Discovery of Potential Parallelism in Sequential Programs

Entdeckung von potentieller Parallelität in sequentiellen Programmen
Zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation von Zhen Li, M.Sc. aus Tianjin, China
Tag der Einreichung: September 1, 2016, Tag der Prüfung: October 28, 2016
Darmstadt 2016 — D 17

1. Gutachten: Prof. Dr. Felix Wolf
2. Gutachten: Prof. Dr. Philippe Clauss

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Parallele Programmierung

Discovery of Potential Parallelism in Sequential Programs
Entdeckung von potentieller Parallelität in sequentiellen Programmen

Genehmigte Dissertation von Zhen Li, M.Sc. aus Tianjin, China

1. Gutachten: Prof. Dr. Felix Wolf
2. Gutachten: Prof. Dr. Philippe Clauss

Tag der Einreichung: September 1, 2016
Tag der Prüfung: October 28, 2016

Darmstadt 2016 — D 17

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den November 4, 2016

(Zhen Li)

Abstract

In the era of multicore processors, the responsibility for performance gains has been shifted onto software developers. Once improvements of the sequential algorithm have been exhausted, software-managed parallelism is the only option left. However, writing parallel code is still difficult, especially when parallelizing sequential code written by someone else. A key task in this process is the identification of suitable parallelization targets in the source code. Parallelism discovery tools help developers to find such targets automatically. Unfortunately, tools that identify parallelism during compilation are usually conservative due to the lack of runtime information, and tools relying on runtime information primarily suffer from high overhead in terms of both time and memory. This dissertation presents a generic framework for parallelism discovery based on dynamic program analysis, supporting various types of parallelism while incurring practically affordable overhead. The framework contains two main components: an efficient data-dependence profiler and a set of parallelism discovery algorithms based on a language-independent concept called Computational Unit.

The data-dependence profiler serves as the foundation of the parallelism discovery framework. Traditional dependence profiling approaches introduce a tremendous amount of time and memory overhead. To lower the overhead, current methods limit their scope to the subset of the dependence information needed for the analysis they have been created for, sacrificing generality and discouraging reuse. In contrast, the profiler shown in this thesis addresses the problem via signature-based memory management and a lock-free parallel design. It produces detailed dependences not only for sequential but also for multi-threaded code without causing prohibitive overhead, allowing it to serve as a generic base for various program analysis techniques.

Computational Units (CUs) provide a language-independent foundation for parallelism discovery. CUs are computations that follow the read-compute-write pattern. Unlike other concepts, they are not restricted to predefined language constructs. A program is represented as a CU graph, in which vertexes are CUs and edges are data dependences. This allows parallelism to be detected that spreads across multiple language constructs, taking code refactoring into consideration. The parallelism discovery algorithms cover both loop and task parallelism.

Results of our experiments show that 1) the efficient data-dependence profiler has a very competitive average slowdown of around 80× with accuracy higher than 99.6%; 2) the framework discovers parallelism with high accuracy, identifying 92.5% of the parallel loops in NAS benchmarks; 3) when parallelizing well-known open-source software following the outputs of the framework, reasonable speedups are obtained. Finally, use cases beyond parallelism discovery are briefly demonstrated to show the generality of the framework.

Kurzfassung

In Zeiten stagnierender Performanz von Einzelprozessoren obliegt die Leistungssteigerung von Programmen deren Entwicklern. Sind alle Möglichkeiten sequentieller Optimierung erschöpft, ist softwaregesteuerte Parallelität die einzig verbleibende Option. Das Schreiben von parallelem Code stellt jedoch immer noch eine Herausforderung dar, besonders wenn der Autor der sequenziellen Version nicht mehr verfügbar ist. Eine Hauptaufgabe ist deshalb die Erkennung potenzieller Parallelität im Quellcode. Werkzeuge zur Entdeckung potenzieller Parallelität vollziehen diese Suche automatisch. Geschieht dies zur Compilezeit, ist das Ergebnis aufgrund mangelnder Laufzeitinformationen eher konservativ. Hingegen leiden Tools, die auf Laufzeitinformationen basieren, vor allem unter großem Overhead – sowohl hinsichtlich Zeit als auch Speicher. Gestützt auf eine dynamische Programmanalysetechnik, präsentiert diese Dissertation ein allgemeines Framework zur Entdeckung verschiedener Arten potenzieller Parallelität mit geringem Overhead. Das Framework besteht aus zwei Hauptkomponenten: einem effizienten Profiler zur Erfassung von Datenabhängigkeiten sowie einer Menge von Algorithmen zur Entdeckung von Parallelität. Den Algorithmen zugrunde liegt das sprachunabhängige Konzept der Computational Units.

Der Profiler dient als Eckpfeiler des Frameworks. Traditionelle Ansätze zum Profiling von Datenabhängigkeiten verursachen signifikanten Overhead. Um diesen zu senken, konzentrieren sich aktuelle Ansätze unter Vernachlässigung von Allgemeingültigkeit und Wiederverwendbarkeit auf diejenige Teilmenge der Abhängigkeitsinformation, die für die jeweilige Analyse benötigt wird. Im Gegensatz dazu begegnet der in dieser Arbeit vorgestellte Profiler der Herausforderung durch signaturbasierte Speicherverwaltung sowie eine lockfreies paralleles Design. Er produziert sowohl für sequentiellen als auch für Thread-parallelisierten Code detaillierte Abhängigkeiten mit praktisch vertretbarem Overhead. Dadurch kann er als allgemeine Basis für ein breites Spektrum an Programmanalysetechniken eingesetzt werden.

Das Konzept der Computational Units (CUs) schafft ein sprachunabhängiges Fundament zur Entdeckung potenzieller Parallelität. CUs sind elementare Programmschritte, die dem Read-Compute-Write Muster folgen. Im Gegensatz zu alternativen Konzepten sind sie nicht auf vordefinierte Sprachkonstrukte beschränkt. Ein Programm wird durch einen CU-Graphen repräsentiert, in dem die Knoten den CUs und die Kanten den Datenabhängigkeiten entsprechen. Dadurch kann Parallelität unter Berücksichtigung von Code-Refaktorisierung auch über die Grenzen einzelner Sprachkonstrukte hinweg erkannt werden.

Die Ergebnisse unserer Experimente zeigen: 1) Der effiziente Abhängigkeitsprofiler bewirkt im Durchschnitt eine sehr konkurrenzfähige Verlangsamung von etwa einem Faktor 80 mit einer Genauigkeit von mehr als 99,6%. 2) Das Framework erkennt Parallelität in NAS Benchmarks mit hoher Genauigkeit. Es identifiziert 92,5% der parallelen Schleifen. 3) Beim Parallelisieren bekannter Open Source-Software gemäß der Ausgabe des Frameworks werden angemessene Geschwindigkeitsgewinne erzielt. Um schließlich die universelle Verwendbarkeit des Frameworks zu demonstrieren, werden beispielhaft Anwendungen jenseits der Erkennung von Parallelität diskutiert.

Acknowledgment

I would like to thank all the people who contributed in a certain way to the work described in this dissertation. First and foremost I offer my sincerest gratitude to my supervisor Prof. Dr. Felix Wolf, head of the Laboratory for Parallel Programming at Technische Universität Darmstadt. He has been supportive since the day I join LPP with his patience and knowledge while giving me the room to work in my own way. Felix has not only guided me academically, but also encouraged me emotionally through the rough road to finish this thesis. Thanks to him I had the opportunity to work on DiscoPoP and presents this thesis, a milestone in almost five years of work. I have been always feeling comfortable working with Felix, and I simply could not wish for a better supervisor.

In my past few years at LPP, I have been advised and aided intensively by Dr. Ali Jannesari, an experienced and enthusiastic researcher in parallel programming. Ali has offered much advice and insight throughout my work on DiscoPoP. Without his inspiration, I could never come up to the idea of computational units. I also want to thank my second referee, Prof. Dr. Philippe Clauss, for being interested in my work and being my thesis reader.

It was a great pleasure to work with a friendly and cheerful group of colleagues at LPP, and I would like to thank them all for their help. Zia Ul Huda, a sincere friend who shared the same office with me for more than three years, made key contributions to DiscoPoP on parallel pattern detection. Rohit Atre spent much time and effort in studying computational units and static parallelism discovery methods. Arya Mazaheri provided a great application of DiscoPoP, which detects parallel communication patterns in programs running on multicore platforms. Mohammad Norouzi made a lot of suggestions on improving DiscoPoP, and has been working on energy-oriented code optimization based on computational units. Additionally, I want to thank Suraj Prabhakaran, a great colleague and my neighbor in 2015, and Elisabeth Altenberger, for their kind help in work and daily life. I also thank Sebastian Rinke and Alexandru Calotoiu for their collaborative assistance in teaching courses at both RWTH Aachen and TU Darmstadt.

I would also like to thank my master students and student assistants for their coding support. Wolfram Gottschlich implemented the original version of the signature described in this thesis. Tuan Dung Nguyen implemented the lock-free parallel version of the DiscoPoP profiler. Michael Beaumont tested the memory skipping technique. Daniel Fried implemented the method of characterizing DOALL loops using machine learning and DiscoPoP.

I want to say thank you to my wife, Mengyu Zhu, for her constant love and support. Thank you for listening to me when I grumble, comforting me when I feel disappointed, encouraging me when I lose faith, and accompanying me when I am alone. It would not be possible for me to finish writing this thesis without your support. I feel lucky to have you as my significant other. You are not just my companion, you are my inspiration.

Finally, I thank my parents for supporting me throughout all my studies at universities, and for always providing me a cozy place whenever I stay in China. Every time I sit at the old wooden table with you, eating tasty home-made food and watching TV, I regret to not having planned a longer stay at home.

Contents

List of Figures

List of Tables

1 Introduction

Although the component density of microprocessors is still rising according to Moore's law, single-core performance is stagnating for more than ten years now. As a consequence, extra transistors are invested into the replication of cores, resulting in the multi- and many-core architectures popular today. The only way for developers to take advantage of this trend if they want to speed up an individual application is to match the replicated hardware with thread-level parallelism. This, however, is often challenging – especially if the sequential version of the application was written by someone else. Unfortunately, in many organizations this situation is more the rule than the exception. Most software systems are created by modifying earlier source code, and most of the work and cost of software development is after the first release, that is, during evolution [1]. To find an entry point for the parallelization of an organization's application portfolio and lower the barrier to sustainable performance improvement, tools are needed that identify the most promising parallelization targets in the source code. These would not only reduce the required manual effort but also provide a psychological incentive for developers to start and a structure for managers along which they can orchestrate the parallelization work flow.

However, constructing parallelism discovery tools is a great challenge. Parallelism is discovered by analyzing dependences in the target program, which so far cannot be obtained both accurately and efficiently. Methods to discover potential parallelism fall into one of two categories: static and dynamic methods. Being closely related to compiler technology, static approaches analyze source or intermediate code and are restricted to information that can be obtained before running the program. Static approaches are fast, but also conservative because they have limited support for objects allocated or identifiable only at runtime. In contrast, dynamic approaches identify dependences only if they exist at runtime. Although dynamic approaches relax the conservative assumptions made by static approaches on dynamic objects, they are input sensitive, that is, their outcome may depend on the particular execution configurations. A more serious limitation of dynamic approaches is their high runtime overhead in terms of both time and space. So far, the high overhead prevents dynamic approaches from practical use.

This thesis presents a generic framework for parallelism discovery based on dynamic dependence analysis, supporting various types of parallelism while incurring practically affordable overhead. The framework contains two main components: an efficient data-dependence profiler (Chapter 2) and a set of parallelism discovery algorithms (Chapter 4) based on a language-independent concept called computational unit (Chapter 3).

In the remainder of this chapter, we first introduce the reason why parallelism discovery tools are needed. After that, data and control dependences, the main obstacle to parallelism, are introduced. Furthermore, we survey the state-of-the-art parallelism discovery tools, and present a summary of the open problems based on the survey. Through the summary, we define the scope of this thesis. Next, we give a brief introduction of LLVM, a collection of compiler and tool chain technologies on top of which our method is built. In the end, we present an overview of our approach and summarize the contributions of this thesis.

1.1 Parallel Computing

Parallel computing is a type of computation in which many calculations are carried out simultaneously, [2] operating on the principle that large problems can often be divided into smaller ones that are then solved at the same time. Nowadays, parallel computing is the key to improve the performance of computer programs.

Moore's law [3] is the empirical observation that the number of transistors in a microprocessor doubles every 18 to 24 months. The additional transistors are used for many architectural improvements, including multi-level caches, sophisticated instruction unit that supports pipelining, support of simultaneous multi-threading, and so on. However, from the mid-1980s until 2004, the additional transistors are mainly used for ramping up processor frequency (know as *frequency scaling*), which was the dominant reason for improvements in computer performance. Increases in frequency also increase the amount of power used in a processor. On May 8, 2004, Intel canceled its Tejas and Jayhawk processors due to their high power consumption, which is generally cited as the end of frequency scaling as the dominant vehicle for performance improvement.

However, Moore's law is still in effect. With the end of frequency scaling, the additional transistors have been used to add extra hardware for parallel computing, leading to the emergence of multi-core processors. A multi-core processor is a single computing component with two or more independent actual processing units called "cores", which are the units that read and execute program instructions. These multiple cores can run multiple instructions at the same time, increasing the overall speed for programs amenable to parallelism.

Unlike frequency scaling, multi-core is not a technology from which programmers can benefit automatically. Sequential programs still run on a single core of a multi-core processor. So far, compilers that equipped with advanced analyses and optimization techniques are able to transform well-formed sequential loops into equivalent parallel loops. These compilers are very successful in scientific computing area since scientific computing programs usually contains many well-formed loops that perform heavy computations. In terms of general-purposed application that usually contains parallelism beyond loops, there is no compiler that supports fully automatic parallelization, which remains a grand challenge due to its need for complex program analysis and the unknown factors (such as input data range) during compilation. As a result, parallel programs are mainly created by programmers. Compared to sequential programming, parallel programming is much more difficult and error-prone. Depending on the concrete task, it may require deep understanding of the algorithm in use, the guarantees of the programming language in use, parallel programming models, details of the target machine, or even all of them. For this reason, tools that can help in parallelization are urgently needed. Currently, approaches to achieve parallelization can be divided into three categories:

- Annotations from programmers to guide compiler parallelization
- Interactive system between programmers and parallelizing tools/compilers to ease parallelization
- Hardware-supported speculative multi-threading

This thesis presents a novel parallelism discovery framework that works as an interactive system to assist parallelization. We select this category because of the following reasons. Firstly, the most efficient parallel programs are so far still written by hands. Secondly, in many cases, parallelization also means code refactoring. Adding annotations without touching the algorithm structure may not yield the best solution. Last but not least, understanding the code is still necessary for programmers because it is not yet possible for a machine to understand a programmer's intention. We will discuss this point further in Chapter 4.

1.2 Parallelism Discovery

Parallelism is mainly prohibited by dependences, and dependences includes data dependences and control dependences. In this section, we introduce both of them and discuss their roles in parallelism discovery.

1.2.1 Data Dependences

A data dependence is a situation in which a program statement (instruction) refers to the data of a preceding statement. In compiler theory, the technique used to discover data dependences among statements (or instructions) is called dependence analysis.

Assuming statement S_1 and S_2, S_2 depends on S_1 if:

$$[I(S_1) \cap O(S_2)] \cup [O(S_1) \cap I(S_2)] \cup [O(S_1) \cap O(S_2)] \neq \varnothing \tag{1.1}$$

where:

- $I(S_i)$ is the set of memory locations read by S_i
- $O(S_j)$ is the set of memory locations written by S_j
- There is a feasible runtime execution path from S1 to S2

This condition is called Bernstein Condition [4], named by A. J. Bernstein. Let $\rightarrow$ be the precedence relationship in terms of expression evaluation order. According to the Bernstein Condition, there are three cases of data dependences:

- Flow dependence: $O(S_1) \cap I(S_2) \neq \emptyset$, $S_1 \rightarrow S_2$. S_2 reads a memory location after S_1 writes to it. A flow dependence is also called read-after-write (RAW) dependence, or true dependence.
- Anti-dependence: $I(S_1) \cap O(S_2) \neq \emptyset$, $S_1 \rightarrow S_2$. S_1 reads a memory location before S_2 writes it. An anti-dependence is also called write-after-read (WAR) dependence.
- Output dependence: $O(S_1) \cap O(S_2) \neq \emptyset$, $S_1 \rightarrow S_2$. Both S_1 and S_2 write to the same memory location(s). An output dependence is also called write-after-write (WAW) dependence.

Understanding data dependences is fundamental to implementing parallel algorithms. No program can run more quickly than the longest chain of dependent calculations, known as the critical path [5], since calculations that depend upon prior calculations in the chain must be executed in order. However, most algorithms do not consist of just a long chain of dependent calculations; there are usually opportunities to execute independent calculations in parallel.

The Bernstein Condition describes whether two program statements can run in parallel or not. In condition 1.1, S_1 and S_2 can be replaced by P_1 and P_2, representing two program segments. In this case, the Bernstein Condition describes whether two program segments can run in parallel or not. In either case, the conclusion is always the same: parallelism can be explored if there is no data dependence between them.

In the scenario of parallelism discovery, the three kinds of data dependences are not equally important. Usually, flow dependences (RAW) cannot be fully resolved, and that is also why they are also called “true dependences”. In contrast, anti-dependences (WAR) and output dependences (WAW) can usually be resolved by renaming the variable where the program statement writes. Thus, anti-dependences and output dependences are also called *name dependences*.

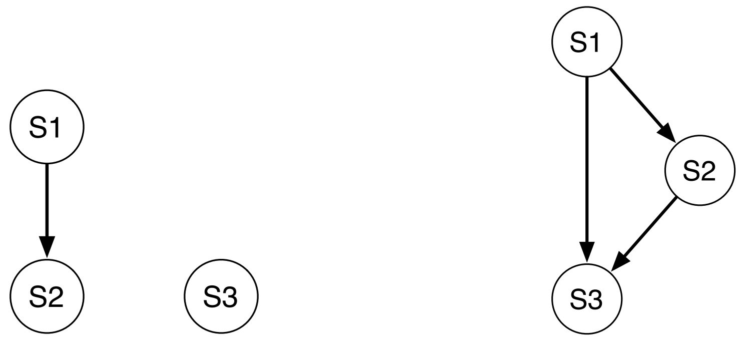
1.2.2 Control Dependences

A program statement (instruction) is control dependent on a preceding statement if the outcome of latter determines whether former should be executed or not. Formally, a statement S_2 is said to be control dependent on another statement S_1 if and only if:

- There exists a path P from S_1 to S_2 such that every statement $S_i \neq S_1$ within P will be followed by S_2 in each possible path to the end of the program (S_2 post-dominates all S_i)
- S_1 will not necessarily be followed by S_2, that is, there is an execution path from S_1 to the end of the program that does not go through S_2 (S_2 does not post-dominate S_1) [6]

For example, consider the following code section:

```
S1. if (a == b)
S2.     a = a + b
S3. b = a + b
```



(a) Control-dependence graph

(b) Control-flow graph

Figure 1.1: Control-dependence graph and control-flow graph of the code snippet shown in section 1.2.2.

Statement S2 is control dependent on statement S1. However, S3 is not control dependent upon S1 because S3 is always executed irrespective of the outcome of S1.

Similar to data dependences, control dependences also produce execution-order constraints between program statements. Unlike data dependences, control dependences can be broken using a technique called *speculative execution*. In the above example, S2 can be executed on another processor speculatively without waiting for the outcome of S1. Later on, if the condition in S1 is evaluated to be true, then the program finishes immediately since S2 is already executed. Otherwise, S3 is executed. In this example, the performance of the program is improved by utilizing one more processor. Speculative execution is usually transparent to programmers. It exists in many schedulers, including both software and hardware implementations.

It is worth mentioning that control dependences are different from *control flow*, which describes an execution path of a program. Again, take the above example, figure 1.1(a) shows its control-dependence graph, and figure 1.1(b) shows its control-flow graph. Usually, compilers produce the control-flow graph, and control dependences can be deduced from it.

Parallelism discovery is challenging mainly because dependences cannot be obtained both accurately and efficiently. Methods to discover potential parallelism fall into one of two categories: static and dynamic methods. Being closely related to compiler technology, static approaches analyze source or intermediate code and are restricted to information that can be obtained before running the program. Static approaches are good at finding the complete control-flow graph and also fast. However, there are several disadvantages associated with them. First, when the program is large and has many branches, the solution search space becomes too big, a problem known as *branch explosion*. Second, they have a limited support of determining dependences among objects allocated or identifiable only at runtime. This is why static approaches are usually considered conservative in their assessment of parallelization opportunities.

In contrast, dynamic methods track dependences at runtime. They treat the execution of a user program as an instruction stream interrupted by previously inserted calls to instrumentation functions that help detect dependences. Dynamic approaches identify control and data dependences only if they really exist at runtime. Since they track only the branches that are actually executed, they do not suffer the branch explosion problem. But the control-flow graph is usually incomplete. In general, dynamic methods are *input sensitive*, that is, their outcome may depend on the particular execution configuration, a disadvantage traded in for not being pessimistic like static methods. A straightforward compromise is selecting a range of representative inputs and repeating the analysis with all of them. A more serious limitation of dynamic approaches is their high runtime overhead in terms of both time and space. The overhead is mainly caused by the underlying dynamic data-dependence analysis that instruments and tracks all the memory accesses of the target program. So far, the extraordinary high overhead prevents dynamic approaches from practical use.

1.3 Parallelism Discovery Tools

As described in Section 1.2.1, data dependences form the critical path of a program, which in turns dictates the upper bound of the application's execution speed. Due to this reason, the fundamental work of parallelism discovery is to look for the absence of data dependences in sequential programs. When the target program contains a considerable number of lines of code (LOC), manually exploring data dependences through the program in program statement level is almost impossible. In this case, automatic methods to detect parallelism and predict parallel performance based on data dependences are necessary. In this section, we cover all the three categories of parallelism discovery tools: data dependence analyzers, semi-automatic parallelism discovery and modeling tools, and automatic parallelization tools.

1.3.1 Data-Dependence Analyzers

Obtaining the data-dependence graph is the first step to discover parallelism. In this section, we introduce tools that reveal data dependences but leave the actual parallelism discovery work to the user.

Usually, tools that can obtain data dependences are capable of extracing more information, such as the control flow graph, the call graph, and use/def – def/use chains. These analyses are usually integrated in the same code analysis framework.

Aristotle Analysis System

The Aristotle Analysis System [7] provides program analysis information and supports the development of software engineering tools using static analyses. It is an open-source tool developed by the Aristotle Research Group from Georgia Institute of Technology.

The Aristotle Analysis System uses a parser to gather control flow, local data flow, and symbol table information for C programs. It processes the data provided by the parser for a variety of tasks, such as data-dependence and control-dependence analysis, graph construction and graph rendering. Parser and tools use database access routines to store information in, and retrieve it from, a data repository. Users can view analysis data textually or graphically. A user interface provides menu-driven access to tools.

Frama-C

Frama-C [8] is a suite of tools dedicated to the analysis of the source code of software written in C. It is an open-source tool developed by teams from two institutions: CEA–LIST (Software Reliability Laboratory) and INRIA-Saclay.

Frama-C contains several static analysis techniques in a single collaborative framework and provides tools such as a code slicer and a dependence analyzer. The main features of Frama-C (including plugins) are:

1. Observe sets of possible values for the variables of the program at each point of the execution
2. Slice the original program into smaller ones with fewer dependences among them
3. Traverse the data-flow graph of the program, from definition to use or from use to definition

DMS® Software Reengineering Toolkit™

The DMS Software Reengineering Toolkit is a set of customizable tools for automatic source program analysis, code generation, and code translation. It is a commercial tool developed by Semantic Designs, Inc., a privately- held corporation headquartered in Austin, Texas. The tool is implemented using PARLANSE [9], an in-house parallel programming language.

DMS works like an extremely generalized compiler. It has a parser, a semantic analyzer, a program transformation engine to do code generation and optimization, and final output formatting components producing source code rather than binary code. However, unlike a conventional compiler, in which each component is specific to its task of translating one source language to one target machine language, each DMS component is highly parameterized, enabling a stunningly wide variety of effects. This means one can change the input language, change the analysis, change the transforms, and change the output in arbitrary ways.

DMS supports control-flow graph construction and data-flow analysis. It covers both use-def chain analysis and def-use chain analysis in the graph. The graph is produced by DMS's C Front End, thus one can infer that the control-flow and data-flow analyses are done statically.

1.3.2 Semi-Automatic Parallelism Discovery Tools

Tools that are in the middle of the spectrum try to locate the potential parallelism in sequential programs rather than showing the data-dependence graph. However, parallelism discovery is a technique much more difficult than dependence analysis. Hence, most of the parallelism discovery tools are still in a prototypical state.

Kremlin

Kremlin [10] is a research parallelism discovery tool developed at University of California, San Diego. It uses the LLVM compiler infrastructure [11] for instrumentation, discovers parallelism based on knowledge of the critical path and supports the programmer in ranking different parallelization opportunities. To address dependences in nested code regions, Kremlin extends the traditional critical-path analysis [12] by making it hierarchical. For this purpose, it introduces a metric called self-parallelism, which quantifies the parallelism of a parent region independent of its children. Kremlin has an OpenMP planner and a Cilk++ planner to suggest parallelism in two different parallel programming models.

Alchemist

Alchemist [13] is a research parallelism discovery tool developed at Purdue University. It is built on top of Valgrind [14], an instrumentation framework for building dynamic analysis tools, to discover parallelism and issue corresponding recommendations. For each region, Alchemist decides whether the region can run asynchronously with its dynamic context by checking the distance between memory references inside and references to the same location that occur during the region's continuation. It thus follows the parallelization strategy underlying the use of futures. A future of a write operation is the code section or construct that contains further reads of the written variable. Alchemist also builds an execution index tree at runtime. This tree is used to differentiate among multiple instances of the same static construct.

Compared to Kremlin and other semi-automatic parallelism discovery tools, Alchemist does not target any specific parallel programming model. This approach is less specific and yet more flexible since it leaves the implementation details to the users.

Parwiz

Parwiz [15] is a parallelism discovery tool developed at INRIA and Université de Strasbourg, France. The main component of Parwiz is a data-dependence profiler that works on binary code. It uses the Intel XED [16] x86 encoder decoder software library to parse the binary, and uses Pin [17] to instrument the program. Data dependences are extracted based on an execution tree. An execution tree is an unfolded static program hierarchy in one execution, including individual instructions, loops and their iterations, and routine calls. Every memory access is an ACCESS node in the execution tree, and the path from the root to an ACCESS node is the global iteration vector of the access. By calculating the least common ancestor of two ACCESS nodes, a data dependence can become apparent on multiple levels of the execution tree. The data dependences together with the execution tree can be used to discover trivial parallel loops, bags of tasks, and code transformations.

Parwiz also includes a few optimizations to lower the overhead of dynamic data-dependence profiling. The idea is to combine contiguous memory accesses that always happen “atomically” into a single block to lower the instrumentation and profiling overhead. To achieve this, Parwiz performs static analysis to find nearby individual accesses when the program updates fields of a structure or accesses in a loop that traverses arrays. These optimizations reduce the profiling time by more than 46%.

Tareador

Tareador [18] is a parallelism discovery tool developed at Barcelona Supercomputing Center, Spain. Tareador analyzes sequential code to find a task decomposition. It provides a set of annotations for marking down tasks in the code. In Tareador 1.x, tasks should be annotated by the user. The tool then generates the data dependence graph according to the annotated tasks, the potential parallel execution of the tasks, and visualization of data usage, that is, the amount of data accessed in each task. Tareador works interactively with the user to find a good task decomposition. The user can refine the task annotation according to the output of Tareador, and determine further improvements based on the new output for the refined tasks.

Tareador 2.x automates the iterative process. It follows a top-down approach by first using the most coarse-grain task annotation, that is, taking the whole main function as a single task, and then iteratively refining the decomposition based on a cost model. The cost model contains three metrics, the length cost, the dependence cost, and the concurrency task. The algorithm breaks tasks that have high costs into smaller tasks. A task with high cost means its instances have long duration, many dependences, and low concurrency. The cost model also evaluates the quality of a decomposition, and the algorithm stops when a newly generated decomposition has a lower quality than the previous solution.

Compared to other semi-automatic parallelism discovery tools, Tareador takes a relatively brute-force approach by enumerating possible decompositions. The cost model serves as heuristics in the searching process. The approach does not take control flow into consideration, which may lead to tasks that are not easy to implement.

Intel® Advisor XE

Intel Advisor XE [19] is a threading prototyping tool for C, C++, C# and Fortran software architects. Intel Advisor XE answers the following questions:

1. Where to parallelize?

   Intel Advisor XE profiles hot spots, that is, code sections (functions and loops, specifically) that consume lots of time to execute, as the parallelization candidates. The criterion is the time spent in a code region, and data dependences are not taken into account. Thus, a profiling in this phase is very close to what `gprof` does. This phase is called “survey”. To accomplish the survey the source code needs to be compiled in release mode.

   The programmer needs to read the code in hot spots, understand its behavior, and insert annotations (provided by Intel Advisor XE) to mark down potential parallelism. Thus, the actual parallel pattern is discovered by the programmer but not the tool. This phase is called “annotation”. To annotate the code the programmer needs to include the corresponding header file that contains the annotations and modify the source code slightly.

2. What is the benefit?

   After the annotation, Intel Advisor XE runs the annotated code and emulates the parallel behavior of the annotated code sections. The output is a report containing:

   - The estimated overall speedup
   - The scalability of the program, from 2 to 512 threads

   This phase is called “suitability test”. To accomplish this phase the source code needs to be compiled again with the annotations in release mode. The suitability test usually has a time overhead of less than a factor of two since it does not profile data, but emulates the behavior based on predefined parameters.

3. Which parallel programming model suits the purpose best?

   The emulation model of Intel Advisor XE contains different sets of parameters for different parallel programming models. On Windows, the supported ones are TBB, Cilk+, OpenMP and Microsoft TPL. A user can compare the suitability test results in different programming models.

   The emulation model covers potential synchronization overheads, too. The user can choose to include or ignore these effects, and the tool can also suggest whether these effects can be ignored or not.

4. What are the potential problems after parallelization?

   Intel Advisor XE can also perform a "correctness check", which is essentially a data-race detection. This check targets potential data sharing problems that can lead to deadlocks or races.

   To accomplish the correctness check the code needs to be compiled again with annotations in debug mode. The correctness check has a huge time overhead. According to the technical documents of the tool, a correctness check may take more than one hour on an annotated region that normally runs in 30 seconds.

Questions that Intel Advisor XE cannot answer:

- How much potential parallelism exits in the sequential code?
- What pattern does the potential parallelism follow?

These two questions can only be answered by profiling data dependence dynamically. Without such information, programmers have to answer these two questions themselves. Moreover, Intel Advisor XE cannot transform the code automatically.

SLX Tool Suite

The SLX Tool Suite [20] is a set of tools for parallel software design automation. The suite contains four tools: SLX Parallelizer, SLX Mapper, SLX Generator, and SLX Explorer. The SLX Tool Suite is developed by Silexica as a commercial product.

