A Graphical Representation of Exposed Parallelism

Gustavo Enrique Rodriguez Villamizar
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd
Part of the Electrical and Computer Engineering Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/6467

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
A Graphical Representation of Exposed Parallelism

Gustavo Enrique Rodriguez Villamizar

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

David A. Penry, Chair
James K. Archibald
Michael Joseph Wirthlin

Department of Electrical and Computer Engineering
Brigham Young University

Copyright © 2017 Gustavo Enrique Rodriguez Villamizar
All Rights Reserved
ABSTRACT

A Graphical Representation of Exposed Parallelism

Gustavo Enrique Rodriguez Villamizar
Department of Electrical and Computer Engineering, BYU
Master of Science

Modern-day microprocessors are measured in part by their parallel performance. Parallelizing sequential programs is a complex task, requiring data dependence analysis of the program constructs. Researchers in the field of parallel optimization are working on shifting the optimization effort from the programmer to the compiler. The goal of this work is for the compiler to visually expose the parallel characteristics of the program to researchers as well as programmers for a better understanding of the parallel properties of their programs. In order to do that we developed Exposed Parallelism Visualization (EPV), a statically-generated graphical tool that builds a parallel task graph of source code after it has been converted to the LLVM compiler framework’s Intermediate Representation (IR). The goal is for this visual representation of IR to provide new insights about the parallel properties of the program without having to execute the program. This will help researchers and programmers to understand if and where parallelism exists in the program at compile time. With this understanding, researchers will be able to more easily develop compiler algorithms that identify parallelism and improve program performance, and programmers will easily identify parallelizable sections of code that can be executed in multiple cores or accelerators such as GPUs or FPGAs. To the best of our knowledge, EPV is the first static visualization tool made for the identification of parallelism.

Keywords: parallel programs, software visualization, parallel loops
ACKNOWLEDGMENTS

I would like to give special thanks to my family, especially to my parents. Their sacrifices, hard work and example are the reason I got to where I am. My pursuit for a master’s degree in science is dedicated to them. I also want to thank Dr. David Penry and Matthew Ashcraft for being great mentors and teaching me all I needed to know in order to understand the world of LLVM and compilers in general. This thesis would have proven impossible without their help. Finally, this work has been supported by the NSF Award 1054075 - CAREER: Compiler-Inserted Runtime Adaptation for Multicore Processors. Special thanks to this great foundation for providing the support required for the development of this research.
# TABLE OF CONTENTS

## LIST OF FIGURES
- vi

## Chapter 1 Introduction
- 1

## Chapter 2 Background
- 5
  2.1 Optimizing Programs for Efficiency
    - 2.1.1 Loops
    - 2.1.2 Parallel Programming
    - 2.1.3 Data Dependence
    - 2.1.4 Exploiting Parallelism Through Data Dependence Analysis
  - 2.2 Program Structure Representation
    - 2.2.1 Basic Blocks
    - 2.2.2 Control Flow Graph
    - 2.2.3 Interval Analysis and Interval Trees
  - 2.3 LLVM
  - 2.4 DOT
  - 2.5 Graphviz

## Chapter 3 Related Works
- 15
  3.1 Trace-Based Visualization
  - 3.2 Dynamic Parallelism Visualization Tools
  - 3.3 Parallelization Tools
  - 3.4 Summary

## Chapter 4 Exposed Parallelism Visualization
- 20
  4.1 The EPV Process
  - 4.2 Graph Format
    - 4.2.1 Node Names
  - 4.3 Features

## Chapter 5 Implementation Details
- 31
  5.1 Data Generation: GraphGenRel
    - 5.1.1 Identification of Loop-Carried Dependences
    - 5.1.2 Generating Graph Layers
  - 5.2 Graphical User Interface Generation

## Chapter 6 Case Study
- 36
  6.1 Rodinia Benchmark Suite
  - 6.2 Particlefilter
  - 6.3 Results
    - 6.3.1 Top Level Analysis
    - 6.3.2 Loop 1 Analysis
LIST OF FIGURES

2.1 Loop figure .................................................. 6
2.2 Data Dependence Examples .................................. 8
2.3 Control Flow Graph example ................................. 11

4.1 EPV: System Overview ..................................... 21
4.2 EPV: Use Cases ............................................. 23
4.3 Sample program with its CFG and Interval Tree .......... 24
4.4 Merger of CFG and interval tree in Figure 4.3 ............ 25
4.5 Function Window ........................................... 26
4.6 Main Window ................................................ 26
4.7 Tab names indicate graph parents ......................... 28
4.8 Node information window .................................. 29
4.9 Legend window displaying information about color code and entry/exit nodes. 30

6.1 Particle filter: pseudo-code code containing only loops, conditionals and function calls. 37
6.2 Particle filter analyzed: Top level view ..................... 39
6.3 Particle filter analyzed: Loop 1, representing the longest top-level for loop in the function particleFilter() of Figure 6.1 (line 76). 40
6.4 Particle filter analyzed: Loop 11 shows the usefulness of red borders and font. 41
6.5 Pseudocode for loop22 in Figure 6.3 ..................... 42

A.1 Function Selection Window ................................ 51
A.2 Main Window ............................................... 52
A.3 Node Information Window ................................. 54
A.4 Legend Window ............................................. 54
CHAPTER 1. INTRODUCTION

Chip multiprocessors (CMP) are common in most markets that require high-performance computing. This tier of performance is very much dependent on the ability of these processors to efficiently execute programs in parallel, putting all of their multiple cores to work simultaneously. Accelerators such as graphics processing units (GPU) and FPGAs can also be used for general purpose computation, with highly parallel programs being sent to the GPU or FPGA for faster execution. The field of Heterogeneous Computing Techniques (HCT), which studies different ways to use multi-processing unit systems for higher performance, has been a hot field in the research community since some of the speedups achieved from combining CPUs with GPUs or FPGAs are considerably large [1–5]. The speedups achieved by systems using HCT are possible due to the existence of programs with parallel properties, without which the efficiency benefits of heterogeneous systems would be greatly diminished.

One of the most common ways to execute a sequential program in parallel is by explicitly exposing the parallel sections such program to the accelerators through the use of directive-based frameworks such as OpenMP [6], CUDA [7] and OpenCL [8]. This task is usually done by programmers who may or may not be the authors of the original program. This trend has motivated researchers to engage in the field of parallel software optimization, which attempts to automate the process of parallelizing source code. However, this is a complex task and it is still primarily up to the programmer to find and expose the parallel characteristics of their program for the runtime to execute. Furthermore, runtimes and the architectures where the programs will run are extremely diverse, which makes the programmer’s job even more complicated if he does not know exactly where his program will be executed or the runtime characteristics.

Tools have been developed to assist programmers with automatic identification and/or parallelization of source code [9–15]. However, all of these tools require multiple executions of the program in order to cover a large number of potential inputs. This approach is not scalable because
it requires a large amount of execution time to analyze complex programs with hundreds of thousands or millions of lines of code. Furthermore, the results of these previously developed tools are highly input-dependent, which requires many runs in order to have a sample that covers most of the possible inputs. Other tools such as Paralax [16] still require the user’s aid by annotating some code sections to aid the compiler to identify parallelism.

Static parallelism discovery and automation without user input is rarely found in the research community, one of the reasons being that static analysis of code struggles with pointer analysis and is very conservative in terms of the parallelism found [9, 10].

Visualizing large and complex systems or sets of information is an efficient and effective way to more easily understand and process the data. Approximately one quarter of our brain is used as an image processing engine [17], making vision one of our strongest senses. Because of this, software visualization techniques are widely used as a means of understanding large programs, allowing for easy identification of patterns and filtering out noise. Automated parallelization is not the target of this work, but rather the facilitation of such a task through visual means. The goal is for researchers to make use of a visualization tool to identify patterns and typical structures of parallel loops in order to come up with algorithms that can track these patterns. Furthermore, a visualization of parallelism would serve as a fast, easy to understand method for programmers who have no experience identifying parallelism or simply do not have the time that such a task requires. It is important to note that often times, programmers who are tasked with parallelization of sequential code are not the original authors of such code. As an example, a programmer tasked with parallelizing a sequential program that he did not write could use the tool presented in this work to easily spot all of the parallel and non-parallel loops in a program. After this, the programmer could refactor the code to eliminate some of the causes of non-parallel loops. Then, the programmer could evaluate which loops out of all of the loops marked as parallel by the tool could have a positive impact in the program if executed in parallel. Once these loops have been identified, the programmer can then add compiler directives that inform the runtime about what specific loops to run in parallel.

