High Performance with Prescriptive Optimization and Debugging

Jensen, Nicklas Bo; Probst, Christian W.; Karlsson, Sven

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

High Performance with Prescriptive Optimization and Debugging

Nicklas Bo Jensen

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

Parallel programming is the dominant approach to achieve high performance in computing today. Correctly writing efficient and fast parallel programs is a big challenge mostly carried out by experts. We investigate optimization and debugging of parallel programs.

We argue that automatic parallelization and automatic vectorization is attractive as it transparently optimizes programs. The thesis contributes an improved dependence analysis for explicitly parallel programs. These improvements lead to more loops being vectorized, on average we achieve a speedup of 1.46 over the existing dependence analysis and vectorizer in GCC.

Automatic optimizations often fail for theoretical and practical reasons. When they fail we argue that a hybrid approach can be effective. Using compiler feedback, we propose to use the programmer's intuition and insight to achieve high performance. Compiler feedback enlightens the programmer why a given optimization was not applied, and suggest how to change the source code to make it more amenable to optimizations. We show how this can yield significant speedups and achieve 2.4 faster execution on a real industrial use case.

To aid in parallel debugging we propose the prescriptive debugging model, which is a user-guided model that allows the programmer to use his intuition to diagnose bugs in parallel programs. The model is scalable, yet capable enough, to be general-purpose. In our evaluation we demonstrate low run time overhead and logarithmic scalability. This enable the model to be used on extremely large parallel systems.

Resume

Parallel programmering er den dominerende måde at opnå høj computer ydeevne i dag. At skrive effektive og korrekte parallelle programmer er en stor udfordring der primært bliver udført af eksperter. Afhandlingen undersøger tre aspekter af parallel programmering.

Først argumenterer vi for at automatisk parallelisering og vektorisering er attraktivt, da det er en transparent måde at opnå høj ydeevne. Denne afhandling bidrager med en forbedret afhængigheds analyse for eksplicit parallelle programmer. Disse forbedringer leder til at flere løkker bliver vektoriseret, vi opnår en gennemsnitlig 1,46 gange hurtigere afvikling af en række programmer.

Automatiske optimeringer fejler desværre ofte af teoretiske og praktiske grunde. Det andet undersøgte aspekt er når de fejler. Her argumenterer vi for at en hybrid tilgang kan være effektiv. Ved at bruge kompiler feedback, foreslår vi at bruge programmørens intuition og indsigt til at opnå høj ydeevne. Kompiler feedback fortæller programmøren hvorfor en given optimering ikke er anvendt, og foreslår hvordan kildekoden kan gøres mere medgørlig i forhold til optimering. Vi viser på en industriel use case, hvordan dette kan lede til forbedringer i ydeevne, op til 2,4 gange hurtigere afvikling.

Det sidste aspekt er parallel debugging. Vi foreslår den prescriptive debugging model, en bruger guidet model der tillader programmøren at kodificere sin intuition til at fejlsøge parallelle programmer. Modellen er skalerbar, og dog stadig generel anvendelig. I vores evaluering viser vi et lavt overhead og logaritmisk skalerbarhed. Dette tillader at anvende modellen på ekstremt store parallelle systemer.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science, in the Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D. degree in engineering.

The Ph.D. study was carried out under supervision of Associate Professor Christian W. Probst and Associate Professor Sven Karlsson in the period from October 2013 to September 2016.

The Ph.D. project was funded by the European collaborative project COPCAMS (Cognitive and Perceptive Camera Systems) funded jointly by the ARTEMIS Joint Undertaking and national governments under GA number 332913.

Kongens Lyngby, 30-September-2016
Nicklas Bo Jensen

Acknowledgments

This thesis would not be possible without the support and help of many people.

I would like to thank my supervisors Christian W. Probst and Sven Karlsson for their valuable time and advice. I know the process of trying to turn me into a scientist have not been easy, but I appreciate your continued effort.

I would also like to thank my awesome colleagues for their help and guidance through this process. This include Ann-Cathrin and Karin Tunder. I have been lucky to work together on projects with Pascal Schleuniger, Maxwell Walter, Andreas Hindborg, Laust Brock-Nannestad, Martin Madsen, Artur Podobas and Lars Bonnichsen. Special thanks to Artur for keeping me company at the end of my PhD and for attempting to keep me sane. It has been a pleasure to share an office with you and hope you achieve your goals.

Thank you to the guys at Lawrence Livermore National Laboratory for hosting me, especially thanks to Greg Lee, Dong Ahn, Matt Legendre, Martin Schultz and Niklas Nielsen for making my stay there a pleasure.

Thanks to Arla and Løgismose for good koldskål, always waiting in the refrigerator or the nearby Netto when needed to keep me alive and happy.

Thanks to my supporting family and friends for their support and understanding. Especially thanks to my girlfriend Charlotte, her help and patience when traveling or frequently being mentally absent.

Publications

This thesis is partially based on the following peer-reviewed publications:

Nicklas Bo Jensen, Per Larsen, Razya Ladelsky, Ayal Zaks, and Sven Karlsson. "Guiding Programmers to Higher Memory Performance". In: *Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)*. 2012

Nicklas Bo Jensen, Sven Karlsson, and Christian W. Probst. "Compiler Feedback using Continuous Dynamic Compilation during Development". In: *Workshop on Dynamic Compilation Everywhere (DCE)*. 2014

Nicklas Bo Jensen, Christian W. Probst, and Sven Karlsson. "Code Commentary and Automatic Refactorings using Feedback from Multiple Compilers". In: *Swedish Workshop on Multicore Computing (MCC)*. 2014

Nicklas Bo Jensen, Sven Karlsson, Niklas Quarfot Nielsen, Gregory L. Lee, Dong H. Ahn, Matthew Legendre, and Martin Schulz. "DySectAPI: Scalable Prescriptive Debugging". In: *Proceedings of the 2014 ACM/IEEE Conference on Supercomputing*. SC. Poster and extended abstract. 2014. URL: `http://sc14.supercomputing.org/sites/all/themes/sc14/files/archive/tech_poster/tech_poster_pages/post237.html`

Nicklas Bo Jensen, Niklas Quarfot Nielsen, Gregory L. Lee, Sven Karlsson, Matthew LeGendre, Martin Schulz, and Dong H. Ahn. "A Scalable Prescriptive Parallel Debugging Model". In: *Proceedings of the International Parallel & Distributed Processing Symposium*. IPDPS. 2015. DOI: `10.1109/IPDPS.2015.15`

Nicklas Bo Jensen and Sven Karlsson. *Improving Loop Dependency Analysis.* Journal manuscript submitted for publication. 2016

The following peer-reviewed publications are closely related to the thesis content, but the publications are not contained in the thesis:

Andreas Erik Hindborg, Pascal Schleuniger, Nicklas Bo Jensen, and Sven Karlsson. "Hardware Realization of an FPGA Processor – Operating System Call Offload and Experiences". In: *Conference on Design and Architectures for Signal and Image Processing (DASIP).* 2014. DOI: 10.1109/DASIP.2014.7115604

Andreas Hindborg, Pascal Schleuniger, Nicklas Bo Jensen, Maxwell Walter, Laust Brock-Nannestad, Lars Bonnichsen, Christian W. Probst, and Sven Karlsson. "Automatic Generation of Application Specific FPGA Multicore Accelerators". In: *The Asilomar Conference on Signals, Systems, and Computers.* 2014. DOI: 10.1109/ACSSC.2014.7094700

Andreas Erik Hindborg, Nicklas Bo Jensen, Pascal Schleuniger, and Sven Karlsson. "State of the Akvario Project". In: *Workshop on Architectural Research Prototyping (WARP).* 2015. URL: http://www.csl.cornell.edu/warp2015/abstracts/hindborg-akvario-warp2015.pdf

Maxwell Walter, Pascal Schleuniger, Andreas Erik Hindborg, Carl Christian Kjærgaard, Nicklas Bo Jensen, and Sven Karlsson. "Experiences Implementing Tinuso in gem5". In: *Second gem5 User Workshop.* 2015. URL: http://www.m5sim.org/wiki/images/f/f5/2015_ws_16_gem5-workshop_mwalter.pptx

Nicklas Bo Jensen, Pascal Schleuniger, Andreas Hindborg, Maxwell Walter, and Sven Karlsson. "Experiences with Compiler Support for Processors with Exposed Pipelines". In: *IEEE International Parallel & Distributed Processing Symposium: Reconfigurable Architectures Workshop.* IPDPSW. 2015. DOI: http://dx.doi.org/10.1109/IPDPSW.2015.9

Last, the thesis is partially based on the following deliverable prepared for the COPCAMS project:

Nicklas Bo Jensen, ed. *D2.5 — Final Results for Exploration Tools.* COPCAMS. Deliverable for the Cognitive and Perceptive Camera Systems project. 2016

Contents

Chapter 1

Introduction

Computing is central to the modern world. We use computers in many forms every day. To keep up high performance, designers have relied on improvements in computer architecture and manufacturing technology allowing them to increase the clock frequency as a driver to increase performance. However, in the 21st century scaling of the clock frequency has stopped due to power consumption.