The SLX Parallelizer performs C code partitioning by analyzing control and data flow within the original sequential code, exposing parallelism. An additional automatic performance estimation allows a fast and accurate prediction of application hotspots and performance gains through identified parallelism. The SLX Mapper performs a fully automated mapping of software tasks and processes onto given multi-core hardware platforms. It also computes optimized data and communication mappings, exploiting memory hierarchies, complex on-chip interconnect fabrics and other memory subsystems, including direct hardware support for FIFO buffers. The SLX Generator follows a source-to-source translation approach that emits architecture-aware and middleware-specific C code as final output. The SLX Explorer facilitates multi-core platform selection by means of a flexible retargetable hardware architecture model.

The technologies behind the SLX Tool Suite originates from the MPSoC Application Programming Studio (MAPS) [21] developed by The Institute for Communication Technologies and Embedded Systems (ICE) of RWTH Aachen University, Germany. MAPS identifies task parallelism in C applications for MPSoC platforms based on the notion of a *coupled block*, which is a single-entry single-exit group of statements tightly coupled by dependences. In the end, a coupled block is treated as a task.

Prism

The Prism Technology Platform [22] is an framework to develop new analysis, visualization or runtime techniques to meet the needs of a particular project. It is an in-house framework developed by Critical Blue, a company based in the UK. Regarding parallelization, the Prism Technology Platform provides the following basic analyses for sequential programs:

- Data-dependence analysis for multicore
- Parallelism 'what-if' modeling
- Multicore scalability modeling

- Dynamic and static code translation
- Multicore data race detection

Prism is built on dynamic binary-level analysis which supports the dynamic instrumentation of compiled software running on most hardware platforms. This allows the capture of performance data down to the detail of individual instruction execution. However, since Prism is an in-house tool and not sold as a product, it is not clear how Prism perform the analyses. For example, we do not know whether the binary-level analysis checks the machine instructions statically or instruments the code. According to the official website of Prism, it models the parallel behavior of sequential programs with potential parallelism marked down by users (parallelism 'what-if' modeling) rather than suggesting the parallelism. This is similar to Intel Advisor XE.

Others

Besides the more well-known ones, there are many semi-automatic parallelism discovery tools that are tailored to specific types of programs or programming languages. JavaSlicer [23] traces Java programs to find parallelism, exploiting knowledge of the critical path. ParaMeter [24] is a tool aiming to find parallelism in task-based applications where computational tasks are added dynamically. It employs a speculative scheduler to decide whether two tasks can be executed concurrently. The tool developed by Tournavitis and Franke [25] and the tool developed by Thies et al. [26] target coarse-grained pipeline-style parallelism in multimedia applications.

1.3.3 Automatic Parallelization Tools

Parallelization assisting tools that fall into the third category aim to automatically convert the sequential code into parallel code. Such tools are known as automatic parallelization tools. Automatic parallelization tools further fall into two categories: compile-time tools and runtime tools. Compile-time tools work like compilers or plugins of a compiler. They perform compile-time analysis to identify code fragments that can run in parallel, and transform the sequential code into equivalent parallel code in either a source-to-source or source-to-binary way. Runtime tools, on the other hand, executes fragments of code that were originally intended to run sequentially in parallel by simply assuming the fragments can be executed in parallel. To ensure correctness, runtime tools check violation of dependences on-the-fly, and fall back to sequential execution when necessary. This approach is also called *speculative parallelization*. Due to the difficulty of automatic task decomposition and code transformation, automatic parallelization tools usually focus on loop parallelism. This applies to both compile-time and runtime tools.

Speculative parallelization has hardware and software solutions. However, many ideas and concepts can be implemented in either way. For this reason, we focus on representative software solutions in this thesis to avoid introducing essential hardware backgrounds, which is less relevant to the other parts of this thesis. A complete survey on speculative parallelization can be found in the work of Estebanez et al. [27].

Intel® C++ Compiler

One well-known compile-time automatic parallelization tool is the Intel C++ Compiler [28], also known as *icc* or *icl*. The Intel C++ Compiler is a group of C and C++ compilers. Compared to other C and C++ compilers, the Intel C++ Compiler specializes in generating optimized code for Intel processors, including processors based on IA-32 and Intel 64 architectures, ATOM processors, and the Intel Xeon Phi coprocessor.

The Intel C++ Compiler supports automatic parallelization of loops. This feature can be enabled by specifying the `-Qparallel` (Windows) or `-parallel` (Linux or Mac OS X) option on the command line. When automatic parallelization is enabled, The Intel C++ compiler performs data-dependence analysis on loops, and generates parallel code that divides the iterations as evenly as possible for loops that are recognized as good candidates. A loop can be parallelized by The Intel C++ Compiler only if it satisfies three requirements [29]. First, the number of iterations must be known in compile time. That means a `while` loop usually cannot be parallelized. Second, there are no control flow jumps into or out of the loop. That means a single `break` statement is usually enough to prevent parallelization. Finally, there must be no data dependences among iterations. However, data dependences due to trivial reductions scenarios such as adding the elements in an array can be resolved automatically.

A technical report [29] from Intel states that “Potential aliasing of pointers or array references is a common impediment to safe parallelization”, and “If the compiler cannot prove that pointers or array references are safe and that iterations are independent, it will not parallelize the loop”. These statements give a clear clue that the Intel C++ compiler utilizes static data-dependence analysis for parallelism discovery. Details of static data-dependence analysis and pointer aliasing analysis are described in Section 2.1. Moreover, the Intel C++ Compiler cannot determine the thread-safety of a loop containing external function calls because it does not know whether the function call has side effects that introduce dependences [29].

Polly

Polly [30] is a high-level loop and data-locality optimizer and optimization infrastructure for LLVM. It is an open-source tool originally developed by Tobias Grosser and Hongbin Zheng. Now the source code of Polly is integrated into the LLVM official repository and released according to the same schedule as LLVM.

Polly uses an abstract mathematical representation based on integer polyhedra to analyze and optimize the memory access pattern of a program. Polly mainly focuses on classical loop transformations, especially loop tiling and loop fusion to improve data locality. It can also exploit OpenMP level parallelism and expose SIMDization opportunities.

Polly adopts a static technique called *polyhedral compilation*. It uses a high-level mathematical representation based on polyhedra [31] or Presburger relations [32] to analyze and optimize computer programs. The polyhedral model can be used to obtain data dependences statically. Compared to classic static data-dependence analysis (introduced in Section 2.1), polyhedral model is much more powerful in extracting data dependences from regular accesses to arrays, which is one of the most common memory access patterns in scientific numerical programs.

Results of testing Polly on Polybench 2.0, a test suite that contains computation kernels from linear algebra routines, stencil computations, image processing and data mining, are also published. The results show that 16 programs out of 30 get major speedup, where 8 programs have speedups bigger than 10 using 24 threads.

Polly is not the only polyhedral optimizer though, and polyhedral optimization is still an active area of research. GCC has a polyhedral optimization framework called Graphite [33], and there are many other polyhedral optimization frameworks such as Omega [34], PolyLib [35], and CLooG [36]. Classic polyhedral optimization requires that the loop bounds and conditions of loop statements are affine functions of the surrounding loop iterators and parameters. Benabderrahmane et al. [37] proposed a method that extends the polyhedral model to support `while` loops, in which loop bounds are non-affine. The method transforms `while` loops into `for` loops with `if` branches to process the loop conditions.

LRPD test

The LRPD test [38] is the origin of software thread-level speculative parallelization. The method executes chunks of iterations of the target loop speculatively, and perform the LRPD test in the end to validate the execution. If the test failed, the target loop is re-executed sequentially.

To lower the possibility that the LRPD test fails, the method firstly transform the target loop through privatization and reduction parallelization. Privatization is to making private copies of shared variables. Reduction parallelization is to identify reduction operations at compile time and replacing the reduction operations with a parallel algorithm. The method then assumes the transformed loop has no inter-iteration dependences.

Apollo

Apollo [39, 40, 41] is a compiler framework dedicated to automatic, dynamic and speculative parallelization and optimization of programs' loop nests. It is developed at Inria and the University of Strasbourg, France. Apollo is a modern runtime tool that supports speculative parallelization.

Apollo consists of two main components. The first component is a set of extensions to the CLANG-LLVM compiler that prepare the program. Specifically, the first component generate two other versions of the program along with the original sequential code: 1) an instrumented version in which memory instructions and updates of scalar values are instrumented, and 2) code bones [42], which are essentially templates of code fragments that can be instantiated at runtime. The second component is a runtime system. The runtime system firstly executes the instrumented version for a small number of iterations to collect memory access information and scalar values. Then it builds a predication model to select an optimal polyhedral transformation for each target loop nest. Once the optimal transformation is decided, it instantiate the code bones to generate the parallel code. The generated parallel code is speculative. In case of invalidation, the original sequential version that contains the chunk of iterations are re-executed, and instrumented version is relaunched to determine a new parallelization strategy.

ParallWare

ParallWare [43] is an auto-parallelizing source-to-source compiler for sequential applications. It is a commercial tool developed by Appentra.

ParallWare automatically discovers the parallelism available in the input sequential code, and generates equivalent parallel source code annotated with compiler directives. The targets are HPC systems based on multi-core processors. ParallWare supports OpenMP and OpenACC. The technical features of ParallWare are:

- Auto-parallelization of convergence loops and propagation loops in scientific numerical applications
- Auto-parallelization of parallel reductions
- Auto-parallelization of for loop nests
- Auto-parallelization of source codes with n-dimensional arrays
- Auto-parallelization of inter-procedural code (e.g., intrinsic and non-intrinsic functions)

Par4All

Par4All [44] is an automatic parallelizing and optimizing compiler that supports programs written in C and FORTRAN. It is an open-source tool maintained by the community, which is mainly supported (technically) by three organizations: SILKAN, MINES ParisTech and Institut TÉLÉCOM/TÉLÉCOM Bretagne/HPCAS.

Par4All is based on the PIPS (Parallelization Infrastructure for Parallel Systems) source-to-source compiler framework. The "p4a" is the basic script interface to produce parallel code from user sources. It takes C or FORTRAN source files and generates OpenMP or CUDA output to run on shared memory multicore processors or GPUs, respectively.

As a compiler, Par4All concentrates on static analyses and mainly transforms loops. Par4All covers many code-generation optimizations, including loop fusion, point-to analysis, and vectorization. Features like automatic instrumentation for loop parameter extraction at runtime are expected in version 2.0.

Par4All offers official evaluation results. On the program *410.bwaves* from the SPEC CPU2006 benchmarks, Par4All achieves a speedup of 4.5 with two Intel Xeon X5670 processors at 2.93GHz (12 cores). On an ordinary matrix multiplication program, it achieves a speedup of 12.1 on the same platform.

PLUTO

PLUTO [45, 46] is an automatic parallelizer and locality optimizer for multi-core programs. It is an open-source tool developed by a team from Ohio State University and Louisiana State University.

Based on the polyhedral model [31], PLUTO transforms sequential C programs to equivalent parallel code. It focuses on coarse-grained parallelism, dealing with big code sections such as entire loop nests. The core transformation framework mainly works by finding affine transformations for efficient loop tiling and loop fusion. The generated OpenMP programs can achieve outer, inner, or pipelined parallelization of loop nests purely with OpenMP directives. PLUTO also has a version generating CUDA code, but it is no longer maintained.

Cetus

Cetus [47] is a compiler infrastructure for the source-to-source transformation of software programs. It is an open-source tool developed by a team from Purdue University.

Cetus is not a dedicated automatic parallelization tool itself, but provides a basic infrastructure for writing such tools or compilers. The basic parallelizing techniques Cetus currently implements are privatization, reduction-variable recognition, and induction variable substitution. The latest version of Cetus also includes a GUI, speedup calculation and graph display, and the Cetus remote server. The Cetus remote server allows users to transform C code through the server. Cetus also has an experimental Hubzero [48] version that allows users to transform C code through a web browser.

Others

Other than the tools mentioned above, Tournavitis et al. [49] applies machine learning techniques to find parallelism in loops and automatically parallelize them using OpenMP. Sambamba [50] integrates three parallelism enabling technologies into one framework: speculation, privatization, and reduction. Instead of performing the classic static analysis on the program dependence graph (PDG), Sambamba solves the problem using integer linear programming (ILP).

1.3.4 Review of the Existing Tools and the Scope of This Thesis

We have introduced the state-of-the-art parallelism discovery tools in the previous sections. Based on the available information from the published documents and papers, we summarize the existing parallelism discovery tools as follows:

1. There is still no mature parallelism discovery tool that works for general-purpose programs with irregular access patterns. Companies that are strong in building program analysis tools sell "parallelism discovery for general-purpose programs" as a service since strong intervention from domain experts and experienced programmers are needed to build such tools. Moreover, automatic code transformation for general-purpose programs is considered to be far beyond the state-of-the-art.
2. On the other hand, parallelism discovery and automatic parallelization for loop-structured programs have been fully automatic. They are either based on the polyhedral model or speculative execution. Although the underlying theory is mature, the techniques build on top of it are still considered advanced. Polyhedral model is limited by its restriction to affine loop nests. Speculative execution can extract parallelism from irregular loops, but it introduces a higher runtime overhead and energy consumption. Furthermore, many tools such as ParallWare and Par4All are still under construction for supporting many core processors (GPUs or accelerators). Building a parallel compiler or front-end is the most popular approach to building automatic parallelization tools.
3. So far, static analysis has been the most widely adopted approach in state-of-the-art tools. The reason is simple: the overhead of the analyses must be low enough to make the tool practical. Among all the tools introduced in this thesis, only Intel Advisor XE and Cetus use profiled runtime data and show the users about the potential time overhead of the profiling. Consequently, there is still no mature technique to lower the overhead of dynamic analyses to a reasonable level.

To narrow the technical gap, this thesis focuses on

- A dynamic data-dependence analysis that has low overhead in terms of both time and memory
- A parallelism discovery approach for general-purpose programs

We do not cover automatic sequential-to-parallel code transformation in this thesis. For scientific numerical programs, several automatic parallelization tools exist. For general-purpose programs, information obtained through program analyses is usually not enough for automatic sequential-to-parallel code transformation. The semantics of the program or the programmer's intentions must be preserved. A detailed discussion is deferred to Section 4.5.

1.4 Introduction to LLVM

The methods presented in this thesis are built on top of the Low Level Virtual Machine (LLVM), which is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines, though it does provide helpful libraries that can be used to build them. The name "LLVM" itself is not an acronym; it is the full name of the project. [51]

LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Nowadays, the LLVM project has grown and holds a huge collection of compiler-related tools. Depending on the context, the name "LLVM" might refer to any of the following:

- **The LLVM project**. This is an umbrella for several projects that together form a complete compiler: frontends, backends, optimizers, assemblers, linkers, and so on.
- **An LLVM-based compiler**. This is a compiler built partially or completely with the LLVM infrastructure. For example, a compiler might use LLVM for the frontend and backend but use GCC and GNU system libraries to perform the final link.
- **LLVM libraries**. This is the reusable code portion of the LLVM infrastructure.
- **The LLVM IR**.This is the LLVM compiler intermediate representation. [52]

In this thesis, the term "LLVM" mostly refer to the LLVM libraries, and the LLVM IR. The meaning should be clear in a given context. Since our approach is built on top of LLVM, it is necessary to introduce its primary components, or subprojects, that are used in this thesis:

- **LLVM core**. The libraries that provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs.
- **Clang**. Clang is an "LLVM native" C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles as well as extremely useful error and warning messages and to provide a platform for building source level tools.
- **Compiler-rt**. The compiler-rt project provides highly tuned implementations of the low-level code generator support routines such as "`__fixunsdfdi`" and other calls generated when a target does not have a short sequence of native instructions to implement a core IR operation. [52] It also provides implementations of runtime libraries for dynamic analysis tools.

The work presented in this thesis are highly related to the LLVM Intermediate Representation and the LLVM Pass Framework. In the following sections, we introduces them in detail.

1.4.1 The LLVM Intermediate Representation

The LLVM Intermediate Representation (IR) is the backbone that connects frontends and backends, allowing LLVM to parse multiple source languages and generate code for multiple targets. Frontends produce the IR, while backends consume it. The IR is also the point where the majority of LLVM target-independent optimizations take place. [52]

The LLVM project started with an IR that operated at a lower level than Java bytecode, thus, the initial acronym was Low Level Virtual Machine. The idea was to explore low-level optimization opportunities and employ link-time optimizations. The link-time optimizations were made possible by writing the IR to disk, just like bytecode. Nowadays, LLVM is neither a Java competitor nor a virtual machine, and it has other intermediate representations to achieve efficiency. In LLVM terms, LLVM IR has two forms: assembly and bitcode. LLVM assembly means the human-readable code, and LLVM bitcode means the binary format. They are equivalent in terms of functionality.

In general, LLVM IR has the following properties:

- **Static Single Assignment (SSA) form**. In the SSA form, names correspond uniquely to specific definition points in the code and each name is defined by one operation, hence the name static single assignment. To reconcile this single-assignment naming discipline with the effects of control flow, the SSA form inserts special operations called ϕ-functions at points where control-flow paths meet. [6]
- **Three-address code**. In the three-address code, most operations have the form `i ← j op k`, where `op` is the operator, `j` and `k` are the operands, and `i` is the result.
- **Infinite number of registers**. Note that local values in the LLVM IR can be any name that starts with the `%` symbol, and there is no restriction on the maximum number of distinct values.

The content of an entire LLVM file, either assembly or bitcode, is said to define an LLVM *module*. The module is the LLVM IR top-level data structure. Each module contains a sequence of functions, which contain a sequence of basic blocks, which contain a sequence of instructions. The module also contains peripheral entities to support this model, such as global variables, the target data layout, and external function prototypes as well as data structure declarations.

Figure 1.2 shows an LLVM IR assembly file, or, a module. The `target datalayout` construct contains information about endianness and type sizes for the target described in `target triple`. In the example shown in Figure 1.2, the target is an `x86_64` processor PC with an a Linux operating system. It is a little-endian target, which is denoted by the first letter in the layout (a lowercase e). Big-endian targets need to use an uppercase E.

The definition of function `foo` in this example is :

```
define i32 @_Z3foov() #2 {...}
```

```
; ModuleID = 'test.cpp'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

...

; Function Attrs: nounwind uwtable
define i32 @_Z3foov() #2 {
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 0, i32* %a, align 4
  store i32 1, i32* %b, align 4
  store i32 0, i32* %c, align 4
  %1 = load i32* %a, align 4
  %2 = load i32* %b, align 4
  %3 = add nsw i32 %1, %2
  store i32 %3, i32* %c, align 4
  %4 = load i32* %c, align 4
  ret i32 %4
}

; Function Attrs: nounwind uwtable
define i32 @main(i32 %argc, i8** %argv) #2 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i8**, align 8
  store i32 0, i32* %1
  store i32 %argc, i32* %2, align 4
  store i8** %argv, i8*** %3, align 8
  %4 = call i32 @_Z3foov()
  ret i32 0
}

attributes #0 = ...
```

Figure 1.2: The content of an LLVM assembly file.

The name of the function has a prefix and a suffix to `foo` because of *mangling*. Name mangling is a technique used to solve various problems caused by the need to resolve conflicting names for programming entities. This function returns a value of the type `i32` and has no parameters. In LLVM IR, local identifiers always need the `%` prefix, whereas global identifiers use `@`. The `#2` tag in the function declaration maps to a set of function attributes.

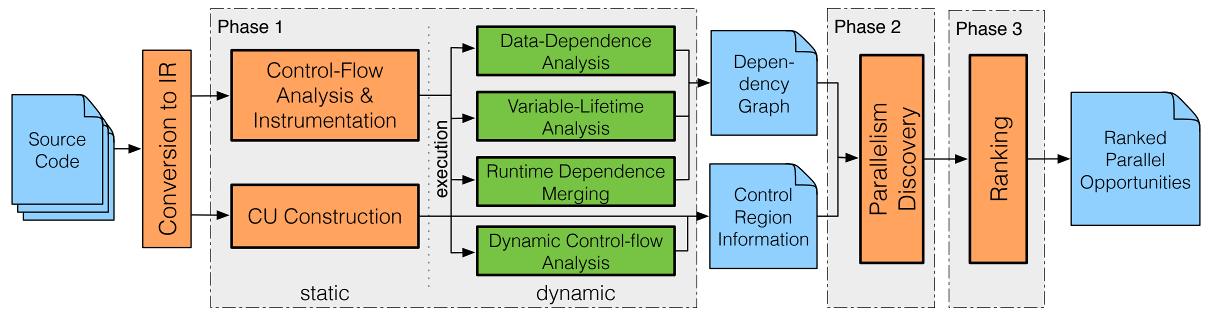
The `alloca` instruction reserves space on the stack frame of the current function. `store` instructions store values to variables, and `load` instructions load values from variables. The `add` instruction adds two operands and puts the result in the third operand. The `nsw` flag specifies that this add operation has "no signed wrap", which indicates an instruction that are known to have no overflow, allowing for some optimizations. The `call` instruction appears in the main function calls another function.

1.4.2 The LLVM Pass Framework

In LLVM, passes perform the transformations and optimizations that make up the compiler, they build the analysis results that are used by these transformations, and they are, above all, a structuring technique for compiler code. All LLVM passes are subclasses of the `Pass` class, which implements functionality by overriding virtual methods inherited from `Pass`.

There are several kinds of passes in LLVM: [53]

- **ModulePass**. The ModulePass is the most general of all superclasses. Deriving from ModulePass indicates that a pass uses the entire program as a unit, referring to function bodies in no predictable order, or adding and removing functions. Because nothing is known about the behavior of ModulePasses, no optimizations can be done for their execution.
- **CallGraphSCCPass**. The CallGraphSCCPass is used by passes that need to traverse the call graph of a program bottom-up (callees before callers). Deriving from CallGraphSCCPass provides some mechanics for building and traversing the call graph, but also allows the system to optimize executions of CallGraphSCCPasses.
- **FunctionPass**. A FunctionPass executes on each function in the program independently of all other functions in the program. FunctionPasses do not require that they are executed in a particular order, and FunctionPasses do not modify external functions.
- **LoopPass**. LoopPasses are similar to FunctionPasses but execute on each loops in the program. LoopPasses process loops in loop-nest order such that the outer most loop is processed last.
- **RegionPass**. RegionPasses are similar to LoopPasses, but execute on each single-entry single-exit region in the function. RegionPasses also process nested regions inside out.
- **BasicBlockPass**. BasicBlockPasses are just like FunctionPasses, except that they must limit their scope of inspection and modification to a single basic block at a time.



Source Code
Conversion to IR
Phase 1
Control-Flow Analysis & Instrumentation
CU Construction
static
execution
Data-Dependence Analysis
Variable-Lifetime Analysis
Runtime Dependence Merging
Dynamic Control-flow Analysis
dynamic
Depen-dency Graph
Control Region Information
Phase 2
Parallelism Discovery
Phase 3
Ranking
Ranked Parallel Opportunities

Figure 1.3: Parallelism discovery workflow.

- **MachineFunctionPass**. A MachineFunctionPass is a part of the LLVM code generator that executes on the machine-dependent representation of each LLVM function in the program.

Passes are managed and scheduled by a *Pass Manager* in LLVM. It takes a list of passes, ensures their prerequisites are set up correctly, and then schedules passes to run efficiently. All of the LLVM tools that run passes use the pass manager for execution of these passes.

1.5 Approach Overview

We name our parallelism discovery framework *DiscoPoP* (Discovery of Potential Parallelism). Figure 1.3 shows our parallelism-discovery workflow. It is divided into three phases: In the first phase, we instrument the target program and execute it. Control flow information and data dependences are obtained in this phase. In the second phase, we search for potential parallelism based on the information produced during the first phase. The output is a list of parallelization opportunities, consisting of several code sections that may run in parallel. Finally, we rank these opportunities and write the result to a file.

1.5.1 Phase 1: Control-Flow Analysis and Data-Dependence Profiling

The first phase includes both static and dynamic analyses. The static part includes:

- Instrumentation. DiscoPoP instruments every memory access, control region, and function in the target program after it has been converted into intermediate representation (IR) using LLVM [11].
- Static control-flow analysis, which determines the boundaries of control regions (loop, if-else, switch-case, etc.).

The instrumented code is then linked to libDiscoPoP, which implements the instrumentation functions, and executed. The dynamic part of this phase then includes:

- Dynamic control-flow analysis. Runtime control information such as entry and exit points of functions and the number of iterations of loops are obtained dynamically.

- Data-dependence profiling. DiscoPoP profiles data dependences using a signature algorithm.
- Variable lifetime analysis. DiscoPoP monitors the lifetime of variables to improve the accuracy of data-dependence detection.
- Data dependence merging. An optimization to decrease the memory overhead.

Note that we instrument the intermediate representation, which is obtained from the source code of the application. That means libraries used in the application can only be instrumented when the source code of the libraries is available. We believe that this approach is sufficient for discovering parallelism since it is nearly impossible to parallelize binary code manually. Besides, the user always has the option of not instrumenting libraries.

1.5.2 Phase 2: Parallelism Discovery

During the second phase, we search for potential parallelism based on the output of the first phase, which is essentially a graph of dependences between source lines. This graph is then transformed into another graph, whose nodes are parts of the code with all parallelism preventing read-after-write (RAW) dependencies explicitly among them. We call these nodes *computational units* (CUs). Based on this CU graph, we can detect potential parallelism and already identify tasks that can run in parallel.

1.5.3 Phase 3: Ranking

Ranking parallelization opportunities of the target program helps users to focus on the most promising ones. For this purpose, we use three metrics: *instruction coverage*, *local speedup*, and *CU imbalance*. Details of the ranking method are introduced in Section 4.3.

1.6 Contributions of This Thesis

This thesis presents the following contributions:

1. An efficient data-dependence profiler [54]. The profiler supports both sequential and parallel programs, and produces detailed dependences that can be used by various program analyses. Thanks to its lock-free parallel implementation, the profiler has an average slowdown of 78× on NAS benchmarks with only 649 MB memory consumption on average. The profiler also includes an optimization that reduces the profiling time on loops by up to 52 % [55].
2. The concept and implementation of computational units [56, 57], which allows parallelism to be discovered among code sections that are not necessarily aligned with source language structures.

3. Parallelism discovery algorithms based on CU graphs for parallelism in loops and parallel tasks [56, 57, 58, 59]. The results show that our method identified 92.5% of the parallelized loops in NAS benchmarks [58], and made correct parallelization decisions on all the 20 hot spots from the Barcelona OpenMP Task Suite [56]. Parallelizing applications manually following the output of DiscoPoP also yields promising speedups. Our parallel version of a face recognition program, FaceDetection, which follows the task graph produced by DiscoPoP results in a speedup of 9.92 when using 32 threads [59].

4. Applications of the parallelization framework presented in this thesis, including characterizing features for DOALL loops (with Daniel Fried) [60], and determining optimal parameters for software transactional memory (with Yang Xiao) [61]. The profiler also enables other applications, one from which is detecting communication patterns on multi-core systems by our colleague Arya Mazaheri.

2 Data-Dependence Analysis

Data-dependence analysis serves as the foundation of many program analysis techniques. Tools for discovering parallelism [10, 57, 13, 62, 15, 63] analyzes data dependences to identify the most promising parallelization opportunities. Runtime scheduling frameworks [64, 65, 66, 67] analyzes data dependences to add more parallelism to programs by dispatching code sections in a more effective way. Automatic parallelization tools [47, 44, 30] analyzes data dependences to transform sequential into parallel code automatically. Software erosion protection tools [68] analyzes data dependences to provide aid in the refurbishment and maintenance of software systems by supporting software understanding and reverse engineering. Common to all of them is that they rely on data-dependence information to achieve their goals.

Data-dependence analysis is not a trivial task. Existing state-of-the-art can be divided into two categories: static and dynamic. While static approaches make conservative assumptions on dynamically allocated memories, dynamic approaches suffer from high time and space overhead. In this chapter, we first review methods from both categories, and then present our dynamic data-dependence profiler in detail. Compared to existing approaches, our data-dependence profiler produces detailed data dependences for both sequential and parallel programs with low time and memory consumption. At the end of this chapter, we show experimental results of our profiler.

2.1 Static Approaches

Static approaches determine data dependences without executing the program. The simplest approach is syntax-driven data-dependence analysis. In syntax-driven approaches, names that are written to and read from are determined by rules based on the syntax of program statements. For example, Bobbie [69] suggests the following pair of rules:

- Singular or composite (valued) variables appearing on the left-hand side of assignment statements are included in the write set
- Singular or composite (valued) variables appearing on the right-hand side of assignment statements are included in the read set

Data dependences are determined based on the write set and the read set. Syntax-driven data-dependence analysis is simple and fast, but language-dependent. For example, the rules stated above need to be changed if the language supports operators like +=. Basically, a set of rules that is designed for a specific language does not work for other languages.

More importantly, syntax-driven data-dependence analysis has a major disadvantage in languages that allow pointers and / or references: it does not distinguish names that point or refer to the same memory location. When analyzing programs written in such languages, the resulting data dependences are far from accurate. Unfortunately, almost all of the popular advanced programming languages today support pointers or references, or even both.

Modern static data-dependence analysis techniques (and many other static analyses and optimizations) are based on *pointer analysis*, or *points-to analysis*, which is still an active research topic today. It answers the following question:

Which memory locations can a pointer expression refer to?

In some context, pointer analysis has another representation called *alias analysis*, which answers the following question:

When do two pointer expressions refer to the same memory location?

In this thesis, we uniformly use the term "pointer analysis". Essentially, pointer analysis finds the names that are equivalent in a program based on the possible memory locations a name can refer to. To understand this, take the following code snippet as an example:

```
int x;
p = &x;  // x and *p alias
q = p;   // *p and *q alias
```

In this code, `x` and `*p` alias, as do `*p` and `*q`, and `x` and `*q`. Thus, `x`, `*p`, and `*q` form an equivalence class because they all refer to the same memory location.

Unfortunately, a complete and precise pointer analysis that is inter-procedural, supporting multi-level pointers and structures is NP-hard. [70] So far, existing pointer analysis methods vary in the following dimensions:

- **Inter-procedural / intra-procedural**: Does the method work at module level (inter-procedural) or function level (intra-procedural)?
- **Flow-sensitive / flow-insensitive**: Does the method compute at each program point (flow-sensitive) or any time (flow-insensitive) of execution?
- **Context-sensitive / context-insensitive**: Are the results affected by the different arguments provided at different call sites?
- **Definiteness**: Does the method guarantee definiteness of the results ("must alias") or not ("May alias")?
- **Heap modeling**: How is dynamically allocated memory represented?
- **Representation**: How are alias relationships represented?

One algorithm that has a critical impact on recent pointer analysis methods is Andersen's algorithm [71], the algorithm described in 1. It divides the program assignment statements into four types, and each type specifies a *subset constraint*. The points-to relationships are constructed according to the constraints and are propagated through the whole program. Since constraints are propagated through the whole program, Andersen's algorithm is inter-procedural and flow insensitive. The constraints in Andersen's algorithm are summarized in Table 2.1.

Table 2.1: Subset constraints in Andersen's algorithm.

