles of parallel programming. Compared to both models, flow-based parallelism admits a more restricted form of parallelism where the topology and
communication interactions are more regular. Thus, the constructs for flow-based programming can be implemented on top of either of these models by restricting some of the constructs used. GPars, the existing library that we are using, builds its flow-based constructs on top of its existing Actor implementation.

7.2 Pattern-based Approach for Parallelism

In Chapter 1, we introduced the ideas of data parallelism, task parallelism and flow-based parallelism as algorithmic structures of parallel programs. We argued that these algorithmic structures have identified useful abstractions for expressing parallelism in ways that provide good performance, understandability and maintainability. The original idea of algorithmic structures comes from the software patterns community.

Christopher Alexander’s *A Pattern Language: Towns, Buildings, Construction* was the original inspiration for a pattern-based approach for describing solutions to recurring problems [AIS77]. Beck and Cunningham first adapted Alexander’s work for use in software in 1987 [BC87]. Since then, the pattern-based approach has been used to effectively describe various software systems including, but not limited to, design patterns [GHJV94], domain-specific languages [Fow11], security [Haf10], software architecture [BMR+96], software testing [Mes07], etc.

Recently, researchers and practitioners have begun embracing patterns as a way to describe the best practices for concurrent and parallel systems. Hohpe and Woolf describe enterprise integrations patterns using asynchronous messaging [HW03]. Campbell et al. used a patterns-based approach to motivate the use of the new data and task parallelism constructs in .NET [CJMT10]. Kim and Snir maintain an online collection of patterns used mostly in scientific applications [KS11].

So far, no comprehensive catalog for flow-based parallelism exists. The closest known work are those of Mattson et al. [MSM04] and Ortega-Arjona [OA10]. Mattson et al. currently have the most comprehensive catalog of different parallel programming patterns. Their work also includes two that are directly applicable to flow-based parallelism, i.e., pipeline parallelism and event-based coordination. Ortega-Arjona’s work focuses more
on patterns for the architecture and design of parallel software. For instance, he focuses on high-level architectures such as pipes & filters and parallel layers.

Our current work builds upon the pipeline parallelism pattern. It would be useful to have a catalog of common patterns for flow-based parallelism. Such a catalog would help guide researchers in designing future transformations for flow-based parallelism.

7.3 Libraries for Flow-based Parallelism

Currently both Intel’s TBB Flow Graph and Microsoft’s TPL Dataflow libraries provide the most support for flow-based parallelism. Groovy’s GPars comes close but does not provide as many constructs. All three libraries, however, share common functionalities and are based on the ideas of source and target nodes. Source nodes produce data while target nodes consume data; a node can also be both a source and a target node. All libraries also come with a predefined set of commonly used nodes that include buffer nodes, broadcast nodes, function nodes and join nodes. Sharing of data via message passing is encouraged but not enforced, developers are still free to use the shared memory facilities of the underlying languages, i.e., C++, Java or C#.

The earliest work on flow-based parallelism for Java was the work on Dataflow Java by Lee and Morris [LM00]. Dataflow Java is not a library but a language extension of Java that introduces constructs for dataflow. The underlying runtime relied on Java threads and provided preliminary support for communication, load balancing and fault tolerance. StreamFlex by Spring et al. is an extension to Java that provides stream processing capabilities [SPGV07]. In stream processing, a continuous stream of data flows through nodes that compute using a functional style, without side effects. StreamFlex is novel because it aims to provide real-time capabilities for streaming programs running on the Java virtual machine. Diva by Chen et al. provides a dataflow programming model for Java based on the concepts of components and channels [CFZW08]. Instead of relying on locks for synchronization, Diva relies on its support for software transactional memory. Because of this, programs in Diva need to run on a custom version of the Java virtual machine.
We do not use these systems in our work because they all require custom extensions to either the Java language or its virtual machine. Our work concentrates on providing support for developers parallelizing their standard Java programs. This was a conscious decision: we want our tools to work with the variety of existing applications without forcing the developer to commit to non-standard tools that are not part of their existing workflow.

Communicating Sequential Processes for Java (JCSP) is a library by the University of Kent [WBM+07]. It provides support for Hoare’s CSP using only features in the standard Java language. JCSP is a strict embodiment of the CSP concurrent model. Each process is represented as a Java thread. Processes communicate synchronously following the semantics of CSP. JCSP is notable because it provides a proven-correct implementation of the alternation (ALT) construct in CSP — a tricky construct to implement correctly.