It was estimated that information and communication technology accounted for 4.6% of the global electricity consumption in 2012, with one projection estimating a 14% share in 2020 [Hed+14]. For supercomputers the performance of running parallel scientific application has increased more than 10,000-fold from 1992 to 2007, while performance per watt only increased 300-fold [FC07]. Thus, improving energy efficiency is one the main challenges of computing. There exist many ways to improve the energy efficiency — for example by utilizing the specialized hardware available and by increasing the performance reducing run time. For systems designed with a fixed power budget it allows for more performance within the same power budget. Improving the energy efficiency will reduce the economical and environment impact of computing.

Chip manufacturers have moved to uphold the increasing need for processor performance using parallelism. This trend has dominated across computing from high-performance supercomputers, warehouse scale computing, servers, general purpose machines, mobile and embedded.

To efficiently program parallel machines, we rely on parallel programming. Current parallel programming models and tools are often too low-level and requires architecture specific tuning to achieve high performance. This makes parallel programming hard. Given the complexity of parallel programming and the sheer number of parallel machines available today, we are not efficiently using the available resources.

Systems available today are so complex that only expert programmers can comprehend all aspects of them when writing software. When a programmer cannot assess and manage the complexities in a system, they will produce lower quality code. In this thesis I aim to show ways moving forward allowing us to take advantage of the specialized hardware available to us today, while placing as little as possible of a burden on the programmer as possible.

Some of the challenges with parallel programming is:

1. How do we decompose a problem into subproblems that can be executed efficiently in parallel?
2. As we scale the amount of parallelism how do we ensure we continue to achieve the desirable speedup?
3. Once we have achieved a parallel implementation, how do we debug it? Debugging parallel programs proposed some unique challenges as it is very different from debugging sequential programs. The extra challenges include non-deterministic execution, parallel bugs such as deadlocks, and the sheer amount of information available at large scale.

These challenges motivate the thesis contributions that are centered around achieving high performance with prescriptive optimizations and prescriptive debugging.

Prescriptive optimizations are defined analogously to prescriptive analytics [RD14] in big data business analytics. Prescriptive optimization not only focuses on the why, how, when and what, but also propose actions to take advantage of the circumstances. Prescriptive optimization is an optimization regime where the programmer prescribes how a code should be optimized, and if not possible prescribe the necessary foundations to do so. Prescriptive optimizations allow programmers to achieve high performance without manually having to optimize code, but merely make the best out of automatic optimizations.

Prescriptive debugging is defined as a debugging model where the programmer's intuition is codified to reduce the error search space. The programmer

prescribes the debugging session before execution. This approach, in contrast to the existing debugging models, is scalable in terms of tool communication and volume of data produced for the programmer, yet capable enough, to serve general-purpose debugging.

Next, we will briefly cover the thesis contributions around prescriptive optimization and debugging.

1.1 Thesis Contributions

The main thesis contributions are:

1. **Compiler Feedback:**

   We show how it is possible to generate feedback in more practical ways with better refactorings. We also show how the feedback can contribute to mitigate issues due to memory optimizations, inlining, automatic parallelization and automatic vectorization. The feedback enables more aggressive compiler optimizations demonstrating its effectiveness.

2. **Improved Compiler Optimizations:**

   We improve the automatic vectorization capabilities of GCC, by taking into account dependences specified for explicitly parallel programs. The improved dependence analysis is able to remove many of the false dependences previously reported. These improvements lead to more loops being automatically vectorized. On average we achieve a speedup of 1.46 for a set of benchmarks.

3. **Prescriptive Debugging:**

   We propose the new prescriptive debugging model. It is scalable in terms of tool communication and volume of data produced for the programmer, yet capable enough, to serve general-purpose debugging. Through our evaluation we show logarithmic scalability allowing us to achieve extreme scalability in terms of system size. Last, the prototype has been applied to a real use case showing its strength in condensing the information presented to the programmer.

1.2 Synopsis

The rest of the dissertation is organized into the following ten chapters:

Chapter 2 reviews the theory and practice necessary for reading the remainder of the thesis.

Chapter 3 identifies the state-of-the-art in automatic vectorization with a study on compiler limitations with respect to automatic vectorization.

Chapter 4 presents the motivation and research questions of the thesis.

Chapter 5 describe related work and state-of-the-art, and relates it to the research in the thesis.

Chapter 6 describe compiler feedback for memory optimizations.

Chapter 7 introduce continuous compiler feedback system integrated into a development environment.

Chapter 8 introduce a feedback system using optimization reports from multiple compilers.

Chapter 9 introduce the OpenMP based dependence analysis and its improvement to automatic vectorization.

Chapter 10 introduce the prescriptive debugging model and evaluate its scalability in terms of system size and information presented.

Chapter 11 concludes the thesis by summarizing the research findings and discusses the potential impact.

Chapter 2

Technical Background

This chapter introduce the terminology necessary to understand the thesis contributions.

The sections each introduce different aspects of computing: the underlying hardware systems in Section 2.1.1, the programming models used Section 2.2 and program optimizations in Section 2.3.

2.1 Processing Elements

A *processor* is a circuit that performs the instructions of a computer program. *Processing elements* (PE) are the units that execute instructions inside a processor. Each processor defines an *instruction set architecture* (ISA). The ISA defines the interface for using the processor, how supported operations are encoded as instructions and how memory locations are accessed. Writing programs manually using the processors instruction set is very difficult and time consuming. Instead, we usually generate the instructions for a processor from higher level languages that are more suitable for writing programs.

An instruction can access two types of storage, a memory location or a register.

A register usually holds a single value, for example 64-bits of information. Access to registers is very fast, but a limited number of them are available. Usually between 14 and 63 registers are available. Memory is addressed indirectly using addresses, for example to implement array accesses such as `Array[index]`. Instructions can also operate on immediate values. In some ISAs most instructions can operate on both registers and memory, in some most ISA instructions can mostly operate on registers. Each ISA defines which instructions can operates on which types of storage.

The processor design strategy and its ISA can be classified as either being either a *reduced instruction set computing* (RISC) architecture or a *complex instruction set computing* (CISC) architecture [HP96]. Both RISC and CISC have advantages and disadvantages [PD80], but today RISC or a hybrid between RISC and CISC machines are dominating as all commercial ISAs in the past 30 years have been RISC or a hybrid between RISC and CISC [Pat15].