Constraint type	Assignment	Constraint	Meaning	Edge
base	`a = &b`	$a \supseteq \{b\}$	$loc(b) \in pts(a)$	no edge
simple	`a = b`	$a \supseteq b$	$pts(a) \supseteq pts(b)$	$b \to a$
complex	`a = *b`	$a \supseteq *b$	$\forall v \in pts(b), pts(a) \supseteq pts(v)$	no edge
complex	`*a = b`	$*a \supseteq b$	$\forall v \in pts(a), pts(v) \supseteq pts(b)$	no edge

```
Initialize a graph $G$ where each vertex is a name in the program. A points-to set (pts) is
  attached to each vertex. $G$ is initialized using base and simple constraints.

Let $W = \{v|pts(v) \neq \varnothing\}$ (all vertices with non-empty points-to sets):

while $W$ not empty do
    $v \leftarrow$ select from $W$
    for each $a \in pts(v)$ do
        for each constraint $p \supseteq *v$ do
            add edge $a \to p$, and add $a$ to $W$ if edge is new
        end
        for each constraint $*v \supseteq q$ do
            add edge $q \to a$, and add $q$ to $W$ if edge is new
        end
    end
    for each edge $v \to q$ do
        $pts(q) = pts(q) \cup pts(v)$, and add $q$ to $W$ if $pts(q)$ changed
    end
end
```

Algorithm 1: Andersen's algorithm for pointer analysis.

$loc(b)$ represents the memory location referred through name `b`, and $pts(a)$ represents the set of memory locations that name `a` possibly points to.

The time complexity of Andersen's algorithm is $O(n^3)$, where n is the number of vertices in the graph, that is, the number of names in a program. Although Andersen's algorithm trades accuracy for speed by not considering control flows, it is still too slow for practical use. Recent pointer analysis methods focus on reducing the time overhead of Andersen's algorithm. Hardekopf and Lin [72] optimized Andersen's algorithm by eliminating circles in the graph first so the algorithm terminates earlier in the last loop. However, it does not reduce the time complexity. Bjarne Steensgaard [73] proposed a similar algorithm that works in almost linear time. The algorithm uses equality constraints instead of subset constraints, further reducing the accuracy.

There are many other pointer-analysis methods for different programming languages, and this thesis cannot cover them all. However, modern methods share the idea of Andersen's algorithm and Steensgaard's algorithm, and are tuned to be either accurate or efficient.

Through the discussion of pointer analysis, we can see that it improves static data-dependence analysis by providing more accurate assumptions on dynamically allocated memory. However, fast pointer analysis is not perfectly accurate. As a result, static data-dependence profiling is still conservative when dealing with dynamically allocated memory, pointers, and dynamically calculated array indices. Nevertheless, static data-dependence analysis enables many advanced program analyses and optimizations, including automatic parallelization in some restricted cases [44, 30]. Nowadays, pointer analysis and static-data dependence analysis are the key in optimizing compilers.

2.2 Dynamic Approaches

After purely static data-dependence analysis turned out to be too conservative in many cases, a range of predominantly dynamic approaches emerged. Dynamic dependence profiling captures only those dependences that actually occur at runtime. Although dependence profiling is inherently input sensitive, the results are still useful in many situations, which is why such profiling forms the basis of many program analysis tools [10, 62, 15]. Besides, input sensitivity can be addressed to some degree by running the target program with changing inputs and computing the union of all collected dependences.

However, a serious limitation of data-dependence profiling is high runtime overhead in terms of both time and space. The former may significantly prolong the analysis, sometimes requiring an entire night [74]. The latter may prevent the analysis completely [75]. This is because dependence profiling requires all memory accesses to be instrumented and records of all accessed memory locations to be kept. In previous work, their overhead was reduced either by tailoring the profiling technique to a specific analysis or by parallelizing it.

Using dependence profiling, Kremlin [10] determines the length of the critical path in a given code region. Based on this knowledge, it calculates a metric called self-parallelism, which quantifies the parallelism of the region. Instead of pair-wise dependences, Kemlin records only the length of the critical path. Alchemist [13], a tool that estimates the effectiveness of parallelizing program regions by asynchronously executing certain language constructs, profiles dependence distance instead of detailed dependences. Although these approaches profile data dependences with low overhead, the underlying profiling technique has difficulty in supporting other program analyses.

There are also approaches that reduce the time overhead of dependence profiling through parallelization. For example, SD3 [75] exploits pipeline and data parallelism to extract data dependences from loops. At the same time, SD3 reduces the significant space overhead of tracing memory accesses by compressing strided accesses using a finite state machine. Multi-slicing [76] follows the same compression approach as SD3 to reduce the memory overhead, but leverages compiler support for parallelization. Before execution, the compiler divides the profiling job into multiple profiling tasks through a series of static analyses, including alias/edge partitioning, equivalence classification, and thinned static analysis. According to published results, the slowdown of these approaches stays close to ours when profiling the hottest 20 loops (70× on average using SD3 with 8 threads), but remains much higher when profiling whole programs (over 500× on average using multi-slicing with 8 threads).

Like SD3 and multi-slicing, we parallelize the data-dependence profiling algorithm instead of customizing it. Unlike these methods, we profile detailed data dependences and control-flow information for not only sequential but also multi-threaded programs. Furthermore, our parallelization is achieved through lock-free programming, ensuring good performance without loss of generality.

2.3 DiscoPoP Profiler

To provide a general foundation for our parallelism discovery framework and other data-dependence-based analysis techniques, we present a generic data dependence profiler called DiscoPoP profiler, using the same name with our parallelism discovery framework. With practical overhead, the DiscoPoP profiler is capable of supporting a broad range of dependence-based program analysis and optimization techniques—both for sequential and parallel programs. To achieve efficiency in time, the profiler is parallelized, taking advantage of lock-free design [77]. To achieve efficiency in space, the profiler leverages signatures [78], a concept borrowed from transactional memory. Both optimizations are application-oblivious, which is why they do not restrict its scope in any way. Our profiler has the following specific features:

- It collects pair-wise data dependences of all the three types (RAW, WAR, WAW) along with runtime control-flow information
- It is efficient with respect to both time and memory (average slowdown of only 86×, average memory consumption of only 1020 MB for benchmarks from NAS and Starbench)
- It supports both sequential and parallel (i.e., multithreaded) target programs
- It provides detailed information, including source-code location, variable name, and thread ID

2.3.1 Representation of Data Dependences

A sample piece of dependence data produced by our profiler is shown in Figure 2.1. A data dependence is represented as a triple `<sink, type, source>`. `type` is the dependence type (RAW, WAR or WAW). Note that a special type `INIT` represents the first write operation to a memory address.

`source` and `sink` are the source code locations of the former and the latter memory accesses, respectively. `sink` is further represented as a pair `<fileID:lineID>`, while `source` is represented as a triple `<fileID:lineID|variableName>`. As shown in Figure 2.1, data dependences with the same `sink` are aggregated together.

The keyword `NOM` (short for “NORMAL") indicates that the source line specified by the aggregated `sink` has no control-flow information. Otherwise, `BGN` and `END` represent the entry and

```
1:60 BGN loop
1:60 NOM {RAW 1:60|i}     {WAR 1:60|i}
         {INIT *}
1:63 NOM {RAW 1:59|temp1} {RAW 1:67|temp1}
1:64 NOM {RAW 1:60|i}
1:65 NOM {RAW 1:59|temp1} {RAW 1:67|temp1}
         {WAR 1:67|temp2} {INIT *}
1:66 NOM {RAW 1:59|temp1} {RAW 1:65|temp2}
         {RAW 1:67|temp1} {INIT *}
1:67 NOM {RAW 1:65|temp2} {WAR 1:66|temp1}
1:70 NOM {RAW 1:67|temp1} {INIT *}
1:74 NOM {RAW 1:41|block}
1:74 END loop 1200
```

Figure 2.1: A fragment of profiled data dependences in a sequential program.

exit point of a control region, respectively. In Figure 2.1, a loop starts at source line 1:60 and ends at source line 1:74. The number following `END loop` shows the actual number of iterations executed, which is 1200 in this case.

2.3.2 Signature-Based Profiling

Traditional data-dependence profiling approaches record memory accesses using shadow memory. In shadow memory, the access history of addresses is stored in a table where the index of an address is the address itself. This approach results in a table covering the memory space from the lowest to the highest address accessed by the target program, which consumes a lot of memory. Although this problem can be partially solved by using multilevel tables, the memory overhead of shadow memory is still too high. According to previous work [75], it is often impossible to profile even small programs using shadow memory if no more than 16 GB of memory is available.

An alternative is to record memory accesses using a hash table, but this approach incurs additional time overhead since when more than one address is hashed into the same bucket, the bucket has to be searched for the address in question. Note that profiling data dependence pair-wise requires an exhaustive instrumentation of all memory accesses in the target program. The number of memory accesses in an ordinary benchmark can easily reach one billion. With all these accesses instrumented, a tiny time cost of the instrumentation function will accumulate into a huge overhead. Based on our experiments, the hash table approach is about 1.5 – 3.7× slower than our approach.

A solution to decrease the profiling overhead is to use an approximate representation rather than instrument every memory access. Previous work [79] tried to ignore memory accesses in a code section when it had been executed more than 2^{32-k} times. However, when setting $k = 10$, only 33.7% of the memory accesses are covered, which can lead to significant inconsistency among the profiled data dependences.

To lower the memory overhead without increasing the time overhead, we record memory accesses in signatures. A signature is a data structure that encodes an approximate representation of an unbounded set of elements with a bounded amount of state [78]. It is widely used in transactional memory systems to uncover conflicts. A signature usually supports three operations:

- Insertion: inserts a new element into the signature. The state of the signature is changed after the insertion.
- Membership check: tests whether an element is already a member of the signature.
- Disambiguation: intersects two signatures. If an element was inserted into both of them, the resulting element must be present in the intersection.

A data dependence is similar to a conflict in transactional memory because it exists only if two or more memory operations access the same memory location in some order. Therefore, a signature is also suitable for detecting data dependences. Usually, a signature is implemented as a bloom filter [80], which is a fixed-size bit array with k different hash functions that together map an element to a number of array indices. Here, we adopt a similar idea, using a fixed-length array combined with a hash function that maps memory addresses to array indices. We use only one hash function to simplify the removal of elements because it is required by variable lifetime analysis, an optimization we implemented to lower the probability of building incorrect dependences. In variable lifetime analysis, addresses that become obsolete after deallocating the corresponding variable are removed from a signature. Also, each slot of the array is three bytes long instead of one bit so that the source line number where the memory access occurs can be stored in it. Because of the fixed length of the data structure, memory consumption can be adjusted as needed.

To detect data dependences, we apply Algorithm 2. It deploys two signatures: one for recording read operations and one for recording write operations. When a memory access c at address x is captured, we first determine the access type (read or write). Then, we run the membership check to see if x exists in the signatures. If x already exists, we build a data dependence and change the source line number to where c occurred. Otherwise, we insert x into the signature. Note that we ignore read-after-read (RAR) dependences because in most program analyses they are not required.

With signatures, we trade a slight degree of accuracy of profiled dependence for profiling speed. When more than one address is hashed into the same slot, false dependences are created instead of building additional data structures to keep the addresses, saving time for maintaining the structures and searching the address from them. Signatures are implemented in fixed-size arrays so that the overhead of new/delete or malloc/free is eliminated.

A signature is an approximate representation where hash collisions can happen. A hash collision in signatures can lead to both false positives and false negatives in profiled dependences.

```
Global signatures sig_write and sig_read

for each memory access c in the program do
    index = hash(c)
    if c is write operation then
        if sig_write[index] is empty then
            c is initialization
        end
        else
            if sig_read[index] is not empty then
                buildWAR()
            end
            buildWAW()
        end
        sig_write[index] = source line number of c
    end
    else
        if sig_write[index] is not empty then
            buildRAW()
        end
        sig_read[index] = source line number of c
    end
end
```

Algorithm 2: Algorithm for signature-based data-dependence profiling (pseudocode).

In Section 2.5.1, we show that the false positive and false negative rates of profiled dependences are negligible if sufficiently large signatures are used. Nonetheless, sufficiently large is still small in comparison to shadow memory. If an estimation of the total number of memory addresses accessed in the target program is available, the signature size can also be estimated using Formula 2.2 in Section 2.5.1. A very practical alternative is to use all the memory of the target system for profiling that remains after subtracting the memory space needed for the target program itself, which is usually more than enough to yield dependences with high accuracy. Consider the following situation:

```
store i32 0, i32* %x  // write x
store i32 1, i32* %y  // write y
%1 = load i32* %x     // read  x
```

where address x and y are hashed into the same slot. In this case, a WAW dependence between write y and write x and a RAW dependence between read x and write y are built, and the RAW dependence between read x and write x is missed (false negative). The former case shows a situation in which false positives appear, and the latter case shows a situation in which false negative appears.

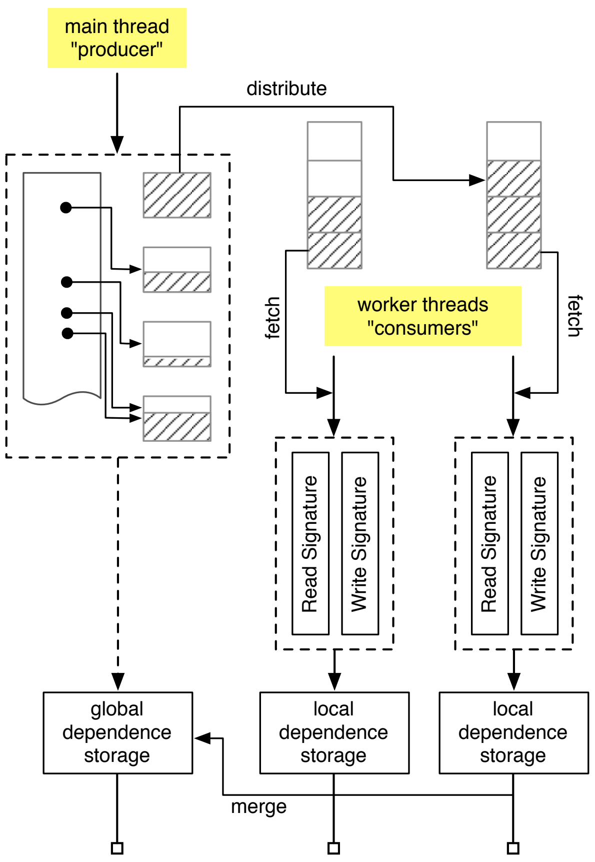


Figure 2.2: Architecture of the parallel DiscoPoP data-dependence profiler for sequential programs.

Finally, we merge identical dependences to reduce the runtime memory overhead and the time needed to write the dependences to disk. Based on our experience, this step is necessary to arrive at a practical solution. Merging identical dependences decreased the average output file size for NAS benchmarks from 6.1 GB to 53 KB, corresponding to an average reduction by a factor of 10^5.

2.3.3 Parallel Data-Dependence Profiling

The basic idea behind the parallelization of our approach is to run the profiling algorithm in parallel on disjoint subsets of the memory addresses. To determine the dependence type (i.e., RAW, WAR, or WAW) correctly, we need to preserve the temporal order of memory accesses to the same address. For this reason, a memory address is assigned to exactly one worker thread, which becomes responsible for all accesses to this address. To buffer incoming memory accesses before they are consumed, we use a separate queue for each worker thread, which can fetch data only from the queue assigned to it.

In our implementation, we apply the producer-consumer pattern. The main thread executes the target program and plays the role of the producer, collecting and sorting memory accesses, whereas the worker threads play the role of consumers, consuming and analyzing memory accesses and reporting data dependences.

Figure 2.2 shows how our parallel design works. The main thread executes the program to be analyzed and collects memory accesses in chunks, whose size can be configured in the interest of scalability. One chunk contains only memory accesses to be assigned to one thread. Once a chunk is full, the main thread pushes it into the queue of the thread responsible for the accesses recorded in it. The worker threads in turn consume chunks from their queues, analyze them, and store detected data dependences in thread-local maps. Empty chunks are recycled and can be reused. The use of maps ensures that identical dependences are not stored more than once. At the end, we merge the data from all local maps into a global map. This step incurs only minor overhead since the local maps are free of duplicates. Since the major synchronization overhead comes from locking and unlocking the queues, we made the queues lock-free to lower the overhead.

Lock-free parallelization

In our parallelization strategy, the major synchronization overhead comes from locking and unlocking the queues. Hence, we made the queues lock-free to lower the overhead. As shown in Figure 2.2, the queues used are single-producer-single-consumer (SPSC) queues, since only the main thread can push chunks into a queue and only the responsible worker thread can fetch chunks from it. Obviously, producer and consumer work on different parts of an SPSC queue most of the time. As long as the tail index is not equal to the front index, there is guaranteed to be at least one element to dequeue. To improve the concurrency further, the producer and the consumer can actually access the queue in parallel without even locking a single node—as long as consistent memory visibility is ensured.

In order to ensure the consistent memory visibility between the producer and consumer, we utilize release-acquire synchronization, which is supported in C++11. After enqueuing a new item, the producer performs an atomic store with memory-order-release. Before dequeuing an item, the consumer performs an atomic load with memory-order-acquire. Once the atomic load is complete, the consumer is guaranteed to see everything the producer wrote to memory. As a consequence, synchronization is narrowed down to the load/store instruction level, and the overhead is much smaller than when locking/unlocking the entire queue.

Load balancing

In our profiler, memory accesses are distributed among worker threads using a simple modulo function:

$$worker\_ID = memory\_address \quad \% \quad W \tag{2.1}$$

with W being the number of worker threads. According to our experiments, this simple function achieves an even distribution of accessed memory addresses. A similar conclusion is also drawn in SD3 [75]. Although memory addresses are distributed evenly, not all of them are accessed with the same frequency. Some addresses may be accessed millions of times while others are only accessed a few times. To avoid the situation where all heavily accessed addresses are assigned to the same worker thread, we also monitor how many times an address is accessed dynamically. These access statistics are stored in a map and updated every time a memory access occurs. The access statistics are needed to ensure that the top ten most heavily accessed addresses are always evenly distributed among worker threads.

The access statistics are evaluated at regular intervals. If we notice that the distribution of heavily accessed memory addresses is out of balance, we initiate redistribution. If an address is moved to another thread, its signature state has to be moved as well. After redistribution, accesses to redistributed addresses will always be directed to the newly assigned worker thread. Redistribution rules are stored in a map and have higher priority than the modulo function.

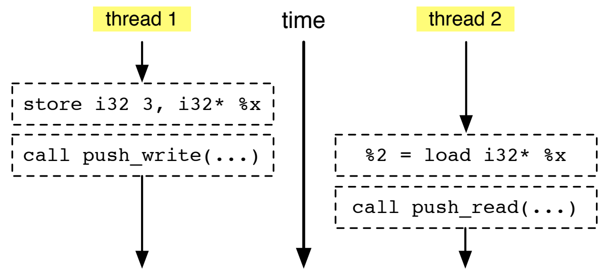
Redistribution is costly, which is why it should not be performed too frequently. In our implementation, we check whether redistribution is needed after every 50,000 chunks. Consequently, for the benchmarks used in this paper, redistribution is performed at most 20 times when profiling a single benchmark, which is enough to have a positive impact on the time overhead.

2.3.4 Supporting Multi-Threaded Target Programs

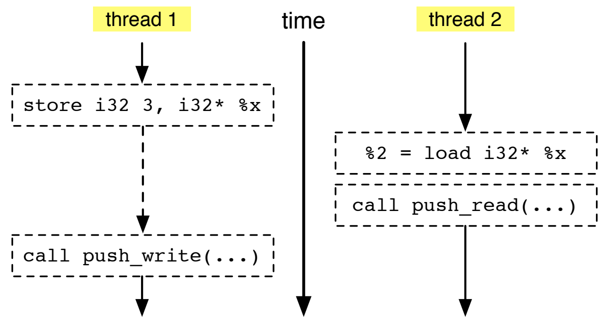
```
4:58|2 NOM {WAR 4:77|2|iter}
4:59|2 NOM {WAR 4:71|2|z_real}
4:64|3 NOM {RAW 3:75|0|maxiter}
           {RAW 4:58|3|iter}    {RAW 4:61|3|z_norm}
           {RAW 4:71|3|z_norm}  {RAW 4:73|3|iter}
4:69|3 NOM {RAW 4:57|3|c_real}
           {RAW 4:66|3|z2_real} {WAR 4:67|3|z_real}
4:71|2 NOM {RAW 4:69|2|z_real}
           {RAW 4:70|2|z_imag}  {WAR 4:64|2|z_norm}
4:80|1 NOM {WAW 4:80|1|green}   {INIT *}
```

Figure 2.3: A fragment of data dependences from a parallel program captured by our profiler. Thread IDs are highlighted.

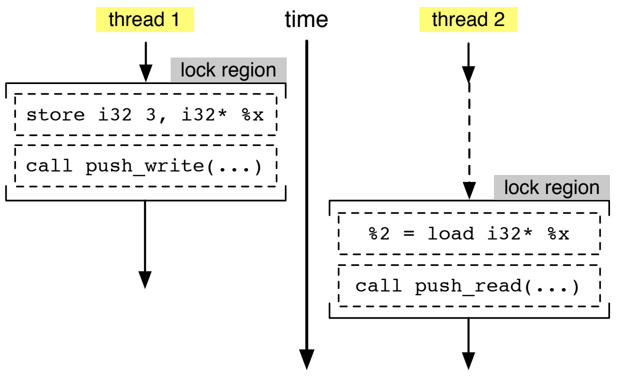
A data dependence in a parallel program is still represented as triple <sink, type, source>. However, to distinguish different threads, we add thread IDs to the sink and source fields. Now, sink has the form <fileID:lineID|threadID> and source has the form <fileID:lineID|threadID| variableName>. Control-flow information is recorded in the same way as shown earlier in Section 2.3.1. Figure 2.3 shows a fragment of dependences captured in a parallel program.



(a) Expected scheduling. `store` happens before `load`. `push_write` and `push_read` record them in the same order. The RAW dependence is detected, which is correct.



(b) Unexpected scheduling. `store` happens before `load`, but due to thread scheduling `push_write` and `push_read` record them in the reversed order. A WAR dependence is detected, which is wrong.



(c) Solution. Instrumentation functions `push_write` and `push_read` are always inserted in the same lock region as the corresponding memory accesses (explicit locking/unlocking primitives in target code required).

Figure 2.4: Thread scheduling affects the correctness of recorded data dependences. `push_read()` and `push_write()` are operations to push memory accesses into chunks.

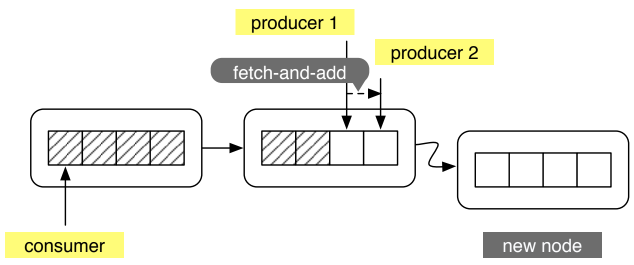


Figure 2.5: A lock-free multiple-producer-single-consumer queue.

Modified parallelization strategy

In a sequential program, the temporal order of memory accesses is automatically preserved. Thanks to this property, we can easily ensure that our parallel profiler produces the same data dependences as the serial version—provided we push a memory access into the corresponding chunk immediately after encountering it. However, parallel programs do not have this property. In a multi-threaded environment, it is not guaranteed that the push operation is always executed immediately after the memory access, resulting in incorrect data dependences.

Figure 2.4 illustrates the problem described above. The expected execution order is shown in Figure 2.4(a). Thread 1 stores 3 to `x` first, then thread 2 loads the value of `x` to a temporary location. The corresponding push operations `push_write` and `push_read` are executed in the same order, so that the RAW dependence is recorded correctly.

However, in a multi-threaded environment, the push operation is not promised to be always executed immediately after the memory access, as shown in Figure 2.4(b). Although thread 1 stores 3 to `x` first, depending on the thread schedule `push_write` may be executed after `push_read` in thread 2. In this case, a RAW dependence is wrongly recorded as WAR.

To solve this problem, we need to make a memory access and its corresponding push operation atomic. Thus, we require that accesses to the same address from multiple threads are protected by locks, and we insert the push operation into the same lock region, as shown in Figure 2.4(c). So far we support only parallel programming languages where locking/unlocking primitives have to be written explicitly in the source code. However, programing languages with implicit synchronization can be easily supported by automatically discovering implicit synchronization patterns [81].

Another difference when profiling parallel programs is that more than one thread may push items into the queue of a worker thread, which is a multiple-producer-single-consumer (MPSC) queue pattern. It means that we have to synchronize producers in an efficient way. For this reason, we implement the lock-free MPSC queue as a linked list of arrays. With these arrays, producers can safely enqueue items at different indices of the array in parallel.

Figure 2.5 shows how our implementation works. Each producer tries to acquire a free index in the array using an atomic fetch-and-add operation. Once the the array in one queue node is full, a new queue node wrapping a new array is created and appended to the tail of the queue. Once all items in a queue node have been dequeued, the node is deallocated. Since fetch-and-add operations are directly supported by the hardware, the synchronization overhead is again minimal.

Data races

We generally do not know whether the cross-thread dependences we report are enforced or not, that is, whether they will be reproduced when the program is run again. In this sense, they can also be regarded as incidental *happens-before* relationships. In most cases, a correct program would always enforce such dependences. An example of an exception is the concurrent update of a flag indicating whether a parallel search was successful. However, these cases are rare in programs. It is usually desirable to know whether a dependence is enforced or not. One way of detecting unenforced dependences is to run the program more than once and hope that a different thread schedule will reverse the order and expose the race. Because this can be a successful strategy for finding races, reporting potentially irreproducible dependences is also valuable from a correctness perspective.

However, there are also cases where we can actually prove the occurrence of a data race even after a single run. The situation where the atomicity of access occurrence and reporting is violated can only happen if there are no explicit locking/unlocking synchronization mechanisms in place to keep the two accesses to memory location mutually exclusive. For this reason, the reported dependence may show the reverse of the actual execution order. To catch such cases, we acquire the timestamp of every memory access and pass it to the corresponding push operation as a parameter. Whenever a worker thread fetches memory accesses from its queue it usually expects increasing timestamps. A violation of this condition indicates that the memory accesses were pushed in a different order from the one in which they occurred. In this case, we mark the dependence accordingly. Moreover, whenever we see such a reversal, we can conclude that the memory accesses were not guaranteed to be mutually exclusive. Although mutual exclusion does not necessarily enforce a particular access order, its absence definitely exposes a potential data race.

2.3.5 Optimization

There are a few optimization techniques implemented to increase either profiling accuracy or performance in terms of time and memory.

Variable lifetime analysis

Although false positives are a basic property of signatures and cannot be completely eliminated, we apply an optimization to lower the false-positive rate further. The main idea is to remove variables from the signature once it is clear that they will never be used again during the remainder of the execution. Thus, we need a way to monitor the lifetime of a variable. The lifetime of a variable is the time between its allocation and deallocation. The lifetime of variables has an impact on the correctness of the data dependence analysis because signature slots of dead variables might be reused for new variables. If this happens, a false dependence will be built between the last access of the dead variable and the first access of the new variable.

To resolve this problem, we perform variable lifetime analysis dynamically. This means we observe the allocation and deallocation of variables, including both explicit methods like `new/delete` and `malloc/free`, and implicit allocation and deallocation of local variables. To achieve this, we exploit dynamic control-flow information, which is helpful to determine the lifetime of local variables allocated inside a control region. Although there is no explicit deallocation of local variables, they die once the program leaves the control region where they have been allocated. In this way, signature slots for local variables can be reused without the danger of building false dependences. With variable lifetime analysis, our signature algorithm can support more variables with the same amount of memory.

Runtime data dependence merging

Recording every data dependence may consume an excessive amount of memory. DiscoPoP performs all the analyses on every instruction that is dynamically executed. Depending on the size of both the source code and the input data, the size of the file containing processed data dependences can quickly grow to several gigabytes for some programs. However, we found that many data dependences are redundant, especially for regions like loops and functions which will be executed many times. Therefore, we merge identical data dependences. This approach significantly reduces the number of data dependences written to disk.

A data dependence is expressed as a triple:

```
<Dependent-Line, dependence-Type, Depends-On-Line>
```

with attributes like variable name, thread ID (only available for multi-threaded programs), and inter-iteration tag. Two data dependences are identical if and only if each element of the triple and all attributes are identical. When a data dependence is found, we check whether it already exists. If there is no match, a new entry for the dependence is created. Otherwise the new dependence is discarded. For a code region that is executed more than once, we maintain only one set of dependences, merging the dependences that occur across multiple instances. When the parallelism-discovery module reads the dependence file, it still treats these multiple execution instances as one. For example, a loop will always be considered as a whole and its iterations will never be expanded.

Merging data dependences may hide parallelism that is only temporarily available. For example, the first half of the iterations of a loop can be parallelized but the second half cannot. With data dependences merged, parallelism that exists in the first half of the iterations can be hidden. We recognize that temporarily available parallelism is definitely promising. However, discovering such parallelism requires a significant amount of time and memory since every iteration must have its own instance of profiled data, and parallelism must be checked between every two instances. We implemented a version without dependence merging, and it failed to profile most of the NAS benchmarks. For those programs it can profile, the size of the dependence file ranged from 330 MB to about 37 GB with input class W (6.1 GB on average).

The effect of merging data dependences is significant. After introducing runtime data dependence merging, all the NAS benchmarks can be profiled and the file size decreased to between 3 KB and 146 KB (53 KB on average), corresponding to an average reduction by a factor of $10^5\times$. Since the parallelism-discovery module redirects the read pointer in the file when encountering function calls rather than processing the file linearly, data dependence merging drastically reduces the time needed for parallelism discovery.

2.3.6 Control Structure Information

To support parallel pattern detection [82, 83], DiscoPoP profiler also produces the Program Execution Tree (PET). We construct a PET using the following information obtained from both static analyses and profiling data:

- Locations of call sites
- Locations of entries and exits of loops
- Locations of entries and exits of functions
- Number of iterations executed of each loop
- Number of IR statements of each scope
- Number of data dependences of each scope

A PET represents a specific execution of the target program. Thus it has a single root node representing the entry point of the execution. A PET contains three kinds of nodes and two kinds of edges:

- Nodes
  - Function node
  - Loop node
  - Block node
- Edges
  - “Calling” edge
  - “Containing” edge

```
for (...) {
  Block 1
  foo();
  Block 2
}
Block 3
while (...) {
  Block 4
}
```

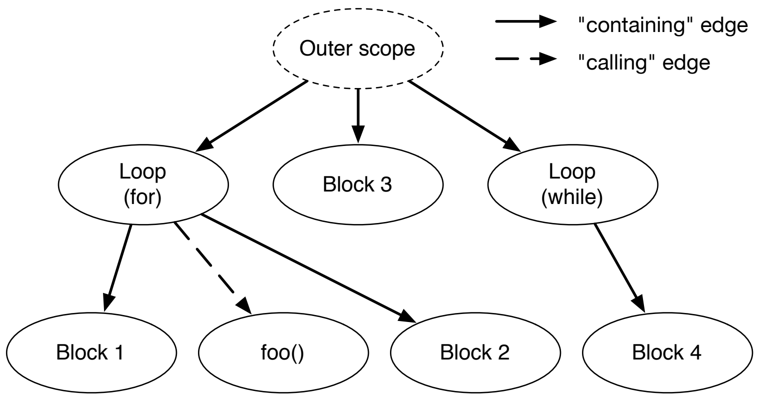


Figure 2.6: An example of the program execution tree (PET).