A library-based approach for flow-based parallelism relies on the developer to explicitly specify the execution strategy. For instance, the programmer has to explicitly connect different components together to specify their flow. This process can be alleviated with a tool such as ours that will connect the different component based on their data dependencies. An orthogonal approach would be to abstract the dependencies and rely on a coordination language and a intelligent runtime for planning a suitable execution strategy. This is the approach advocated by Linda [GC92] and, more recently, by Intel’s Concurrent Collections [Inta]. For instance, in Linda, a developer specifies the execution constraints for each component (the input, output) and the runtime automatically executes those components in a way that satisfies those constraints. All input and output are made available in a tuple space and components can read and write from that tuple space.

7.4 Interactive Transformations for Parallelism

The idea of interactive tools for parallelization has been explored in various forms. Two notable tools are SUIF Explorer [LDB+99] and Parascope Editor [KMT91], which focused on transformations for loop parallelism for scientific applications written in Fortran. Both
tools were unique at the time because they allowed the developer to examine the outcomes of the analysis of the compiler in a rich environment that included various visualizations for program dependencies. Both tools utilized deep interprocedural analyses, which are inherently conservative, and sought developer input to help refine the outcomes of the analyses. These tools were targeted at knowledgeable users. To effectively use these tools, the user must have a fairly intimate knowledge of program analysis and transformations. This, unfortunately, limited the number of potential users.

More recently, Dig et al. demonstrated that a transformation-based approach for introducing concurrency and parallelism into sequential code via library constructs is practical. Their initial work focused on providing support for three kinds of transformations: (i) convert int to AtomicInteger, (ii) convert HashMap to ConcurrentHashMap and (iii) convert recursion to ForkJoinTask [DME09]. Dig et al. then created ReLOOPER, a tool for converting array operations to use data parallelism through the new ParallelArray construct in Java [DTR+09]. Kjolstad et al. continued work in this area by introducing transformations to convert mutable classes into immutable classes, i.e., value objects [KDAS11]. In the same vein, Wloka et al. have proposed a series of transformations to make programs reentrant. Their approach replaces global state with thread-local state and performing each execution in a fresh thread [WST09].

As mentioned in Chapter 5, our current analyses only detect potential data races, but do not provide any assistance in removing those data races. The tools we mentioned in the previous paragraph could be used in this regard. In general, the process of parallelizing an application is too broad to be handled by just a single tool. Instead, a developer is likely to compose several tools together, and even perform some transformations by hand. Thus, it is useful to have a repertoire of such tools at the developer’s disposal.

An orthogonal approach of source-to-source transformations would be to rely on developer annotations. In this approach, developers annotate their source code to indicate sections to be parallelized. A compiler reads those annotations and generates binary versions of the parallelized program. The most prominent annotation based approach is OpenMP which provides support for data parallelism and, more recently, task parallelism [OMP]. In OpenMP, the annotations direct the compiler to perform the requested
transformation to the binary code. In most cases, the compiler does not check (or performs very simple syntactic checks) on the safety of these annotations. Instead, the compiler relies exclusively on the developer to guarantee that the annotations will not introduce any concurrency bugs, e.g., data races.

In other systems, the annotations serve as hints to provide additional information. For instance, in the Paralax system, the developer provides annotations to help the underlying interprocedural dependence analysis [VRDB10]. During compilation, the compiler uses those annotations to guide its analysis. While annotation-based approaches typically target data and task parallelism, some systems have tried to target other forms of parallelism. Thies et al. [TCA07] were the first to propose and implement an annotation-based method for automatically detecting and parallelizing pipeline parallelism in C programs. The annotations serve as hints to a dynamic dependence analyzer that would monitor for data accesses. A main disadvantage of annotation-based parallelizing is that the transformations are opaque. The transformed program is in binary form, making it hard to examine and debug the parallelized program especially if the annotations were mistakenly specified.

Another orthogonal approach relies on algorithmic skeletons [Col91]. Developers use a top-down approach to parallel application development. They first determine the desired algorithmic structure and then select from several predefined skeletons (similar to parallel programming patterns) to create that structure. Predefined skeletons include farm (master-slave), pipe (pipeline parallelism), divide-and-conquer, etc. After the skeletons have been composed, the system generates the underlying code with placeholders and hooks for the developer to customize. Notable systems that use this approach include CO₂P₃S [MAB⁺02], JaSkel [FSP06] and Skandium [NIC]. Because such systems require programs to be designed from scratch, it is not possible to use them on legacy code.