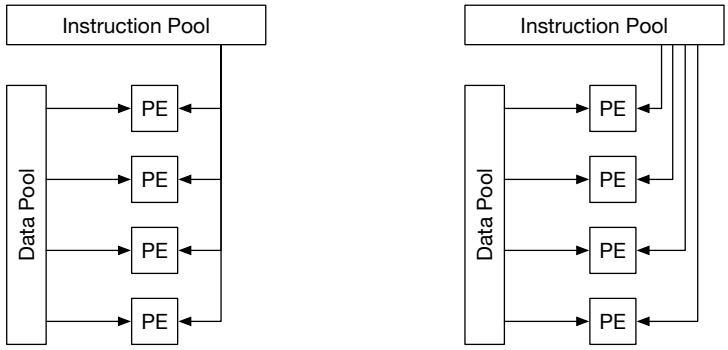
PEs can be organized in several ways to form a computer. There are three classifications of computers according to Flynn's taxonomy [Fly72] that are used in mainstream computing. *Single instruction, single data* (SISD) describing a sequential computer with no parallelism in the instruction and data streams. The *single instruction, multiple data* (SIMD) where each instruction operates on multiple data elements as illustrated in Figure 2.1a. The *multiple instruction, multiple data* (MIMD) consisting of multiple cores operating independently on different data as illustrated in Figure 2.1b. When multiple PEs are organized into a single computer we call it a *multi-core* computer. Today's modern mainstream architectures usually combine the MIMD and SIMD paradigms into a multi-core processor. Each core executes an independent thread, short for *thread* of execution. Each executing thread can furthermore make use of SIMD parallelism achieving the best of both worlds in a versatile system.

2.1.1 Driving Factors for Multi-Core SIMD Hardware

Moore's law as we know it today was put forward by Moore in 1975 predicting approximately a doubling in transistor counts every 18 months on a single chip [Moo75]. Moore's law is still in effect as seen in Figure 2.2. Closely, related to Moore's law is *Dennard scaling* [Den+74].

The interplay between Moore's law and Dennard scaling was highlighted by Bob Colwell in his DAC 2013 keynote: "Moore's Law gave us more transistors, Dennard scaling made them useful".

Dennard scaling is defined as:



(a) SIMD **(b)** MIMD

Figure 2.1: SIMD architecture illustrated in 2.1a and MIMD architecture illustrated in 2.1b.

$$\text{Dynamic Power} = \alpha C F V^2$$

Where α is the percentage time the transistors switch from logical zero to one, C is the capacitance or area, F is the frequency and V is the voltage. Dennard's observation was that voltage and current should be proportional to the dimensions of a transistor. This entails that as transistors gets smaller due to innovations in process technology, the necessary voltage and current also gets smaller. This allowed circuits to operate at a higher frequency at the same power. A higher frequency equals the processor performing instructions faster. However, Dennard scaling does not take into factors such as leakage current, which starts becoming important as transistors gets smaller [Boh07]. Dennard scaling have broken down around 2005 [Esm+11].

The effect of Dennard scaling failing is seen around 2005 where the frequency scaling of single-core processors is no longer one of the primary drivers behind single-core performance enhancements. This change in hardware has caused designers to hit several walls — the Power Wall, the Memory Wall and the ILP (instruction-level parallelism) Wall. The Power Wall is where we can put more transistors on a chip than we can afford to turn on due to power issues [Esm+11]. The Memory Wall is where the memory speed does not scale at the same rate as computation speed [WM95]. This entails that memory operations are expensive and computation is cheap relative to each other. The ILP Wall describes how the amount of parallelism on the instruction level is diminishing [Wal91].

To these ends processor designers have moved to multi-core systems, where the transistor count is spent integrating several cores, bigger caches and more

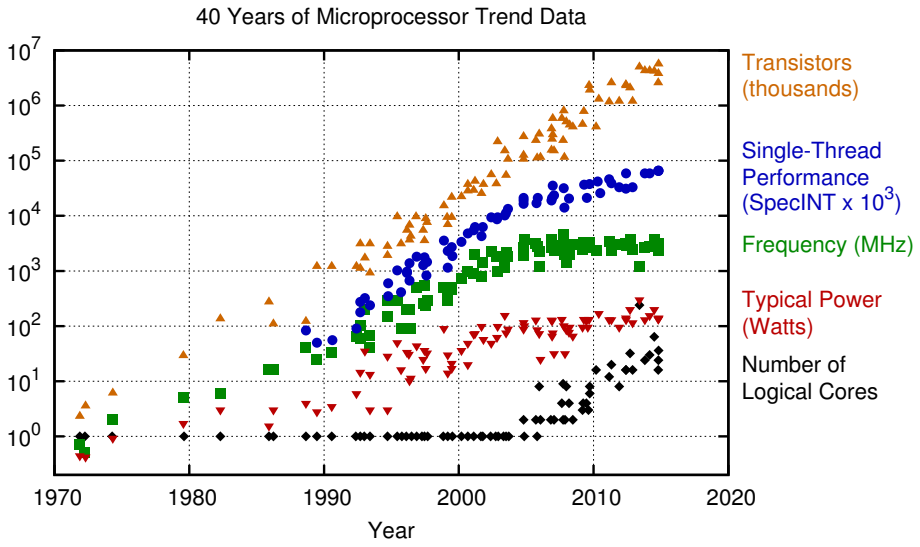


Figure 2.2: 40 years of processor trend. Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. New plot and data collected for the year 2010 to 2014 by K. Rupp under a Creative Commons Attribution 4.0 International Public License.

SIMD capabilities. A multi-core processor contains two or more processing cores in a single package. Each core operates independently executing different instructions on different data as per the MIMD architecture [Fly72]. At the same time each processor core can independently execute single instructions operating on multiple data as per the SIMD architecture [Fly72]. Most of today's processors, even processors designed for mobile devices, contain multiple cores with SIMD capabilities.

Looking at the Intel processor generations from the last decade we see how the numbers of cores has been growing steadily from one core in 2004 for Intel microarchitecture codenamed Nocona to 500 cores for Intel microarchitecture codenamed Knights Corner in 2014. Regarding SIMD the width of the vector units has increased significantly. From 128 bits in the 2004 Intel Nocona architecture to 512 in the Intel Knights Landing Architecture as we see in Figure 2.3. Furthermore, the number of vector registers have increased and the number of vector instructions in the instruction sets have increased as well. In fact, most new instructions added to Intel's instruction set architecture for the past decade have been vector instructions. For example, when the Intel Streaming SIMD Extensions (SSE) were introduced in 1999 they contained just 62 instructions in the ISA, in contrast the modern Advanced Vector Extensions 2 (AVX2) contains 1557 SIMD instructions as seen in Figure 2.3. These numbers show how the SIMD units in modern Intel processors have become increasingly powerful, in both width and number of instructions. Last, for AVX and AVX2 we have 16 256-bit registers compared to 32 512-bit registers for the newest AVX512 extensions. Clearly, this shows that SIMD units have gotten more advanced and complex. This complexity emphasizes how we need compiler support to efficiently take advantage of SIMD without placing an enormous burden on programmers.

2.1.2 Memory Organization

Memory is organized into a hierarchy simply because main memory operates much slower than the processor. Operating directly on main memory would be very inefficient for many programs. Instead, frequently used data is kept in a hierarchy of smaller and faster memories.

These smaller memories are called *caches*. Caches provide quick accesses to frequently accessed data. The closer to the processor, the faster the memory accesses, but the size of the cache will typically be smaller. To this end, caches are organized in a hierarchy, usually within two or three levels.

A typical cache hierarchy contains three layers, with a shared Level 3 cache and

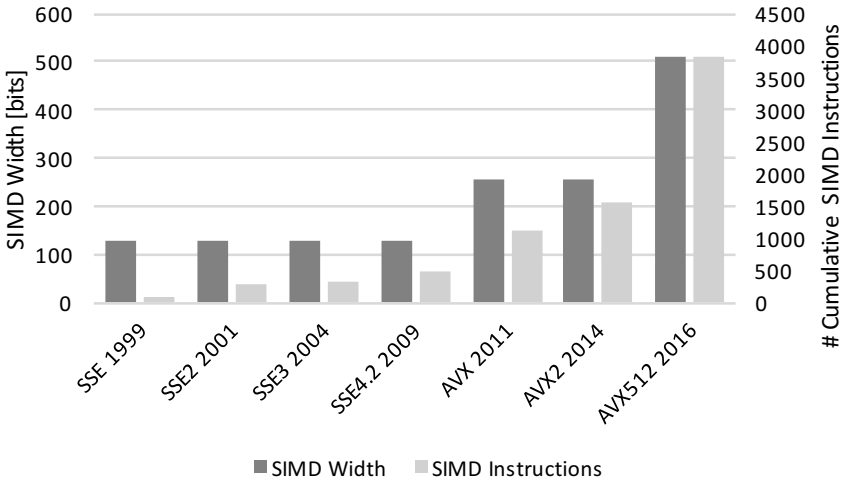


Figure 2.3: Intel processor trends in terms of SIMD width and cumulative counts of SIMD instructions for modern generations of Intel SIMD extensions. Product specifications are available at `ark.intel.com`. Cumulative SIMD instruction counts by Ayal Zaks [Zak15], used with permission.