Function nodes represent functions of a program. Incoming edges of function nodes are “calling” edges (functions are called by other functions), and outgoing edges can be either “calling” edges (calling other functions) or “containing” edges (contains loops or blocks of code). When considering only function nodes and “calling” edges, a PET is similar to a call graph excluding functions that are not executed.

Loop nodes represent loops of a program. Both incoming and outgoing edges are “containing” edges since loops cannot be called or invoked. In contrast to other nodes, each loop node has a counter recording the number of executed iterations.

Block nodes represent blocks of code of a program. They are plain blocks that do not contain control-flow constructs inside. Block nodes are always leaf nodes of a PET. Incoming edges are “containing” edges, and there are no outgoing edges.

Each node, despite its type, has several metrics characterizing the scope it represents, including the number of IR statements, the number of data dependences, the length of its critical path, and so on. These metrics can be used in selecting the most interesting code sections under different requirements. In this thesis, they are used to rank parallelization opportunities (Section 4.3). Figure 2.6 shows an example of a PET along side the corresponding code structure.

The notion of a PET is the key to detecting parallel patterns. Attaching data dependences to a PET results in a comprehensive tree of dependences among functions, loops, and blocks of code in a hierarchical way. When examining parallelism between two functions, data dependences within each of them can be easily ignored. Such features allow the straightforward application of pattern matching technique to detect parallel patterns. Details about parallel pattern detection is introduced in related work [82, 83].

2.3.7 Limitations

At the moment, the DiscoPoP profiler has the following limitations:

- It does not guarantee 100% accurate data dependences

- It does not guarantee correctness when profiling parallel programs with implicit synchronization mechanisms

To produce 100% accurate data dependences, we have to use a classic shadow memory solution. DiscoPoP provides a shadow memory implemented using a hash table, which is slower and consumes more memory. The user can choose this option if 100% accuracy is a must.

Parallel programs with implicit mutual exclusion and synchronization mechanism can be supported by existing approaches of detecting synchronizations automatically [81]. This is considered as one of the future-work items.

2.4 Skipping Repeatedly-Executed Memory Operations in Loops

In general, a profiler obtains information about the target program by reading hardware counters, sampling instructions during runtime, or inserting instrumentation functions. Some profilers utilize more than one technique to obtain information. Specifically, a data-dependence profiler usually contains two parts: an instrumentation component that inserts analysis functions for memory operations, and a runtime library that implements the analysis functions and data structures. Instrumented code will be linked against the runtime library and executed. The runtime library is further divided into two components. The first component is usually called *shadow memory*. Note that in this context, the term "shadow memory" has a broader meaning, referring to any technique that maintains status information in a separate memory space. This is different to the meaning of shadow memory in Section 2.3.2, which refers to a narrower definition that every byte used in the target program has a shadow word to record its access status. The second component is the data-dependence storage, where data dependences are built and stored when the status of memory locations in the shadow memory changes.

In this section, we say a memory instruction when we refer to a machine instruction that accesses memory in the dynamic execution instruction sequence, and a memory operation when we refer to an intermediate representation statement that operates on memory. A memory operation residing in a loop usually leads to multiple memory instructions. In short, memory instructions are in dynamic execution sequence, while memory operations refer to static code. In most of the cases, a memory operation leads to exactly one memory instruction. However, a memory operation that resides in a loop and accesses memory through a pointer will lead to multiple memory instructions.

Instrumentation can be done statically, and the time overhead of instrumentation is usually negligible. The main time overhead is caused by the remaining two phases: updating shadow memory and building dependences. Both shadow memory and dependence storage are typically implemented based on table-like data structures where each memory address or data dependence has an entry. Given that the number of memory instructions and data dependences is usually very large, the overhead is mainly incurred by searching and updating the data structures, and inserting elements into them. As a result, data-dependence profiling typically slows the program down by a factor ranging from 100 to 500.

```
while (k > 0) {
  sum += k * 2;
  k--;
}
```

Figure 2.7: A simple loop where data dependences will not change over iterations.

Table 2.2: Data dependences of the loop shown in Figure 2.7.

ID	sink	source	type	variable	loop-carried
1	2	2	write after read (WAR)	sum	no
2	3	1	write after read (WAR)	k	no
3	3	2	write after read (WAR)	k	no
4	3	3	write after read (WAR)	k	no
5	1	3	read after write (RAW)	k	yes
6	2	2	read after write (RAW)	sum	yes
7	2	3	read after write (RAW)	k	yes
8	3	3	read after write (RAW)	k	yes

However, not every memory instruction has to be processed through all the three phases. Let us take the loop shown in Figure 2.7 as an example. After profiling two iterations of the loop, the data dependences are complete. Table 2.2 shows the dependences. *Source* and *sink* are the source code locations of the first and the second memory instruction, respectively. *Type* is the dependence type, including read after write (RAW), write after read (WAR), and write after write (WAW). *Variable* is the variable that causes a dependence. When source and sink of a dependence belong to different iterations of a loop, we call the dependence a *loop-carried* dependence.

Among the dependences shown in Table 2.2, dependence 1–4 can be obtained within the first iteration, and dependence 5–8 will be added once the second iteration is done. After that, no more data dependence will be built, no matter how many iterations the loop has. In this case, profiling the remaining memory instructions in this loop over and over again is not necessary. It may be necessary to keep updating the status information in the shadow memory for correctness, but we definitely do not want to touch the dependence storage when profiling the same code section after the data dependences for the code section are complete. In the next section, we show how we skip these memory instructions after the dependences are fully obtained to accelerate the profiling process.

Before describing our method, we first briefly introduce basic implementation concepts of a data-dependence profiler since it helps understand our method. We have known that a data-dependence profiler has an instrumentation component that inserts analysis functions for memory operations. An analysis function for a memory operation looks like this:

```
analyze_mem_op(accessType, accessInfo, addr)
```

For a memory operation, `accessType` can be either read or write. It does not change over time. In practice, two analysis functions will be created for read and write operations, respectively. Necessary information needed to update the shadow memory are stored in `accessInfo`, and passed into the analysis function. Usually, `accessInfo` is the identifier of the associated memory instruction. For example, the address of the operation, the source line location, the variable name, or a combination of such information. Depending on the specific implementation, `accessInfo` may or may not be unique to each memory operation. However, for one memory operation, its `accessInfo` does not change. Finally, `addr` is the memory address accessed by the memory operation. It can change if the address is referred to by pointers.

2.4.1 Condition on addr

If a memory instruction can be safely skipped, at least its corresponding memory operation must have been profiled before and the memory address it accesses must not change. For simplicity, we create a variable called `lastAddr` for each memory operation *op* storing the memory address accessed by the last memory instruction translated from *op* before the current memory instruction. And we require

```
addr == lastAddr
```

to be a necessary condition if a memory instruction can be safely skipped. `lastAddr` should be initialized with an address which is never accessed in user code in practice, such as `0x0`.

When the condition on `addr` holds, it only means that the memory operation corresponding to the current memory instruction has been profiled before. It does not mean all the data dependences that are related to the memory operation have been obtained. Again, let us take the loop shown in Figure 2.7 as an example. All the memory instructions in the first iteration will be profiled, and dependences 1–4 in Table 2.2 are obtained. When only applying the condition on `addr`, all the memory instructions are skipped from the second iteration on because the addresses accessed by all the memory instructions do not change. Thus, we name the condition on `addr` a necessary condition, and we still need other conditions to decide whether a memory instruction can be skipped.

2.4.2 Condition on `accessInfo`

The key to cover all data dependences is to decide when to resume profiling once the profiling has been paused. Our solution is to have a mechanism that allows an analysis function to be notified if the access status of its memory operation has changed, so that the subsequent memory instructions translated from the memory operation must be profiled again.

To track the access status of a memory address, the shadow memory stores `accessInfo` of the most recent read instruction and the most recent write instruction that accesses the address. We call them `statusRead` and `statusWrite`, respectively. We then create two variables `lastStatusRead` and `lastStatusWrite` for each memory operation *op*, storing the `accessInfo` of the most recent read instruction and the most recent write instruction that accessed the memory address accessed by *op* when *op* was profiled the last time, respectively. Then we require

```
statusRead == lastStatusRead &&
statusWrite == lastStatusWrite
```

to be another necessary condition if a memory instruction can be safely skipped. Both `lastStatusRead` and `lastStatusWrite` should be initialized with values that have no meanings for `accessInfo`.

When the condition on `accessInfo` holds, it means that the access status of the memory address was seen before. We say "was seen before" because the address may change, and the access status of the current memory address may just coincidentally be the same as the access status of another address. This is very likely to happen when `accessInfo` is not unique to each memory operation. However, combing the two conditions on `addr` and `accessInfo` will give a sufficient condition to decide whether a memory instruction can be safely skipped: the corresponding memory operation has been profiled before, the memory address it accesses does not change, and the access status of the memory address has not changed since it was profiled the last time.

When the conditions do not hold anymore, it means either the new memory instruction accesses a different memory address, or the access status of the memory address has changed. No matter what, the new memory instruction must be profiled in order to cover new data dependences.

2.4.3 Example

In this section, we show how our method works on a simple example. And we also present a special case where a memory instructions can be skipped even without updating the status of its memory address in shadow memory.

Figure 2.8 shows a loop with four memory operations (`op1`–`op4`). All the memory operations access the same memory address `x`. We show memory operations instead of source code so that the profiling process can be clearly illustrated. The data dependences of the loop shown in Figure 2.8 are listed in Table 2.3.

How the values stored in `lastStatusRead` and `lastStatusWrite` are changed for each memory instruction is shown in Table 2.4. "1st", "2nd", and "3rd" refer to the first, the second, and the third iteration of the loop, respectively. An "S" means the memory instruction is

```
loop:
  op1:  write x
  op2:  read  x
  op3:  read  x
  op4:  write x
end

```

Figure 2.8: A loop containing four memory operations on the same memory address.

Table 2.3: Data dependences of the loop shown in Figure 2.8.

ID	sink	source	type	variable	loop-carried
1	op2	op1	read after write (RAW)	x	no
2	op3	op1	read after write (RAW)	x	no
3	op4	op3	write after read (WAR)	x	no
4	op1	op4	write after write (WAW)	x	yes

Table 2.4: How the values of `lastStatusRead` and `lastStatusWrite` are changed during the profiling process for the loop shown in Figure 2.8.

Op	lastStatusRead				lastStatusWrite			
	init	1st	2nd	3rd	init	1st	2nd	3rd
write x	—	0	op3	S	—	0	op4	S
read x	—	0	op3	S	—	op1	op1	S
read x	—	op2	S	S	—	op1	S	S
write x	—	op3	S	S	—	op1	S	S

Table 2.5: How the status in shadow memory is changed during the profiling process for the loop shown in Figure 2.8.

execution	init	op1	op2	op3	op4	op1	op2	op3	op4
statusRead	0	0	op2	op3	op3	op3	op2	op3	op3
statusWrite	0	op1	op1	op1	op4	op1	op1	op1	op4

skipped, otherwise the memory instruction is profiled and the value of `lastStatusRead` and `lastStatusWrite` are updated.

How the access status of `x` is changed in the shadow memory is shown in Table 2.5. We adopt the most common design, where for each memory address the corresponding memory operations of the last read instruction and the last write instruction that access the address are stored.

Let us examine the profiling process step by step. In the beginning, all the variables are initialized. Now comes `op1` in the first iteration. Since `addr` is not equal to `lastAddr`, `op1` is profiled. The access status of `x` in shadow memory is read into `lastStatusRead` and `lastStatusWrite`, which are both 0 in the case of `op1`. Then `op1` updates the shadow memory. `statusWrite` of `x` is now 1.

The same process is applied to `op2` in the first iteration. The difference is that when `op2` is executed, `statusRead` and `statusWrite` of `x` have been changed to 0 and 1, respectively. With `statusWrite` being no longer zero, a read-after-write (RAW) dependence from `op2` to `op1` is built, which is the first dependence shown in Figure 2.3. The profiling process continues, and dependences 2 and 3 are built when `op3` and `op4` are profiled.

Now the profiling process enters the second iteration, and the second memory instruction translated from `op1` comes. Although the condition on `addr` holds this time, the condition on `AccessInfo` fails. The last time `op1` was profiled, the corresponding memory operations of the last read instruction (stored in `lastStatusRead`) and the last write instruction (in `lastStatusWrite`) accessing `x` were 0. After the first iteration is completed, they are 3 and 4. The second memory instruction translated from `op1` must be profiled to cover new dependences. Thus, the last data dependence in Table 2.3 is built. The same situation also happens to `op2`, but it only leads to a read-after-read (RAR) dependence, which is ignored in most of the data- dependence profilers.

Both conditions hold when the second memory instruction translated from `op3` is executed, and it is skipped. No dependence instance is built, and no query to the dependence storage occurs. Note that the shadow memory is still updated for ensuring the consistency between the instruction stream and the access status in shadow memory. From then on, all further memory instructions accessing `x` in the same loop are skipped, and no dependences are missed. The dependence storage is touched only four times, which matches exactly the number of dependences the loop contains.

Special case

When the loop contains only `op1`, `op2`, and `op3`, `statusWrite` of `x` will always be 1. This is a special case where the following condition holds:

```
currentWrite == statusWrite == lastStatusWrite.
```

In this case, a write instruction can be skipped without updating the shadow memory. The same applies to read instructions as well.

2.5 Evaluation

We conducted a range of experiments to evaluate both the accuracy of the profiled dependences and the performance of our implementation. Test cases are the SNU NAS Parallel Benchmarks

3.3.1 [84, 85] (NAS), a suite of programs derived from real-world computational fluid-dynamics applications, and the Starbench parallel benchmark suite [86] (Starbench), which covers programs from diverse domains, including image processing, information security, machine learning and so on. Whenever possible, we tried different inputs to compensate for the input sensitivity of dynamic dependence profiling.

Note that the original NAS Parallel Benchmarks [85] are FORTRAN programs, and SNU NAS Parallel Benchmarks [84] are the C equivalents. Since our method is implemented based on LLVM and uses Clang as the compiler, we use the C version of the benchmarks. The short term "NAS" in this thesis always refer to the SNU NAS Parallel Benchmarks.

2.5.1 Accuracy of Profiled Dependences

We first evaluate the accuracy of the profiled data dependences since we build upon the idea of a signature as an approximate representation of memory accesses. As it is described in Section 2.3.2, the membership check of this approximate representation can deliver false positives, which further lead to false-positive and false-negative dependences.

To measure the false positive rate (FPR) and the false negative rate (FNR) of the profiled dependences, we implemented a "perfect signature", in which hash collisions are guaranteed not to happen. Essentially, the perfect signature is a table where each memory address has its own entry, so that false positives are never produced. We use the perfect signature as the baseline to quantify the FPR and the FNR of the dependences delivered by our profiler.

Table 2.6 shows the results for Starbench. Three groups of FPR and FNR are shown under three different signature sizes in terms of the total number of slots. When using 1.0E+6 slots, the average FPR and FNR are 24.47% and 5.42%, respectively. The values are significantly reduced to 4.71% and 0.71% when the signature size is increased to 1.0E+7. Finally, hardly any incorrect dependences appear when the signature has 1.0E+8 slots as the average value of both FPR and FNR are lower than 0.4%. In our implementation, each slot is four bytes. Thus, 1.0E+8 slots consume only 382 MB of memory, which is adequate for any ordinary PC.

c-ray, *rgbyuv*, *rotate*, *rot-cc* and *bodytrack* have higher FPR and FNR than other programs because they access a large number of different addresses. This observation matches the theory of predicting the false positive rate of a signature. Assume that we use a hash function that selects each array slot with equal probability. Let m be the number of slots in the array. Then, the estimated false positive rate (P_{fp}), that is, the probability that a certain slot is *used* after inserting n elements is:

$$P_{fp} = 1 - (1 - \frac{1}{m})^n. \tag{2.2}$$

Clearly, P_{fp} is inversely proportional to m and proportional to n. In our case, m is the size of the signature and n is the number of addresses.

Table 2.6: False positive and false negative rates of profiled dependences for Starbench.

Program	LOC	# addresses	# accesses	# dependences	# slots = 1.0E+6		# slots = 1.0E+7		# slots = 1.0E+8	
					FPR	FNR	FPR	FNR	FPR	FNR
c-ray	620	1.1E+6	1.9E+9	574	19.95	1.34	6.00	0.24	0.00	0.00
kmeans	603	6.9E+5	1.9E+9	281	5.46	0.75	0.21	0.00	0.00	0.00
md5	661	2.6E+5	4.8E+8	859	3.08	0.15	0.03	0.00	0.00	0.00
ray-rot	1425	4.0E+5	9.8E+8	862	11.82	1.64	1.19	0.00	0.00	0.00
rgbyuv	483	6.3E+6	2.1E+8	155	47.67	15.74	4.44	1.90	0.21	0.05
rotate	871	3.1E+6	3.7E+8	278	55.92	15.68	4.50	3.02	0.00	0.00
rot-cc	1122	6.3E+6	4.9E+8	372	63.15	19.52	24.08	2.04	0.39	0.00
streamcluster	860	8.6E+3	1.2E+7	780	1.55	0.42	0.84	0.12	0.55	0.06
tinyjpeg	1922	4.2E+2	2.3E+7	1711	17.37	0.54	4.23	0.11	0.02	0.00
bodytrack	3614	4.4E+6	4.8E+9	3422	25.07	3.56	3.69	0.32	0.75	0.30
h264dec	42 822	8.7E+5	3.6E+8	31 138	18.10	0.23	2.63	0.03	1.95	0.01
average	—	—	—	—	24.47	5.42	4.71	0.71	0.35	0.04

2.5.2 Performance

We conducted our performance experiments on a server with 2 x 8-core Intel Xeon E5-2650 2 GHz processors with 32 GB memory, running Ubuntu 12.04 (64-bit server edition). All the test programs were compiled with option `-g -O2` using Clang 3.3. For NAS, we used the input set W; for Starbench, we used the reference input set.

Time overhead

First, we examine the time overhead of our profiler. The number of threads for profiling is set to 8 and 16. The slowdown figures are average values of three executions compared with the execution time of uninstrumented runs. The negligible time spent in the instrumentation is not included in the overhead. For NAS and Starbench, instrumentation was always done in two seconds.

The slowdown of our profiler when profiling sequential programs is shown in Figure 2.9. The average slowdowns for the two benchmark suites ("NAS-average" and "Starbench-average") are also included. As the figure shows, our serial profiler has a 190× slowdown on average for NAS benchmarks and a 191× slowdown on average for Starbench programs. The overhead is not surprising since we perform an exhaustive profiling for the whole program.

When using 8 threads, our lock-free parallel profiler gives a 97× slowdown on average for NAS benchmarks and a 101× slowdown on average for Starbench programs. After increasing the number of threads to 16, the average slowdown is only 78× for NAS benchmarks, and 93× for Starbench programs. Compared to the serial profiler, our lock-free parallel profiler achieves a 2.4× and a 2.1× speedup using 16 threads on NAS and Starbench benchmark suites, respectively.

Our profiler may seems slightly slower than SD3, which has a 70× slowdown on average using eight threads [75]. However, the slowdown of SD3 is measured by profiling the hottest 20 loops from each benchmark. Multi-slicing [76], another parallel dependence profiler that shares its sequential design with SD3, results with eight threads in a slowdown of more than 500× on average when applied to the entire target program.

The speedup is not linear for two reasons. Firstly, data-dependence profiling always has imbalanced workload due to uneven accesses, as we discussed in Section 2.3.3. In this case, simply introducing more worker threads does not help balance the workload. Similar behavior is also observed in related work [76]. Profiling performance is affected by this problem on five benchmarks: *kMeans*, *rgbyuv*, *rotate*, *bodytrack* and *h264dec*.

Secondly, determining detailed data dependence types (RAW, WAR, WAW) requires retaining the temporal order of memory accesses to the same address, which means such accesses have to be processed sequentially. Obviously, determining only a binary value (whether a dependence exists or not) instead of detailed types would allow a more balanced workload and lead to better performance. Moreover, the performance of the profiler can be further improved via set-based profiling, which tells whether a data dependence exists between two code sections instead of two statements. However, all these optimization will decrease the generality of the profiler, which is contrary to our purpose.

Figure 2.9 also shows the slowdown of our lock-based profiler when eight threads are used. Compared to the lock-based version, our lock-free version gives a 1.6× speedup on average for NAS benchmarks, and a 1.3× speedup on average for Starbench programs. A faster lock-free implementation that only allocates memory but never de-allocates will further boost performance, but increase the memory overhead significantly.

When profiling multi-threaded code, our profiler has a higher time overhead because more contentions are introduced. The native execution time of the parallel benchmarks is calculated by accumulating the time spent in each thread.

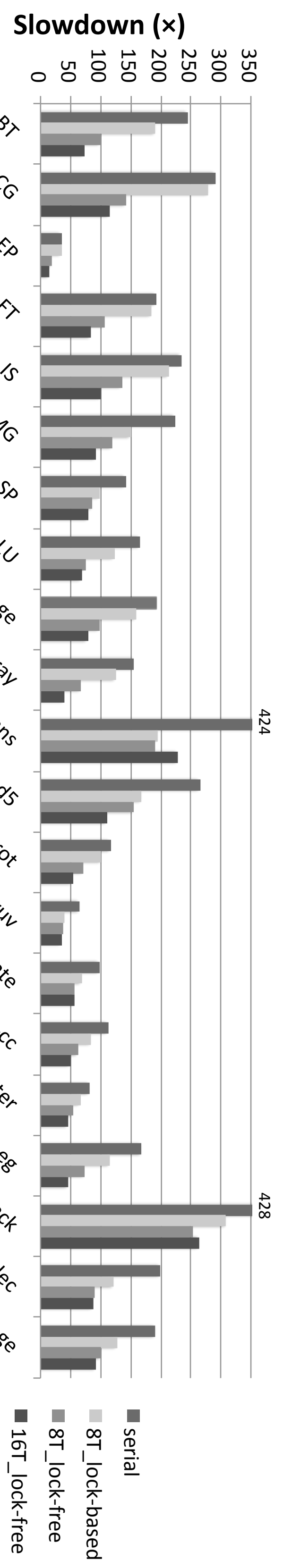
Slowdowns of our profiler for parallel Starbench programs (pthread version, 4 threads) are shown in Figure 2.10. We only tested Starbench because our profiler currently requires parallel programs with explicit locking/unlocking primitives. Using eight threads for profiling, the average slowdown of our profiler for Starbench is 346×, and further decreases to 261× when 16 threads are used for profiling. Again, *kMeans*, *rgbyuv*, *rotate*, *bodytrack* and *h264dec* do not scale well because of their imbalanced memory access pattern.

Memory consumption

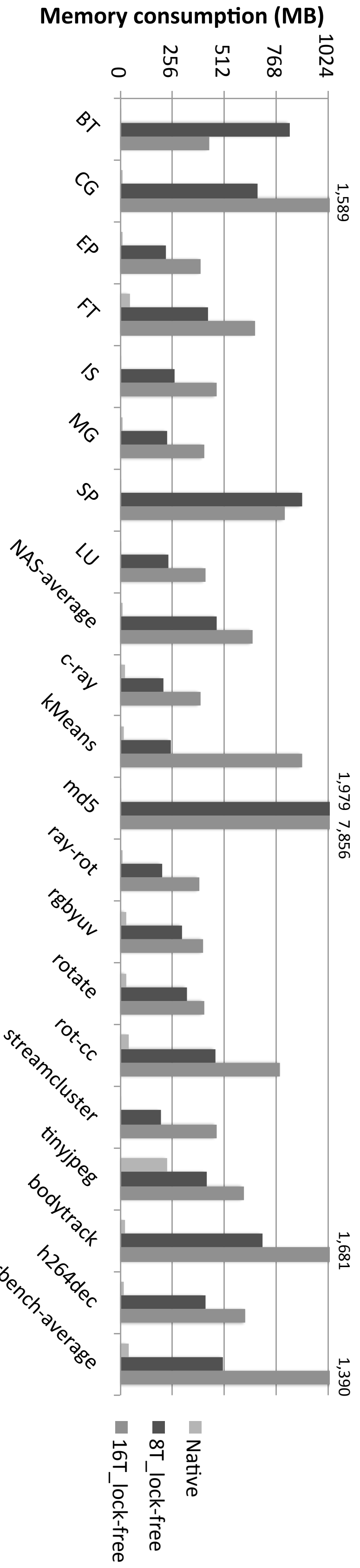
We measure memory consumption using the *maximum resident set size* value provided by `/usr/bin/time` with the verbose (`-v`) option. Figure 2.9(b) shows the results when 6.25E+6 signature slots are used in each thread, which summed to 1.0E+8 slots in total of 16 threads. This configuration leads to 191 MB and 382 MB of memory to be consumed by the signatures for 8 threads and 16 threads, respectively.

When using 8 threads, our profiler consumes 473 MB of memory on average for NAS benchmarks and 505 MB of memory on average for Starbench programs. After increasing the number of threads to 16, the average memory consumption is increased to 649 MB and 1390 MB for NAS and Starbench programs, respectively. The worst case happens when 16 threads are used to profile *md5*, which consumes about 7.6 GB of memory. Although this may exceed the memory capacity configured in a three-year-old PC, it is still adequate for up-to-date machines, not to mention servers that are usually configured with 16 GB memory or more.

The memory consumption of our profiler for parallel Starbench programs (pthread version, 4 threads) is shown in Figure 2.11. Our profiler consumes 995 MB and 1920 MB memory on average using 8 and 16 threads for profiling, respectively. The consumption is higher than when profiling sequential benchmarks (505 MB and 1390 MB) because of the implementation of the lock-free queues, additional data structures to record thread interleaving events, and an extended representation of data dependences. However, the consumption is still moderate for an ordinary PC.



(a) Slowdowns of the data-dependence profiler on sequential NAS and Starbench benchmarks.



(b) Memory consumption of the profiler for sequential NAS and Starbench benchmarks.

Figure 2.9: Performance results of the data-dependence profiler on sequential NAS and Starbench benchmarks.

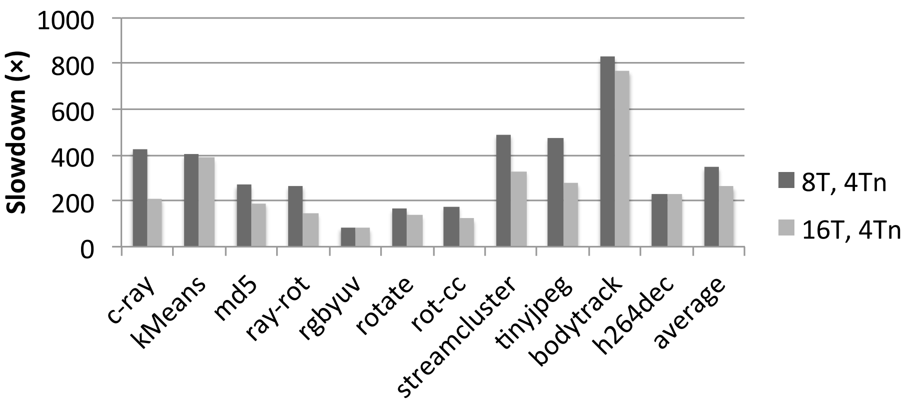


Figure 2.10: Slowdown of the profiler for parallel Starbench programs (pthread version, T = thread for profiling, T_n = thread for benchmark).

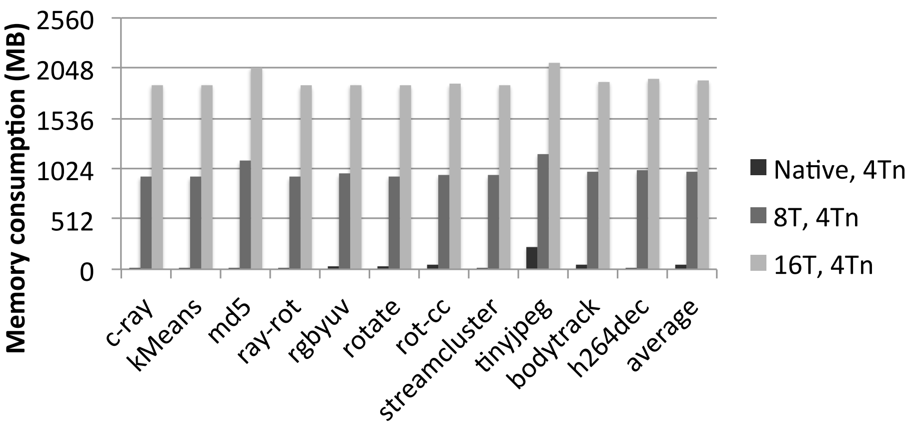
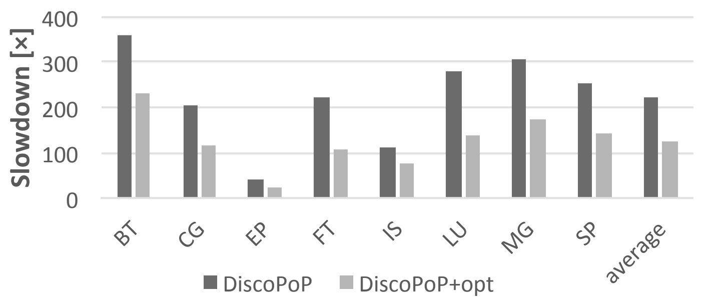
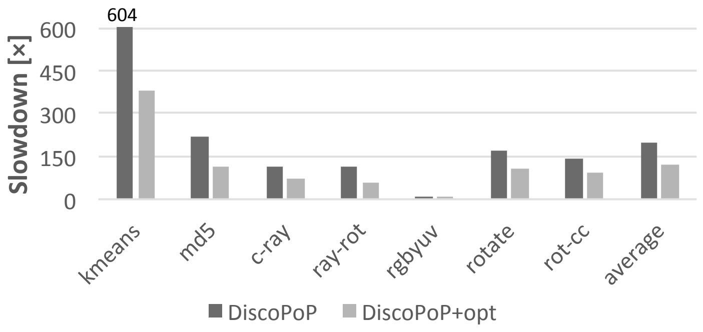


Figure 2.11: Memory consumption of the profiler for parallel Starbench programs (pthread version, T = thread for profiling, T_n = thread for benchmark).



(a) Slowdown of the DiscoPoP profiler applied to NAS.



(b) Slowdown of the DiscoPoP profiler applied to Starbench.

Figure 2.12: Slowdowns of the DiscoPoP profiler when applied to NAS and Starbench benchmarks with (DiscoPoP+opt) and without (DiscoPoP) skipping repeatedly executed memory operations.

Effectiveness of skipping memory instructions in loops

We evaluated the effectiveness of skipping memory accesses in loops in separate experiments so that the results are not affected by other optimization techniques. To set up a ground truth, shadow memories used in this section are based on non-approximate data structures, meaning no false positives or false negatives will be built. For simplicity, the experiments are done with the sequential version of the profiler.

Test cases are the SNU NAS Parallel Benchmarks 3.3.1 [84] (NAS), a suite of programs derived from real-world computational fluid-dynamics applications, and a few applications from the Starbench parallel benchmark suite [86], which covers programs from diverse domains, including image processing, information security, machine learning and so on.