We believe that the development of automated parallelization will have a great impact in the efficiency of software worldwide given the widespread use of multicore processors in commodity computers and also the popular use of accelerators in industry.
The vast majority of software visualization tools that currently exist have primarily focused on representing the structure of object-oriented programs and dependencies between different files, functions and objects. Some other early visualization tools focused on visualizing the execution of large programs in distributed systems [18, 19], which is not the focus of this work. Finally, there are some tools that have been developed for visualization of parallel programs [20–26]. Although these tools have proven useful for small and mid-size programs, they start to lose value for large programs because the tools require prior execution of the program. Therefore, a static visualization tool which exposes potential parallelism in sequential programs is necessary.

This work presents Exposed Parallelism Visualization (EPV), a compiler-generated graphical tool created to aid researchers and programmers in the task of automated static parallelization by providing them with a parallel task graph. EPV’s primary focus is the display of loops that can be safely parallelized. It accomplishes this by searching for loops at any nesting level that have no loop-carried dependencies. The lack of data dependencies indicates that the loops can be executed in parallel without compromising program correctness. A secondary goal is to easily identify non-parallel loops and study their properties in order to gain insights about patterns of instructions that caused such lack of parallelism. We focus on loops for two reasons: First, complex programs spend most of their execution time in loop bodies, and second, loop structures are the most parallelizable by definition because they repeat a sequence of instructions multiple times.

The graph generated in EPV does not directly represent source code. Instead, it is a task graph of the LLVM compiler framework’s internal representation (IR) [27]. The reason we use LLVM IR is twofold: first, by making EPV depend solely on LLVM’s IR, we can use EPV on any programming language that can be compiled with LLVM. Second, working with the IR allows us to apply compiler optimizations and analyses to the program in order to identify dependencies between instructions. The analysis used for the creation of our graph is based on a set of relationships generated through an LLVM pass created in previous research.

The primary contribution presented in this work is a visualization tool which takes source code as its input, runs a sequence of LLVM passes on it, and generates a dynamic graph where parallel loops in the compiler’s IR are exposed to the user. The graph highlights both parallel and non-parallel loops through a color-code. EPV has a set of features which make it easy for the user to identify where these loops are and what they contain, among other things. By analyzing our
graph, researchers will be able to identify patterns and properties of parallel loops which they can then search for in compiler algorithms to automate the process of exposing parallelism at compile-time. Furthermore, programmers will easily identify the parallelizable sections of their code which will allow them to more easily place directives for parallel execution in multi-processors, GPUs or FPGAs.

The rest of this work is organized as follows: Chapter 2 introduces parallel programming, program structure and visualization and some software tools used by EPV. Chapter 3 summarizes past research on automatic parallelization and visualization tools for parallel programs. Chapter 4 presents EPV, its importance and contributions, some of the design decisions that were made for this tool and introduces its features. Chapter 5 explains implementation details in depth. Chapter 6 walks the reader through a case study that shows the usefulness of EPV and presents the results of other benchmarks. Finally, chapter 7 concludes this thesis. An appendix is included with a reference manual.
CHAPTER 2. BACKGROUND

This chapter serves as an introduction to parallel programming and its importance. It also introduces some of the software tools used in our research. The following sections will summarize the concepts necessary to understand parallelism, loops, data dependencies, program structure, and graphs as tools for visualization of program structure. Then we will introduce LLVM, the compiler framework that generates the data dependence data for EPV and how this data is generated through the LLVM pass GenRels, which is a product of previous research. After this we will introduce DOT, a graph description language used by EPV, and finally we will end the chapter with an introduction to Graphviz, a popular C library for visualizing graphs.

2.1 Optimizing Programs for Efficiency

In this section we will describe loops, parallel programming and data dependence analysis and how these concepts are used for the optimization of program execution time.

2.1.1 Loops

Loops are a programming construct defined by the repetition of a set of instructions until a condition is met. The general form of a loop is shown in Figure 2.1. Depending on the programming language being used, there are multiple forms of loops, but the two most common types are while loops and for loops. A while loop executes the statements inside it for as long as the condition in the while statement is true. A for loop is identified by having a counter (usually an integer). The counter can be initialized in the loop, and the statements inside the loop are executed if the counter meets the condition. Finally, an operation is done on the counter after the statements are done and the condition is checked again for a new iteration.
2.1.2 Parallel Programming

Before the era of manycore processors and the use of accelerators for general purpose computation (FPGAs, GPUs, etc), most programs were sequential. In a sequential program instructions are executed one at a time in the order they are written. The development of multicore processors and accelerators naturally calls for the more common development of parallel programs for improved performance. Most programs tend to spend most of their execution time in loops [28] since the number of iterations per loop in a given program usually far exceeds the total number of lines of code (or individual instructions). This is an important observation because if we want to minimize a program’s execution time, we can reduce this large problem to the smaller problem of minimizing the execution time of the loops in the program and get the same outcome. An effective way to reduce loop execution time is by executing different iterations of the loop in parallel in different hardware, which is known as parallel processing. Parallel processing is achieved through

Figure 2.1: Loop form.
parallel programming, which is a form of programming where programmers indicate to the runtime which sections of code should be executed in parallel through a series of tools. In order to annotate the parallel sections of a program, programmers must first analyze the source code to find those sections, since only sections with a specific set of properties can be processed in parallel. Such analysis is known as data dependence analysis, which will be described next.

2.1.3 Data Dependence

A data dependence is a relationship between two instructions executed in order in a program path that have two properties: First, both operations use the same variable, and second, at least one of the operations writes to said variable. We say that there is a data dependence because the latter instruction depends on data that was modified or read by the former instruction. The significance of identifying data dependencies is that if two instructions that have a data dependence are not executed in the right order, the result will change (thus making the program’s output incorrect). There are three types of data dependences between any two instructions \( I_1 \) and \( I_2 \) (where \( I_1 \) precedes \( I_2 \)):

- Anti-dependence: \( I_1 \) reads the variable and \( I_2 \) writes to it, or a write-after-read sequence.

- True dependence: \( I_1 \) writes the variable and \( I_2 \) reads it after, or a read-after-write.

- Output dependence: \( I_1 \) writes the variable and \( I_2 \) writes it again, or a write-after-write.

The order of instruction execution in these three variable access patterns must be respected for program correctness (i.e. \( I_2 \) can’t begin until \( I_1 \) finishes). As an example, the code in Figure 2.2(a) represents a true dependence: line 4 depends on the result computed in line 3. Therefore, the two instructions cannot be executed in parallel. This is a loop independent dependence because even though there is a dependence between two instructions, both instructions are in the same loop iteration. Although in this example line 3 could easily be computed in line 4 to avoid the dependence, we wrote it this way as an illustration to show what a loop independent, true dependence looks like. A loop-carried dependence is shown in Figure 2.2(b). In this example, an instruction in a loop iteration has a data dependence with an instruction from previous iterations of the same loop. This constrains the execution of line 4 to wait until the previous loop iteration is finished since the
value being read in iteration \( i \) is written to on iteration \( i - 1 \). Loop independent dependences are not a serious issue for parallel processing because the greatest performance gain is achieved from executing each iteration in parallel in each processor or accelerator core, but the loop body is still executed sequentially. However, loop-carried dependences are problematic because they establish an order between iterations, preventing the parallelization of the loop iterations.

**Data Dependence Analysis**

Data dependence analysis checks a program for all data dependences. This analysis can prove difficult in type-unsafe languages (such as C++) because these languages allow pointers to be cast to point to any memory location. Another difficulty arises when attempting to perform data dependence analysis at compile time. This is because compilers cannot make assumptions about memory accesses that depend on user input. Thus, compilers usually assume the worst-case scenario for input-dependent memory accesses and report these accesses as potential data dependencies.

There are multiple techniques that can detect data dependences. Some of the more important ones are array data dependence analysis and pointer-alias analysis. In array data dependence analysis, array indices are analyzed to check for array accesses to the same index. Pointer-alias analysis checks for pointers that access the same memory address (this is the definition of pointer aliasing).