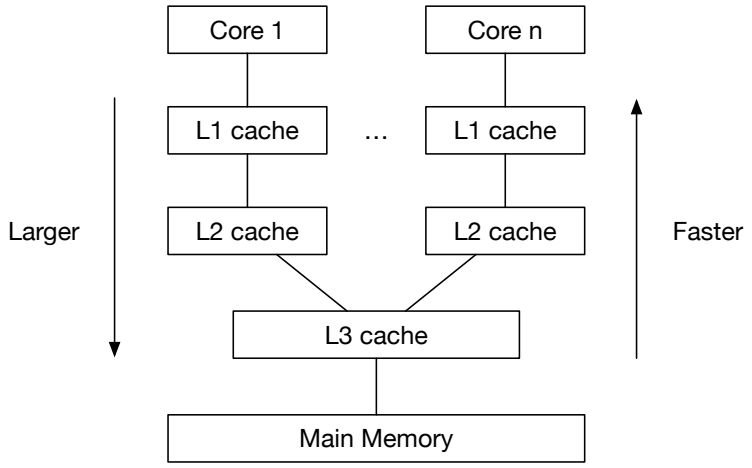


Figure 2.4: Example cache hierarchy organized in three layers.

private Level 2 and Level 1 caches as seen in Figure 2.4. The highest level of cache is shared between the cores in a multi-core processor to efficiently make use of the caches and allow the cores to share data.

2.2 Programming Models

A *programming model* defines the interface between the software and the hardware. It is an abstraction presented to the programmer of the underlying machine executing the software. Different Programming models target different types of machines, provide varying productivity for the programmer and more or less performance.

Parallel programming models define how a parallel machine is programmed. Typically, a parallel programming model is either under the *shared memory* or the *distributed memory* abstraction. The machine being programmed is typically using the same memory abstraction as the programming model. In the shared memory programming model, the programmer is given the abstraction that communication with other concurrently running threads is performed through shared memory. In the distributed memory programming model, program communicates using *message passing*.

2.2.1 Threads

A thread is an entity that can execute on a PE. A thread consists of an execution context together with the program instructions. A program can separate its execution into multiple threads. Each thread can run concurrently on the same processor core or in parallel several processor cores. A program can contain several threads grouped into a process. Threads are used to implement MIMD parallelism in shared memory programming models. One programming model for shared memory programming is OpenMP introduced next.

2.2.2 OpenMP

OpenMP is de-facto standard for developing parallel applications. OpenMP consists of a set of compiler directives, also called pragmas, and library routines. Loops annotated with OpenMP directives such as a *worksharing* loop, where work is shared among several threads, are characterized by the absence of dependences between loop iterations, i.e., each iteration of these loops can execute in parallel. The OpenMP specification defines the interface for compilers and runtime systems to create a compliant implementation [Boa15].

OpenMP is a shared memory programming model. OpenMP mainly focus on fork-join parallelism. A master thread distributes work to a set of worker threads when entering a parallel region. The master thread typically executes the sequential parts of the program.

Worksharing loops can have several schedules for how iterations are mapped to available threads. The iterations can statically be partitioned evenly among the threads. They can also be dynamically scheduled where thread executes a specified chunk of iterations before requesting a new chunk. The default chunk size is 1 if not specified by the programmer. Guided scheduling is similar to dynamic scheduling, but starts out with large chunks and decreases to a smaller chunk size to handle load imbalance. Auto lets the compiler and runtime choose a mapping and runtime defers the decision until runtime.

A sequential program specifies a total execution ordering describing the happens-before relationship. For a loop with memory operations, the order of these has to comply with the memory model. For an OpenMP worksharing loop, the parallel semantics define a partial execution where each iteration of the loop can be executed independently.

OpenMP also supports task-level parallelism. Tasking is powerful when units

```
1 #pragma omp parallel for
2 for(i=0; i<N; i++) {
3   for(j=0; j<N; j++) {
4     A[i][j] = A[i][j] + B[i][j];
5   }
6 }
```

Figure 2.5: OpenMP worksharing loop.

of work are generated dynamically. When a task encounters a task construct a new task is spawned. Tasks are dynamically scheduled to be executed by the set of threads available.

2.2.2.1 OpenMP SIMD

Automatic vectorization where the compiler automatically converts a sequential code into a using SIMD is an inherently hard problem that is not fully solved simply due to its complexity. Many have suggested programmer-guided vectorization. OpenMP 4.0 [Boa13] added support for programmer guided vectorization with the `#pragma omp simd` construct influenced by work from Caballero et al. [Cab+15] and Klemm et al. [Kle+12]. OpenMP 4.0 is a significant step forward in terms of programmer productivity compared with manual vectorization, but still requires significant effort compared to automatic vectorization.

It enables more vectorization by disabling alias and dependence analysis, and enabling more aggressive vectorization transformations. The programmer is required to determine that it is legal to vectorizable the loop and specify each reduction variable and other properties in the loop. OpenMP SIMD therefore puts significant burden on the programmer.

OpenMP SIMD supports several clauses for guiding the vectorization; these include `safelen` for safe vector lengths and `aligned` for specifying data alignment.

2.2.3 Message Passing

The distributed memory abstraction relies on message passing for communication given the lack of shared memory. All communication in message passing

abstractions are through either point-to-point messages or through collective operations such as a broadcast or reduction.

Usually, message passing is implemented with the single program, multiple data (SPMD) abstraction, a subset of MIMD. A single program is executed with its own data in parallel. Often a single execution of the program will be the master orchestrating the execution. The best known message passing implementation is the standardized Message Passing Interface (MPI) [For15]. MPI is implemented in a library to lower the development effort, but correctly implementing distributed programs is hard often leading to subtle bugs. The MPI-3 interface defines more than 430 routines [For15] and is thus a complicated library to use correctly and efficiently.

2.2.4 Writing Parallel Programs

Amdahl's law describes the relation between serial and parallel parts of a program and its effect on the theoretical speedup [Amd67].

One of the biggest limitations to parallel programming is the well-known Amdahl's law limiting the theoretical speedup due to the serial parts of a program that cannot be made parallel:

$$S = \frac{1}{(1 - P) + \frac{P}{N}} \tag{2.1}$$

Where S is the theoretical speedup, P is the fraction of the program that can be made parallel and N is the number of threads executing in parallel or speed of the vector unit relative to the scalar unit available on the system. Amdahl's law applies to both MIMD and SIMD parallelism. Combining both MIMD and SIMD parallelism we obtain even more significant speedups. See Figure 2.6 for an example of the speedup for varying fractions of the program that can be parallelized for both a MIMD machine, a SIMD machine and both types combined.

Amdahl's law implies that we should maximize the parallel regions of our programs and if possible parallelize with multiple types of parallelism. This often means focusing on loops as many programs spent a majority of their execution in loops. Another implication of Amdahl's law is that the sequential parts of a parallel program can take up a large part of the execution time and optimizing the sequential is thus very important.

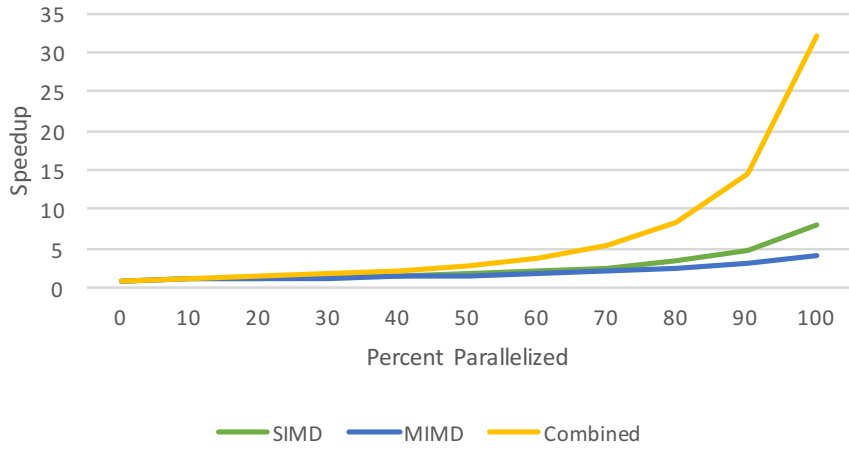


Figure 2.6: Speedup as a function of the fraction of the program parallelized with Amdahl's law for MIMD parallelism, SIMD parallelism and both types of parallelism combined. We assume a vector width of 8 and 4 threads.