Figure 2.12 shows the slowdowns of the data-dependence profiler when applied to NAS benchmarks and Starbench with (dp+opt) and without (dp) applying the mechanism of skipping memory operations that are repeatedly executed in loops. As shown, our method reduces the slowdown of data-dependence profiling in all of the test cases. The highest slowdown reduction appears with *FT* (52.0 %), and the lowest shows appears with *rot-cc* (31.1 %). On average, our method reduces the time overhead of data-dependence profiling by 41.3 %. The output after applying our optimization was compared to the original one, and no difference was observed.

Whether our method reduces the time overhead of data-dependence profiling depends on the computation pattern of the target application. Theoretically, the more work is done in loops or in any other repetitive manner, the more effective our method will be. If a program does not have any code sections that are executed more than once, which is obviously very uncommon for a real-world application, our method should actually bring a minor time overhead caused by condition checking. In the test cases *FT*, *LU*, and *CG*, the biggest hot spots are all loops. Applying our method to these test cases yields slowdown reductions of 52 %, 51 %, and 44 %, respectively.

The memory access pattern is another factor that can affect the effectiveness of our method. In the worst case, the accessed memory addresses change in every iteration, which means the profiling process cannot be paused. This usually happens when the computation is based on arrays or matricies. The results on four test cases *BT*, *IS*, *rotate*, and *rot-cc* are affected by this problem.

Our method introduces a minor overhead on the memory consumption of data-dependence profiling because of the variables created for the condition check. However, compared to the memory overhead of shadow memory, the memory overhead of our method can be ignored. In our experiments, one 64-bit integer (`lastAddr`) and two 32-bit integers (`lastStatusRead` and `lastStatusWrite`) are created for each memory operation. However, the number of memory operations is usually much smaller than the number of dynamic memory instructions due to loops and other code blocks that are repeatedly executed. For example, *kmeans* has 10^9 memory operations in total and iterates 300 times. Thus, the number of distinct memory operations in *kmeans* is roughly 3×10^6. With 16 bytes memory overhead each, our method results in about 50 MB memory consumption. The memory overhead of shadow memory, however, is almost ten times of that. The memory consumption of the state-of-the-art data-dependence profilers [75, 54] ranges from several hundred megabytes to several gigabytes. Trading 10 % extra memory for 30-50 % reduction of time overhead is preferred in most of the cases.

Statistics on skipped memory instructions

We also obtain statistics of the memory instructions that lead to data dependences but are skipped in each test case. As most of the data-dependence profilers do, read-after-read (RAR) dependences are not profiled in our experiment.

Table 2.7 shows the statistics. In each column group, percent shows how much percent of the dynamic memory instructions are skipped. As shown, on average 80.06 % of the memory instructions that lead to data dependences were skipped. It is surprising that the full data dependence set of an application can be obtained by profiling only 20% of its memory instructions or even less because those do not lead to dependences are ignored anyway. The results give

Table 2.7: Statistics of memory instructions that lead to data dependence but are skipped by the profiler when profiling NAS benchmarks and Starbench.

Benchmark	read			write			read+write		
	total	skip	percent	total	skip	percent	total	skip	percent
BT	743 969 748	535 241 484	71.94	104 153 401	23 600 681	22.66	848 123 149	558 842 165	65.89
CG	562 665 608	445 634 496	79.20	82 428 819	76 098 646	92.32	645 094 427	521 733 142	80.88
EP	1 268 263 496	1 227 090 185	96.75	528 633 275	470 479 986	89.00	1 796 896 771	1 697 570 171	94.47
FT	1 034 144 426	1 030 885 977	99.68	274 436 113	273 133 439	99.53	1 308 580 539	1 304 019 416	99.65
IS	26 061 226	21 549 539	82.69	10 596 042	7 791 594	73.53	36 657 268	29 341 133	80.04
LU	368 187 710	320 650 899	87.09	36 303 260	15 220 037	41.92	404 490 970	335 870 936	83.04
MG	66 160 096	54 647 704	82.60	5 876 449	3 166 502	53.88	72 036 545	57 814 206	80.26
SP	450 997 264	376 765 812	83.54	51 853 149	22 978 550	44.31	502 850 413	399 744 362	79.50
kmeans	1 124 603 733	734 002 549	65.27	225 500 303	198 374 067	87.97	1 350 104 036	932 376 616	69.06
md5	3 908 055	3 558 461	91.05	1 368 725	1 341 157	97.99	5 276 780	4 899 618	92.85
c-ray	1 251 777 658	810 820 904	64.77	264 217 429	127 746 061	48.35	1 515 995 087	938 566 965	61.91
ray-rot	500 462 138	282 670 259	56.48	133 222 408	63 474 244	47.65	633 684 546	346 144 503	54.62
rgbyuv	25 639 777	22 892 199	89.28	15 977 310	13 631 373	85.32	41 617 087	36 523 572	87.76
rotate	328 610 773	293 020 100	89.17	53 662 659	30 367 659	56.59	382 273 432	323 387 759	84.60
rot-cc	427 139 027	391 548 122	91.67	76 733 411	44 001 082	57.34	503 872 438	435 549 204	86.44
average	—	—	82.08	—	—	66.56	—	—	80.06

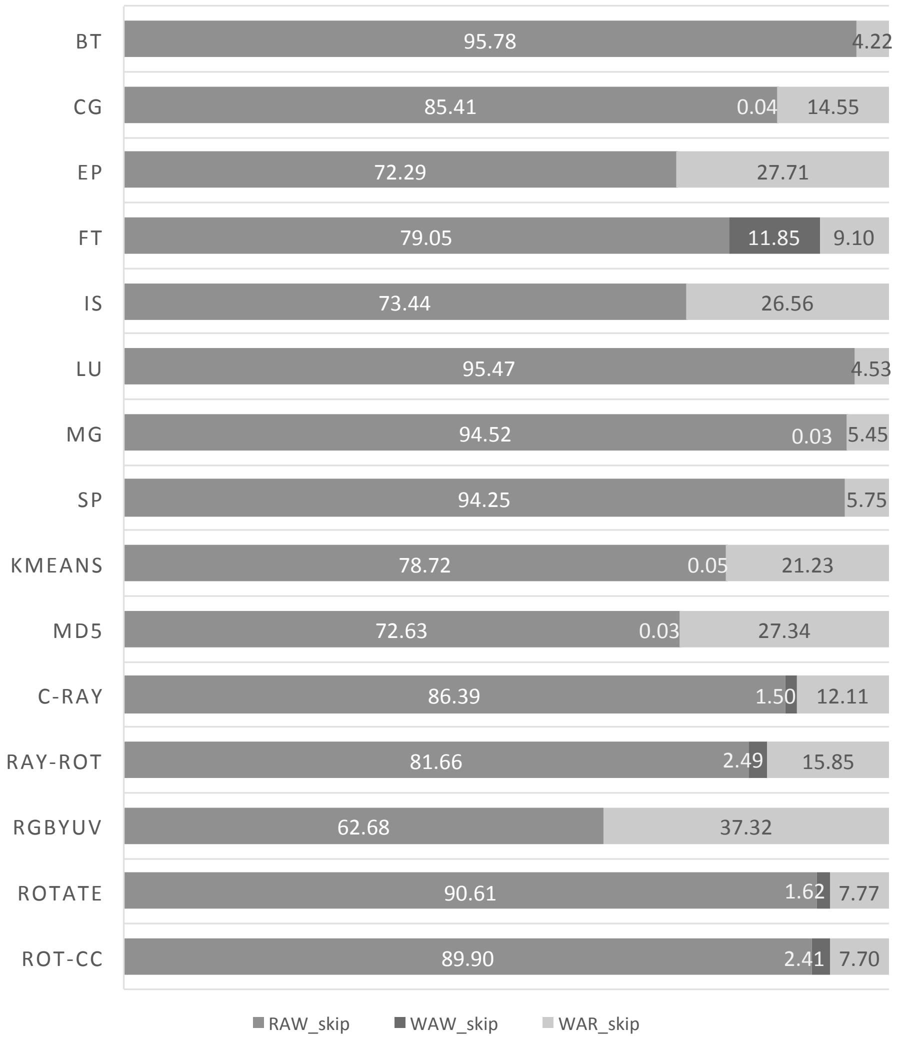


Figure 2.13: Distribution of skipped memory instructions according to the type of data dependences they would create.

us an insight of how much time were wasted in a classic data-dependence profiler that profiles identical data dependences over and over again.

Although on average about 80% of the memory instructions that lead to data dependence are skipped, the slowdown reductions shown in Figure 2.12 never achieve 60%. There are two reasons for this. First, in most cases, skipping a memory instruction means skipping the phase of building data dependences. Overhead is still incurred when updating the shadow memory. The second reason is that profiling a write instruction is more complex than profiling a read instruction, and the percentage of skipped write instructions (66.56 %) is less than the percentage of read instructions (82.08 %). When profiling a write instruction, we need to check both WAW and WAR dependences, while we need to check only RAW dependences when profiling a read instruction.

We also characterized the distribution of skipped memory instructions according to the type of data dependences they would create. Results are shown in Figure 2.13. “RAW_skip", “WAR_skip", and “WAW_skip" represent the percentage of skipped memory instructions that lead to read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependences, respectively. Take *BT* as an example, 95.78 % of the skipped memory instructions would lead to RAW dependences, while 4.22 % would lead to WAR dependences. In this case, none of the skipped memory instructions would result in WAW dependences.

In six benchmarks (*BT*, *EP*, *IS*, *LU*, *SP*, and *rgbyuv*), no skipped memory instructions would lead to WAW dependences. In eight benchmarks (*CG*, *MG*, *kmeans*, *md5*, *c-ray*, *ray-rot*, *rotate*, *rot-cc*), the percentages of skipped memory instructions that would lead to WAW dependences are below 2.50 %. The reason is straightforward: WAW dependences are rare in most programs. In our experiment, we build WAW dependence only for consecutive write instructions to the same address. Obviously, this is not a common way of writing programs. Surprisingly, the percentage of skipped memory instructions leading to WAW dependences in *FT* is more than 10 %.

```
for (k = 1; k < d3; k++) {
  dummy = randlc(&start, an);
  RanStarts[k] = start;
}
```

Figure 2.14: Write-after-write dependences are frequently built in *FT* because of the use of variable `dummy`.

We found some code snippets in *FT* that can explain this behavior, and one of these code snippet is shown in Figure 2.14. The variable `dummy` is used to store the return value of `randlc()`, but it is never used later on. Many write-after-write dependences are built because of the use of `dummy`. Similar code snippets appear at different places in `FT`. We believe that the percentage of skipped memory instructions that lead to WAW in *FT* should also be close to zero if dummy variables are removed.

The distributions shown in Figure 2.13 do not necessarily represent the distribution of data dependences for each benchmark. They reflect characteristic of the workload of a data-dependence profiler rather than its output. As shown, *BT*, *LU*, *MG*, *SP*, *rotate*, and *rot-cc* have similar workload distributions. In these benchmarks, skipped memory instructions that would lead to WAR dependence are around 4 % – 8 %. *EP*, *IS*, *kmeans*, and *md5* form another group, with 21 % – 28 % skipped memory instructions that would lead to WAR dependences. *CG*, *c-ray*, *ray-rot*, *rotate*, and *rot-cc* are similar to one another, with 7 % – 16 % that would lead to WAR dependences, and a small percentage (<2.5 %) that would lead to WAW dependences. Again, *FT* belongs to none of the two groups due to the high percentage of memory instructions that would lead to WAW dependences, and the same applies to *rgbyuv* because of the high percentage of memory instructions that would lead to WAR dependences.

2.6 Summary

We started our tour of data dependence analysis from reviewing advantages and disadvantages of static and dynamic approaches. Static approaches are fast and necessary to enable advanced code optimization, but conservative on dynamically allocated memory, pointers, and dynamically calculated array indices. Dynamic approaches, on the other hand, cover all dynamic memory instructions in one execution, but incur high runtime overhead in terms of both time and space.

In this thesis, we present the DiscoPoP profiler, a generic data-dependence profiler with practical overhead for both sequential and parallel programs. To achieve efficiency in time, the profiler is parallelized, taking advantage of lock-free design. To achieve efficiency in space, the profiler leverages signatures, a concept borrowed from transactional memory. Both techniques are application-oblivious, which is why they do not restrict the profiler's scope in any way. The profiler also produces the Program Execution Tree (PET) to support parallel pattern detection. Together with other optimization techniques such as variable lifetime analysis and dependence merging, DiscoPoP profiler achieves a slowdown of 86 on average for NAS and Starbench benchmarks, with on average memory consumption of 1020 MB.

An aggressive optimization that skips memory instructions in loops lower the time overhead of profiling further. Without any other optimization technique, skipping memory instructions in loops shortens the profiling time by 41.3% without incurring significant space overhead. Moreover, it provides interesting insights into the distributions of memory instructions and data dependences in NAS and Starbench benchmarks.

3 Computational Units

Existing approaches limit the scope of their search for parallelism to predefined language constructs. For example, the method proposed in [63] is designed to find parallelism only between functions. Other approaches such as [10, 13, 87] are more flexible in that they consider multiple and also in principle arbitrary construct types. Common to all of them, however, is the restriction that they can only answer questions of the following type: (i) Can a construct or region with given entry and exit points be parallelized? (ii) Can a construct with given entry and exit points run asynchronously with other parts of the program? Thus, their underlying strategy first identify the regions of investigation, usually following the structure of the programming language, and then reason about their parallelization.

In contrast to the classic methods, we try to cover parallelism that is not aligned with language constructs. This means we need a new representation of a program where the smallest unit does not contain any unexplored parallelism, and this unit may not be aligned with language constructs. We should analyze dependences among such units for parallelism, and it should be also possible to utilize such units from fine grain to coarse grain. In this chapter, we define the *computational unit* (CU) to serve as the smallest unit mentioned above. We show algorithms to construct CUs, as well as our new representation of program execution: the CU graph.

3.1 Definition

We define a new language-independent code-granularity level for both program analysis and reflection of parallelism, which we call computational units (CUs). A CU is the smallest unit of code we map onto a thread, that is, while potentially running in parallel to other CUs, a CU itself is not subject to any further (internal) parallelization—at least not within the scope of our method.

The notion of CUs was inspired by our earlier work [88], where a variation of this concept was applied to detect data races on correlated variables. In this thesis, a CU is a collection of instructions following the *read-compute-write* pattern: a set of variables is read by a collection of instructions and used to perform computation, then the result is written back to another set of variables. We call the two sets *read set* and *write set*, respectively. The two sets do not have to be disjoint. The load instructions reading the variables in the read set form the *read phase* of the CU, and the store instructions writing the variables in the write set form the *write phase* of the CU.

Definition of a CU. Given a code section C, let GV_c be the set of variables that are global to C. Let I_x and O_x be the sets of instructions reading and writing variable x, respectively. C is a computational unit if it satisfies the following condition:

$$\forall v \in GV_c, I_v \rightarrow O_v. \tag{3.1}$$

"$\rightarrow$" is the happens-before relationship [89]. Note that "$\rightarrow$" is defined on a single variable. Read and write operations on two different variables can be executed in any order if there is no indirect data dependence. It does not conflict with the concept that a CU does not contain any unexplored parallelism: instruction-level parallelism is explored and automatically utilized by the hardware.

Following the definition, the read phase and the write phase of a CU are $\cup_{v \in GV} I_v$ and $\cup_{v \in GV} O_v$, respectively. When considering only the read phase and the write phase, a CU does not hide any true dependences (RAWs) inside that are essential to the data flow of the program, meaning all relevant parallelization opportunities can be analyzed on the level of CUs. Moreover, via control-flow analysis we ensure that CUs never cross the boundaries of a control region. While being small enough, typically not covering more than a few lines of code, to express very fine-grained parallelism, this property ensures that CUs can be easily combined to higher-level constructs such as loops or functions. This allows the reflection of parallelism to be lifted to arbitrarily high levels of abstraction, making our approach general. Note that CUs never crossing control boundaries is not in conflict with the idea that CUs may not be aligned with language constructs: a CU may be part of a construct.

3.2 Construction

The definition of a CU distinguishes variables that are global and local to a code section. In Section 3.2.1, we show how we distinguish the two categories of variables. Control dependences are also important since CUs are not allowed to cross control-region boundaries. If the source code of the target program is available, obtaining control dependences is trivial since every ordinary compiler is able to perform control-flow analysis on the source code. However, if only the binary of the target program is available, obtaining control dependences can be difficult because the original control structures can only be inferred from the binary code. In this thesis, we present a method to obtain control dependences when the source code of the target program is not available, which is described in Section 3.2.2. Finally, the CU construction algorithm is described in Section 3.2.3.

3.2.1 Global and Local Variables

The first task in constructing CUs is to determine the variables that are global to a control region. For this reason, we determine global variables of a control region by analyzing variable scope information, which is available in any ordinary compiler. Note that the global variables in the read set and the write set do not have to be global to the whole program. They can be variables that are local to an encapsulating scope, but global to the target scope.

In the LLVM IR, metadata that conveys extra information about the code to the optimizers and code generator is attached to program instructions. One example application of metadata is source-level debug information. There are more than twenty kinds of specialized metadata structures, called metadata nodes in LLVM IR, among which we are interested in two: `DIGlobalVariable` and `DILocalVariable`.

`DIGlobalVariable` nodes represent global variables in the source program. A global integer named "`foo`" has the following metadata node:

```
!0 = !DIGlobalVariable(name: "foo", linkageName: "foo", scope: !1,
                       file: !2, line: 7, type: !3, isLocal: true,
                       isDefinition: false, variable: i32* @foo,
                       declaration: !4)
```

Variables global to the whole program are certainly global to any of the control regions. They are always included in the `globalVars` set. Global variables can be obtained through the `globals` field of the `DICompileUnit` metadata node.

We further analyze all the `DILocalVariable` nodes, which represent local variables in the source program. Examples of `DILocalVariable` nodes are as the following:

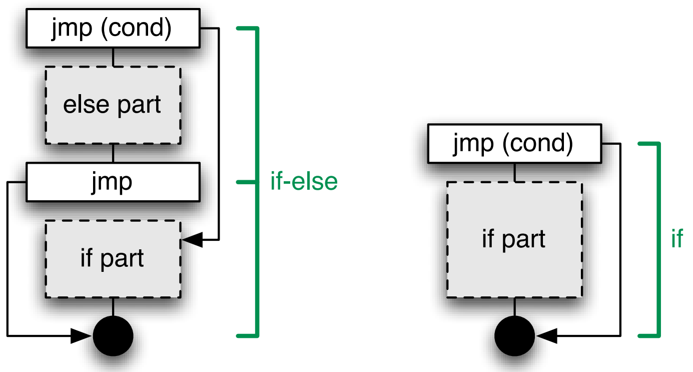
```
!1 = !DILocalVariable(name: "x", arg: 2, scope: !4, file: !2, line: 7,
                      type: !3)
!2 = !DILocalVariable(name: "y", scope: !5, file: !2, line: 7, type: !3)
```

If the `arg` field is non-zero, then this variable is a subprogram (function) parameter. Analyzing `DILocalVariable` nodes gives variables that are local to a function but not any local scopes nested inside the function because the LLVM IR has only two syntactic scopes – a global scope and a function scope. When it is necessary to construct CUs within a function, we have to record all the variables that are defined and used in different code sections. If a variable is defined and used in only one code section, it is local to the code section. Otherwise, the variable is global to all the code sections.

It is worth mentioning that defining CUs based on the notion of global variables is slightly stricter than necessary. Imagine the situation where a variable named g is defined global to the whole program but used only in a relatively small code section. Because g is globally defined, it has to be included in the `globalVars` set of any code section. A better option to define a CU is based on "communicating" variables – those variables causing data dependences among CUs. Global variables are an approximation to communicating variables as there may be global variables that do not cause any data dependences. However, such definition brings a circle: CUs are defined on communicating variables, while communicating variables are defined based on CUs. For this reason, global variables are used since they can be obtained in a much easier way. Another solution shares the similar concept of expectation maximization (EM) method in machine learning. First, let the set of global variables be the initial guess of the set of communicating variables and build CUs based on the guess. After the CUs are built, we calculate the set of communicating variables based on the CUs. The new set of communicating variables is then used as the improved guess in the second iteration of the same process. The algorithm iterates until the the of communicating variables does not change any more.

3.2.2 Dynamic Control-Dependence Analysis

In this section, we introduce the method of to obtain control dependences when the source code of the target program is not available. A control dependence between two instructions op_i and op_j exists if op_j is conditionally guarded by op_i. Without source code, to decide whether an instruction is conditionally guarded we need to know the *re-convergence point*, which is the point where the different branch alternatives end and unconditional execution resumes. To circumvent that dynamic analysis has usually no access to the complete control-flow graph because not all branches of the program are actually executed, we use a look-ahead technique. Before the real branch is executed, we follow every possible branch first and terminate this look-ahead once we encounter the re-convergence point, which is the first instruction that comes after the basic blocks defined by the branch alternatives. Our method described in this section is implemented on top of Valgrind [14] because it disassembles basic blocks belonging to all branch alternatives when a branch is encountered. This feature greatly reduces the difficulty of implementing our method. We traverse the blocks representing the the branch alternatives without actually executing them, simply following jump instructions until we find the re-convergence point. An example of finding the re-convergence point of an `if-else` and a simple `if` statement is shown in Figure 3.1.



(a) `if-else` construct (b) `if` construct

Figure 3.1: Finding the re-convergence point (solid black circle).

We instrument jump operations and maintain a stack where we record the scope of the currently active control regions. When we encounter a control region, we push a triple `<start, type, end>` onto the stack. When we leave a control region, we remove the topmost entry. We determine the type of a region (branch or loop) and its re-convergence point using our look-ahead technique. We also respond to function calls. If a function is called inside a control region, we simply keep the current top of the stack untouched and continue pushing control regions we find in the callee on the stack. When the callee function returns, all control regions it contains should also terminate and the calling region is again on the top of the stack.

```
1   for (i = 0; i < MAX_ITER; i++) {
2       if (i == 0)
3           x = 3
4       a = x + rand() / x
5       b = x -  rand() / x
6       x = a + b
        ...
k   }
```

Figure 3.2: A simple code example.

The example shown in Figure 3.2 illustrates our algorithm. It contains several control and data dependencies. Applying our algorithm for finding re-convergence points to the example yields Figure 3.3, where the re-convergence points (solid black circles) are exactly the first lines encountered after the corresponding control structure ends.

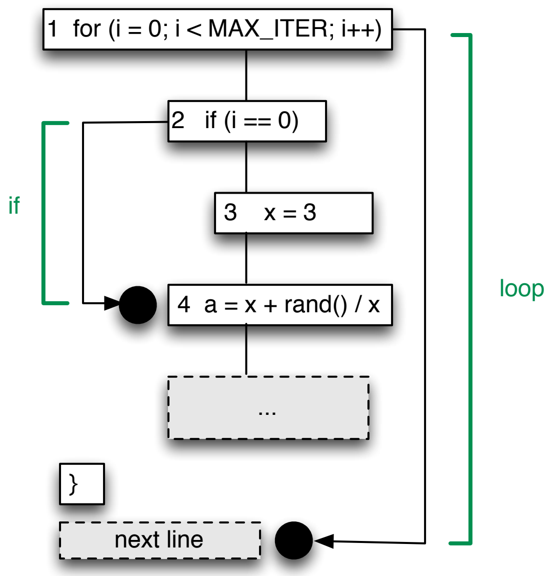


Figure 3.3: Re-convergence points of the example in Figure 3.2.

3.2.3 The Algorithm of Building CUs

Since a CU may not be aligned with a predefined language construct, it is not sufficient to build CUs according to the control structure of a program. There are two ways to build CUs: the bottom-up approach that builds CUs from instructions and merge them as bigger CUs, and the top-down approach that builds CUs from functions and seeks for opportunities to divide the CU into smaller CUs when the whole code section is not a CU.

The bottom-up approach

Imagine the execution of the program as a sequence of instructions $\{op_0, op_1, \ldots, op_{n-1}\}$. Let R be the subsequence of these instructions that belongs to the current control region, not including instructions belonging to regions nested inside. Let us assume we have already processed all instructions in R up to but not including op_i. Then we can apply the following algorithm to op_i and all remaining instructions in R:

1. When we encounter a new control region nested inside the current one, we suspend the current region until we have processed the region nested inside.
2. When an instruction $op_i \in R$ is executed:
   - If the variable v that op_i operates on is defined in R (local to R), ignore op_i and exclude dependences involving v from building CUs. Otherwise we build a CU that just contains op_i.
   - We *merge* the CU of op_i with all the CUs of instructions $op_{j<i} \in R$ that op_i directly depends on via anti-dependences.
   - If op_i directly depends on $op_{j<i}$ via a true data dependence, we create a directed edge from the CU of op_i to the CU of op_j, expressing that op_i truly depends on op_j. Note that op_j does not necessarily have to be an element of R.
   - If op_i is the first write of a variable in the program, we mark it as *initialization*.
   - Repeat the algorithm for all remaining instructions in R.
3. At the end, *merge* the CUs of all adjacent *initialization* operations into one INIT node. Adjacent means that their instructions form a contiguous subsequence of R.

The advantage of the bottom-up approach is that it builds CUs on-the-fly. It is a purely dynamic approach that does not rely on any pre-execution static analysis. In step 2, it checks whether a variable is defined in an instruction right after executing the instruction, and utilizes such information to distinguish variables that are local to the current control region. It always builds a CU from a single instruction, and merges it with previous CUs if it depends on previous CUs via anti-dependences (WAR). This is consistent with the definition that the read phase happens before the write phase. If the new CU depends on previous CUs via true dependences (RAW), it means the read-after-write pattern is violated.

There are two main disadvantages of the bottom-up approach, both rooted in the merge step. First, due to the complexity of the instruction stream, true dependences are frequently observed, meaning the approach produces a huge amount of CUs that represent only a few instructions. This is fine according to the definition, but practically not helpful for parallelism discovery. Second, the frequent merging operation incurs high time overhead, making the algorithm slow.

Since the algorithm is usually performed on-the-fly, it slows down the original program by an unacceptable factor.

For a detailed description of the bottom-up approach and the parallelism discovery method based on it, please refer to Li et al. [57]. Results show that CUs produced by the bottom-up approach are too fine to discover coarse-grained parallel tasks. For this reason, we developed another CU construction algorithm that works in a top-down manner.

The top-down approach

Algorithm 3 shows the top-down CU construction algorithm. It starts from functions, examining whether the whole control region satisfies the definition of a CU. It relies on a set `globalVars` containing all the variables that are global to the control region. The algorithm first builds the read phase ($\cup_{v \in GV} I_v$) and the write phase ($\cup_{v \in GV} O_v$) following the definition. It then checks whether the the read phase happens before the right phase, satisfying the read-compute-write pattern. If so, the whole control region is a CU. Otherwise, the algorithm records all the read instructions that violate the read-compute-write pattern, and tries to build CUs for all code snippets within the region that are separated by the violating read instructions. In this way, multiple CUs may be build for a control region, and parallelism may be explored among these CUs.

The top-down approach constructs coarse-grained CUs first. Coarse-grain parallelism is usually utilized using parallel patterns like master-worker, fork-join, pipeline, and so on. In contrast to the instruction-level parallelism discovered by the bottom-up approach, thread-level parallelism is not explored and automatically utilized by the hardware. Thus, discovering coarse-grain thread-level parallelism is more interesting, and more beneficial to users.

We start from functions because they are the biggest constructs that could potentially resemble the concept of a CU in a program. A function receives arguments, performs computation, and returns results, which follows the read- compute-write pattern by nature. We cannot directly treat every function as a CU because a function may have side effects, like modifying global status. And a function that has side effects is very common in C and C++.

The top-down approach is fast. It simply checks whether a control-region satisfies the read-compute-write pattern. However, it requires pre-execution static analysis to produce the `globalVars` of each control region. The top-down approach is better performed off-line because it deals with a whole control region at a time. In the parallelism discovery framework described in this thesis, the top-down CU construction algorithm is used and implemented as a compiler pass, which is called after global variable analysis but before instrumentation.

3.2.4 Example of Building CUs Using the Top-Down Appoach

In this section, we show an example of building CUs for a simple code snippet. The example is shown in Figure 3.4.

```
for each region R in the program do
    globalVars = variables that are global to R
    violated = false
    for each variable v in globalVars do
        if v is read then
            readSet += v
            for each instruction Irv reading v do
                readPhase += Irv
            end
        end
        if v is written then
            writeSet += v
            for each instruction Iwv writing v do
                writePhase += Iwv
            end
        end
    end
    violateSet = empty
    for each variable v in readSet do
        for each instruction Ir reading v do
            for each instruction Iw writing v do
                if Ir happens after Iw then
                    violated = true
                    violateSet += Ir
                end
            end
        end
    end
    if violated == false then
        cu = new computational unit
        cu.readSet = readSet
        cu.writeSet = writeSet
        cu.readPhase = readPhase
        cu.writePhase = writePhase
        cu.computationPhase = (instructions in R) - (readPhase + writePhase)
    end
    else
        for each read instruction Iv in violateSet do
            build CU for instructions do not belong to any CU before Iv
        end
    end
end
```

Algorithm 3: The algorithm of building CUs (top-down).

```
1 int x = 3;
2 for (int i = 0; i < MAX_ITER; ++i) {
3     int a = x + rand() / x;
4     int b = x - rand() / x;
5     x = a + b;
6 }
```

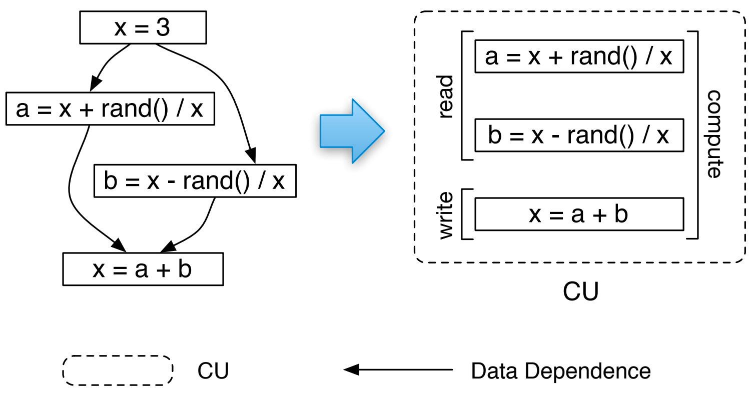


Figure 3.4: Building a CU.

In this example, *readSet* and *writeSet* are both {x}. Each loop iteration calculates a new value of x by first reading the old value of x and then by computing a new value via local variables a and b. Finally, the new value is written back to x. For a single iteration, all the reads of x happen before the write to x. Following the read-compute-write pattern, lines 3–5 are in one CU, as shown in Figure 3.4. At the source-line level, the compute phase (line 3–5) of the CU overlaps with its read phase (line 3–4) and write phase (line 5). At the instruction level, the three phases are separate to one another. If a and b were declared outside the loop, then they would be considered global to the loop as well. This would mean the loop would be made up of two CUs with lines 3-4 being one CU and line 5 being the second CU.