(a) Example of a true dependence

```c
int A[N];
for (int i = 0; i < N-1; i += 2) {
    A[i] = i*2;
}
```

(b) Loop-carried dependence

```c
int A[N];
A[0] = 1;
for (int i = 1; i < N; i++) {
}
```

Figure 2.2: Data Dependence Examples
Pointer-alias analysis can be done at different scope levels. Some analyses focus on entire programs (*interprocedural alias analysis*), some others are done on a function basis (*global alias analysis*), and some even look at a single block of code. Additionally, an interprocedural analysis can be context-sensitive or context-insensitive. Context-sensitive interprocedural analysis checks each function as many times as it is called, taking the calling context into account on each analysis. This allows the analysis to make more precise inferences about the effects that the analyzed functions might have, at the cost of a more computationally expensive analysis. Context-insensitive analysis only checks each function once regardless of its calling context. Thus, the inferences that can be made about each function are more conservative since the analysis does not have any information about the calling context (see [28] for a more in-depth discussion about context sensitivity).

2.1.4 Exploiting Parallelism Through Data Dependence Analysis

As stated in the previous sections, programs can be optimized for faster execution through the discovery and implementation of parallel sections of code. Such parallelism can be detected through analysis of a program’s source code. In particular, loops represent ideal structures for parallel processing, making loop parallelization an effective technique for program optimization. A loop can be executed in parallel if it contains no loop-carried dependences. Therefore, a programmer’s task to speed up a program’s execution through parallel processing is facilitated by analyzing the program’s loops and determining which loops are candidates for parallelization.

2.2 Program Structure Representation

Understanding program structure is an effective way to identify problems and also parts of the program that can be optimized. One of the most effective ways to study program structure is through graphic representations that highlight certain aspects of the structure while filtering out others. Two common representations which aid in the understanding of program structure are the *control flow graph* (CFG) and *interval trees*. We will explain these and their benefits and weaknesses in the next subsections, but first the concept of a *basic block* will be introduced.
2.2.1 Basic Blocks

A basic block is a set of sequential instructions with a single control entry point at the beginning and a single exit at the end. Entries and exits are marked by jumps or jump targets. The concept of basic blocks is key because it allows for easy analysis and optimization of code. Once a program starts executing a basic block, an assumption can be made that every instruction in the basic block will be executed in the order in which it was written without any control jumps in between. This assumption is helpful for a set of compiler optimizations and useful graph definitions (e.g. the aforementioned CFG and interval trees).

2.2.2 Control Flow Graph

A control flow graph (CFG) is a set of vertices and directed edges \((V, E)\) that represents all possible paths that can be taken by the program for any given input. Every vertex \(V_i\) represents a basic block and every directed edge \(E_i\) represents a jump between two basic blocks in the program. For example, in the program in Figure 2.3(a) the returned value depends on the conditional and can be represented by the CFG found in Figure 2.3(b). As programs become increasingly large and complex, visualizations such as the CFG allow us to determine patterns and discover properties of programs which would otherwise be incredibly hard to conclude by analyzing the source code.

2.2.3 Interval Analysis and Interval Trees

An interval is a section of a CFG with a single entry point. Interval Analysis is a technique which aims to represent the structural hierarchy of a program by grouping program intervals according to the control flow inside the interval and how it affects its successors or children. An interval type gives information about the control flow of the part of the program inside the interval, the number of exit control flow edges going out of the interval, and in some cases the hierarchy of the interval as compared to other intervals. Below is a list of the different types of intervals with a short explanation:

- **BLOCK** Represents a basic block (see Section 2.2.1).
- **LOOP** Contains a set of intervals that form a loop (there is a back edge to the entry block).
int bar(int c) {
    int r;
    if (c)
        r = 2*c;
    else
        r = 1;
    return r;
}

(a) Sample function

(b) CFG of function in (a)

Figure 2.3: Control Flow Graph example

- **CHOICE** Equivalent to a diamond structure in a CFG. The path taken by the program depends on a condition.

- **PROPER** There is an arbitrary control flow structure inside it, but all paths converge to create a single exit at the end.

- **IMPROPER** Similar to PROPER, contains arbitrary control flow inside. However, an IMPROPER region is not required to converge at the exit, leading to multiple exit paths.

- **LINEAR** Interval where the next level of hierarchy (i.e. its contents) have no control flow between them. Single entry, single exit.

- **ROOT** This is the top-level node. It contains all other intervals in the program or function analyzed.
An *Interval Tree* is a tree data structure based off of interval analysis. It is built bottom-up starting at the leaf intervals (blocks) and incrementally merges them into larger intervals until it becomes root. It also keeps track of connections between parents and children and even keeps track of successors and predecessors, which can be compared to the edges in a CFG at the interval granularity. This structure is incredibly useful for understanding the hierarchy of a program at different levels of granularity. An interval tree can be constructed at compile-time.

### 2.3 LLVM

Low Level Virtual Machine (LLVM) [27] is a compiler framework written in C++ but usable with any programming language for the creation of both front ends and back ends. It contains a series of compiler passes which can be used for analysis and/or optimization of the original code. It is out of the scope of this section to explain every detail of LLVM, as this is a very complex and large project. Instead, this section will summarize the LLVM tools used for this research.

LLVM, like most compilers, transforms the source code into a data structure called an *intermediate representation* (IR) in order to facilitate analysis and optimization of a program. This representation looks similar to assembly code, but it has some unique properties. The most important property of the IR is static single assignment form (SSA), which requires that every variable only be assigned to once and only is read after being assigned. The fact that IR is in SSA form facilitates a large number of optimizations to be done to the IR, which can be mapped back to source code if debug information is generated. Identification of parallelism is one of the things that SSA form is useful for, since it allows for easy identification of data dependences between instructions: true dependences, anti-dependences and output dependences. LLVM provides its users with a large amount of standard analysis and optimization passes, as well as the ability to write their own passes for customized analysis and/or optimization. This has made LLVM a very popular compiler framework which is maintained and updated every year for research and commercial purposes.
**Relationship Generation Pass**

Because EPV was built to analyze sequential programs, we needed to perform a data dependence analysis to effectively find potential parallelism at compile-time. This information forms the basis for EPV because it allows us to identify loop-carried dependences. We perform such analysis through an LLVM pass named GenRels. GenRels performs a context-sensitive, field-sensitive interprocedural pointer-alias analysis based on Lattner’s Data Structure Analysis [29]. GenRels examines all load and store instructions in the program. For every load and store, it computes a function that will give the address accessed by the instruction. This function takes as input any loop indices or non-loop index variables that affect the instruction. In other words, it creates functions that map loop indices and non-loop variables to the memory addresses accessed by a load or store instruction dependent on those indices and/or variables. These functions are called the *address functions*. After that, GenRels tries to compute the set of combinations of indices and other variables such that two functions are equal (i.e. the addresses mapped are the same). We call these two functions the *left and right-hand side* of the relationship. That set then represents the situation under which the addresses are equal. We call this set of pairs of functions our *relationships*. A more in-depth explanation of address functions and relationships can be found in Chapter 5 where we describe the implementation of EPV.

**2.4 DOT**

DOT is a graph description language that allows users to represent a wide spectrum of graphs by writing them down in a textfile. Like all languages, it has a specific syntax that the user needs to follow when creating a graph in a *dot file* (ending with the extension *dot*). DOT-formatted graphs contain a set of attributes for nodes, edges, graphs and subgraphs which change the appearance of the graph when rendered (e.g. color, shape, label, etc). Because of the versatility of this language, there are many tools (e.g. *xdot*) that take dot files as input and allow the user to dynamically look at the graph and even modify it by adding or removing edges or changing attributes.
2.5 Graphviz

Graphviz [30] is a C library which was developed by AT&T Laboratories for easy drawing of graphs. EPV uses graphviz because some of its functions provide the user with complex and efficient algorithms such as laying out a graph given its edges and nodes. It also allows its users to programatically create, modify and delete graphs. Finally, because it mainly works with graphs in the DOT graph language, it also allows users to add, modify or remove attributes from graphs, nodes or edges such as color, position, shape, etc.
CHAPTER 3. RELATED WORKS

Parallelization of sequential programs has been an active field of research for several years, especially since multiprocessors became normal in supercomputers as well as commodity computers. Some tools for software visualization of parallel programs have also been developed in the past, but this field does not get as much activity as the field of sequential program parallelization. This is probably because researchers are more interested in the goal (fully automatic parallelization) rather than the development of tools to help us achieve such goals.