There are several ways to obtain a parallel version of a program, each with advantages and disadvantages. For example, to create a SIMD version of a program the following three options are widely used:

- *Automatic vectorization:* where the compiler takes care of the heavy lifting. It analyzes the code and proves that it is legal to vectorize it and automatically perform the required transformations. This is attractive as it puts a very small burden on the programmer.
- *Guided vectorization:* where the programmer specifies on the loop level how a loop should be vectorized and let the compiler do the heavy lifting.
- *Intrinsics:* where the programmer implements the individual vector operations using intrinsic functions. The programmer has to implement all the low level SIMD operations and the compiler then turn the function calls into SIMD.

Each method is attractive for different reasons. Automatic optimization places no burden on the programmer. Guided vectorization can require significant programmer effort for determining the correctness of the described transformations,

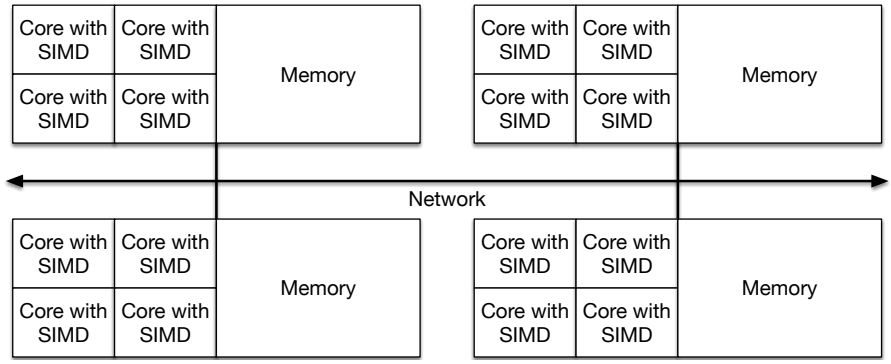


Figure 2.7: Parallel system with several types of parallelism.

but significantly less than using intrinsics where most of the burden is put on the programmer.

When we have obtained a vectorized SIMD version of a program, we can combine that with other types of available parallelism. SIMD can be combined with shared memory programming models and distributed memory programming models to best exploit parallelism at different level. In High Performance Computing (HPC) both SIMD, OpenMP and MPI is often combined to achieve high performance [Kar+13] on systems such as the one seen in Figure 2.7 with multiple nodes. For smaller single node systems using SIMD and OpenMP is attractive to exploit all the available parallelism.

2.3 Program Analysis and Optimization

A compiler transform source code written in one language into another. Typically, it transforms languages such as C and C++ into other, usually lower, languages targeting a specific architecture. A static compiler transforms code prior to execution. Programs can also be interpreted dynamically at run time and compiled at run time. Compilers are traditionally organized into three parts: the front-end, an optimizer and the back-end [Muc97] as seen in figure 2.8.

The front-end parses and check source code for errors. It is also responsible for constructing an intermediate representation well-suited for optimization.

The optimizer performs optimizations on the intermediate representation. They all have the goal of improving the performance of the compiled code.

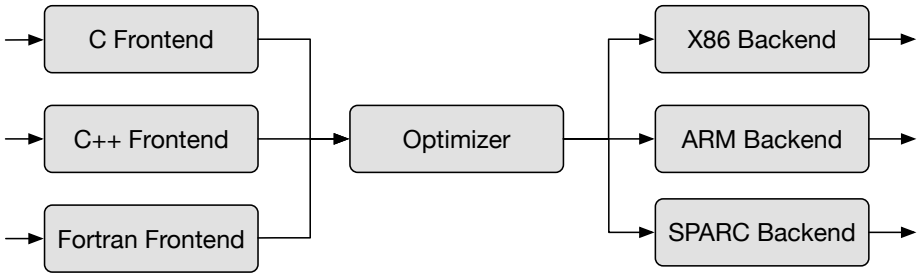


Figure 2.8: A classic retargetable static compiler design.

The backend transforms the intermediate representation to a specific architecture as well as architecture specific optimization.

Static compilers are very often built with this architecture or a closely related organization. Examples includes the GNU Compiler Collection, GCC [Fou]. This organization has the benefit of a high level of reuse. Implementing a new language only requires adding a new frontend. A new optimization pass can benefit more than one programming language and architecture. Last, adding support for a new architecture only requires adding a new backend.

The optimizer has the goal of improving performance using program optimizations. The optimization can usually either lead to the program executing faster or use fewer resources such as power or space. Both the programmer and the compiler can perform program optimization. Programmers have to change the source code to perform optimization, which can affect the quality of the source code through lower readability and maintainability. Automatic optimization done by the compiler has the benefit of being the most transparent way of optimizing programs.

Transformations need to be semantics preserving, valid and safe. Automatic optimization is therefore based on program analysis. Static program analysis is performed without executing the program. It statically predicts safe and computable values and behaviors. To stay computable static analysis can only compute approximate answers [NNH99]. The approximate nature of program analysis is often a limitation in optimizations.

Loops are often the target of optimizations, as they due to their iterative nature usually take of a large part of the program execution time. Many loops perform the same operations across array elements. Optimization targeting these types of loops are therefore very important. Loop vectorization transforms these operations into using instructions that applies the operations to multiple data

elements simultaneously.

2.3.1 Intermediate Representations

Program code has several representations inside a compiler each engineered with specific goals enabling optimization or ease of performing specific transformations. These representations are called intermediate representations (IR).

First, each language frontend is represented as an abstract syntax tree (AST). Every tree node is a construct in the program being compiled.

The frontend then translates the AST to a language independent representation. This representation usually consists of instructions grouped into a *basic block*. A basic block has a list of instructions to be executed in order. At the end of a basic block we may have branches to other basic blocks used for implementing *control flow* to define the order in which basic blocks are executed. The control flow can be represented using a control flow graph (CFG). Each basic block is represented as node in the CFG and edges represent branches. Loops and other code constructs are represented in this way, for example with a cyclic CFG.

The instructions inside a basic block can be represented in different ways. One popular choice is Static Single Assignment, SSA, which is a specific type of IR. SSA form has the property that variables are only assigned at most once. SSA form was developed in the 1980's at IBM [Cyt+91] and has gained popularity as it makes many common optimizations simpler and more powerful. SSA form has the following properties:

- Each definition of a variable has a distinct name.
- Each use of a variable refers to a single specific definition.

SSA form inserts ϕ-functions into programs with multiple paths where a single variable can be assigned. ϕ-functions merge variables to make the single-assignment property hold. SSA code is usually in three-address code form, where an instruction has the form `i = j op k`, where `j` and `k` are the operands, `op` is the operator and the result is placed in `i`. ϕ-functions are not in three-address form as they can merge arbitrary many variables.

Many optimizations benefit from SSA form and as each use has a single definition because each use of a variable must refer to a specific definition of the variable. Optimizations that benefit from SSA form are constant propagation, sparse

```
1  for(i=0; i < N; i++)
2    A[i] = A[i-1]+1;
```

Figure 2.9: Loop with a true-dependence.

conditional propagation, dead code elimination and global value numbering. Due to these advantages of SSA form, many research and production compilers use it.

Usually compilers convert to SSA form for use in the target independent optimizer and convert out of SSA for the backend and code generation. Some research argue how SSA form is also useful for register allocation [HGG06] and other compiler backend passes.

2.3.2 Dependences

There is a dependence between statements if they share a memory location and one of them writes to it. Data dependences impose a constraining execution ordering on the program execution, as it may not be legal to reorder statements or execute them in parallel.