3.2.5 Special Variables in Building CUs

Function parameters and return values deserve special treatment when determining the read set and the write set of a function. We treat them as follows:

- All function parameters are included in the read set
- Function parameters passed by value are *not* included in the write set
- The return value is stored in a virtual variable called `ret`, and `ret` is included in the write set

The first rule is obvious. We follow the second rule because parameters passed by value are copied into functions, thus modifications to them do not affect their original copies. The return

```
1  // source code
2  int a = 0, b = 1, c = 0;
3  c = a + b;
4
```

```
1   ; LLVM IR
2   %a = alloca i32, align 4
3   %b = alloca i32, align 4
4   %c = alloca i32, align 4
5   store i32 0, i32* %a, align 4
6   store i32 1, i32* %b, align 4
7   store i32 0, i32* %c, align 4
8   %4 = load i32* %a, align 4
9   %5 = load i32* %b, align 4
10  %6 = add nsw i32 %4, %5
11  store i32 %6, i32* %c, align 4
12
```

Figure 3.5: The LLVM IR of a sample C++ code section.

value must be included in the write set. However, it is common that the return value does not have a name. That is why we always call it `ret` when building a CU statically.

Loop iteration variables also require special treatment. Specifically, the following rules apply to them:

- By default, loop iteration variables are considered as local to loops
- If a loop iteration variable is written inside the body of a loop, it is considered as global to the loop

We treat loop iteration variables in a special way because inter-iteration dependences on them in loop headers do not prevent parallelism. However, if their values are updated inside the loop body, the normal iteration process may be interrupted, and dependences on loop-iteration variables must be taken int o account when deciding whether the loop can be parallelized.

3.3 Granularity

It is important to understand that CUs are built by analyzing IR statements of the target program. For an IR in single assignment (SA) form, it is common that a source language statement is translated into multiple IR statements. Figure 3.5 shows an example of representing a simple C++ code section in LLVM IR, which is in static single assignment (SSA) form.

When talking about read and write instructions in the CU construction algorithm, we refer to the load and store instructions and many other memory access instructions shown on the right side of Figure 3.5. In this example, source line 3 is translated into four IR statements at line 8 – 11, which follows the read-compute-write pattern perfectly. We can build a CU out of these four IR statements if we assume the variables a, b, and c are global to them. When using the bottom-up approach, CUs are built at such granularity and merged if necessary. When mapping this CU back onto the source code, it contains only line 3. This is an example where a CU contains "only a few instructions".

Table 3.1: Possible forms of an edge in a CU graph.

	of the same CU	of different CUs
from write phase to read phase (RAW)	✓	✓
from read phase to write phase (WAR)	–	✓
from write phase to write phase (WAW)	–	✓

Note that the two IR statements at lines 8 and 9 can be executed in parallel since there is no data dependence between them. It is very common for compilers and hardware to schedule load instructions in such a way that a long stall is avoided. However, we put them into the same CU, and ignore the parallelism between instructions. As mentioned in Section 3.2.3, instruction- level parallelism is automatically explored and utilized by the hardware with the help of compilers.

Now it is clear why the bottom-up approach is not preferred: it produces too many fine-grained CUs. Practically, it does not help much to know that two source lines can run in parallel, especially if such suggestions form the majority. The users want to explore parallelism among functions, loops, and potential tasks, but usually not among individual source lines.

Note that the top-down approach could also eventually get into the same fine granularity as the bottom-up approach. The difference is that, by setting a threshold, the top-down approach can quickly get avoid of analyzing code sections that do not form big CUs, while the bottom-up approach must try its very best to merge no matter in what situation. The top-down approach is more flexible: it stops at a level where finer-grain parallelism in not interesting anymore, or goes down to cover fine-grained parallelism if coarse-grained parallelism is not found.

3.4 Computational Unit Graph

CUs and the data dependence among them form a *CU graph*. Data dependences among CUs are always among instructions in the read phases and the write phases. Given that the number of variables global to a code section is usually much smaller than the number of local variables, a CU graph is a significant simplification of the classic dependence graph. Table 3.1 summarizes the possible forms of an edge in a CU graph.

Two forms of edges are not included in a CU graph, the edges starting from the read phase and ending at the write phase (WAR-dependence edges) of the same CU, and the edges starting from the write phase and ending at the write phase (WAW-dependence edges) of the same CU. They are not included because they provide no contribution to parallelism discovery.

The WAR-dependence edges of the same CU indicate that the read set and the write set of the CU share common elements. Having such dependences or not, the write phase and read phase of a CU cannot be executed in parallel because the “read-compute-write” pattern means internal RAW dependences that force the write phase to be executed after the read and the compute phase. For this reason, the WAR-dependence edges of the same CU are not included.

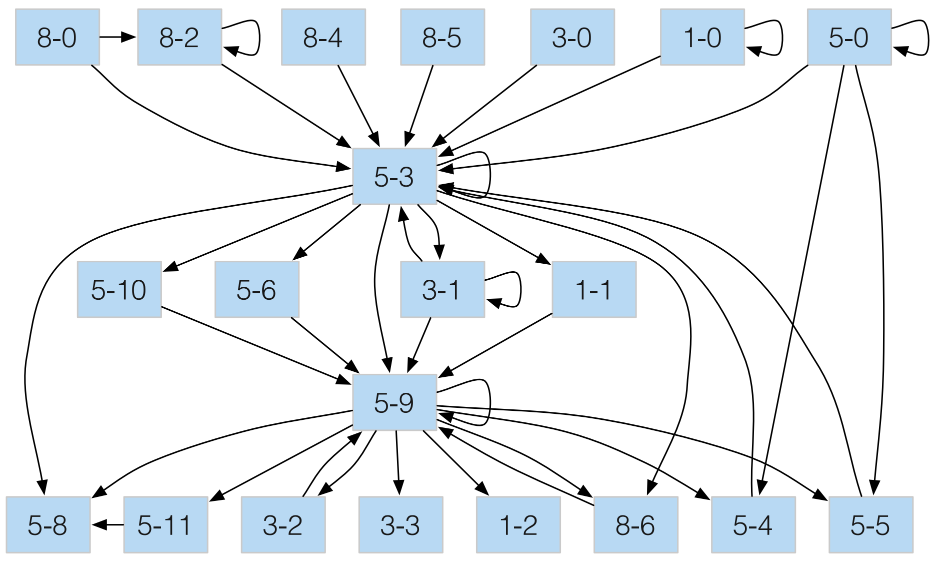


Figure 3.6: Part of the CU graph of *rot-cc*.

The WAW-dependence edges of the same CU mean the values of some variables are overwritten within the same write phase. Overwriting is OK as long as it happens in the same write phase because they are in a sequential order, and compilers are good at discovering such optimization opportunities by using static analyses like def-use chain analysis and constant propagation. For this reason, WAW-dependence edges of the same CU are not included.

On the other hand, the RAW-dependence edges of the same CU must be included. If a CU has such edges, it means two things: 1) the CU has been executed multiple times, and 2) in each execution, the CU uses the outputs from the last execution as the input of the current execution. Obviously, it is the most common iterative computation pattern. Whether the RAW-dependence edges exist gives a clue about whether the iterations are independent from one another. Thus, the RAW-dependence edges of the same CU are included.

All the three kinds of edges between different CUs are include in a CU graph. Again, RAW-dependence edges must be included as they reveal the true dependences that cannot be easily broken. Whether the WAR-dependence and WAW dependence edges can be removed depends on the semantics of the program. If the variables being written to can be renamed without violating these semantics, it is possible to remove these edges in order to explore more parallelism. Currently, the user has to decide whether it is safe to remove these edges.

Figure 3.6 shows a part of the CU graph of *rot-cc*, a benchmark from the Starbench parallel benchmark suite [86]. Numbers in vertices are CU IDs in the format of `module ID - local CU ID`. The CU graph shows all the main computational units and only the RAW-dependence edges, that is, the true data dependences that cannot be broken. In this example, CUs are built using the top-down approach. The figure shows that the program can be organized in a three-step manner, with two computations serving as barriers. Moreover, part of the computations in each step can also run in parallel, such as CU 8-4 and 8-5, CU 5-10 and 5-6.

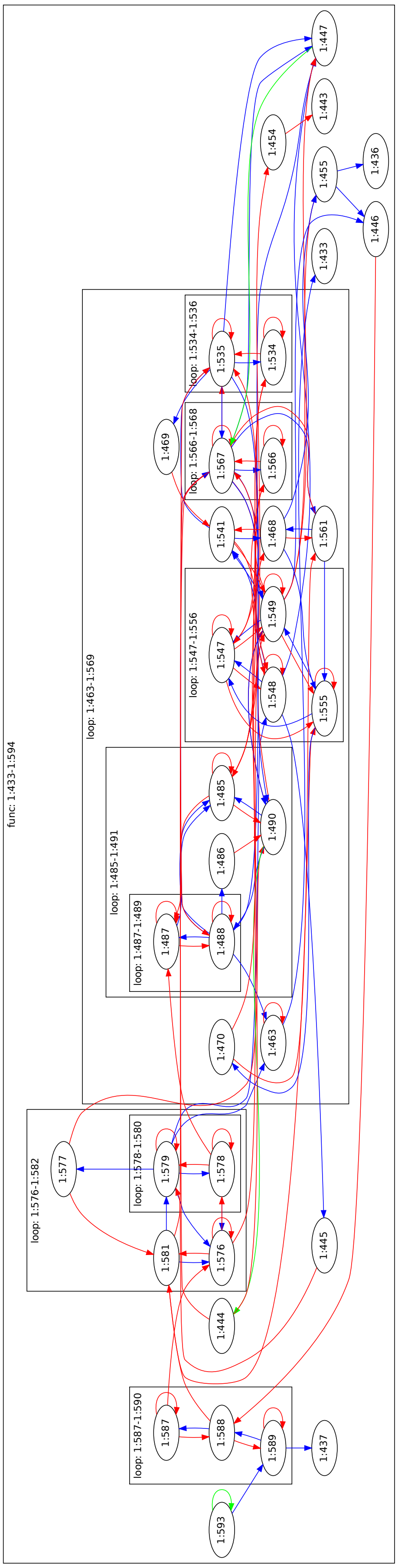


Figure 3.7: Part of the CU graph of *CG* combined with control region information.

To detect loop parallelism and parallel patterns [90, 91], it is useful to combine the control region information of the PET and the CU graph. Figure 3.7 shows a combined graph of a function in *CG*, a benchmark from NAS Parallel Benchmarks [85]. The CU graph contains all the CUs belonging to the function and three kinds of data dependences (red – RAW; blue – WAR; green – WAW). In this example, CUs are built using the bottom-up approach. Obviously, the combined graph is much more complex, and it is almost impossible for users to manually explore the parallelism it contains.

3.5 Computational Units and Pure Functions

It is mentioned in Section 3.2.3 that a function cannot be directly treated as a CU because it may have side effects. This suggests that the notion of a CU is related to a pure function. A pure function is surely a CU, but the inverse is not true. In computer science, we say "pure functions" when we refer to functions that work in a mathematics' way, and we say "functions" when we refer to procedures in programming. The difference is that, a pure function is a function where the return value is only determined by its input values, without observable side effects.

A function or expression is said to have a side effect if it modifies some external state or has an observable interaction with calling functions or the outside world. For example, a particular function might modify a global variable or static variable, modify one of its arguments, raise an exception, write data to a display or file, read data, or call other side-effecting functions [92].

Now it should be clear that a pure function is a CU because it does not rely on any global variable. The return value is the only thing visible to the outside world, and it is included in the write set (see Section 3.2.5).

Consider a CU that is not a function. Its read set and write set contain variables that are global to itself. Thus, it depends on external status, and may modify a global variable as well. To make a CU resemble a pure function, we have to prevent it from modifying variables that are global to it. Moreover, to guarantee that the computation always yields the same result, its read set cannot be modified by any other CU, either. Thus, a CU resembles a pure function if it satisfies the following conditions:

- Variables in its read set cannot be modified by itself nor by other CUs
- The special variable `ret` is the only variable allowed in its write set, if not empty
- The computation of the CU always yields the same result if the values of the variables in its read set do not change

Now it is clear that a CU is a weaker concept than a pure function. A purely functional program can be represented by a data-flow graph. A CU graph is similar, but it takes side effects of imperative programs into consideration. A CU graph reveals how the side effects of a CU affect other CUs so that users can develop improvement strategies. Basically, a CU-based parallelism discovery method encourages a purely functional programming style. The more pure functions, the easier parallelism can be revealed.

3.6 Summary

In this chapter, we introduce computational units, which represent the smallest units of a program that do not contain parallelism that are worth parallelizing in thread level. Computational units may not be aligned with predefined language constructs. A CU is a code section that follows the read- compute-write pattern: a set of variables is read by a collection of instructions and used to perform computation, then the result is written back to another set of variables.

We utilize metadata nodes in LLVM IR to determine variables global to a code section, and use static control-flow analysis to obtain control region boundaries. When the source code is not available, we obtain control region boundaries by finding the re-convergence point dynamically. CUs can be built in two different ways: bottom-up and top-down. Although the bottom-up approach can be performed on-the-fly, it produces a huge amount of single-instruction CUs, which produces distractive parallelism discovery results. In contrast, the top-down approach produces coarse-grained CUs that are suitable for detecting task-level parallelism. However, it requires pre-execution static analysis to obtain the set of global variables.

The CUs of a program and the data dependences among them form a CU graph. Two forms of edges are not included in a CU graph, the WAR-dependence edges that start from and end at the same CU, and the WAW-dependence edges that start from and end at the same CU. They are not included because they do not contribute to parallelism discovery. A CU graph shares a similar concept with the data-flow graph in functional programming. However, a CU graph also reveals side effects of CUs, making it a useful tool to discover parallelism in imperative programs. The fewer edges a CU graph has, the more parallelism the target program has. Basically, CU-based parallelism discovery method encourages a purely functional programming style.

4 CU-Based Parallelism Discovery

As described in Section 1.3.2, semi-automatic parallelism discovery tools try to locate the potential parallelism in sequential programs rather than show data-dependence graph. Unlike previous approaches, we introduce the concept of computational units (CUs) and represent a sequential program as a CU graph instead of the traditional dependence graph. In this chapter, we present parallelism discovery methods based on CU graphs, covering both parallelism in loops and parallel tasks. A ranking method is also presented to help users focus on the most promising parallelization opportunities. Evaluation results and a short discussion on limitations are presented at the end of this chapter.

4.1 Parallelism in Loops

Loops have been the main optimization targets in computer programs for decades. Many optimization passes in compilers are related to loops, such as *loop unrolling*, *loop fusion*, *loop-invariant code motion*, *hoisting*, and many others. [6] In the era of parallel programming, loops are the main targets of exploring parallelism. The key problem is to determine whether there are dependences between different iterations of a loop. The answer to this question divides loops that can be parallelized in two categories: *DOALL loops* and *DOACROSS loops*.

4.1.1 DOALL Loops

A loop can be categorized as a DOALL loop if there is no inter-iteration dependence. For nested loops, whether an inner loop is DOALL or not does not affect outer loops. This is the easiest type of parallelism to be discovered since it only requires verification whether there is an inter-iteration dependence among the CUs belonging to the body of the target loop.

When checking inter-iteration dependences, we check read-after-write (RAW) dependences only. The condition is relaxed because usually inter-iteration write-after-read (WAR) and write-after-write (WAW) dependences do not prevent parallelism (suppose a variable is always assigned a new value at the beginning of each iteration). This may lead to false positives, but we expect that false positives are rare. Thus, our algorithm detecting DOALL loops is optimistic. Note that data dependences on loop-iteration variables are already taken care of by the special treatment described in Section 3.2.5.

Rule of determining DOALL loops

A loop is classified as a DOALL loop if there is no inter-iteration RAW dependence.

```c
for (i = 0; i < nz; i++) {
    for (k = 0; k < ny; k++) {
        for (j = 0; j < nx; j++) {
            y[i][k][j] = dcmplx_mul2(y[i][k][j], twiddle[i][k][j]);
            x[i][k][j] = y[i][k][j];
        }
    }
}
```

Figure 4.1: The nested loops in function `evolve` of the SNU NAS benchmark *FT*.

DOALL loops are common in benchmarks performing numerical computations. Figure 4.1 shows a loop nest (`auxfnct.c`, line 180) in the SNU NAS benchmark *FT*. In this example, the three loops iterate over a 3D-array, each along a different dimension. The innermost loop multiplies two complex numbers from two arrays `y` and `twiddle` at the same index. The result is stored to both array `y` and `x`, also at the same index. Obviously, computations at different locations of the arrays are independent from one another. Thus, there are no inter-iteration dependences in all the three loops, meaning they are all DOALL loops.

Following the rule for DOALL loops mentioned above, our approach reports all the three loops as DOALL loops. However, it may not be a good idea to parallelize all of them. Generally, nested parallelism requires careful consideration. The overhead of thread management may be high if the number of threads is big, and the workload of a single task has to be big enough so that parallelization yields speedup rather than slowdown. The users must be aware of this problem since classic hotspot profiling techniques do not distinguish the outermost and innermost loop (both are hotspots with roughly the same execution time). We will see a similar situation when discussing SPMD-style tasks in Section 4.2.1.

4.1.2 DOACROSS Loops

When a loop has inter-iteration dependences, it is possible to further analyze the dependence distances of the inter-iteration dependences to discover DOACROSS [93] loops. A DOACROSS loop has inter-iteration dependences, but the dependence are not between the first line of an iteration and the last line of the previous iteration. This means in a DOACROSS loop, iterations are not independent but can partly overlap with one another, providing parallelism that can be utilized by implementing reduction or pipeline. Dependence distances can be easily measured since data dependences are indexed by source line numbers.

Rule of determining DOACROSS loops

A loop is classified as a DOACROSS loop if it is not a DOALL loop, and there is no inter-iteration dependence that starts from the read phase of the first CU (in single-iteration execution order) and *ends at the write phase of the last CU of the loop body. Note that the first CU and the last CU can be the same.*

Many loops are classified as DOACROSS loops due to inter-iteration dependences on one or more variables used for reduction. As an example, Figure 4.2 shows a loop in the *nqueens* benchmark from BOTS. The variable `*solutions` is used for accumulating results returned from function `nqueens()`, resulting in inter-iteration RAW dependences.

```
int* solutions = 0;
for (i = 0; i < n; i++) {
    a[i] = (char) i;
    if (ok(i + 1, a)) {
        nqueens(n, i + 1, a, &res);
        // reduction
        *solutions += res;
    }
}
```

Figure 4.2: A loop in the BOTS benchmark *nqueens*.

A special case of DOACROSS loops is the loops where inter-iteration dependences are only caused by reductions. These loops can be divided into a DOALL loop and a reduction loop. Many parallel programming models support reductions in loops, like the `reduction` clause in OpenMP. A detailed description of reduction detection for loops is provided by our master student Sergei Krestianskov [94].

4.2 Parallel Tasks

So far, parallelism discovery tools have been focusing on data parallelism in loops, which can be exploited by distributing iterations of a loop among multiple threads. However, as more programming models such as OpenMP and Intel TBB [95] aim at task-based parallelism, this original focus of parallelism discovery becomes too narrow. In contrast to loop-based data parallelism, task parallelism does not require every thread to execute the same code. Tasking can exploit parallelism between arbitrary code sections, including parallelism within individual iterations of a loop or between different loops.

4.2.1 SPMD-Style Tasks

As its name suggests, *single-program-multiple-data (SPMD) tasks* execute the same code but work on different data. It is similar to data decomposition. To identify SPMD task parallelism, one only needs to check whether a CU depends on itself because tasks execute the same code.

Note that iterations in a DOALL loop can also be considered as SPMD task parallelism. However, since DOALL loops can usually be parallelized using specialized mechanisms that are more efficient (like `#pragma parallel for` in OpenMP, and `tbb::parallel_for()` in TBB), we categorize DOALL loops separately. In this paper, SPMD task parallelism refer to independent calls to the same function with different parameters, possibly combined with recursive pattern.

Rule of determining SPMD tasks

A function is classified as a SPMD-style task if it meets the following conditions:

- *The function is called more than once in the program*
- *The function is a CU, and its read phase does not depend on its write phase*

It is worth mentioning that the rule stated above is conservative. Since data dependences are obtained dynamically, there is no chance to tell whether a CU depends on itself if the CU is executed only once. However, it does not mean that the CU is definitely not a SPMD task. Self-dependences may be revealed when using another input of the program. In our experiments, we try different inputs whenever possible to minimize the effect of input sensitivity.

Consider the classic program that computes the n^{th} Fibonacci number in a recursive way. The function is shown in Figure 4.3. It is well known that the two recursive calls `Fibonacci(n-1)` and `Fibonacci(n-2)` can run in parallel. They can be interpreted as SPMD-style tasks since the same function is called twice, each time with a different set of arguments.

```
int Fibonacci(int n) {
    if(n <= 0)
        return 0;
    else if(n == 1)
        return 1;
    else
        return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Figure 4.3: A program that computes the n^{th} Fibonacci number.

However, any programmer that has some experience in parallel programming will realize that it is a bad idea to parallelize the two calls to `Fibonacci` without any care. When $n > 1$, each call to `Fibonacci` spawns two tasks, and the current task needs to wait until the two new tasks complete. The number of threads created by the program grows exponentially, and the computation of each task is just an add operation. Parallelizing this program is like hiring a different typist to type each character of a novel. In practice, a recursive program that computes subtasks repetitively can benefit from dynamic programming, which caches results of subtasks to avoid repetitive computation.

Nevertheless, recursive algorithms are often good candidates for parallelization, particularly if they split the job into smaller jobs that can be performed independently. The trick is to know when to stop parallelizing, i.e., the minimum workload of a task that can benefit from parallelization. So far our tool does not suggest "when to stop" since this question is related to the underlying hardware and operating system.

4.2.2 MPMD-Style Tasks

In contrast to SPMD task parallelism, *multiple-program-multiple-data (MPMD) tasks* execute different code from one another. Once tasks are allowed to execute different code, identifying only independent tasks is not sufficient. Multiple MPMD tasks that are dependent on o one another may lead to a pipeline or task graph. Note that a task graph is acyclic. Thus, we report MPMD task parallelism if dependences among CUs that belong to target code sections do not form a circle. That said, MPMD task parallelism is the most general type of parallelism we identify in this thesis.

Note that the implementation of MPMD task parallelism can be different, and the resulting performance varies. When a task graph is implemented, the performance is greatly influenced by the scheduler.

Rule of determining MPMD tasks

Two CUs are classified as MPMD-style tasks if there is no data dependence between them.

The rule for discovering MPMD-style tasks is loose. We do not include control dependences here because if there is no data dependence, the CU guarded by a condition can actually be executed speculatively given the machine has enough computational resources.