This chapter will summarize some of the more important works in these two areas as a base for EPV. By learning about past studies about parallelization of sequential programs and their visualization, we can better understand the strengths of EPV and its novel approach for discovering parallelism. Some of the earliest visualization tools that displayed information obtained from trace files will be described first. After this, a number of profiling tools that display information about potential parallelism are discussed. Finally, we summarize some dynamic tools developed for automatic parallelization of sequential programs both at compile-time as well as runtime.

3.1 Trace-Based Visualization

A trace file is a formatted text generated by a program during its execution. This file contains event logs such as communication between tasks, interrupts, etc. There are a number of visualization tools created in late 80s and early 90s that used the program’s trace file in order to graphically display the data contained in these files. These tools were developed for the analysis of programs executed in distributed systems.

LeBlanc presented a visualization toolkit [20] that generates multiple views of a parallel program’s trace. The views can be customized by the user to show a combination of the following three categories: process interactions, process states and time. The tool can provide insights
ranging from global characteristics of a program’s state down to communication between tasks in a process.

ParaGraph [19] displays multiple synchronized graphs with information about utilization, communication and task activity information. Users can open graphs with information about total communication traffic over time, task activity, size of processors’ incoming message queues, individual processor use, etc. This tool allows the user to have a direct look into the usage of the distributed system over different time intervals.

PARAVER [18] allows the user to visualize the state of CPUs, processes, threads and other similar objects through a Gantt diagram containing a time interval of the program’s execution. Communication between the objects is also displayed. This tool reads in a trace file with a specific required format to generate these diagrams. Users are able to open as many windows as they wish to cover different time intervals with different scales and different objects.

These tools were introduced because they represent a first step towards the usage of visualization to understand complex programs. However, our focus is that of facilitating discovery of parallelism, which is different than analyzing a program’s trace file post-execution to see resource utilization and communication.

### 3.2 Dynamic Parallelism Visualization Tools

The previous section focused on tools that displayed synchronization and state information of programs executed in distributed system. This section presents visualization tools created in the multicore era for discovery and evaluation of parallelism.

Given the conservative nature of compile-time data dependence analysis and the difficulty in statically finding such dependencies, tools for visualization of parallelism in sequential programs rely on prior program execution. In fact, to the best of our knowledge, no tools have been created which generate a parallelism visualization at compile-time. Here we present some of the more prevalent examples of these tools.

A loop visualization tool was developed [21] which shows the user a graph of the program’s iteration space with dependencies between iterations in three dimensions. The user can then drag the graph and look at it from different angles in order to analyze the parallelism in the loops of the
program. However, this tool requires instrumentation and multiple executions of the program with
different inputs in order to be representative of all possible inputs.

ThreadScope [22] builds a graph to help programmers understand the behavior of a parallel
program and identify bugs caused by its multithreaded nature such as deadlocks, data races, and
other synchronization errors. Similar to EPV, the graph is laid out and rendered with the Graphviz
[30] framework. Nodes represent either blocks of sequential code or memory objects (read/write)
and edges represent thread operations such as spawns and joins. The tool requires a trace file
in order to generate a description file containing information about thread and communication
operations. The event description file is then used as an input to the algorithm that generates the
final graph. This approach focuses on identifying inter-thread communication and other thread
operations and its purpose is to debug parallel programs rather than to help identify parallelism.

Trumper’s tool [23] generates a visualization of the trace file where different threads and
their respective function calls and delays are depicted over time in multiple 2D graphs. The objective
is to help developers gain a better understanding of complex multithreaded programs and the
patterns found in the execution of threads. This tool also facilitates debugging and identification
of bottlenecks.

HPCToolKit [24] contains a set of tools for analysis of sequential and parallel programs.
One of its tools, hpctraceview, asynchronously forms trace files that are presented in a 2D space-
time visualization which shows execution of the program and all of its threads and processes over
time.

Parceive [25] aids in parallelization of source code by performing a dynamic analysis of an
instrumented program. Using the trace data, Parceive generates information about function calls,
memory accesses and their relationships. After this analysis, the tool generates a graph where
functions and memory accesses are represented by nodes, and functions accessing specific mem-
ory locations are represented by edges. The graph also shows execution time per function through
color-coding. Finally, the authors of Parceive further developed a framework [26] based on the
aforementioned tool. This highly scalable and efficient visualization framework aids programmers
in identifying parallelism in their code by applying both a static and dynamic approach to pro-
gram analysis. Using Parceive’s trace and profiling data, the framework obtains information about
execution time, memory accesses, and function calls. Three views are generated to the user: Per-
formance view, which shows information about time spent in each function; calling context tree view, which allows data dependence analysis through a graph where call nodes, loop nodes and memory access nodes are shown; and finally source view, where the tool links source code with the other views for easier comprehension.

### 3.3 Parallelization Tools

This section introduces some tools that have been developed for the purpose of automatic parallelization of sequential programs. The authors of this research field would benefit from tools like EPV by saving time and simplifying the parallelism discovery process.

Alchemist [13] is a profiling tool that identifies sections of sequential code which are potentially parallelizable. In particular, Alchemist marks program constructs (e.g. conditionals, loops, functions) to indicate that they are candidates for parallelization through the analysis of read-after-write, write-after-read and write-after-write dependences. It also creates a notion of distance between a parallelizable construct and its continuation which helps identify which constructs are more amenable for parallelization.

Prospector [10] recommends a set of candidates for manual parallelization to the user after performing loop and data dependence profiling. It does some partial analysis at instrumentation-time but most of the information is obtained at runtime.

SD3 [14] identifies parallel loops in a program. It does this by applying the pairwise method, which keeps a table of pending and history accesses to any given memory address and uses the concept of killed addresses in order to differentiate between loop-independent and loop-carried dependences. To get rid of the large memory footprint of the pairwise method, the algorithm works with compressed formats (thus avoiding recompression). Also, in order to reduce the runtime overhead of the method, the technique parallelizes the data dependence profiling itself.

Kremlin [12] identifies and ranks parallel regions of sequential code. It tells the programmer information about what sections of the program have higher parallel potential through a metric called self-parallelism. Kremlin implements a hierarchical critical path analysis in order to identify parallel parent and children regions in every nested structure of the program.

Parwiz [15] identifies parallel sections and presents strategies to parallelize these regions in source code through the identification of data dependences at runtime. The programmer provides
Parwiz with a sample of input that should cover all possibilities and the program is run a few times and identifies data dependences which then are suggested to the programmer for corroboration. The source code is instrumented and the main constructs analyzed are instructions, loops and function calls.

Li’s tool [9] identifies parallelizable sections of code through the notion of Computational Units (CU), which are blocks where a variable follows the read-compute-write pattern. CUs are useful because the assumption can be made that there are no dependences within a single CU. Instead, edges connecting different CUs represent dependencies. The approach in this method is different as it looks for the absence of dependencies (this is a bottom-up approach rather than top-down as usual). An advantage of this approach is that there is no need for a granularity specification before the search begins. One last noteworthy aspect of this work is that the tool builds a CU graph which is used to identify parallelism.

Finally, [11] introduces a generic profiler for identifying data dependences which works for both sequential and parallel programs and also reduces runtime and space overhead. To reduce space overhead, the profiler records memory accesses in signatures, which also reduces runtime as compared to other space efficient methods. To further improve runtime, the profiling algorithm is run in parallel on disjoint subsets of the memory accesses. The key contributions are its speed and low space requirements and also the fact that it is generic (other profilers were tailored to specific applications to be as fast/space conservative).

3.4 Summary

The tools presented aid in the discovery, understanding and automation of parallelism. Virtually all of these tools can be classified as profilers since they rely on trace files. This is what makes EPV unique: It relies completely on static information generated at compile time, which results in significant time savings. Static analysis also guarantees input-independence. Another distinction that can be made between the tools presented and EPV is that EPV focuses primarily on loops as a main source of parallelism, saving its users from spending time optimizing unimportant parts of a program.
CHAPTER 4. EXPOSED PARALLELISM VISUALIZATION

Exposed Parallelism Visualization is a tool available at compile-time for identifying parallelism in a program. Its name is derived from its usage of compiler-generated data dependence information (using the pass GenRels, which we described in Section 2.3). Our tool uses this data (which we call the program’s exposed parallelism) to generate a powerful visualization of the program’s structure and control flow, highlighting loops and their potential parallelism. EPV is not currently open-source, making it only available to the faculty and students of Brigham Young University.