One classification of data dependences is according to the operation type and their ordering:

True-Dependence: Also known as a read-after-write dependence. This dependence is present when a read depends on a write from a previous operation. With regards to automatic parallelization and automatic vectorization this dependence can be reordered in some special cases.

The loop in Figure 2.9 represents such a true-dependence.

Anti-Dependence: Also known as a write-after-read dependence. This dependence is present when one operation read a memory location that is later written to by another operation. Even though there is no transfer of data between the two operations, they cannot generally be reordered when considering automatic parallelization and automatic vectorization.

The loop in Figure 2.10 contains such an anti-dependence.

Output-Dependence: Also called a write-after-write dependence. This data

```
1  for(i=0; i < N; i++)
2    A[i-i] = A[i]+1;
```

Figure 2.10: Loop with an anti-dependence.

```
1  sum = 0;
2  for(i=0; i < N; i++)
3    sum = sum + A[i];
```

Figure 2.11: Reduction example.

dependence is present when two operations both write to the same memory location. In general, this type of dependence cannot be reordered either and is unsafe for parallelization and vectorization.

All of the above: Some code patterns can contain all of the above dependences, for example the reduction pattern example in Figure 2.11. The compiler recognizes the pattern and uses its built-in support for reductions to handle the dependence.

It has been known for a long time that precisely determining the dependences of memory operations is essential for many loop optimizations [Bra88]. Transformations that rely on dependences are automatic parallelization, automatic vectorization, renaming, expansion, dead code elimination, etc. Loop memory operations are coarsely classified as either loop independent or as loop-carried dependences. For automatic vectorization, a statement in a loop can directly be vectorized if it does not depend on itself. Dependence analysis is therefore an important part of automatic vectorization.

To exemplify the difficulty in determining dependences consider the loop in Figure 2.12a. The statement in the body of the loop does not depend on itself, as `A[i]` refers to the old value when reading it, and no other iteration reads it once `A[i]` has been updated. In Figure 2.12b a loop carried dependence exist. Each iteration writes to `A[i+1]` read by the next iteration.

Dependence analysis is usually designed for array loop bounds and data accesses that are affine functions. An affine function is defined as a linear function with a constant. For example, data accesses such as $X[i]$, $X[i+j+10]$, $X[3*i]$ and $X[i*j]$ are affine. Analyzing non affine functions is often considered too hard and expensive to do inside a compiler.

```
1  for(int i=0; i < N; i
       ++)
2    A[i] = A[i] + C;
```

(a) Loop independent iterations.

```
1  for(int i=0; i < N; i
       ++)
2    A[i+1] = A[i] + C;
```

(b) Loop carried dependence.

Figure 2.12: Loop with loop independent memory operations in a and a loop with a loop carried dependence in b.

A notation and terminology similar to that of Banerjee and Skeppstedt is adopted [Ban93]; [Ske12] for the remainder of this section.

Definition 2.1 A perfect loop nest L is a loop nest such that the innermost loop consists of a sequence of assignment statements and the outer loops do not contain assignment statements. The loop bounds and data accesses of L are furthermore affine functions.

We denote m as the number of nested loops. If m has the value 1 only a single loop is present, the value 2 is a double loop and so on. We denote the index vector I of loop nest L as $I = (I_1, I_2, ..., I_m)$. The index values of L are the values of I.

$\mathbb{Z}$ is the set of all integers. The Cartesian product $\mathbb{Z}^m$ denotes the integer vectors of size m. The index space of L is the subset of $\mathbb{Z}^m$ consisting of all the index points.

Definition 2.2 We define a partial order relation $\prec_\ell$ in $\mathbb{Z}^m$ for $1 \leq \ell \leq m$ by $i \prec_\ell j$ if $i_1 = j_1, i_2 = j_2, ..., i_{\ell-1} = j_{\ell_1}$ and $i_\ell < j_\ell$.

An example of the partial order is $(1, 2, 3) \prec_3 (1, 2, 4)$.

Definition 2.3 The lexicographic ordering $\prec$ in $\mathbb{Z}^m$ is the union of all the relations $\prec_\ell$: $i \prec j$ iff $i \prec_\ell j$ for some ℓ in $1 \leq \ell \leq m$.

For two statements S_x and S_y, with loop indices i and j respectively, we denote them as $S_x(i_1, ..., i_m)$ and $S_y(j_1, ..., j_m)$. There is a dependence between the two statements if:

1. Both $S_x(i)$ and $S_y(j)$ access the same memory location $\mathcal{M}$.
2. $S_x(i)$ is executed before $S_y(j)$ in L.

3. When L is executed, the memory location $\mathcal{M}$ is not written to in the period from the end of execution of $S_x(i)$ to the beginning of execution of $S_y(j)$.

The two statements S_xq and S_y does not need to be distinct, but the two instances $S_x(i)$ and $S_y(j)$ need to be distinct according to requirement 2.

Definition 2.4 The data dependence distance is calculated as $\sigma_n = j_n - i_n$. The data dependence distance vector as $\sigma = (\sigma_1, ..., \sigma_m)$.

Definition 2.5 The data dependence direction ρ_n is defined as negative, zero or positive if the distance is negative, zero or positive denoted as $\rho_n = sig(\sigma_n)$. The direction vector is defined as $\rho = (\rho_1, ..., \rho_m)$.

There are several ways to classify loop carried dependences, either according to operation type, chronological order or dependence.

Chronologically ordered loop-carried dependences can be divided into the three different classes:

- Lexical forward dependence if the direction is negative.
- Self-dependence if the direction is zero.
- Lexical backwards dependence if the direction is positive.

The compiler calculates the loop distance and its direction vectors to determine if any dependences exist between the loop iterations. Dependences can often not be determined due to the approximate nature of program analysis necessary for it to stay computable.

2.3.2.1 Dependence Testing

It has been proven that the loop dependence analysis problem is actually to solve a system of equations [All83]. Finding a solution to the arising system of equations has been shown to be NP-complete. Therefore, approaches for conservatively estimating a solution are used in practice. A common solution is the `GCD` test [Coh73], also implemented in GCC as part of its dependence analysis.

```
1 for(int j=0; j < N; j++ {
2   for(int i=0; i < N; i++) {
3     A[2*i] = ...
4     ... = A[2*j+1]
5   }
6 }
```

Figure 2.13: Loop example with no loop carried dependences.

Consider the example of the code fragment in Figure 2.13. We want to prove that the memory accesses to `A[2*i]` does not conflict with the memory accesses to `A[2*j+1]`. For this simple example it is clearly the case that there are no dependences between the two accesses as `A[2*i]` touches even elements in `A` while `A[2*j+1]` touches odd elements in `A`. To prove this, we can set up the following equation:

$$2i = 2j + 1 \tag{2.2}$$

A dependence exists if we there exist integers i and j, such that the equation 2.2 is satisfied. As no such i and j exists that can satisfy equation 2.2 there is no dependence between the read and write access. This equation is called a Diophantine equation.

Definition 2.6 A linear Diophantine equation of the form

$$a_1x_1 + a_2x_2 + ... + a_nx_n = c$$

has a solution for $x_1, x_2, ..., x_n$ if $gcd(a_1, a_2, ..., a_n)$ divides with c.

We rewrite equation 2.2 to $2i - 2i = 1$. Given how $gcd(2, -2) = 2$ and the constant 1 does not divide 2 there is no solution. If a solution exists we there may or may not exist a dependence. In these cases, we can apply stronger dependence testing methods such as Fourier-Motzkin eliminations [Ban93].

2.3.3 Alias Analysis

A big obstacle for dependence analysis is the use of pointers. Some programming languages allow the use of symbols to refer to memory locations as pointers.

When multiple pointers can point to the same memory location the pointers are said to be aliased.

To be effective dependence analysis need to take aliasing into account through alias analysis or points to analysis as aliasing greatly affects dependence analysis.

Alias analysis is particular difficult for C programs as C allows arbitrary operations to be performed on pointers. Taking an integer as input from the user and assigning it to a pointer is valid C code, potentially aliasing all pointers in the program.