```
1   result = compute(input);
2   if (SANITY_CHECK == true) {
3       bool ok = sanity_check();
4       if (!ok)
5           exit(-1);
6   }
```

Figure 4.4: A code snippet showing the role of control dependence in MPMD-style tasks.

Take the code snippet shown in Figure 4.4 as an example. If there is no data dependence between function `compute` and `sanity_check`, `sanity_check` could run in parallel with `compute` in prior to the evaluation of the condition at line 2. Later on, the result of `sanity_check` can either be directly used or discarded based on the evaluation results of the condition. This is a classic example of speculative execution.

We say the rule is loose also because it covers almost all the remaining parallelism other than loop parallelism and SPMD-style tasks. However, reporting all the MPMD-style tasks would be overwhelming to the user. Moreover, unlike DOALL loops and SPMD-style tasks, MPMD-style tasks are based on data flow rather than data decomposition. Compared to DOACROSS loops, the data flow pattern among MPMD-style tasks is more general. For these reasons, we produce

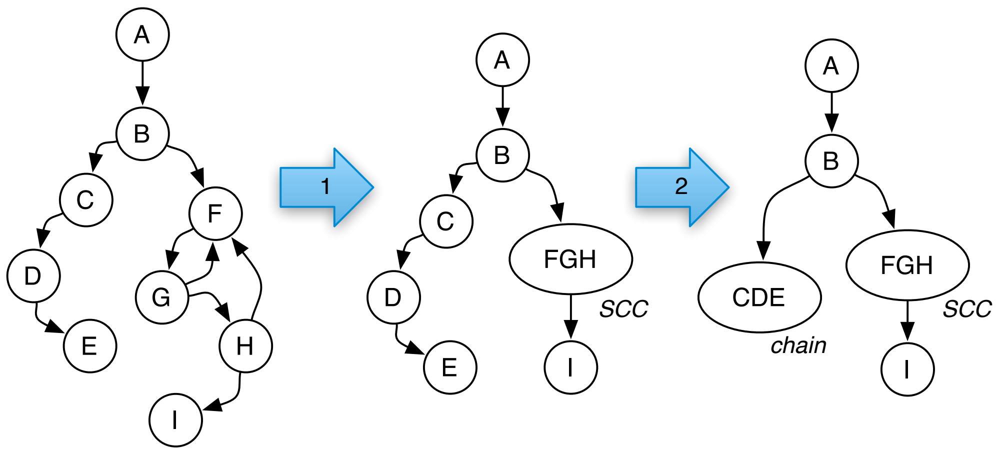


Figure 4.5: Simplifying CU graph by substituting SCCs and chains of CUs with vertices.

a task graph instead of reporting individual MPMD-style tasks so that users can have a better understanding of the data flow.

The task graph is a simplified CU graph by merging CUs contained in *strongly connected components* (SCCs) or in *chains*. The idea of merging CUs in SCCs is inspired by Ottoni et al. [64]. In graph theory, an SCC is a subgraph in which every vertex is reachable from every other vertex. Thus, every CU in an SCC of the CU graph depends on every other CU either directly or indirectly, forming a complex knot of dependences that is likely to defy internal parallelization. Identifying SCCs is important for two reasons:

- **Algorithm design**. Complex dependences are usually the result of highly optimized sequential algorithm design oblivious of potential parallelization. In this case, breaking such dependence requires a parallel algorithm, which is beyond the scope of our method.
- **Coding effort**. Even if such complex dependences are not created by design, breaking them is usually time-consuming, error-prone, and may cause significant synchronization overhead that may outweigh the benefit of parallelization.

Hence, we hide complex dependences inside SSCs, exposing parallelization opportunities outside, where only a few dependences need to be considered. Figure 4.5 shows the graph simplification process by substituting vertices with SCCs and chains of CUs. In step 1, CU F, G and H are grouped into SCC_{FGH}. After contracting each SCC to a single vertex, the graph becomes a directed acyclic graph (DAG). Moreover, we group CUs that are connected in a row without a branch or reconvergence point in between into a *chain* of CUs since a chain of CUs does not contain significant parallelism inside, and merging them can lower the communication overhead among tasks. In step 2, CU C, D and E are grouped into $chain_{CDE}$. We call the simplified graph *task graph*. Finally, we declare each vertex in the task graph a potential task.

4.3 Ranking of Parallelization Targets

Ranking parallelization opportunities of the target program helps users to focus on the most promising ones. Three metrics are involved: *instruction coverage*, *local speedup*, and *CU imbalance*.

4.3.1 Instruction Coverage

The *instruction coverage (IC)* provides an estimate of how much time will be spent in a code section. The estimation is based on the simplifying assumption that each kind of instruction costs about the same amount of time. Given a code section i and the whole program P,

$$IC(i) = \frac{N_{inst}(i)}{N_{inst}(P)} \tag{4.1}$$

where $N_{inst}(i)$ and $N_{inst}(P)$ are the number of instructions of code section i and the whole program P, respectively. Note that N_{inst} always represents the total number of IR instructions that are actually executed at runtime. For example, in a loop, N_{inst} is the sum of the number of IR instructions across all iterations.

4.3.2 Local Speedup

The *local speedup (LS)* reflects the potential speedup that would be achieved if a code section was parallelized according to the suggestion under the assumption that computational resources are unlimited. Since the speedup refers only to a given code section and not necessarily to the whole program, it is called local. The local speedup is based on the critical path, that is, the longest series of operations that have to be performed sequentially due to data dependences and Amdahl's law, which is why super-linear effects are not considered. Note that LS is used to approximate the benefit of parallelization rather than an exact prediction of the real speedup. Given a code section i of the target program:

$$LS(i) = min(N_{threads}, \frac{N_{inst}(i)}{length(CP(i))}) \tag{4.2}$$

where $N_{inst}(i)$ is the total number of instructions of code section i, and length(CP) is the length of the critical path of i—again, based on the assumption that each kind of instruction costs the same amount of time. $N_{threads}$ is the number of threads. If the local speedup exceeds the number of threads, it will be just equal to the number of threads.

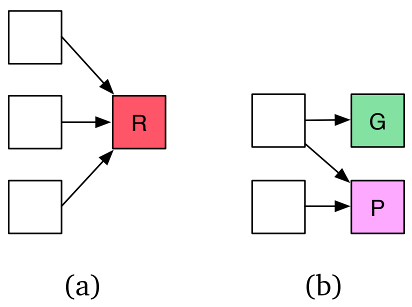


Figure 4.6: Scenarios with different degrees of CU imbalance.

4.3.3 CU Imbalance

The *CU imbalance* reflects how evenly CUs are distributed in each stage of the critical path. A stage of the critical path is a computation step separated by data dependences. A stage may contain multiple CUs that can run in parallel at the step. CU imbalance measures whether every thread has some work to do in each step of the computation. Otherwise, some of the threads have to wait because of data dependences, which means the suggested parallelization may have a bottleneck. We define the CU imbalance for a code section i as

$$CI(i) = \frac{\sigma(i)}{MP(i)} \tag{4.3}$$

where $\sigma(i)$ is the standard deviation of the number of CUs in each stage of the critical path, and $MP(i)$ is the number of CUs in the largest stage of the critical path of code section i. The CU imbalance is a value in $[0, +\infty)$. The more balanced the CU ensemble is, the smaller the value becomes.

Figure 4.6 provides an example. Under the assumption that each CU has the same number of instructions, both situations (a) and (b) have a local speedup of two and will complete all the tasks in two units of time, assuming the system has an unlimited number of threads available. However, the arrangement in Figure 4.6(a) requires three threads while 4.6(b) requires only two. The red CU (R) in 4.6(a) needs the results from three CUs, constituting a bottleneck of the execution. Although the purple CU (P) in 4.6(b) is in the similar situation, the other thread still has some work to do (green CU) so that it does not need to wait. The CU imbalance values of the two situations (4.6(a): $\sqrt{2}/3 = 0.47$, 4.6(b): $0/2 = 0$) reflect such a difference. Note that a code section containing no parallelism (CUs are sequentially dependent) will also show a CU imbalance of zero, which is consistent with our definition.

Our ranking method now works as follows: Parallelization opportunities are ranked by their estimated global speedup (GS) in descending order, with

$$GS = \frac{1}{\sum_i \frac{IC(i)}{LS(i)} + (1 - \sum_i IC(i))}. \tag{4.4}$$

Should two or more opportunities exhibit the same amount of global speedup, they will be ranked by their CU imbalance in ascending order. Note that since LS is never bigger than the number of threads and IC is always smaller than 1, GS can never exceed the number of threads, either.

4.4 Evaluation

We conducted a range of experiments to evaluate the effectiveness of our approach. We applied our method to benchmarks from the Barcelona OpenMP Task Suite (BOTS) [96], the PARSEC benchmark suite [97], the SNU NAS Parallel Benchmarks (NAS) [84, 85], and the Starbench benchmark [86]. All the four benchmark suites contain sequential benchmark applications as well as their equivalent parallel versions.

There are two evaluation methods. After applying our method to the sequential benchmark applications, we 1) compare the identified parallelization opportunities to the existing parallel versions in the benchmark suites. For the benchmarks of which existing parallel versions are not available, we 2) implemented our own parallel versions and measure the speedups.

Our approach is implemented using LLVM [11] 3.6.1, and all benchmarks are compiled using Clang [98] 3.6.1 with `-g -O0` for instrumentation, and `-O2` for execution. Experiments were run on a server with 2 x 8-core Intel Xeon E5-2650, 2 GHz processors with 32 GB memory, running Ubuntu 12.04 (64-bit server edition). The performance results reported are an average of five independent executions. Whenever possible, we tried different inputs to compensate for the input sensitivity of the data-dependence profiling approach, resulting in more complete data dependences for each benchmark.

4.4.1 DOALL Loops

The purpose of the first experiment was to detect DOALL loops and see how the signature-based approximation used by the data dependence profiler affects the accuracy of the suggestions on parallelism. We first took our test cases from the NAS benchmarks, a suite of programs derived from real-world computational fluid dynamics applications. The suite includes both sequential and OpenMP-based parallel versions of each program, facilitating the quantitative assessment of our tool's ability to spot potential loop parallelism. We searched for parallelizable loops in sequential NPB programs and compared the results with the parallel versions provided by NPB.

Table 4.1 shows the results of the experiment. The data listed in the column set "Executed" are obtained dynamically. Column "# loops" gives the total number of loops which were actually executed. The number of loops that we identified as parallelizable are listed under "#

Table 4.1: Detection of parallelizable loops in NAS Parallel Benchmark programs.

Benchmark	Executed		OpenMP-annotated loops			
	# loops	# parallelizable	# OMP	# identified	# in top 30%	# in top 10
BT	184	176	30	30	22	9
SP	252	231	34	34	26	9
LU	173	164	33	33	23	7
IS	25	20	11	8	2	2
EP	10	8	1	1	1	1
CG	32	21	16	9	5	5
MG	74	66	14	14	11	7
FT	37	34	8	7	6	5
Overall	787	720	147	136	96	45

parallelizable". At this stage, prior to the ranking, DiscoPoP considers only data dependences, which is why still many loops carrying no dependence but bearing only a negligible amount of work are reported. The second set of columns shows the number of annotated loops in OpenMP versions of the programs (# OMP). Under "# identified" we list how many annotated loops were identified as parallelizable by DiscoPoP.

As shown in Table 4.1, DiscoPoP identified 92.5% (136/147) of the annotated loops, which is the same as using a perfect signature. [54] These results proved that the effect of the signature approximation to be negligible. A comparison with other tools is challenging because none of them is available for download. A comparison based exclusively on the literature has to account for differences in evaluation benchmarks and methods. Kremlin [10], which was also evaluated with NPB, selects only loops whose expected speedup is high. While Kremlin reported 55.0% of the loops annotated in NPB, the top 30% of DiscoPoP's ranked result list cover 65.3% (96/147).

We further evaluated our tool on a set of small applications that are commonly used in teaching parallel programming. The parallel versions of these applications are not available, but the parallel solutions are obvious to experienced programmers. The purpose is to see whether our tool discovers these obvious solutions. We parallelized these applications manually by adopting the suggestions generated by our tool and measured the speedup we gained. The parallelization is either based on Pthreads or OpenMP, and the parallel versions always use four threads. Table 4.2 summarizes the results. Values shown in the table are averages of five runs. The details of each application are discussed below.

Histogram visualization

This program receives an array whose elements can belong to N different types and sorts them into buckets, putting data with type N_i into the i^{th} bucket. The items in every bucket are counted to produce the histogram. We use this example to illustrate details of the suggestions produced by our tool, which are shown in Table 4.3. Our tool successfully finds the main

Table 4.2: Speedups achieved when parallelizing textbook programs adopting the suggestions produced by our method using four threads.

	histogram	mandelbrot	light propagation	ANN training
LOC	102	521	74	107
Input size	50,000,000 numbers	square matrix (dim = 1024)	500,000 random points	matrix 50×500 and 500×4
Number of suggestions	5	2	1	10
# Adopted	1	2	1	2
Seq. time (s)	0.36	46.02	5.67	5.11
Par. time (s)	0.098	22.73 (11.61)	2.33	1.66
Speedup	3.67	2.02 (3.96)	2.43	3.07

computational loop as a good candidate to be parallelized. The loop iterates over the input array with no data dependences inside, indicating the numbers in the array can be processed in parallel. The other loops are also parallelizable, but belong either to the initialization or output stage and do not promise significant speedup for larger input problems. Moreover, we do not follow suggestion 5 because the loop contains only one line without function call and iterates four times. To measure the speedup, we use an array of 50,000,000 numbers as input. The serial version of the program runs in 0.36 seconds, whereas the parallel version with four threads runs in 0.098 seconds, resulting in a speedup of 3.67.

Mandelbrot set

The Mandelbrot set is the set of values c in the complex plane for which the orbit of zero under iteration of the complex quadratic polynomial $z_{n+1} = z_n^2 + c$ remains bounded. Our test program produces a 1024×1024 resolution image for the Mandelbrot set. The program iterates over rows and columns, checking whether a point belongs to the set. The problem exhibits a high degree of data parallelism, since every point on the plane can be examined independently. Our tool reports that the innermost loop cannot be parallelized because of RAW dependences between iterations, involving variables z_{real} and z_{imag}. This loop iterates 50,000 times at most to test whether the complex number $z_{real} + z_{imag}i$ satisfies the equation. However, the outer loops are reported as parallelizable. The outermost loop iterates over the rows of the matrix, and the loop direct nested inside iterates over its columns. We parallelize the program with Pthreads by dividing the matrix among four threads. While the serial version of the program takes 46.02 seconds, the parallel version takes 22.73 seconds, resulting in a speedup of 2.02. With the fastest thread running only 0.15 seconds, the disappointing speedup is the result of imbalanced workload. After introducing a dynamic load-balancing scheme, the four threads consume about the same time, resulting in an almost linear speedup of 3.96.

Table 4.3: Suggestions for histogram visualization.

Number	Location	# Iter.	Loop size	Adopted	Reason
1	line 46	50	6 lines	Yes	-
2	line 21	50	3 lines	No	initialization
3	line 54	53	1 line	No	output
4	line 34	50	1 line	No	output
5	line 44	4	1 line	No	too small

Simulation of light propagation using Monte Carlo

This program simulates light propagation from a point source in an infinite medium with isotropic scattering using the Monte Carlo method. Photons are modeled as pairs of randomly produced numbers and each photon is simulated independently. Nevertheless, a global array of `heat` must be calculated. It is therefore possible that two photons write the same element of the `heat` array. For some executions, our tool reports that the loop iterating over the photons can be parallelized, and for some other executions the loop has inter-iteration dependences. This is because the algorithm is a Monte Carlo method, and whether two photons write the same element of the `heat` array is a random event. When the input size is small, the probability of writing the same element of the `heat` array is low. After getting this observation, we parallelize the main loop and protect each element of `heat` with a separate lock. We run the parallel version with four threads. This simple approach results in a slowdown of 15.75. The serial version runs only 5.67 seconds, but the parallel version runs 89.28 seconds. After an investigation, we found that the function `rand()` maintains internal global states that must be protected in parallel execution. After replacing all occurrences of the `rand()` function with a thread-safe alternative `rand_r()`, the adjusted parallel version runs 2.33 seconds, resulting a speedup of 2.43.

Artificial neural network training

The Artificial Neural Network training algorithm adjusts the weight matrices of the network by iteratively examining training data provided as input. Because new weight values always depend on their former values, it is hard to run different iterations in parallel. However, during the same iteration, it is possible to parallelize the calculation of the weight matrix in one dimension. Our tool successfully identified two loops, both of which iterate along one dimension of the weight matrices. Adopting the suggestion from our tool, we parallelize the training program using OpenMP and run it with four threads. Because the training algorithm usually needs quite a long time to reach convergence if it reaches it at all, we took the liberty of placing an upper bound on the number of iterations to make the program terminate in a reasonable time frame. Our neural test network comprises 50×500×4 neurons. The serial version runs 5.11 seconds, while our parallel version runs 1.66 seconds, resulting in a speedup of 3.07. This is actually quite close to the results provided by Alfred Strey [99], in which three parallel versions of ANN training are tested and the approach B is almost the same as when following our tool's

Table 4.4: Detection of DOACROSS loops in benchmarks from Starbench and NAS. The biggest hot loops in terms of execution time of each benchmark are summarized in the table.

Benchmark		Exec. time [%]	DOACROSS	Implemented in parallel version	# CUs
Starbench	rgbyuv	99.9	✓	pipeline (DOALL)	5
	tinyjpeg	99.9	✓	pipeline	2
	kmeans	99.5	✓	reduction	4
BOTS	nqueens	~100	✓	reduction	1
NAS	CG	96.9	✓	reduction	4
	BT	99.1	✗	no	n.a.
	SP	99.1	✗	no	n.a.
	FT	49.3	✗	no	n.a.
	MG	49.8	✗	no	n.a.

suggestions. In Strey's work, the approach B is implemented using OpenMP, results in a speedup of about 3.0 with 40×100×10 neurons.

4.4.2 DOACROSS Loops

A DOACROSS loop has inter-iteration dependences, but the dependence are not between the first line of an iteration and the last line of the previous iteration. This means in a DOACROSS loop, iterations are not independent but can partly overlap with one another, providing parallelism that can be utilized by implementing reduction or pipeline.

It is obvious that parallelizing small loops (in terms of workload) with inter-iteration dependences is not beneficial. Thus we focus on DOACROSS loops that are hotspots in terms of execution time. Table 4.4 summarizes the biggest DOACROSS loops in benchmarks from Starbench [86], BOTS [96], and NAS. As shown in the table, the target loops in *BT*, *SP*, *FT*, and *MG* are not DOACROSS loops. The column "Implemented" shows the implementation mechanism in the existing parallel versions.

Among the loops that are identified as DOACROSS, two (in *rgbyuv* and *tinyjpeg*) are suitable for pipeline implementation while the other three (*kmeans*, *nqueens*, and *CG*) can be parallelized with reduction. As we mentioned before, the implementation choice has to be made by the user. However, distinguishing which implementation is the best for a DOACROSS loop is relatively easy since the inter-iteration dependences are reported. We verified that the DOACROSS loop identified in *tinyjpeg* is implemented as a pipeline in the official parallel implementation. However, the target loop in *rgbyuv* is an interesting case.

```
1    for(int j = 0; j < args->pixels; j++) {
2        R = *in++;
3        G = *in++;
4        B = *in++;
5
6        Y = round(0.256788*R + 0.504129*G + 0.097906*B) + 16;
7        U = round(-0.148223*R - 0.290993*G + 0.439216*B) + 128;
8        V = round(0.439216*R - 0.367788*G - 0.071427*B) + 128;
9
10       *pY++ = Y;
11       *pU++ = U;
12       *pV++ = V;
13   }
```

Figure 4.7: The target loop in *rgbyuv* (bmark.c, line 151).

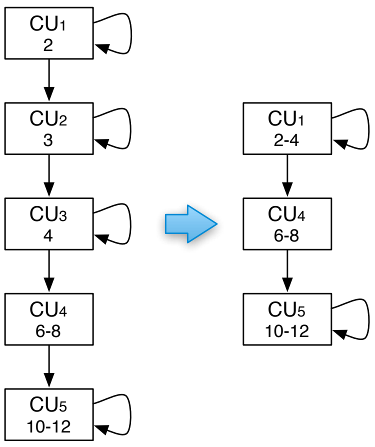


Figure 4.8: CU graphs of the loop body of the loop in *rgbyuv* (bmark.c, line 151).

rgbyuv (Starbench)

The target loop in *rgbyuv* is in bmark.c, line 151. The source code of the loop is shown in Figure 4.7. The target loop has five CUs: CU_1 (line 2), CU_2 (line 3), CU_3 (line 4), CU_4 (line 6–8), and CU_5 (line 10–12). The CU graph of the loop body is shown on the left side of Figure 4.8. Obviously, CU_1, CU_2, and CU_3 are too small, so we consider them as a single computation without losing significant parallelism, leading to the simplified CU graph shown on the right side of Figure 4.8.

At the beginning we know nothing about the code of *rgbyuv*, just like a programmer who parallelizes sequential code written by someone else. Simply following the simplified CU graph in Figure 4.8, we found the loop can be parallelized as a three-stage pipeline. Since CU_1 and CU_5 have self-dependences, the first stage and the third stage have to be sequential stages, while the second stage can be a parallel stage. A parallel stage is a stage where data can be

Table 4.5: Parallelism discovery results of gzip 1.3.5 and bzip2 1.0.2 compared to existing parallel implementations. The table summarizes the number of suggestions and the most important parallelization opportunity for each application.

	gzip 1.3.5	bzip2 1.0.2
Number of suggestions	43	62
Location parallelized in parallel implementation	pigz.c: 1478	bzip2smp.c: 81
Matching suggestion	gzip.c: 1595	bzip2.c: 3793
# Iteration	284	104
Loop size	101 lines	34 lines

further divided and processed in parallel. We implement the pipeline using Intel TBB [95]. Each stage is implemented as a filter class, and stages are connected using `tbb::parallel_pipeline`. Moreover, the `filter::serial_in_order` attribute is specified for stages 1 and 3. In a word, everything was done following the output of our tool, and we did not bother understanding the code.

The best performance of our implementation appears when using 4 threads, with a speedup of 2.29. Using more threads than the number of stages of a pipeline usually does not give better performance, especially when most of the stages are sequential. When examining the official parallel implementation of *rgbyuv*, we found that the target loop is parallelized as DOALL, not DOACROSS. This means the inter-iteration dependences on CU_1 and CU_5 do not prevent parallelism. This is true because the inter-iteration dependences are on pointers (`in`, `pY`, `pU`, and `pV`), not the data to which they point. Thus, to utilize the DOALL parallelism we just need to make the pointers local.

This example shows that simply following the output of our tool yields good speedup, and understanding the code is still important. Nevertheless, our tool reveals interesting parallelization opportunities and data dependences that potentially prevent parallelism, helping the users to achieve a better implementation much faster.

DOACROSS loops identified in *kmeans*, *nqueens* and *CG* are implemented using reduction in the official parallel implementations. The DOACROSS loops in kmeans and *CG* are similar to the example shown in Figure 4.2, but the code is more complicated.

We further analyzed two well-known open-source programs gzip 1.3.5 and bzip2 1.0.2 for DOACROSS parallelism. We choose these two applications because they are compression tools and compression tools are famous for their pipeline work flow in which data is divided into small chunks for processing. We want to see if our tool can detect such pipeline parallelism resides in DOACROSS loops. For this reason, we use pigz 2.2.4, a parallel version of gzip 1.3.5, and bzip2SMP 1.0, a parallel version of bzip2 1.0.2, as the parallel reference implementations for comparison. Table 4.5 summarizes the results.

gzip 1.3.5

gzip is a widely used file-compression tool and pigz [100] a popular parallel implementation based on Pthreads. In gzip, files are broken down into blocks, and the algorithm iterates over blocks, compressing them one by one. In the output of our tool, we find that the loop starting at line 1595 is classified as DOACROSS loop, which iterates 284 times while other structures are usually executed not more than ten times. Although four dependences are reported inside the loop, the fact that it contains more than 100 lines of code and iterates 284 times makes it an attractive parallelization target. After analyzing the code in detail, we realize that all four dependences refer to global variables, which are used when compressing individual file blocks. Based on these insights, we think that in spite of the four dependences this structure is worth to be parallelized, given the large amount of work it performs.

The loop is parallelized in pigz. In the function `parallel_compress` at line 1478 in pigz.c, pigz breaks the input into blocks of 128 KB and compresses them concurrently. However, this function does more than what has been suggested by our simple discovery. It also calculates the individual check values for each block in parallel, and contains some optimizations for parallel IO. Nevertheless, the main idea of the underlying parallelization strategy is correctly identified.

Our tool also lists other interesting places as potential parallelization targets. For example, there is a loop in the main function starting at line 3400, which iterates over user input files after processing user options. Obviously, it would also be a good parallelization candidate since compressing different files exhibits data parallelism. But it would require some effort to resolve dependences, since the buffers in the sequential program are reused. pigz does not parallelize this part. In the parallel implementation suggested by Ding et al. [101], this part is also identified.

bzip2 1.0.2

bzip2 is another well-known compression tool. A number of parallel implementations exist, but their approaches differ. We chose bzip2SMP [102], a parallel implementation based on Pthreads, for comparison because our methods are mainly designed for the shared-memory platform. Our tool suggests that the loop starting at line 3793 inside the function `handle_compress` iterates hundreds of times and consumes 83% of the function's execution time. It is identified as a DOACROSS loop, and data dependences come from accesses to the global data structure `EState* s`. The loop contains two parts: one for the preparation of a new block and the other for the compression of the block. They exchange state information through `s`, leading to a RAW dependency between iterations. By examining the call graph starting from `handle_compress`, we find calls to `BZ2_compressBlock` and `BZ2_blockSort`. Dependences inside them are also anchored in the structure `s`, since the pointer of `s` is passed to these two functions as a parameter. According to our understanding of the original bzip2 algorithm, we find that by duplicating the `EState` structure, the block sorting stage of the pipeline can be parallelized, which means that the blocks of a file can be compressed in parallel. Bzip2SMP adopts exactly the same idea. The function performing the parallel block sort is `threadFunction` starting at line 81 in

Table 4.6: SPMD-style tasks in BOTS benchmarks.

Benchmark	Function	SPMD-task	Implemented in parallel version	Execution time (%)
sort	cilkmerge	✓	✓	34.4
	cilksort	✓	✓	74.8
	seqquick	✗	✗	22.6
	seqmerge	✗	✗	52.0
	sort	✗	✗	74.9
fib	fib	✓	✓	~100
fft	fft	✗	✗	~100
	fft_aux	✓	✓	97.2
	fft_twiddle_16	✓	✓	83.0
	fft_unshuffle_16	✓	✓	12.7
floorplan	add_cell	✓	✓	~100
health	sim_village	✓	✓	~100
sparselu	sparselu	✓	✓	34.4
	bmod	✗	✗	89.6
strassen	strassen_main	✗	✗	95.2
	OptimizedStrassenMultiply	✓	✓	95.2
	MultiplyByDivideAndConquer	✓	✓	82.0
	FastNaiveMatrixMultiply	✗	✗	21.4
	FastAdditiveNaiveMatrixMultiply	✗	✗	61.9
uts	serTreeSearch	✓	✓	99.6

bzip2smp.c. However, the real parallel strategy is much more complex than we expected. The same parallelization target was also found by Zhang et al. [13].

4.4.3 SPMD Tasks

We applied our approach to the BOTS [96] benchmarks to evaluate the discovery of SPMD-style tasks. We choose BOTS because they contains many of the SPMD-style tasks we are looking for, and such tasks rarely occur in other benchmarks. Results of the experiments are summarized in Table 4.6. The evaluation work in this section is done by two members of our group Zia Ul Huda and Rohit Atre.

Similar to the approach of analyzing DOACROSS loops, only hotspots in terms of execution time are examined. In total, 20 hotspot functions from the BOTS benchmarks are analyzed and 12 of them are classified as SPMD-style tasks, all of which are parallelized in the existing parallel versions of the benchmarks.

```
 1 void fft_twiddle_16(int a, int b, COMPLEX * in, COMPLEX * out, COMPLEX * W,
 2                     int nW, int nWdn, int m)
 3 {
 4   int l1, i;
 5   COMPLEX *jp, *kp;
 6   REAL tmpr, tmpi, wr, wi;
 7   if ((b - a) < 128) {
 8     for (i = a, l1 = nWdn * i, kp = out + i; i < b; i++, l1 += nWdn, kp++) {
 9       ... // omit 336-line loop body
10     }
11   }
12   else {
13     int ab = (a + b) / 2;
14     fft_twiddle_16(a, ab, in, out, W, nW, nWdn, m);
15     fft_twiddle_16(ab, b, in, out, W, nW, nWdn, m);
16   }
17 }
```

Figure 4.9: Souce code fragments of function `fft_twiddle_16` in the BOTS benchmark *fft*.

fft (BOTS)

The code of one hotspot function, `fft_twiddle_16`, is shown in Figure 4.9. The loop body is omitted since it does not affect the SPMD parallelism and is too long to fit in this section. This function recursively calls itself twice (line 14, 15) using different sets of parameters. Moreover, the two calls are guaranteed to run on different data ranges (specified by formal parameters `a` and `b`) according to line 13. Since the two calls do not depend on each other, these are exactly the SPMD-style tasks we discussed in Section 4.2.1.

Similar patterns are found at multiple places in `fft`, and we do not discuss them again. Basically, all the `fft_twiddle_` functions and `fft_unshuffle_` functions follow the same pattern. These functions differ in the suffix, from 8 to 32. They are selected based on the input data size. Thus, all are identified as SPMD-style tasks.

4.4.4 MPMD Tasks

To evaluate the detection of MPMD tasks, we applied our method to PARSEC benchmarks [97] and two other applications: the open-source Ogg codec *libVorbis* and an Intel Concurrent Collections (CnC) sample program *FaceDetection*. In contrast to SPMD tasks that widely exist in BOTS benchmarks, MPMD tasks execute different code. Generally speaking, programs containing MPMD tasks perform multiple kinds of computations rather than a single computation on big input data. This implies that it is more likely to find MPMD-tasks in larger programs in terms of lines of code (LOC). This is the reason why the programs we use in this section are generally bigger in terms of code size. Moreover, it is well known that pipeline and flow graph

Table 4.7: Detection of MPMD tasks in PARSEC benchmarks and the multimedia applications libVorbis, and FaceDetection.

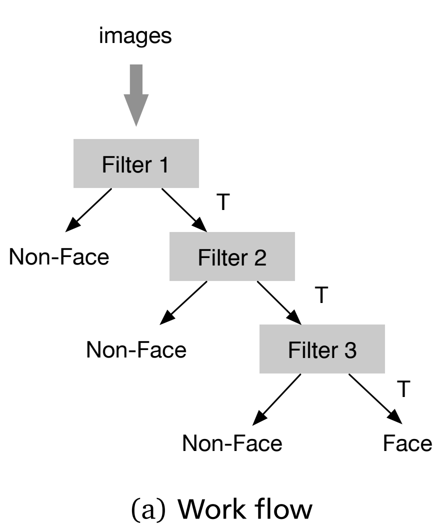
Benchmark	Function	Implemented in parallel version	Our solution	# threads	Speedup
blackscholes	CNDF	no	omp sections	4	1.00
canneal	routing_cost_given_loc	no	omp sections	4	1.00
fluidanimate	RebuildGrid	no	omp sections	4	1.00
fluidanimate	ProcessCollisions	no	omp sections	4	1.00
fluidanimate	ComputeForces	data decomposition	pipeline	3	**1.52**
libVorbis	main (encoder)	no parallel version	pipeline	4	**3.62**
FaceDetection	facedetector	pipeline	pipeline	32	**9.92**

patterns are common in multimedia processing applications. Two programs that process audio (*libVorbis*) and images (*FaceDetection*) are included.

Table 4.7 summarizes the results of evaluating the detection of MPMD tasks. As the results show, MPMD tasks are not the main type of parallelism in applications that performs simulations. Only three benchmarks (*blackscholes*, *canneal*, and *fluidanimate*) contain MPMD tasks. However, all the MPMD tasks found in these programs are from non-hotspot computations. They are not parallelized in the official parallel implementations, and parallelizing them using `omp section` does not give any speedup. The only interesting place in these programs is the `ComputeForces` function in *fluidanimate*. The parallelization story, however, is similar to the case study shown in Section 4.1.2. We parallelized the function body following the output CU graph using TBB and achieved a speedup of 1.52 using three threads. On the contrary, the official parallel version of *fluidanimate* shows this function is parallelized using data decomposition, yielding almost linear speedup.

FaceDetection

FaceDetection is a simplified version of a cascade face detector used in the computer vision community. The face detector consists of three different filters. As shown in Figure 4.10(a), each filter rejects non-face images and lets face images pass to the next layer of the cascade. An image will be considered a face if and only if all layers of the cascade classify it as a face. The corresponding TBB flow graph is shown in Figure 4.10(b). A join node is inserted to buffer all the boolean values. In order to decide whether an image is a face, every boolean value corresponding to that specific image is needed. For this reason, we use the `tag_matching`



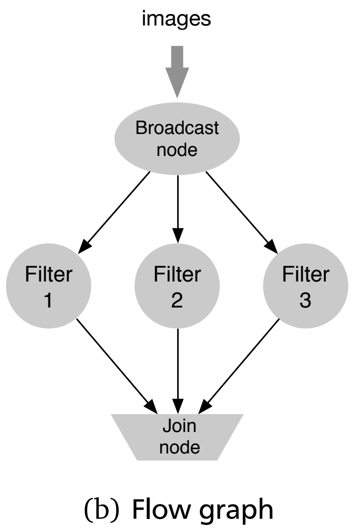


Figure 4.10: Work flow and flow graph of FaceDetection.

buffering policy in the join node. `tag_matching` policy creates an output tuple only when it has received messages at all the ports that have matching keys.

The three filters take 99.9% of the sequential execution time. We use 20,000 images as input. The speedup of our TBB flow graph parallel version is 9.92× using 32 threads. To evaluate the scalability of the parallel code, we compare the speedups achieved by the official Intel CnC parallel version and our TBB flow graph version using different numbers of threads. The result is shown in Figure 4.11.

The performance is comparable using two and four threads. When more than eight threads are used, the official CnC parallel version outperforms ours. The reason is that the official CnC parallel code is heavily optimized and restructured. For example, some data structures are altered from `vector` to CnC `item_collection`. As shown in Figure 4.11, the official CnC version is already two times faster than our TBB version when using a single thread because of the optimization.

libVorbis

libVorbis is a reference implementation of the Ogg Vorbis codec. It provides both a standard encoder and decoder for the Ogg Vorbis audio format. In this study, we analyzed the encoder part. The suggested pipeline resides in the body of the loop that starts at file encoder_example.c, line 212, which is inside the main function of the encoder. The pipeline contains only two stages: `vorbis_analysis()`, which applies some transformation to audio blocks according to the selected encoding mode (this process is called analysis), and the remaining part that actually encodes the audio block. After investigating the loop of the encoding part further, we found it to have two sub-stages: encoding and output.

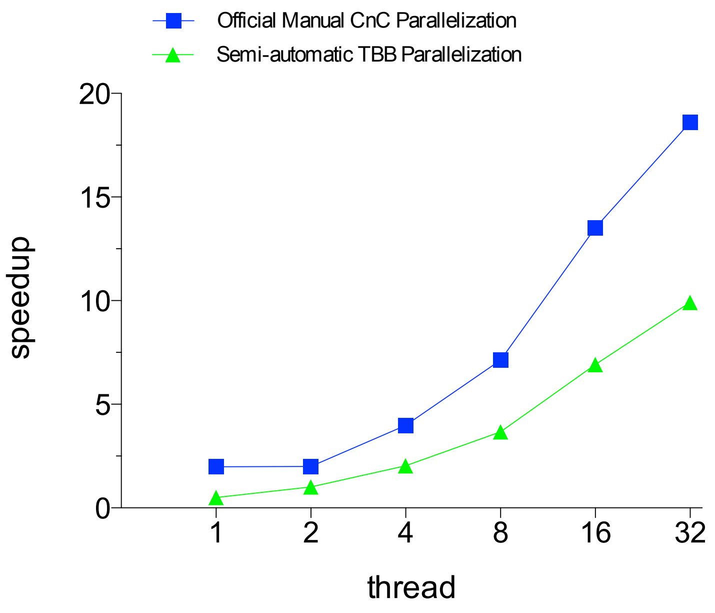


Figure 4.11: FaceDetection speedups with different numbers of threads.

We added one more stage to the pipeline for serialization, in which we reorder the audio blocks because we do not force audio blocks to be processed in order in the analysis and the encoding phase. We end at with a four-stage pipeline with one stage each for analysis, encoding, serialization, and output, respectively. We ran the test using a set of uncompressed wave files with different sizes, ranging from 4 MB to 47 MB. As a result, the parallel version achieved an average speedup of 3.62 with four threads.

4.4.5 Ranking Method

We also evaluated the precision of our ranking method. The results are shown in Table 4.1. Column “# in top 30%" lists the number of suggestions matched by actual parallelization in the OpenMP version (# identified) that end up in the top thirty percent after ranking. We believe that only few programmers would examine all the suggestions one by one and that for most the first 30% would be the upper limit. As one can see, 70.6% (96/136) of the matched suggestions can be found among the top 30%. This means by examining only 30% of the suggestions, 70% of the actually implemented parallelism can be explored.

We also verified whether the top 10 suggestions for each program are really parallelized in the official OpenMP version. The results are listed in the column “# in top 10”. For most of the programs, more than a half (for some of them even 90%) of the top 10 suggestions are parallelized, proving the effectiveness of our ranking method.

4.5 Limitations

Even assuming the data dependences obtained from a sequential program are complete and precise, parallelism discovery approaches based on data and control dependence analyses still produce false positives due to the lack of semantic information. We present an example in this section to demonstrate the problem.