This work is not a solution to the problem of automatically parallelizing sequential programs, for that is a complex task that is out of the scope of this thesis. Instead, EPV can be thought of as a tool that fills a gap in the area of static program visualization tools for parallelism discovery. It shows its user a simple yet powerful picture of the program, the loops contained in it and their parallel characteristics in a graphical view. As mentioned in previous chapters, EPV is the first tool which does a data dependence analysis at compile-time and creates a visualization of loop parallelism. It is also the first parallelism visualization tool that focuses solely on program loops.

In order to highlight and differentiate parallel and non-parallel loops, every loop node is colored following a color-coding scheme. This scheme also provides the user with information about other properties of the loop such as whether the loop contains other nested loops or not. The color-code will be described in Section 4.3.

4.1 The EPV Process

Figure 4.1 describes the step-by-step process for generating EPV’s final result, starting with the user’s source code and ending with a graphical user interface (GUI) where the graphs are displayed. In the figure, ovals represent inputs and outputs (or in/out), and rectangles represent steps. A description of every step in the figure follows:
**Step 1**: The source code is compiled with LLVM. The GenRels and IntervalTreeAnalysis passes are run. This step outputs data structures containing LLVM IR, data dependence information and the interval tree.

**Step 2**: Algorithms (developed in this work) for detecting loop-carried dependences, filtering out relationships that are not loop-related, identifying loop-carried dependences and generating all graph layers are executed. This is the core step of EPV since most of the algorithms designed to discover loop parallelism and generating the visualization lie here. More details on these algorithms will be provided in Chapter 5. The output is graphviz objects representing all graph layers with parallelism information attached as attributes.

**Step 3**: The graphs generated in step 2 are run through graphviz’s layout algorithm. This algorithm takes a graph with its set of nodes and edges and generates a layout. The layout algorithm does its best to position nodes and edges in such a way that edges do not cross
each other, nodes are spaced out and the graph is easy to read. After this step, the graph objects created in Step 2 contain position attributes for later rendering.

- **Step 4:** In this step, the graph objects created and laid out in steps 2 and 3 are read and the EPV GUI is generated with C++’s Qt library, a popular API for graphical user interface creation. After this step, EPV’s graphical user interface is executed and a dynamic GUI opens to the user which contains all loop-parallelism information.

   In order to understand the big picture of EPV’s utility from a user perspective, Figure 4.2 shows the potential use cases of EPV. These users (i.e., programmers and researchers) get access to a visualization of their programs in seconds, with which they can make decisions such as whether to parallelize some loops through compiler directives, rewrite loop bodies to eliminate loop-carried dependences, or simply study the properties of the program in question.

   We will now describe some properties of EPV’s graph, followed by a list of features with a description of their use and importance. Further screenshots of all features and windows will be included in Appendix A, which will serve as a reference manual.

4.2 Graph Format

   The graph represented in EPV is a merger between a CFG and an Interval Tree (see sections 2.2.2 and 2.2.3). We made this decision as a way to convey the most information about the program structure and its loops in the most powerful yet simple way.

   A CFG would not be good enough on its own because complex programs have too many basic blocks that are not relevant to the discovery of parallelism. We had to use a data structure which could merge sets of basic blocks that we are not interested in from a parallelism perspective. On the other hand, an interval tree lacks information about the control flow, making it difficult for users to understand the graph and establish the connections between the LLVM IR and the interval tree. The control flow gives the easiest understanding of the graph, while the interval tree simplifies the graph by merging multiple non-parallelizable sections of the program into single nodes. These reasons led us to decide on a merger graph between both a CFG and an interval tree in order to get the best properties of both graphs.
As an illustration, Figure 4.3(a) shows a function containing two loops and a conditional. A CFG of this function is shown in Figure 4.3(b), followed by an interval tree of the same function in Figure 4.3(c). Notice how the CFG is too detailed and could potentially become very noisy for large and complex functions. On the other hand, the interval tree makes it easy to understand the hierarchical structure of the function, but there is no notion of control flow. Finally, Figure 4.4 represents the two layers of the merger between a CFG and an interval tree of *foo*. This merger graph is ideal because it keeps the best properties of both a CFG and an interval tree (i.e., control flow and hierarchy structure), while keeping the visualization concise.
Matrix* foo(int r, int c) {
    Matrix* result = new Matrix(r, c);
    for(int i = 0; i < r; i++) {
        if(i % 2 == 0) {
            for(int j = 0; j < c; j++) {
                result->data(i, j) = (-1)*rand();
            }
        } else {
            for(int j = 0; j < c; j++) {
                result->data(i, j) = rand();
            }
        }
    }
    return result;
}

(a) Sample function

(b) CFG of function in (a)  
(c) Interval Tree of function in (a)

Figure 4.3: Sample program with its CFG and Interval Tree

4.2.1 Node Names

Nodes in EPV are named based on the interval type of the current node such as “block”, “loop”, and “choice” (for more information see Section 2.2.3). We made this decision after trying to use the entry basic block as the node’s name, which caused the graph to become very confusing
since LLVM IR’s naming convention is not easy to understand in a graph and often gives basic blocks very long names.

### 4.3 Features

Below is a list of the features of EPV. This section will only describe each feature along with its primary purpose or importance. A user reference manual is included in Appendix A with screenshots and instructions on how to use every feature.

- **Function Window**: This is the first window shown to the user when the LLVM pass which generates EPV is kicked off and is shown in Figure 4.5. It contains a scrollable list of all functions of the sequential code being analyzed. A user can go back to this window at any time to re-open a visualization for the same function or other functions as needed. More information about why we decided to evaluate each function individually can be found in Chapter 5.

- **Main Window**: The main window, shown in Figure 4.6 contains three elements: The graph pane covering most of the window, a list pane on the left side, and a legend button below the list pane. These are explained in more detail below.
- **Graph Pane**: This is where the graph of the current function is shown. Initially, the graph pane shows the top level graph, but users can “expand” loop nodes that contain
nested loops inside. The new graph layer is opened in a new tab in the graph pane. The features of the graph displayed in this pane are listed below:

* **Color-Code**: Loop nodes’ fill, border and font colors indicate the loop contents and parallelism. Fill colors define whether a loop is parallel or not (based on their loop-carried dependences content) and it also indicates whether the loop contains other nested loops inside it or not. We call a loop with no inner loops a *leaf loop*. There are 4 fill colors:
  - Red indicates a leaf loop that is not parallelizable because it contains loop-carried dependences.
  - Green indicates a leaf loop that is parallelizable.
  - Yellow indicates a non-leaf loop that is parallelizable.
  - Orange indicates a non-leaf loop that is not parallelizable.

A red border and font color on a node indicates that the node contains a nested non-parallel loop inside it. This feature was added for users that are interested in finding all loop-carried dependences easily in order to make changes to the source code or to study their properties. Color-coding is one of the most important features in EPV because it conveys quick and easy information about parallelism and allows the user to identify non-parallelizable loops from upper layers as well as the current layer’s parallel and non-parallel loops. The color code is presented to the user in the legend window, described later in this section.

* **Expandable Loops**: Loop nodes which are not leaf loops can be expanded by the user with a double click. The new graph layer will be opened in a new tab in the graph pane, allowing the user to switch back and forth between different graph layers.

* **Tabs indicate tree structure**: The names of each tab opened by the user form a ”trace” of the layer’s parent, all the way to the top. This helps the user to remember what loop is currently being looked at if it is multiple layers deep. See Figure 4.7.

* **Zooming**: Users can zoom-in or out. This facilitates emphasis on certain areas of the graph vs. a visualization of the entire graph.
Figure 4.7: Tab names indicate graph parents

* **Dragging**: Users can drag the entire graph. This is a basic feature required for interacting with the graph dynamically.

* **Node Information**: Users can open a window with information about each node by right-clicking on the node. This window contains a list of basic blocks contained by the current node as well as the instructions (in LLVM’s IR) that cause the loop-carried dependence if there is one. This allows the user to correlate the graph to the IR much more easily. See Figure 4.8.