### 7.5 Inferring Side Effects

To infer the side effects of statements, we relied on a mod-ref analysis. An alternative approach is to rely on types and effects systems [BAD⁺09, LPHZ02, LG88]. While types and
effects systems can be more precise, an inherent difficulty is that they require extensive
annotations from the developer. Work has been done to alleviate this by inferring some
of these annotations [Kel05, Nik06, VDB+09]. Another limitation of such type systems is
that they usually require custom language syntax. This makes them challenging to adopt
into existing programs written in legacy programming languages. In this regard, ap-
proaches based on pluggable type systems are attractive because they allow a type system
to be seamlessly added to an existing programming language [DDE+11].

We rely on static analysis to determine if an application can be parallelized safely. As
demonstrated in Chapter 6, static analysis tends to be conservative and has trouble under-
standing some common programming idioms. Realizing the limitations of static analysis,
some researchers have proposed a dynamic approach to program analysis through profil-
ing at runtime. These approaches rely on the developer to provide a representative test
program for the data dependence analyses to work. A faulty test program could easily
jeopardize the entire analysis and transformation process.

Though unsound, dynamic approaches can be quite effective. Thies et al. [TCA07]
first proposed an annotation-based method for automatically detecting and parallelizing
pipeline parallelism in C programs. Rul et al. [RVDB10] and Tournavitis et al. [TF10]
improved on their work and automatically detect and parallelize pipeline parallelism in
applications without any annotations. Unfortunately, this approach is not always ideal
for every workflow. Running, collecting and analyzing the data is slow and consumes
a lot of memory and storage. Work by Kim et al. attempt to alleviate this overhead by
parallelizing and compressing the data collection process [KKL10b].

Static and dynamic analysis need not be mutually exclusive. The OoOJava project
combines both [JED11]. It uses disjoint reachability analysis to statically analyze the
code [JhED11]. When the analysis cannot safely prove that threads operate on disjoint
objects, it inserts a run-time check. During run time, it compares the addresses of both
objects to determine if they are indeed disjoint. Such an approach requires the use of a
specialized runtime and might not be feasible for all scenarios. In its current implementa-
tion, OoOJava sidesteps some of the complexities of analyzing the internal Java libraries
(e.g. Java File API) by translating programs into C and using simpler library calls.
Modern applications are complex. They tend to use many different programming idioms and many library APIs. Relying solely on fully automated parallelizing compilers is likely to fail and yield unsatisfactory results. On the other hand, our interactive approach is both useful and practical. By leveraging fast analyses, our approach tightens the feedback loop, allowing developers to invoke our tool, act on the feedback and repeat the process until they have successfully parallelized their application.

While static analysis is constantly improving to account for more programming idioms, it can never handle all of them. As Michael Hind remarked in the appropriately titled paper “Pointer Analysis: Haven’t We Solved This Problem Yet?” [Hin01]:

During the past twenty-one years [referring to 2001, the year the paper was published], over seventy-five papers and nine Ph.D. theses have been published on pointer analysis. Given the tomes of work on this topic, one may wonder, “Haven't we solved this problem yet?”

His answer in 2001 was “No!” And, more recently, during his talk at the 2013 Dagstuhl Seminar on pointer analysis, he believes that there will continue to be work in this area given how many of the problems previously identified in 2001 still remain open.

We believe that our approach of incorporating the developer as part of a tool's workflow complements static analysis and is a practical approach that yields great benefits. By engaging the developer, we are able to parallelize more applications than would have been possible through static analysis alone.
8.1 Summary of Contributions

The two key challenges to parallelizing an application are *analysis* and *transformation*. Both are typically done manually. Manual analysis is error-prone: given a sufficiently large and unfamiliar code base, it is hard for developers to identify code portions that inhibit parallelism, e.g., data races. Manual transformation is tedious: after analyzing the code to parallelize, developers still have to write repetitive, boilerplate code to initiate and stop the parallel code, e.g., starting and joining threads.

This dissertation addresses both challenges through interactive source-to-source tools. The thesis statement, as introduced in Chapter 1, is:

Interactive analyses and transformation tools that target the constructs of a parallel library are useful and practical ways to help developers parallelize their flow-based applications.