Alias analysis is in general undecidable [Ram94]. Alias analysis is often designed to give a safe conservative approximation of pairs of aliasing into the categories: *must alias, may alias and no alias*. Many algorithms exist, some intra-procedural and some that are inter-procedural. Even if a compiler implements an inter-procedural alias analysis, it is often only limited to the current compilation scope (e.g. a file being compiled). This greatly limits the precision of alias analysis on function calls.

C99 introduced the `restrict` keyword [JTC99]. The `restrict` keyword states how only the pointer itself, or a value derived directly from the it, is used for all accesses to that object. This makes the programmers intent clearer to the compiler, mitigating many issues preventing optimizations.

2.3.4 Induction Variable Analysis

Induction variable analysis is important for many loop optimizations fundamental to automatic parallelization and vectorization. Array indexes of the accessed memory locations in a loop that are affine expressions of loop indexes enables us to reason about data dependences across iterations.

A variable is an *induction variable* in a region if within the region it is only incremented or decremented with a region constant. A region is a strongly connected component in the control flow graph(CGF). A region constant is a variable whose definition dominates every edge in the CFG for the region or a compile time constant [Ske12]. The most common induction variable is a loop index being incremented by one in every iteration.

2.3.5 Automatic Parallelization

Writing correct and fast parallel code is hard as it adds several new dimensions of complexity. Automatic parallelization is attractive as it puts the burden of finding and exploiting parallelism on the compiler or runtime system. Without automatic parallelization the programmer has to manually parallelize the applications requiring significant effort.

Automatic parallelization converts sequential code into parallel code. Automatic parallelization usually focuses on loops as large speedups can be gained for them. Automatic parallelization can refer to both thread level parallelism and SIMD parallelism. Normally automatic parallelization refers to thread level parallelism and automatic vectorization refers to SIMD parallelism.

Loop iterations with loop independent operations can be executed in parallel. Loops with loop carried dependences may be transformed to remove loop carried dependences. In contrast to scalar optimizations just one data dependence that cannot be proven independent can render an entire important loop unparallelizable. In scalar optimizations inaccuracies in the analysis does not have the same impact on optimizations, missing one optimization opportunity usually has little impact.

To Automatically parallelize program code involves performing alias analysis, dependence analysis and induction variable analysis.

Many compilers exploit existing OpenMP facilities already built into them for fork-join style parallelism. Once the compiler has proven correctness of parallelizing, it applies a cost function to estimate if a speedup will be obtained and finally emits the OpenMP code. It distributes the loop iterations to a number of threads using the existing OpenMP primitives.

2.3.6 Automatic Vectorization

Automatic vectorization is the process of turning sequential code into parallel code automatically utilizing the SIMD capabilities in processors. It is thus a form of parallelization exploiting data-parallelism by executing iterations concurrently.

We can model the performance of a vectorized loop by:

$$t = t_o + nt_e \tag{2.3}$$

Where t is the time it takes to execute a vectorized loop, t_0 is the startup time, n is the numbers of iterations in the vectorized loop and t_e is the time it takes to execute a particular vectorized iteration of the loop [LMM85]; [LCD91]. This model allows us to reason about the theoretical speedup while vectorizing. The model takes into account the strip-mine nature of loop vectorization on modern register-to-register SIMD architectures. A loop usually operates on more elements than the SIMD units support. For example, if we operate on 128 elements and the SIMD units can operate on 16 elements the loop processed in eight strips of 16 elements.

Automatic vectorization shares many of the same challenges as automatic parallelization for thread level parallelism. Automatic vectorization relies on alias analysis, dependence analysis and induction variable analysis. Compilers typically implement two types of vectorization — traditional loop vectorization [AK87]; [KL00]; [EWO04]; [Bik+02] and Superword Level Parallelism vectorization [LA00]; [BZS10].

Traditional loop vectorization started out as a way to accelerate scientific workloads on vector machines, for example the Cray machines [Rus78]. Automatic vectorization today exploits the SIMD extensions in modern instruction sets. For example, Intel processors support the MMX/SSE/AVX SIMD extensions [Int15b], with AVX-512 extensions supported on their newest processors. The current trend is to have wider vector execution units, more vector registers and richer instruction extensions as seen Figure 2.3.

Typically, automatic vectorization of inner loops proceeds in the following fashion known as the unroll-and-jam approach:

1. Unrolling the loop by the *vector factor* (`VF`).
2. Scheduling the unrolled instructions so they are adjacent.
3. Replace the adjacent instructions with their corresponding vector variants.

Traditional loop vectorization typically targets inner loops, but for some loop nests outer loop vectorization can yield good performance if the outer loop has more data-level parallelism and locality than the inner loop. Outer loop

vectorization has traditionally been performed by interchanging the two loops. Nuzman et al. describe a more direct approach strip-mining or blocking the outer loop and collapsing the strip-mined iterations with the inner loop [NRZ06]. This technique is especially suited for short SIMD machines and non-perfect loop nests.

SIMD have a high theoretical computational throughput but using the vector units often incurs several additional overheads. SIMD operations can only access data that is packed correctly in vector registers. To load and store vector registers, conventional SIMD platforms incorporate mechanisms for both strided and contiguous memory access. Furthermore, SIMD extensions include shuffle operations for data in vector registers. These shuffle operations can be expensive if required for each iteration in a loop.

An alternative method for automatic vectorization is the *Superword Level Parallelism* (SLP) [LA00] focusing on vectorization of one or more basic blocks. They detect statements inside a basic block performing the same operations in a compatible ordering. These statements are then combined to form SIMD operations. SLP have been extended to uncover more parallelism using dynamic programming achieving a performance improvement of 13.78% relative to SLP [BZS10]. Last, SLP techniques have been extended to handle control-flow using the predicated execution supported on PowerPC Altivec [SHC05]. Predicated execution is the conditional execution of instructions. These approaches for SLP have the limitation that they mainly focus on straight line code. If loops are considered the traditional approaches are more mature and can therefore handle more special cases.

2.3.7 If-Conversion

Control-flow is a significant hurdle to vectorization of loops. Given how multiple iterations of a loop is executing concurrently, it poses a challenge when control flow is diverging between the iterations. If-statements can be replaced with a sequence of predicated instructions in a method called if-conversion. An example of if-conversion is shown in Figure 2.14. We see how the control flow is replaced with predicated instructions. In this way all branches of the control flow are executed. This transformation results in loops that can be vectorized using the predicated store and loads available in many SIMD extensions.

There are many obstacles to if-conversion. It can introduce faulting traps changing the program behavior and have to maintain the *precise exception* semantics. Precise exceptions assert that the state of the processor is consistent before and after a faulting instructions. Traps can be caused by writes in a read only

```
1 for(int i=0; i<N; i
    ++)
2   if(cond)
3     x = a;
4   else
5     x = b;
6   A[i] = x;
```

(a) Original code.

```
1 for(int i=0; i<N; i
    ++)
2   x_0 = a;
3   x_1 = b;
4   x_2 = cond ? x_0 :
      x_1;
5   A[i] = x_2 + C;
```

(b) If-converted.

Figure 2.14: If-conversion demonstrated by converting the original code in a to the if-converted code in b.

memory, accessing out-of-range memory, invalid pointers and division by zero. To prove correctness of if-conversion the compiler relies on alias analysis and dependence analysis as previously described.

Given how all code paths are executed after if-conversion, it adds a significant overhead. This overhead needs to be smaller than the benefit of the optimization it enables. In the example in Figure 2.14 one extra instruction is executed in each iteration, but with SIMD we achieve a speedup far out weighting this cost. Depending on the loops being vectorized, this might not be beneficial.

Chapter 3

State-of-the-Art — Limitations of Modern Compilers

As described in the previous chapter compiler optimizations analyze and transform codes into versions that achieves better performance. Optimization are important as they are the most transparent way of improving application performance. The analyses and transformations required to perform optimizations are complex. Given the complexity, optimizations often fail for a multitude of reasons. We argue that both regular application programmer and compiler engineers should pay attention to when compiler optimizations fail. If we can understand the reason behind missed optimization opportunities, we can either address them in the compiler or the application programmer can slightly modify the code as a cost effective way to increase application performance. In this chapter, we will motivate work on various optimizations aspects by studying how one major compiler, GCC, performs on a wide set of benchmarks.