* **Entry and Exit Nodes**: Entry nodes at lower graph layers are marked by an incoming vertical arrow to let the user know that that is the entry node of the loop expanded. Exit nodes (nodes that have edges connecting to an upper layer) are also marked with a vertical outgoing arrow below the node that is not connected to anything. These can be seen in Figure 4.6: block1 is an entry node, and loop1 and block2 are exit nodes.
– **Node Pane**: This is a scrollable list where nodes are listed by their names as displayed in the graph pane. A useful feature of the node list is that it lists the node’s entry basic block after its name in order to make it easier to relate the generic node names created for the graph to the LLVM IR for the source code. Another important feature is that by clicking on a node, the graph in the graph pane is dragged, leaving the selected node in the center. The node pane can be seen on the left side of Figure 4.6.

– **Legend Button**: This button opens a window with the legend where the color-code is explained. The legend button can be seen on the bottom-left corner of Figure 4.6, and the legend window is shown in Figure 4.9.
Figure 4.9: Legend window displaying information about color code and entry/exit nodes.
CHAPTER 5. IMPLEMENTATION DETAILS

The implementation of EPV will be described in detail in this chapter. The data collection will be discussed first, which is where most of the work is done: An LLVM pass, named GraphGenRels, calls other LLVM passes for data dependence analysis, interval analysis, construction of all graph layers and then performs identification of loop-carried dependences. After this, we will describe the implementation details for the user interface developed in the Qt framework for C++ as well as the transfer of data from GraphGenRels to the GUI.

5.1 Data Generation: GraphGenRels

GraphGenRels is an LLVM module pass implemented in order to generate all of the information necessary for identification of loop-carried dependences as well as the creation of all graph layers including nodes, edges, layout and marks on entry nodes, exit nodes, and parallel and non-parallel loops at both leaf level as well as higher interval levels. One of the main characteristics of this pass is that it runs separately on each function of the source code, ignoring standard library functions that do not modify memory. The decision to evaluate a program on a per-function basis was made because we are not concerned with the function call structure of the program but rather we want to analyze loops in every function for parallelism. However, we leave it for future work to study parallelism in the call structure of a program.

For each function, the first step in GraphGenRels is to run two other key LLVM passes developed in previous research: IntervalTreeAnalysis and GenRels. The former generates an interval tree data structure for the program being evaluated. The latter performs data dependence analysis and populates a list of relationships (see Section 2.3 for more detailed information). Then, we filter out all relationships from the list that are not between instructions inside loop bodies because we are only concerned with loop parallelism. The remaining relationships are analyzed and tested to evaluate if they are loop-carried dependences. The process to identify loop-carried dependences

31
is explained in detail below since it represents one of the key mechanisms required for EPV’s purposes.

5.1.1 Identification of Loop-Carried Dependences

Recall that GenRels generates address functions and a set of indices that cause a relationship. Address functions map loop indices and non-loop variables to the memory addresses accessed by a load or store instruction. These functions are

\[ f_1(<i_1, \ldots, i_n, v_1, \ldots, v_p>) \]
\[ f_2(<j_1, \ldots, j_n, v_1, \ldots, v_p>), \]

where \(<i_1, \ldots, i_n>\) and \(<j_1, \ldots, j_n>\) are the loop indices of each side of the relationship at each loop nesting level and \(<v_1, \ldots, v_p>\) is an arbitrary number of non-loop variables. Now, the set of indices on the left and right-side of a relationship generated by GenRels to identify aliasing memory accesses is

\[ S = \{<i_1, \ldots, i_n, j_1, \ldots, j_n, v_1, \ldots, v_p>| f_1(<i_1, \ldots, i_n, v_1, \ldots, v_p>) = f_2(j_1, \ldots, j_n, v_1, \ldots, v_p>)\}. \]

All of the set manipulations and operations required to compute \(S\) are done through a C library called the integer set library (ISL) [31], which offers a set of functions that can be used for performing operations on sets and relations of integer points under linear constraints. Because ISL only works with linear constraints, not all sets can be represented. If a set has non-linear constraints, \(S\) has \(v\) variables which are non-affine functions of index variables. In practice, when \(v\) variables exist in \(S\), this relationship is automatically classified as a loop-carried dependence because GenRels has no information about the actual value of these variables.

Now, a loop at nesting depth \(m\) is loop-carried if:

\[ \left( S \land \{\bigwedge_{k=1}^{m-1} i_k = j_k \land i_m \neq j_m\} \right) \neq \emptyset. \] (5.1)
We evaluated each relationship following Equation 5.1 to determine if a loop-carried dependence exists. After this step, we are left with two data structures: the interval tree of the function currently being evaluated, and a set of relationships where each relationship is within the same loop and represents a loop-carried dependence. With these two data structures, the program counts with everything it needs in order to create all of the graph layers for the function.

5.1.2 Generating Graph Layers

The algorithm for generating graph layers begins by creating the nodes for the current layer. The node creation function is an interval refining process that begins with the root interval node and only refines an interval node if it is the ancestor node of a loop node or it is a loop itself. Pseudo-code for the node creation function can be found in Algorithm 1. After the nodes have been created for the current graph layer, edges between nodes are created by accessing the successors and predecessors of each interval tree node in the list of nodes previously created. An edge is created and saved if one of two conditions are true:

- the predecessor or successor being evaluated is a node in the current graph layer.
- the predecessor or successor being evaluated is a child or descendant of a node in the current graph layer.

If the predecessor or successor is at a higher level graph layer, the node is added to a list of exit nodes. After all nodes and edges have been created for the current graph layer, our algorithm proceeds to create all other layers.

Finally, once all layers have been created, our algorithm takes all of the information in each graph layer and creates graphviz graph objects. Graphviz allows us to attach attributes to nodes such as their color, position, size, etc. In this step, we give graph nodes generic labels based on their interval type (e.g. block1, block2, loop1) since right-clicking on the node shows all of the basic blocks contained. Parallel loops are also color-coded in this step by checking if the graphviz node matches with with an entry in our list of loop-carried relationships. If a match exists, the node is colored red if it is a leaf loop or orange if it is not. If it doesn’t match with any of the relationships representing loop-carried dependences, the node is colored green if it is a leaf loop.
or yellow if it’s not. Finally, whenever a non-leaf loop contains a nested loop with a loop-carried
dependence, the edge of that loop is colored red. This was done for programmers who are looking
for non-parallel loops in order to modify the source code to get rid of the dependence. All of this
color coding is shown to the user through a legend implemented in the GUI. The graphviz nodes
get their positioning attributes through Graphviz’s layout algorithm, which takes in a graph and
lays it out in such a way that the nodes don’t collapse with each other and edges rarely cross over.

Algorithm 1 Create Nodes

Require: \( n \) is an interval tree node
Require: \( LC[n] \) is the set of loops contained in node \( n \)
Require: \( N \) is the set of nodes for the current graph layer
Require: \( toExpand \) is the set of root nodes from which a new layer will be created.
Require: \( n.contents \) is the set of interval nodes contained in node \( n \)

\[
\begin{align*}
1: & \quad \text{function CREATE\_NODES}(n) \\
2: & \quad \text{if } LC[n] \neq \emptyset \text{ then} \\
3: & \quad \quad \text{if } n \in LC[n] \text{ then} \\
4: & \quad \quad \quad N.insert(n); \\
5: & \quad \quad \quad \text{if } LC[n].size > 1 \text{ then} \\
6: & \quad \quad \quad \text{toExpand.insert}(n) \\
7: & \quad \quad \quad \text{end if} \\
8: & \quad \quad \text{return} \\
9: & \quad \text{else} \\
10: & \quad \quad \text{for all } child \in n.contents \text{ do} \\
11: & \quad \quad \quad \text{CREATE\_NODES}(child) \\
12: & \quad \quad \text{end for} \\
13: & \quad \text{end if} \\
14: & \quad \text{else} \\
15: & \quad \quad N.insert(n); \\
16: & \quad \text{return} \\
17: & \quad \text{end if} \\
18: & \text{end function}
\end{align*}
\]

5.2 Graphical User Interface Generation

As mentioned in the previous section, the GUI for EPV was created through Qt’s GUI
library. One of the first challenges we encountered was passing the data structures from our LLVM
pass to the GUI. We did this through Graphviz since it provides functions for dumping graphs into
textfiles with the dot format. Our LLVM pass creates a temporary directory that it fills up with dot files and text files with all the information necessary to build the GUI. The very last step of the pass is to fork a process which calls the executable for the GUI and waits for the GUI to be closed by the user to resume execution.