In this dissertation, we have argued and provided evidence to support the thesis statement through the following:

**Library Constructs** Chapter 2 presents the results of our case study demonstrating that using library constructs provides comparable performance, while increasing understandability and maintainability compared to using low-level threads. More importantly, it suggests that there is a structured series of transformations (the 3-step process) that developers follow for parallelizing their applications using library constructs.

**Analyses & Transformations** Chapter 5 presents our novel approach for analyzing and transforming sequential applications into parallel applications that target the constructs of a parallel library. We implemented our analyses and transformations to mimic the 3-step process that developers follow. The heart of our analyses is a fast staged-approach for detecting potential *inter* and *intra* node data-races.

**Interaction** Chapter 6 presents the evaluation of our approach. We parallelized seven
applications from a diverse set of emerging applications. Our evaluation demonstrates that with minimal interactions from the developer, we can successfully parallelize the applications with good performance (offering up to 3.45x speedup on a 4-core machine) and is fast enough to be used interactively as part of a developer’s workflow.

Researchers and practitioners can build upon our work to create new tools to support the growing field of flow-based parallelism.

8.2 Limitations

8.2.1 Exception Handling

As mentioned in Section 5.1.1, the first stage of our analyses checks that each node has well-structured control flow, i.e., a single exit. One feature of modern object-oriented languages that complicates control flow is the semantics of exception handling. Exceptions may be nested; in addition, they may be caught by an enclosing method higher in the call stack.

Exception handling poses challenges for parallelization tools both during and after the parallelization process. During the parallelization process, deeply nested exceptions make it difficult to find suitable partitions for flow-based parallelism. This is somewhat exacerbated by the fact that languages such as Java support the notion of checked exceptions that must be explicitly caught and handled. For instance, all the I/O APIs could throw a IOException that must be handled. In our current implementation, we sidestep this issue by requiring that each node should not throw an exception that it does not catch. However, this constraint is not always feasible in real code with deeply nested exception handlers. A developer might need to transform her code to change the way exceptions are caught and handled before she can use our tool to partition her code.

After the parallelization process, exception handling is also a challenge. In modern object-oriented languages such as Java, most calls to any parallel construct (including those from a parallel library) need to handle InterruptedException, which is thrown
when a thread is interrupted. This exception is frequently used to signal cancellation of a long-running task by a developer. Catching and handling this exception lets the cancelable activity clean up any work in progress, restore invariants, notify other activities of the cancellation, and then terminate [GPG+06, Lea00a].

We do not have a way to satisfiably handle InterruptedExceptions. The original code, being sequential, never has to deal with such an exception so we cannot refer to it for guidance. Currently, we do the minimum that is permitted by the language semantics. We catch the InterruptedException in an empty try ... catch block. The developer then fills in the details of how to handle the exception manually.

Given how pervasive exception handling is in modern object-oriented applications, both these challenges warrant further investigation. Krischer’s PhD Thesis, “Advanced Concepts in Asynchronous Exception Handling”, provide a good starting point to explore some of those challenges [Kri11].

8.2.2 Analyzing Multi-threaded Applications

Our current analyses and transformations help developers parallelize their sequential application. Thus, our analyses only handle the sequential semantics of an application. However, once a developer has parallelized an application, that application now has both sequential and parallel semantics. A developer might want to further parallelize or optimize her newly parallelized application. How would we modify our current analyses to work on parallel applications?

Handling both sequential and parallel semantics is hard. Our analyses for data dependencies have to change. We now need to consider the possibility of a data dependence from a different thread. We need to also consider the underlying memory model semantics for how threads can publish their data to other threads [MPA05]. Even if we are using the constructs of a parallel library, we need to consider how data is published to concurrently executing parts of the application.

Considering data dependencies in concurrent programs has been explored before in the literature, mostly in the context of program slicing. Cheng proposed three new kinds
of dependencies: (i) selection dependence to model control dependencies arising from non-deterministic selection, (ii) synchronization dependence to model synchronization between processes, and (iii) communication dependence to model inter-process communication [Che93]. Zhao presented a dependence-based representation called the multi-threaded dependence graph, which extends previous dependence-based representations, to represent program dependencies in concurrent Java programs [Zha99]. However, extending existing dependency analysis to account for parallel dependencies requires more complicated algorithms that consume more time and resources; it has yet to be empirically demonstrated if any of these techniques can scale to real applications.