```
...
Time start = get_current_time();
...
// computations
...
Time end = get_current_time();
TimeInterval elapsed = end - start;
...
```

Figure 4.12: A simple but common time measurement method in benchmarks.

The code snippet shown in Figure 4.12 shows a very common time measurement method in benchmarks. `start` and `end` record the time stamp right before and after the computations, and the time elapsed during the computations are calculated based on the two recorded time stamps. There is no data dependence between the variables used in recording time and the variables used in computations. When constructing CUs using the bottom-up approach, time-related computations form CUs that are independent of the CUs corresponding to the computations to be measured, leading to MPMD-style parallel tasks. Such a result is a false-positive because although there is no dependence between time measurements and actual computations, they cannot run in parallel without losing their intended semantics.

Note that this example does not mean the CUs are incorrect. Actually, they are absolutely correct according to their definition, which makes it possible to detect MPMD-style tasks that are not aligned with source-language structures. This example shows that besides control and data dependences, semantic information may also have an impact on parallelism.

It is very difficult to infer a programmer's intention by analyzing the code unless the programmer specifies such information by certain means. To the best of our knowledge, it is an open problem of dependence-based parallelism discovery approaches.

4.6 Summary

In this chapter, we have introduced CU-based parallelism discovery methods covering four kinds of parallelism:

- **DOALL loops**: Loops that have no inter-iteration dependences

- **DOACROSS loops**: Loops that have inter-iteration dependences, but iterations can still overlap partially to explore parallelism
- **SPMD-style tasks**: Functions, that individual calls to which can run in parallel
- **MPMD-style tasks**: Different code sections that do not depend on each other via data dependences

A ranking method is also presented to help users focus on the most interesting parallelization opportunities.

We evaluated our CU-based parallelism discovery methods using four benchmark suites: BOTS, PARSEC, NAS, and Starbench. All of the four suites contain sequential benchmark applications as well as their equivalent parallel versions. We have presented two evaluation methods. After applying our method to the sequential benchmark applications, we 1) compared the identified parallelization opportunities to the existing parallel versions. For the benchmarks that do not have corresponding parallel versions, we 2) implemented our own parallel versions for these applications and measure the speedups.

In the experiment of DOALL loop discovery, our tool identified 92.5% of the loops parallelized in the NAS benchmarks. Reasonable speedups are obtained when parallelizing textbook examples in parallel programming, following the suggestions made by our tool. In the experiments of DOACROSS loop and SPMD-style task discovery, all the hotspot loops and functions that are classified as parallelizable are parallelized in the existing parallel versions of the benchmarks, but the concrete implementations sometimes differ from the suggestions produced by our tool. In the experiment of MPMD-style task discovery, not many MPMD tasks are found in benchmarks and parallelizing them does not yield satisfying speedups. However, when analyzing multimedia applications FaceDetection and libVorbis, MPMD-style tasks are common. Parallelizing FaceDetection following the task graph produced by our tool gives a speedup of 9.92 using 32 threads.

5 Further Applications of the Framework

When designing and implementing the DiscoPoP dependence profiler, we want it to be a profiler that supports multiple data-dependence based analyses. As described in Chapter 2, the DiscoPoP profiler provides a general foundation for parallelism discovery and other data-dependence-based analysis techniques for both sequential and parallel programs. In this chapter, we introduce three additional applications of the profiler: 1) characterizing features of DOALL loops, 2) determining optimal parameters for software transactional memory, and 3) detecting of communication patterns on multicore systems.

5.1 Characterizing Features for DOALL Loops

Instead of classifying DOALL loops only based on the existence of inter-iteration dependences, Daniel Fried, an exchange program student worked in our group, tried to take more code features into consideration, and discover those features that are important to decide whether a loop is a DOALL loop. [60] Features are extracted based on the output of our profiler, including data dependences, control-flow information, and metrics integrated into the program execution tree. Table 5.1 summarizes the extracted features. Data dependences are further categorized based on their directions and the scope of their source and sink.

Further, the method trains a classification model using supervised learning. In supervised learning, the model adjusts itself by "learning" from a training data set. The data in the training set is labeled: the expected output is always attached to a specific input. In this work, loops in NAS benchmarks are divided into two sets, one used as the training set (630 loops, 126 with positive labels, that is, parallelized in the existing parallel versions) and the other used as the test set (160 loops, 21 with positive labels), for a split of roughly 80% / 20%. Three different classification models are compared in this work: support vector machine (SVM) [103], decision tree [104], and an ensemble of decision trees boosted with adaptive boosting (AdaBoost) [105].

Table 5.1: Dynamic features used for DOALL loop classification.

Feature	Description
N_Inst	Number of instructions within the loop
exec_times	Total number of times the loop is executed
CFL	Critical path length
ESP	Estimated speedup
incoming_dep	Dependency count of external instructions on loop instructions
internal_dep	Dependency count between loop instructions
outgoing_dep	Dependency count of loop instructions on external instructions

Table 5.2: Feature importance in decision trees, calculated using weighted error reduction in an AdaBoost ensemble of trees.

Feature	Importance	Feature	Importance
N_Inst	0.12	internal_dep	0.06
internal_dep_RAW	0.09	incoming_dep_WAR	0.06
outgoing_dep_RAW	0.08	outgoing_dep_WAR	0.06
incoming_dep	0.08	CFL	0.05
incoming_dep_RAW	0.08	internal_dep_WAW	0.02
internal_dep_WAR	0.08	ESP	0.02
outgoing_dep	0.07	outgoing_dep_WAW	0.02
exec_times	0.07	incoming_dep_WAW	0.02

To analyze which features have a bigger impact on classifying DOALL looops, we compute he feature importance in a decision tree by calculating a weighted sum of the reduction in the impurity criterion that each feature provides across all nodes for which it is the splitting point [106].

Intuitively, features that receive higher importance scores were used to split larger number of training instances and resulted in larger impurity reductions in these splits. Importance for a single tree may not be informative if the tree is a weak classifier, but if we have an ensemble of trees (as we do in AdaBoost), we can average these feature importances across all the trees in the ensemble, producing more robust feature scores. Table 5.2 shows the relative feature importance calculated in this manner for the loops in the training set.

Unlike most of the related work which takes data dependences as the main or even the only criterion of discovering parallelism, the top feature that decides whether a loop is parallelized by an expert programmer is the number of instructions within the loop. That is why we always consider hots pots first in Chapter 4. The order of the remaining features are within expectation. RAW dependences are the most important dependences, and WAW dependences generally do not prevent the parallelization of loops.

In the end, the three models are compared using two sets of features: all features and top features. Top features are features with an importance score of 0.08 or greater. The results of the comparison are summarized in Table 5.3.

When using all features, the SVM and decision-tree classifiers achieve nearly identical scores. Boosting (with AdaBoost) significantly improves the precision of the decision tree resulting in a higher F1 score, a measure of the accuracy of a test in statistical analysis. When using only the most important features, as ranked by importance in the boosted ensemble of decision trees, the performance of SVM and decision tree both increase in accuracy and F1 score, while the performance of AdaBoost ensemble decreases slightly.

Table 5.3: Classification scores on the held-out evaluation set, separated by loops with pragmas and loops without pragmas.

Classifier	Identifying pragma presence			Identifying pragma absence			Accuracy
	Precision	Recall	F1	Precision	Recall	F1	
Baseline	0.00	0.00	0.00	**1.00**	**1.00**	**1.00**	0.81
SVM - AF	0.46	0.62	0.53	0.94	0.89	0.92	0.85
Decision Tree - AF	0.45	0.62	0.52	0.94	0.88	0.91	0.85
AdaBoost DT - AF	**0.72**	0.62	**0.67**	0.94	0.96	0.95	**0.92**
SVM - TF	0.53	**0.81**	0.64	0.97	0.89	0.93	0.88
Decision Tree - TF	0.63	0.57	0.60	0.94	0.95	0.94	0.90
AdaBoost DT - TF	0.71	0.48	0.57	0.92	0.97	0.95	0.91

This application shows that the DiscoPoP profiler is capable of performing analysis other than dependence-based parallelism discovery. The rich number of metrics produced by the profiler allow many interesting properties of sequential programs or their structures to be studied.

5.2 Determining Optimal Parameters for Software Transactional Memory

Software transactional memory (STM) is becoming increasingly popular as a convenient way of writing parallel programs. STM provides an atomic construct, called transaction, which is used to protect shared memory locations from concurrent accesses by threads. Intermediate transactional values are not visible to other transactions. STM executes transactions speculatively in parallel and monitors memory locations accessed by active transactions. If executing transactions do not conflict over shared memory locations, then they safely commit. In the event of a conflict, only one transaction can proceed and the rest must abort and restart. Transactions log operations during their execution so that they can restore the state of the program before the transaction if a rollback is needed.

The size of a transaction has a significant impact on performance. If the transaction is too short, then the overhead of STM APIs exceeds the performance gain of parallel execution and may lead to an STM program which is slower than the sequential version of the same program. On the other side, if the transaction is too large, then the cost of rollbacks in applications with a high abort rate may reduce the speedup in STM applications.

Xiao et al. [61] presents a method of predicting the close-to-optimal size of a transaction using linear regression and decision trees. The authors first identify potential transactions in NAS benchmarks. Potential transactions are determined by applying DiscoPoP profiler on the benchmarks and analyzing the CU graphs produced by the profiler. Table 5.4 shows the number of transactions found in each of the benchmarks. Among the 46 transactions, 34 are used for training the linear regression model and 12 are used for the validation.

The optimization method considers two more features other than the size of a transaction: the size of the write-set and the size of the read-set. A transaction uses a write-set and a read-set

Table 5.4: Number of transactions in NAS benchmarks. Transactions are determined by analyzing the output of the DiscoPoP profiler.

Benchmark	Number of transactions
LU	6
BT	12
CG	4
EP	3
IS	6
MG	6
FT	9
total	46

to record memory locations that it writes and reads, respectively. The write-set and the read-set are usually maintained as linked lists. Take a write-set as an example. When a transaction writes into a shared memory location, it inserts a new node to the linked list. During commit, the transaction traverses the linked list to acquire locks and update the memory with new transactional data. If the transaction fails to acquire a lock, then it aborts and restarts. So, a transaction with a large write-set is more likely to abort.

The linear regression model trained using the three features is not as accurate as expected. The R^2 metric, which indicates how the data fit the model, is only 45% for the trained model. Taking more features into consideration does not improve the results. However, further investigations of the trained model reveals that the error rate for transactions falls into three categories: transactions with large negative error (class1), transactions with large positive error (class2), and transactions with small error (class3). This results inspired us to use a decision tree to decide the class first, and then use a separate linear regression model for each class. On average, the mixed model decreases the error rate from from 59% to 2.8%.

This application shows that the parallelization opportunities produced by our method are not limited to a specific parallel programming model. In Chapter 4, we have parallelized programs and benchmarks using OpenMP and TBB. And this application shows that they can also be implemented using software transactional memory.

5.3 Detecting Communication Patterns on Multicore Systems

The performance of parallel applications very often depends on efficient communication. This is as true for message passing as it is for communication via shared variables. Knowing the communication pattern of a shared-memory kernel can therefore be important to discover performance bottlenecks such as true sharing or to support software-hardware co-design. In shared-memory programming, communication often follows the pattern of producer and consumer. The producer thread writes a variable, after which the consumer thread reads the written

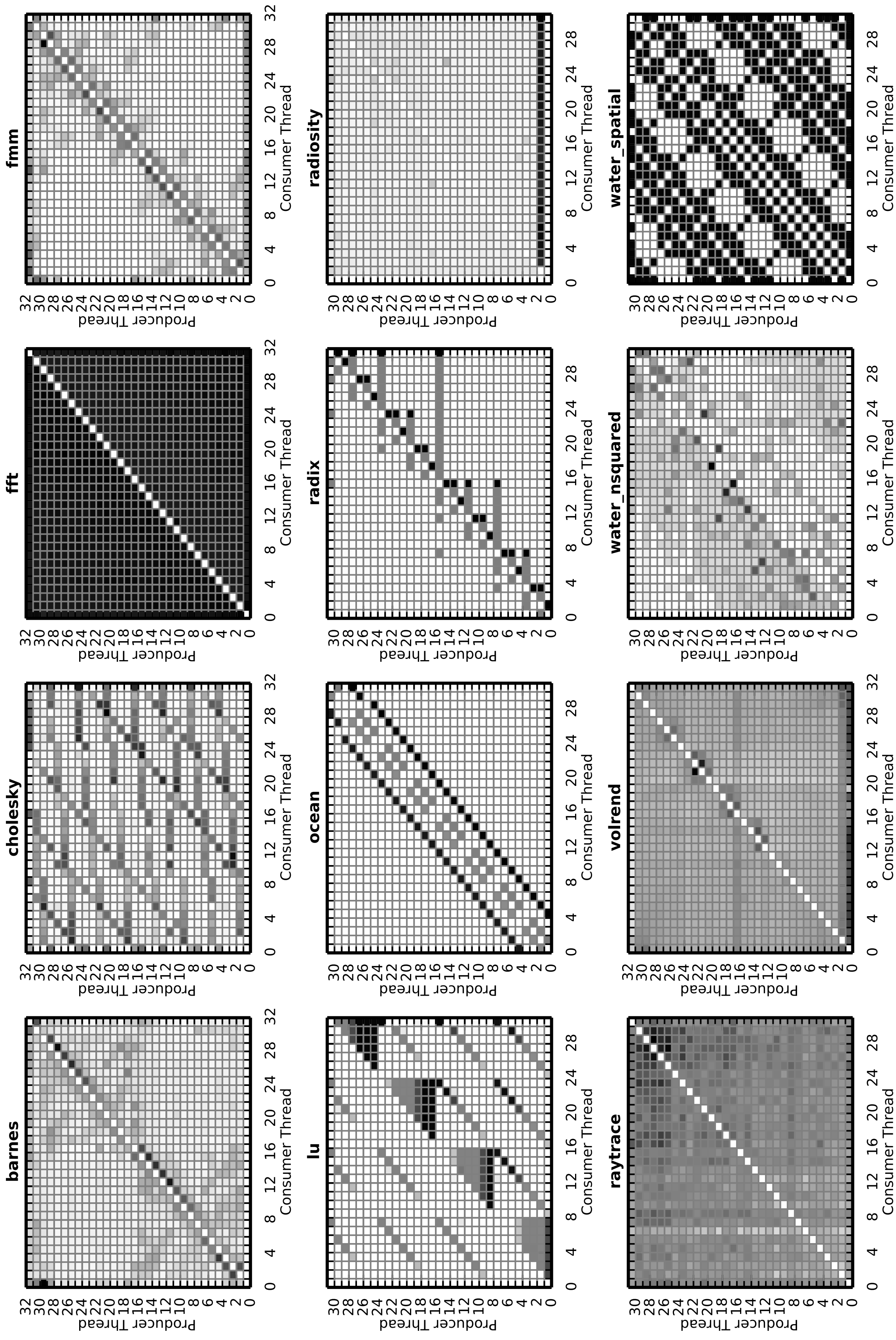


Figure 5.1: Communication patterns of splash2x benchmarks derived from the output of the DiscoPoP profiler.

value. The read happens before the next write occurs. Such a pattern can be represented as a matrix, showing the communication intensity between producer and consumer threads.

Producer-consumer behavior describes a read-after-write relation between memory operations, which can be easily derived from the RAW dependences produced by our profiler. With detailed information such as thread IDs available, we can generate the communication matrix directly from the output of our profiler. Arya Mazaheri et al. [107] produces the communication patterns for splash2x benchmarks [108], which are shown in Figure 5.1. The ticks of the vertical and horizontal axes represent producer and consumer threads, respectively. The darker the square the stronger the communication between the the two threads. Compared to a former analysis by Barrow-Williams et al. [109], we identified exactly the same communication patterns.

Former approaches [109, 110] that characterize communication patterns are usually built on top of simulators, which can easily have a slowdown of more than a factor of 1,000× if in-order processing is required. Unfortunately, in-order processing is required to produce communication patterns because producers and consumers need to be distinguished. With the help of our profiler, the same communication patterns can be obtained more efficiently since our profiler has only a 261× slowdown on average when profiling multi-threaded Starbench benchmarks.

This applications shows that the output of DiscoPoP profiler is capable of supporting various program analysis and optimization techniques, including program behavior analysis and auto tuning for both sequential and parallel programs.

5.4 Summary

In this chapter, we have introduced three applications of the profiler other than parallelism discovery, which are 1) characterizing features of DOALL loops, 2) determining optimal parameters for software transactional memory, and 3) detecting of communication patterns on multicore systems. These applications show that the DiscoPoP dependence profiler is a generic data-dependence profiler that supports multiple data-dependence based program analyses.

6 Conclusion and Outlook

This thesis presents a novel dynamic program analysis framework for the discovery of potential parallelism in sequential programs. The framework contains two main components: an efficient data-dependence profiler and a set of parallelism discovery algorithms based on a language-independent concept called computational unit. The framework is designed to be generic to support program analysis techniques other than parallelism discovery.

Dynamic data-dependence profilers are well-known for their high overhead in time and memory consumption. To keep the profiling overhead within reasonable limits, traditional profilers are usually customized so that only the information needed for a specific analysis or tool is collected. This solution leads to a dissatisfactory situation: every time a new analysis tool is constructed, existing profilers cannot be reused. Creating a new one is not only expensive and inconvenient, but it also makes the final analyses or tools hard to compare since they are based on different profiling techniques.

To enable reuse without having to accept compromises in terms of efficiency, we presented a parallel and lock-free data-dependence profiler that can serve as a uniform basis for different dependence-based analyses. While its time and space overhead stays within practical limits, our profiler also supports multi-threaded code. In this way, it supports not only date-dependence analyses for multi-threaded code, but also tuning and debugging approaches where the necessary information can be derived from dependences. While performing an exhaustive dependence search with 16 profiling threads, our lock-free parallel design limited the average slowdown to 78× and 93× for sequential NAS and Starbench applications, respectively. Using a signature with 10^8 slots, the memory consumption did not exceed 649 MB (NAS) and 1390 MB (Starbench), while producing less than 0.4% false positives and less than 0.1% false negatives.

In this thesis, we introduce a novel concept called computational unit (CU), and use CUs to represent sequential programs. CUs enable parallelism discovery for code sections that are not aligned with source language structures. A sequential program is represented as a set of CUs and data dependences among them, which we call a CU graph. We further introduced CU-based parallelism discovery methods that unify the identification of DOALL loops, DOACROSS loops, SPMD tasks, and MPMD tasks. Our approach found 92.5% of the parallel loops in NAS benchmarks and successfully identified SPMD tasks and MPMD tasks at different level of language constructs. Furthermore, we provide an effective ranking method, selecting the most appropriate parallel opportunities for the user. Our results show that 70% of the implemented parallelism in NPB can be explored by examining only the top 30% of our suggestions.

Nonetheless, several enhancement opportunities arise. For the data-dependence profiler, we believe that combining our method with static techniques will further reduce the time and space overhead substantially, if a slightly lower accuracy is still acceptable. Moreover, designing the shadow memory in a more efficient way could reduce the memory footprint. For the parallelism discovery methods, further efforts will be directed towards a more precise estimation of parallelization effort and expected speedup to give the users a more comprehensible overview of the parallelism contained in the target program. We also want to explore more potentials of the concept of CUs to support multiple parallel programming models and different program analyses other than parallelism discovery. We have shown one main application, parallelism discovery, and three other applications of our program analysis framework in this thesis, which are characterizing features of DOALL loops, determining optimal parameters for software transactional memory, and detecting of communication patterns on multicore systems. We are using our framework for developing other program analyses, such as techniques needed for energy-oriented program analysis and auto-tuning. Overall, we believe that the work presented in this thesis provides the foundation for a both comprehensive and practical tool that can significantly help programmers parallelize large numbers of sequential legacy code.

Bibliography

[1] Ralph E. Johnson. Software development is program transformation. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 177–180. ACM, 2010.

[2] `https://en.wikipedia.org/wiki/Parallel_computing`. [Online; accessed 22-March-2016].

[3] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[4] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, Oct 1966.

[5] James E. Kelley, Jr and Morgan R. Walker. Critical-path planning and scheduling. In *Proceedings of the 1959 Eastern Joint IRE-AIEE-ACM Computer Conference*, pages 160–173, New York, NY, USA, dec. ACM.

[6] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[7] Mary Jean Harrold, Loren Larsen, John Lloyd, David Nedved, Melanie Page, Gregg Rothermel, Manvinder Singh, and Michael Smith. Aristotle: A system for development of program analysis based tools. In *Proceedings of the 33rd Annual on Southeast Regional Conference*, ACM-SE 33, pages 110–119, New York, NY, USA, 1995. ACM.

[8] `http://frama-c.com/index.html`. [Online; accessed 12-April-2015].

[9] `http://www.semdesigns.com/Products/Parlanse/index.html`. [Online; accessed 12-April-2015].

[10] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 458–469. ACM, 2011.

[11] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, CGO '04, pages 75–88, Washington, DC, USA, 2004. IEEE Computer Society.

[12] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9):1088–1098, September 1988.

[13] Xiangyu Zhang, Armand Navabi, and Suresh Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 47–58. IEEE Computer Society, 2009.

[14] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, June 2007.

[15] Alain Ketterlin and Philippe Clauss. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 45, pages 437–448. IEEE Computer Society, 2012.

[16] `https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library`. [Online; accessed 26-July-2016].

[17] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[18] Vladimir Subotic, Eduard Ayguadé, Jesus Labarta, and Mateo Valero. Automatic exploration of potential parallelism in sequential applications. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Proceedings of the 29th International Supercomputing Conference (ISC)*, pages 156–171. Springer International Publishing, 2014.

[19] `http://software.intel.com/en-us/intel-advisor-xe`. [Online; accessed 12-April-2015].

[20] `https://silexica.com/products`. [Online; accessed 15-March-2016].

[21] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS: An integrated framework for mpsoc application parallelization. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 754–759. ACM, 2008.

[22] `http://www.criticalblue.com/prism-technology.html`. [Online; accessed 12-April-2015].

[23] `http://www.st.cs.uni-saarland.de/javaslicer/`. [Online; accessed 22-March-2016].

[24] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 3–14, New York, NY, USA, 2009. ACM.

[25] Georgios Tournavitis and Björn Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 377–388, New York, NY, USA, 2010. ACM.

[26] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[27] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. A survey on thread-level speculation techniques. *ACM Computing Surveys (CSUR)*, 49(2):22:1–22:39, June 2016.

[28] `https://software.intel.com/en-us/articles/intel-c-and-c-compilers-features-and-supported-platforms`. [Online; accessed 27-July-2016].

[29] Automatic parallelization with intel compilers. Technical report, Intel Corporation, 2011.

[30] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[31] Vincent Loechner and Doran K Wilde. Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, 25(6):525–549, 1997.

[32] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1248–1278, 1994.

[33] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Vasilache Nicolas . Graphite: Loop optimization based on the polyhedral model for GCC. In *4th GCC Developer's Summit*, 2006.

[34] `http://www.cs.umd.edu/projects/omega/`. [Online; accessed 18-September-2016].

[35] `http://icps.u-strasbg.fr/polylib/`. [Online; accessed 18-September-2016].

[36] `http://www.cloog.org/`. [Online; accessed 18-September-2016].

[37] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[38] Lawrence Rauchwerger and David Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *ACM SIGPLAN Notices*, 30(6):218–232, June 1995.

[39] Alexandra Jimborean, Philippe Clauss, Benoît Pradelle, Luis Mastrangelo, and Vincent Loechner. Adapting the polyhedral model as a framework for efficient speculative parallelization. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 295–296, New York, NY, USA, 2012. ACM.

[40] Alexandra Jimborean, Luis Mastrangelo, Vincent Loechner, and Philippe Clauss. VMAD: An advanced dynamic program analysis and instrumentation framework. In *Proceedings of the 21st International Conference on Compiler Construction*, pages 220–239, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[41] Alexandra Jimborean, Philippe Clauss, Jean-François Dollinger, Vincent Loechner, and Juan Manuel Martinez Caamaño. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *International Journal of Parallel Programming*, 42(4):529–545, 2014.

[42] Juan Manuel Martinez Caamaño, Willy Wolff, and Philippe Clauss. Code bones: Fast and flexible code generation for dynamic and speculative polyhedral optimization. In *Proceedings of the 22nd International European Conference on Parallel and Distributed Computing*, Euro-Par 2016, pages 225–237. Springer International Publishing, August 2016.

[43] `http://www.appentra.com/products/parallware/`. [Online; accessed 12-April-2015].

[44] Mehdi Amini, Onig Goubier, Serge Guelton, Janice Onanian Mcmahon, François-Xavier Pasquier, Grégoire Péan, and Pierre Villalon. Par4All: From convex array regions to heterogeneous computing. In *Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques*, IMPACT 2012, 2012.

[45] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Proceedings of the 17th International Conference on Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 132–146. Springer Berlin Heidelberg, 2008.

[46] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[47] Sang-Ik Lee, TroyA. Johnson, and Rudolf Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 539–553. Springer Berlin Heidelberg, 2004.

[48] `http://en.wikipedia.org/wiki/Nanohub`. [Online; accessed 12-April-2015].

[49] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN Con-*

ference on Programming Language Design and Implementation, PLDI '09, pages 177–187, New York, NY, USA, 2009. ACM.

[50] Kevin Streit, Clemens Hammacher, Johannes Doerfert, Sebastian Hack, and Andreas Zeller. Generalized Task Parallism. *ACM Transactions on Architecture and Code Optimiziation*, 12(1), March 2015.

[51] `http://llvm.org/`. [Online; accessed 26-July-2016].

[52] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.

[53] `http://llvm.org/docs/WritingAnLLVMPass.html`. [Online; accessed 15-March-2016].

[54] Zhen Li, Ali Jannesari, and Felix Wolf. An efficient data-dependence profiler for sequential and parallel programs. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*, IPDPS '15, pages 484–493, 2015.

[55] Zhen Li, Michael Beaumont, Ali Jannesari, and Felix Wolf. Fast data-dependence profiling by skipping repeatedly executed memory operations. In *Proceedings of 15th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), Zhangjiajie, China*, volume 9531 of *Lecture Notes in Computer Science*, pages 583–596. Springer International Publishing, November 2015.

[56] Zhen Li, Rohit Atre, Zia Ul Huda, Ali Jannesari, and Felix Wolf. Unveiling parallelization opportunities in sequential programs. *Journal of Systems and Software*, 117:282–295, Jul 2016.

[57] Zhen Li, Ali Jannesari, and Felix Wolf. Discovery of potential parallelism in sequential programs. In *Proceedings of the 42nd International Conference on Parallel Processing*, PSTI '13, pages 1004–1013. IEEE Computer Society, 2013.

[58] Zhen Li, Rohit Atre, Zia Ul-Huda, Ali Jannesari, and Felix Wolf. DiscoPoP: A profiling tool to identify parallelization opportunities. In *Tools for High Performance Computing 2014*, pages 37–54. Springer International Publishing, 2015.

[59] Zhen Li, Bo Zhao, Ali Jannesari, and Felix Wolf. Beyond data parallelism: Identifying parallel tasks in sequential programs. In *Proceedings of 15th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), Zhangjiajie, China*, volume 9531 of *Lecture Notes in Computer Science*, pages 569–582. Springer International Publishing, November 2015.

[60] Daniel Fried, Zhen Li, Ali Jannesari, and Felix Wolf. Predicting parallelization of sequential programs using supervised learning. In *Proceedings of the 12th IEEE International Conference on Machine Learning and Applications (ICMLA), Miami, FL, USA*, pages 72–77. IEEE Computer Society, December 2013.

[61] Yang Xiao, Zhen Li, Ehsan Atoofian, and Ali Jannesari. Automatic optimization of software transactional memory through linear regression and decision tree. In *Proceedings*

of 15th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), Zhangjiajie, China, volume 9531 of *Lecture Notes in Computer Science*, pages 61–73. Springer International Publishing, November 2015.

[62] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. Prospector: Discovering parallelism via dynamic data-dependence profiling. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*, HOTPAR '10, 2010.

[63] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. Function level parallelism driven by data dependencies. *ACM SIGARCH Computer Architecture News*, 35(1):55–62, March 2007.

[64] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 105–118. IEEE Computer Society, 2005.

[65] David I. August, Jialu Huang, Stephen R. Beard, Nick P. Johnson, and Thomas B. Jablin. Automatically exploiting cross-invocation parallelism using runtime information. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '13, pages 1–11. IEEE Computer Society, 2013.

[66] R. Govindarajan and Jayvant Anantpur. Runtime dependence computation and execution of loops on heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '13, pages 1–10. IEEE Computer Society, 2013.

[67] John M. Ye and Tianzhou Chen. Exploring potential parallelism of sequential programs with superblock reordering. In *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, HPCC '12, pages 9–16. IEEE Computer Society, 2012.

[68] Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus: A tool suite for program analysis and reverse engineering. In *Proceedings of the 11th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe'06, pages 71–82, Berlin, Heidelberg, 2006. Springer-Verlag.

[69] PO Bobbie. Partitioning programs for parallel execution: A case study in the intel ipsc/2 environment. *International journal of mini & microcomputers*, 19(2):84–96, 1997.

[70] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM.

[71] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[72] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.

[73] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[74] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Computing*, 36(9):531–551, September 2010.

[75] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. SD3: A scalable approach to dynamic data-dependence profiling. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 43, pages 535–546. IEEE Computer Society, 2010.

[76] Hongtao Yu and Zhiyuan Li. Multi-slicing: A compiler-supported parallel approach to data dependence profiling. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 23–33. ACM, 2012.

[77] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2), May 2007.

[78] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 123–133. IEEE Computer Society, 2007.

[79] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with LLVM compiler. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 110–114. Springer-Verlag, 2012.

[80] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[81] Ali Jannesari and Walter F. Tichy. Library-independent data race detection. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2606–2616, October 2014.

[82] Zia Ul Huda, Ali Jannesari, and Felix Wolf. Using template matching to infer parallel design patterns. *ACM Transactions on Architecture and Code Optimization*, 11(4):64:1–64:21, January 2015.

[83] Zia Ul Huda, Ali Jannesari, and Felix Wolf. Automatic parallel pattern detection in the algorithm structure design space. In *Proceedings of the 30th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 43–52, Chicago, USA, May 2016.

[84] S. Seo, G. Jo, and J. Lee. Performance characterization of the nas parallel benchmarks in opencl. In *Proceedings of the 2011 IEEE International Symposium on Workload Character-*

ization (IISWC), pages 137–148, Nov 2011.

[85] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[86] Michael Andersch, Ben Juurlink, and Chi Ching Chi. A benchmark suite for evaluating parallel programming models. In *Proceedings 24th Workshop on Parallel Systems and Algorithms*, PARS '11, pages 7–17, 2011.

[87] Korbinian Molitorisz, Jochen Schimmel, and Frank Otto. Automatic parallelization using autofutures. In *Proceedings of the 2012 International Conference on Multicore Software Engineering, Performance, and Tools*, MSEPT'12, pages 78–81, Berlin, Heidelberg, 2012. Springer-Verlag.

[88] Ali Jannesari, Markus Westphal-Furuya, and Walter F. Tichy. Dynamic data race detection for correlated variables. In *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP'11, pages 14–26, Berlin, Heidelberg, 2011. Springer-Verlag.

[89] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[90] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 1st edition, 2004.

[91] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[92] `https://en.wikipedia.org/wiki/Side_effect_(computer_science)`. [Online; accessed 22-March-2016].

[93] Ron Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of International Conference on Parallel Processing*, pages 836–844. IEEE Computer Society, aug 1986.

[94] Sergei Krestianskov. Detection of loop-specific parallel patterns in sequential programs. Master's thesis, RWTH Aachen, May 2015.

[95] James Reinders. *Intel Threading Building Blocks*. O'Reilly Media, July 2007.

[96] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[97] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[98] `http://clang.llvm.org/`. [Online; accessed 22-March-2016].

[99] Alfred Strey. A comparison of OpenMP and MPI for neural network simulations on a sunfire 6800. In *PARCO*, pages 201–208, 2003.

[100] `http://zlib.net/pigz/`. [Online; accessed 22-March-2016].

[101] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software behavior oriented parallelization. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’07, pages 223–234, New York, NY, USA, 2007. ACM.

[102] `http://bzip2smp.sourceforge.net/`. [Online; accessed 22-March-2016].

[103] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995.

[104] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[105] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119 – 139, 1997.

[106] Trevor. Hastie, Robert. Tibshirani, and J Jerome H Friedman. *The elements of statistical learning*, volume 1. Springer New York, 2001.

[107] Arya Mazaheri, Ali Jannesari, Abdolreza Mirzaei, and Felix Wolf. Characterizing loop-level communication patterns in shared memory applications. In *Proceedings of the 44th International Conference on Parallel Processing*, pages 759–768, September 2015.

[108] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA ’95, pages 24–36. ACM, 1995.

[109] Nick Barrow-Williams, Christian Fensch, and Simon Moore. A communication characterisation of Splash-2 and Parsec. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, IISWC ’09, pages 86–97. IEEE Computer Society, 2009.

[110] Sucheta Chodnekar, Viji Srinivasan, Aniruddha S. Vaidya, Anand Sivasubramaniam, and Chita R. Das. Towards a communication characterization methodology for parallel applications. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, HPCA ’97, pages 310–. IEEE Computer Society, 1997.