The GUI's implementation has two main windows: the function window, containing a list of all functions and the main window, containing the tabbed graph pane, node list pane and legend button. When the binary is executed by the LLVM pass, the function window is opened and loads all of the program’s function names by reading a textfile in the temporary directory created by our LLVM pass. When the user selects the function, the main window is created. Once control goes back to GraphGenRels (our LLVM pass) after the user closes the GUI, the temporary directory is deleted to avoid permanently taking disk space from the user of EPV.

From an implementation standpoint, these windows and their widgets are implemented following Qt’s documentation. They follow the concept of signals and slots. A signal is emitted by widgets through user input such as clicks, scrolling, dragging, selecting a node from a list, etc. Slots are functions which are connected to signals for a response to user input.
CHAPTER 6. CASE STUDY

This chapter will present a case study where we used EPV to analyze a popular heterogeneous computing benchmark from the Rodinia Benchmark Suite [32] named particlefilter. The following sections will introduce the benchmark and describe the results of using EPV for the discovery of parallelism in particlefilter.

6.1 Rodinia Benchmark Suite

The Rodinia Benchmark Suite is very popular in the field of heterogeneous systems. The suite contains a set of computationally heavy programs that are rich in loops and parallel sections. The benchmarks were designed to measure the efficiency of scheduling algorithms for heterogeneous systems. These are algorithms work by scheduling threads to different execution units for parallel execution.

We chose a Rodinia benchmark for our case study because all of the suite’s benchmarks have been manually parallelized through the addition of compiler directives. This proved helpful as a comparison point between the loop parallelism that can be found through EPV and the manual discovery of parallelism performed by the benchmark’s authors.

6.2 Particlefilter

We picked particlefilter from the Rodinia benchmarks because it is one of the richest benchmarks in the suite in terms of loops that are both parallel and non-parallel. Particlefilter estimates the location of a target object given noisy measurements of the object’s location and the object’s path. We made a copy of the particlefilter source code and modified it by removing all compiler directives to make the code fully sequential. Then we ran EPV to see the result of running our tool on a program with over 30 loops. Pseudo-code containing only the loops, conditionals, functions declarations and function calls for particlefilter is shown in Figure 6.1.
void videoSequence(params) {
  for (...) {...}
  call imdilate_disk(params);
  for (...) {
    for (...) {
      for (...) {...}
    }
  }
  call setIf(params);
  call setIf(params);
  call addNoise(params);
}
void imdilate_disk(params) {
  for (...) {
    for (...) {
      for (...) {
        if (cond) {
          call dilate_matrix(params);
        }
      }
    }
  }
}
void setIf(params) {
  for (...) {
    for (...) {
      for (...) {
        if (cond) {...}
      }
    }
  }
}
void addNoise(params) {
  for (...) {
    for (...) {
      for (...) {...}
    }
  }
}
void dilate_matrix(params) {
  for (...) {
    for (...) {
      if (cond) {...}
    }
  }
}
void findIndex(params) {
  for (...) {
    for (...) {
      if (cond) {...}
    }
  }
}
int main(params) {
  for (...) {...}
  call videoSequence(params);
  call particleFilter(params);
  return 0;
}

Figure 6.1: Particlefilter: pseudo-code code containing only loops, conditionals and function calls.
6.3 Results

Because of the large number of loops in particlefilter, we will only show two of the graph layers generated by EPV in this chapter: the top-level in Figure 6.2, and a layer that represents a large loop nest inside the function particleFilter. The particleFilter function starts on line 66 of Figure 6.1, but the sublayer that we include in Figure 6.3 represents the loop starting on line 76. Additionally, a loop that EPV named loop11 is highlighted for the purposes of this case study (see Figure 6.4). An explanation of this decision is in a later subsection where loop11 is discussed. One observation derived from these figures as well as the IR generated is the fact that LLVM inlined all functions into main. Compilers often do this to facilitate optimization efforts since function calls usually complicate the analysis of IR.

6.3.1 Top Level Analysis

The first observation to make when looking at the top level layer of particlefilter on EPV (see Figure 6.2) is the ease with which a complex program can be understood through the graph. Since control flow is preserved, it is easy to see branches and identify the loops in the program simply by scanning top-to-bottom or vice versa. For example, the first loop in the program is the topmost colored node (colored green in this case). The user can come to this conclusion by inspecting the graph for no more than a few seconds. Additionally, it is easy to see that this loop corresponds to the for loop on line 98 in Figure 6.1. To identify other loops, it suffices to keep scanning loops top-down in the graph while comparing them to the pseudo code in Figure 6.1. Similarly, it can be concluded that the bottommost loop (colored orange) is the last top-level loop in the program, which can be found on line 76 by inspection. This technique can be used to identify line numbers for every loop at this level and every other graph layer or loop depth. Thus, we see the simplicity with which users of EPV can relate the tool’s visualization to the source code even without having to scan the IR.

Another advantage of EPV observed in this example is the fact that our merger graph abstracts away basic block nodes that are unimportant for parallelism discovery. The CFG of this same example is much harder to read because it contains so many basic blocks that it looks like a
Figure 6.2: Particlefilter analyzed: Top level view.

long vertical line of dots when zoomed out enough to fit the whole graph on the screen (this is also the reason why a screenshot of the CFG is not included here, as it would be unreadable).
Figure 6.3: Particlefilter analyzed: Loop 1, representing the longest top-level for loop in the function particleFilter() of Figure 6.1 (line 76).
6.3.2 Loop 1 Analysis

*Particlefilter* was manually analyzed and parallelized by its authors. In total, it contains 10 loops with compiler directives for parallel execution. Loop 1 in Figure 6.3 contains 8 out of those 10 loops. These are all of the loops in the figure except for loops 14 and 17. This is an important observation because out of those 8 loops, EPV marked 7 as parallel. Loops 14 and 17 were not manually parallelized by the benchmark authors. This could be due to two reasons: one, the benchmark authors’ analysis erroneously led them to believe these loops are not parallel; two, perhaps the authors chose not to execute these loops in parallel since they do not represent an
for( x = 0; i < Nparticles; x++) {
    for( y = 0; y < countOnes; y++) {
        if( array[x*countOnes + y] >= maxSize) {
            array[x*countOnes + y] = 0;
        }
    }
}

Figure 6.5: Pseudocode for loop22 in Figure 6.3

This is an example of the time savings that EPV could provide to programmers or researchers looking to quickly optimize their programs without performing an in-depth analysis of the source code. It is also a great example of a situation where a loop that is actually parallelizable contains what the compiler conservatively (and erroneously) considers a loop-carried dependence. This is the case of loop22. Pseudo-code for loop22 is shown in Figure 6.5. The data dependence analysis cannot determine the value of the array index in lines 3 and 4 of the pseudocode. Since the array index is dependent on both for loop counters, the data dependence analysis cannot deterministically know if there are two permutations of values of x and y that will access the same array element (causing a memory alias). Because of this, it marks this loop as non-parallel even though it can still be parallelized. This is an example of a situation where EPV reports a loop-carried dependence incorrectly, which is also a valuable contribution to researchers looking for patterns of instructions that “trick” compilers into thinking that such instructions may alias. By using EPV, researchers can quickly inspect all non-parallel loops because they’re marked with a color code. Then they can quickly determine if the loop-carried dependences really exist by inspecting the IR since it’s easy to connect the visualization to the IR through EPV’s node information window and also to the source code through inspection of loop structure, as discussed in Section 6.3.1.
6.3.3 Loop 11 Analysis

Loop 11 is shown in Figure 6.4. This is a loop displayed at the top level graph of EPV’s visualization of *particlefilter*. Analysis of this loop in our case study revealed another one of the contributions of EPV: the ability to identify loop-carried dependences existing at deep layers by glancing at the top-level layer. In this particular case, loop 11 is a loop nest that is 5 layers deep, with the loop-carried dependence happening at the deepest level. The dependence occurs because an array is accessed with an index that depends on all 5 loop counters. Because of this loop-carried dependence, all of the nested loops become non-parallel as they all share a loop-carried dependence. EPV quickly shows the loop-carried dependence at the highest level of loop nesting. Even if the first 4 levels were parallel loops and only the last loop had a loop-carried dependence, the red borders and font would indicate this to the user.