As more applications are parallelized, the ability to extend our current tools to handle both sequential and parallel semantics becomes increasingly important. There has been some work on extending the current refactoring tools in modern IDEs to be aware of the concurrent semantics of the programming language. Schafer et al. examined the current automated refactoring tools in available in IDEs for Java and conclude that most of them continue to preserve program behavior when applied to a concurrent program. They also discuss instances of refactorings that no longer preserve program behavior and propose a framework for correctly preserving program behavior with respect to the underlying Java memory model [SDS+10].

8.2.3 Eliminating Data Races

Our current analyses and transformations detect possible inter and intra-node data races and warns the developer of them. However, we do not provide any automated tool to help eliminate these data races. Some of these data races are easy to eliminate, e.g., by converting a variable to use an AtomicInteger or by protecting all accesses to a variable through a lock. Some of the data races are harder to fix and might require creating private copies of each variable for each node and having some mechanism for aggregating their values. It would be useful to provide a set of guidelines on how to eliminate such data races and, perhaps, create automated tool support to handle some of the smaller, well-defined methods of eliminating data races.
8.3 Future Work

This dissertation adopts the interactive approach for refactoring sequential applications to use pipeline parallelism, the most common form of flow-based parallelism that we have observed. However, this only scratches the surface of flow-based parallelism. Many more styles of flow-based parallelism exist and it would be helpful to equip developers with tools to handle them. Future work in this area falls into two key themes: decomposition and re-composition.

Decomposition focuses on helping the developer partition the original sequential application into nodes of a computation flow graph. Currently we rely on the developer to annotate the nodes in a pipeline. This works well for a pipeline with its linear structure. However, many more complex control flows exist and work remains to improve our analyses to handle them (see Section 8.2). An approach worth investigating would be the creation of a visualization tool to help developers visualize the different options for partitioning. We have experimented with some basic visualizations for the control flow and data dependencies (using static analyses). While it works for small examples, it does not scale for larger examples. Larger examples with complex control and data dependencies create too many vertices and edges in the visualization, quickly overwhelming the developer. Visualization is not only useful for partitioning but could also be useful for debugging by pinpointing the specific data dependency that could cause a data race.

Re-composition, on the other hand, focuses on helping the developer transform the partitioned application to run in parallel. This involves composing the most appropriate parallel library constructs to use and tuning them not only for performance but also for extensibility. The current versions of Intel’s TBB and Microsoft’s TPL Dataflow provide at least 10 different parallel constructs, e.g., join_node, split_node, etc. Having a tool that could suggest or recommend which existing library construct to use could help the developer in her task. The suggestions could be based on control and data flow in the original sequential application. Or it could be based on a deeper analytical model of the characteristic of the application, perhaps through a simulation or dynamic profiling of the application.
References


[BFH+] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In SIGGRAPH ’04.


[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994.


LPHZ02  K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data
groups to specify and check side effects. In Proceedings of the ACM SIGPLAN

LSB09  Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The Design of a
Task Parallel Library. In Proceeding of the 24th ACM SIGPLAN conference on

LTNS10  Percy Liang, Omer Tripp, Mayur Naik, and Mooly Sagiv. A dynamic evalua-
tion of the precision of static heap abstractions. In Proceedings of the ACM
international conference on Object oriented programming systems languages
and applications, OOPSLA ’10, pages 411–427, New York, NY, USA, 2010. ACM.

From Patterns to Frameworks to Parallel Programs. Parallel Comput.,

MCKO08  Chi Cao Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. Stamp:
Stanford transactional applications for multi-processing. In Workload Character-

MCSW07  E. Mealy, D. Carrington, P. Strooper, and P. Wyeth. Improving Usability

Mes07  Gerard Meszaros. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley

Mes09  Message Passing Interface Forum. MPI: A Message-Passing Interface Stan-

sourceforge.net/projects/flow-based-pgm.

Mor10  John Paul Morrison. Flow-Based Programming: A New Approach to Application

MP85  S. Matwin and T. Pietrzykowski. PROGRAPH: A preliminary report. Com-
puter Languages, 10(2):91 – 126, 1985.

MPA05  Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model.
In Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles
of programming languages, POPL ’05, pages 378–391, New York, NY, USA,
2005. ACM.

MRR05  Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object