6.4 Summary of Results and Lessons Learned

We saw through the analysis of *particlefilter* that EPV effectively represents a program’s loops in a visually appealing way. We were able to easily discover all parallel loops and loop-carried dependences. Additionally, this case study took a total execution time (including all dependent passes) of 10 seconds on a commodity machine from 2012. Below is a list of lessons learned:

- It is easy to correlate the graph to the IR through the node information window, which shows the basic blocks contained in each node.

- Correlation between the graph and the source code is also possible with little effort by following the loop structure of the top level graph (as well as other layers) and comparing it to the loops found in the source code.

- Unimportant basic block nodes are filtered out by the properties of the graph used in EPV, only leaving the branches that lead to loops for easier understanding.

- This case study also identified 9 out of 10 of the loops that were parallelized by hand by researchers who performed an in-depth analysis of the code in order to discover this loop parallelism, and it did this in under 10 seconds of compile-time and analysis.
• We were able to easily discover a “false” loop-carried dependence identified by the compiler at a deep loop nesting from the top level through the node’s color code. This will help future researchers to study loop-carried dependences much faster should they decide to visualize their benchmarks with EPV.

• EPV marked a total of 19 loops at different nesting levels as parallel. Out of the 19, 9 were actually manually parallelized by the developers of the benchmark. The other loops were not parallelized because they are only executed once and/or they have a low number of iterations, so the execution time of these loops is minimal. Thus, by using EPV, users can quickly identify all parallel loops and then determine which ones are worth parallelizing through a quick analysis of the loop bounds and location in the source code.
CHAPTER 7. CONCLUSIONS

This work introduced Exposed Parallelism Visualization, a tool for researchers and programmers that helps in the discovery of loop parallelism in sequential programs at compile-time through an interactive graphical user interface. EPV has several unique properties:

- It is the first parallelism visualization tool generated at compile-time. To the best of our knowledge, all other parallelism visualization tools require multiple executions of the program (they are profilers).

- EPV is the first visualization tool to focus on loop-carried dependence information entirely, given that large programs spend most of their execution time in loops.

- Since EPV creates a visualization of LLVM’s Intermediate Representation rather than source code, it extends to every language that can be compiled with LLVM.

- Finally, EPV represents programs in a completely new type of graph which is a merger between an interval tree and a control flow graph.

EPV’s graph representation allows it to keep the most important properties of interval trees and control flow graphs: the structural information derived from the interval analysis, and the popular control-flow information that keeps that graph simple and easy to connect back to the program’s IR. These properties will provide researchers and programmers with insights about program patterns that can both cause and prevent loop-carried dependences.

We hope that extensions to EPV are created in future work because we believe in the power of software visualization tools for the discovery of parallelism in sequential programs. Vision is, after all, the most powerful human sense for understanding large amounts of complex information, so we must take advantage of that to better understand properties of parallel software. This is especially true in a world where society relies on programs to do more for us every year. Below we present some potential future work related to the purposes of this thesis.
7.1 Future Work

- **Callgraph Analysis:** Another source of parallelism lies in the call structure of a program. Figuring out the relationships between functions and function calls as well as each function’s parallel properties can help to further optimize programs. We propose an extension to EPV where a program can also be analyzed as a whole and shown as a function call graph that exposes the potential parallelism to the user.

- **Connecting the graph nodes to the source code:** It would be possible to connect the graph shown in EPV directly back to the source code with debug information. However, this information is not currently implemented in our data dependence pass and other LLVM passes. If it were to be implemented, we could connect a specific loop-carried dependence to a line of source code, adding some more utility value to EPV since not every programmer understands LLVM IR (but most researchers in the field of parallel optimization do).

- **In-depth analysis of loop-carried dependences:** In some cases, the data structure analysis that EPV relies on identifies false loop-carried dependences which result in non-parallel loop nodes in EPV. A more in-depth analysis of the instructions causing loop-carried dependences could improve the accuracy of EPV by reducing the number of false non-parallel loops displayed. An idea is to modify EPV to color-code loops that seem not to be parallel but that are more likely to benefit from parallelism. These loops could be identified by searching for patterns in the loop contents that lead to false dependences.

- **Adding profiling features:** Adding a profiling option for the less conservative detection of parallelism could be added. Although that was not the purpose of this research, some users could have the time to execute the program a number of times if that will provide them with more accurate parallelism discovery. Another feature that could be added by profiling the program is an estimation of the average percentage of execution time spent in each loop. By displaying such information to the user, the most important loops in the program (i.e. the loops that take the longest percentage of the execution time) could be easily identified.

- **Integration of support for directive-based framework:** Already-parallelized programs that use frameworks such as OpenMP, CUDA, OpenCL, etc. could benefit from EPV if support
was added for these libraries. As an example, a program containing directives could be analyzed to identify potential parallelism that is not being exploited by the framework’s directives.

- **Output parallel loops found:** It could be useful to generate a textfile or some data structures containing a list of parallel loops for other tools in a toolchain to potentially parallelize the code automatically. This is a more tangential idea since the purposes of this research are only the visualization for easy parallelism discovery, not automatic parallelization of code.

We hope that Exposed Parallelism Visualization becomes a widely used tool in the development of efficient automatic parallelization algorithms at both compile time and runtime. Furthermore, programmers that choose to use EPV will benefit from easy and fast discovery of loop parallelism. Manual discovery of such parallelism requires expertise in data dependence analysis and is a tedious and time-consuming task. This is why we believe that EPV will represent an important contribution to programmers across the world tasked with parallelizing their own programs or even other people’s programs after quickly discovering parallel loops and using this information to make decisions about what loops to execute in parallel.
REFERENCES


[31] Integer Set Library: Manual. Sven Verdoolaege. 32

APPENDIX A. REFERENCE MANUAL

This appendix serves as a reference manual for users of EPV. Instructions on how to install LLVM or execute passes are outside of the scope of this appendix. Furthermore, the LLVM pass that generates EPV (GraphGenRels) depends on two other passes that are not part of the LLVM package (GenRels and IntervalTreeAnalysis). Our primary focus here will be handling the GUI and its features once it’s been generated.

A.1 Function Selection Window

The function selection window is the first window that the user will see when the LLVM pass GraphGenRels is run. A screenshot of this window is shown in Figure A.1. In order to open any given function, left-click on the function name in the list. This will open the main window, described below. As long as the function selection window is open, the user can keep opening main windows for different functions (or the same function).

![Function Selection Window](image-url)

Figure A.1: Function Selection Window.
A.2 Main Window

A screenshot of the main window for one of our benchmarks is shown in Figure A.2. The main window contains three main panes: the tabbed graph pane, marked by the red rectangle and the number "1" in the figure, the node selection pane, marked by the blue rectangle and number 2, and finally the legend button, marked by the blue rectangle and the number 3. There are a number of user features in each pane of the main window that allow the user to interact with the window. The subsections below explain each of these features along with instructions on how to use them.

A.2.1 Dragging the Graph

This is done by left-clicking anywhere in the graph pane and moving the mouse while holding the left click. This will move the graph following the mouse movement. When the left click is lifted, the dragging action stops.
A.2.2 Zoom

Zooming is done by using the mouse scrollbar while the mouse cursor is on the graph pane. Scrolling up zooms in, while scrolling down zooms out. The graph will zoom around the location of the cursor.

A.2.3 Expanding Loop Nodes

Loop nodes that are not leaf loops (i.e. they have other nested loops inside them) can be expanded by double left-clicking on them. This will open a new graph on a different tab in the graph pane. The node selection list will be updated according to the currently selected tab. Users can switch back and forth between different tabs freely.

A.2.4 Opening the Node Information Window

Users can learn more about nodes in any given graph by right-clicking on any node. This will pop up a node information window where two pieces of information will be shown: On the top half, the basic blocks contained in that interval node will be listed by name. On the bottom half, instructions that cause a loop-carried dependence will be shown if the node clicked is a non-parallel loop. See Figure A.3 for an example.

A.2.5 Selecting a Node in the Node Selection Pane

To center the graph around a specific node, find the node in the node selection pane (labeled 2 in A.2) and left-click it. This will automatically position the node in the center of the graph pane (i.e. the entire graph will be repositioned).

A.2.6 Legend Button and Window

By clicking on the legend button (with a left click), a legend window will be shown. This legend has information about color-coding of loops as well as entry and exit nodes marked in lower graph layers. See Figure A.4.
Figure A.3: Node Information Window.

Figure A.4: Legend Window.