In general, we find three main reasons for the limitations in today's production quality compilers:

- Static analysis precision

- Lacking support for required transformation
- Profitability analysis

Static analysis is approximate by nature and therefore the precision is often inadequate to perform optimizations. Second, compilers often assume certain common code patterns to reduce the complexity of the compiler implementation while still achieving high performance. Therefore, a specific code pattern is often missing to perform an important optimization. Last, the profitability analysis is based on heuristics and therefore often fails to produce the correct answer either leading to a slowdown where an optimization should not have been applied or a missed optimization that could lead to a speedup.

3.1 Automatic Vectorization

Automatic vectorization is either applied or not applied at all. This is in contrast to scalar optimizations that are less affected by the inaccuracies in the analysis. Missing one scalar optimization opportunity usually has little impact. In automatic vectorization a single limitation in one of the required analyses can render an entire loop unvectorizable by the compiler. Many other loop optimizations relying on the same analyses exhibit this pattern.

Given the complexity of automatic vectorization several studies have looked at how good production compilers are at automatic vectorization.

Callahan et al. studied 100 synthetic loops written in Fortran with the purpose of testing the vectorization effectiveness of 19 compilers [CDL88]. On average, the compilers vectorized or partly vectorized 61% of the loops, the best compiler vectorized 80% of the loops.

A follow-up study by Maleki et al. show how the production quality compilers GNU GCC, IBM XLC [IBM15] and Intel ICC [Int] automatically vectorize a set of synthetic loops from the Test Suite for Vectorizing Compilers, TSVC [Mal+11]. 123 out of 151 of the synthetic loops were found to be vectorizable on Intel platforms with the current hardware mechanisms for gather. GCC could vectorize 47%, IBM XLC 55% and Intel ICC 73% of the 123 vectorizable loops. All loops were amenable to vectorization and have been available to the compiler engineer.

We reproduced the results from Maleki et al. using the newest compilers available for AVX-2 from Intel and GCC. The system and compilers used can be

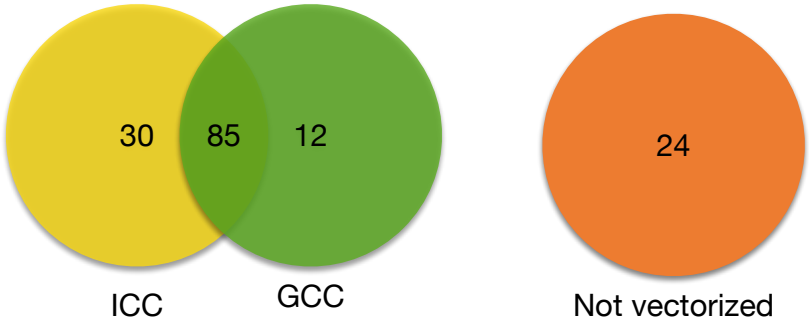


Figure 3.1: Loops vectorized by ICC and GCC in the Test Suite for Vectorizing Compilers.

seen in Table 3.1. The results can be seen in Figure 3.1. In contrast to the results obtained by Maleki et al. recent versions of GCC are vectorizing more of the TSVC loops. Where GCC 4.7.0 could vectorize 47% of the loops, GCC 6.1 can vectorize 64% of the loops. ICC 12 could vectorize 73% of the loops where ICC 16 can vectorize 76% of the TSVC loops. We conclude that GCC has improved its automatic vectorization capabilities significantly when it comes to the features required for the loops in TSVC, even though ICC still has a small advantage.

We also see how even though if a loop is not optimized by one compiler, it will often be optimized by another due to the strength and weaknesses of the individual compiler optimizations. We can use this information to guide the programmer focus on the loops that we know are possible to optimize more aggressively.

Maleki et al. also studied two benchmark suites. GCC could only vectorize 21% of the most computationally intensive loops in the two benchmark suites. Automatic vectorization with GCC achieved a speedup factor of 1.2 in contrast to manual vectorization with a speedup factor of 2.7. Thus, while automatic vectorization is effective there is significant room for improvement compared to manual vectorization.

Similar to the two previous studies Larsen [Lar11] presented how four compilers, ICC from Intel, XLC from IBM, PGCC from Portland Group and SUNCC from Oracle, optimized the loops in the EEMBC benchmark suite. The motivation was to show how synergies between compilers can be used for categorizing missed optimizations as resolvable. Out of the 3490 missed optimizations generated by the four compilers, 43% could be categorized as potentially resolvable or unprofitable.

3.2 Limitations in GCC 6.1

While the related studies are very interesting, they fail to address the general case. The related studies either only study synthetic loops or few benchmarks. In this section we will study how the newest version of GCC does in term of automatic vectorization. We evaluate on a wide set of compute intensive benchmarks, targeting a broad set of application areas attempting to evaluate the general case. We classify hot loops that take up a significant part of the execution time, determine if the hot loops are automatically vectorized and if not, exactly why not.

The included benchmark suites are SPEC CPU2006 [Hen06], SPEC OMP2012 [Mül+12], NAS Parallel Benchmarks 3.3.1 [Bai+91], Rodinia Benchmark Suite 3.1 [Che+09] and SPLASH-2x [Bie11]; [Woo+95]:

SPEC CPU2006 The foremost sequential benchmark suite. The suite contains a mix of integer and floating point benchmark attempting to evaluate the widest possible set of hardware features. Often used to evaluate compiler performance.

SPEC OMP2012 Benchmark suite from the SPEC consortium for performance evaluation using OpenMP 3.1 for shared-memory parallel systems. In this evaluation we only use one thread.

NAS Parallel Benchmarks 3.3.1 Benchmark suite from NASA targeting high performance computing and thus often used for evaluation of supercomputers and HPC systems.

Rodinia Benchmark Suite 3.1 A benchmark suite first released in 2010 targeting heterogeneous systems updated regularly. It contains OpenMP, OpenCL and CUDA implementations. We use the OpenMP benchmarks targeting the host processor in the systems to evaluate CPU performance.

SPLASH-2x Suite of highly parallel benchmarks. In this study we only use the sequential version. SPLASH-2x has been created by the PARSEC benchmarks maintainers, the only difference is the scaled up input sets to make evaluating realistic on modern machines. The suite is popular in architectures studies.

Table 3.1: Experimental Machine Specification

Processor	Intel® Xeon® CPU E3-1276 v3
Frequency	3.6 GHz
Cores	4
Caches	256 KiB L1D, 1 MiB L2 and 8 MiB L3
Memory	16GB DDR
Processor vector capabilities	AVX2 256-bit SIMD
OS	Debian 8.4, Linux kernel 3.16.7
Compilers	GCC 6.1 and Intel Composer XE 16.0.3
Compiler Options	-Ofast -mavx2

From these 5 benchmark suites we obtain 87 individual benchmarks. Unfortunately, 5 benchmarks had compilation or run-time errors and were not included in the evaluation leaving a total of 82 benchmarks in the study.

The targeted machine is an Intel Haswell as seen in Table 3.1. The GCC 6.1 compiler is used as it is the most recent release at the time of writing. We use the `-Ofast` compilation option, giving the most aggressive optimizations and allowing floating point vectorization that can lead to some numerical inaccuracy. This did not introduce numerical instability and correctness issues for the studied benchmarks.

As we aim to evaluate performance improvements, we only study loops that account for significant execution time. We used a loop profiler and determined that 5% of execution time includes many of the important loops and excludes many loops that are not important. Even though 5% of the execution times is low, many of the benchmarks lack hot loops. The maximum number of hot loops is seven with an average of around two hot loops per benchmark.

After classifying the hot loops, the compiler output has manually been analyzed either studying the assembly code, compiler reports or the compiler internals often modifying the compiler to emit more diagnostics. We report the first issue encountered during automatic vectorization. Thus, if one limitation is mitigated in the compiler, the loop may still not be automatic vectorizable. The results of this classification can be seen in Figure 3.2.

An impressive 40% percent of the hot loops are automatically vectorized and doing so has been assessed to be profitable. In the following sections we will go the biggest limitations and explain their nature using examples.

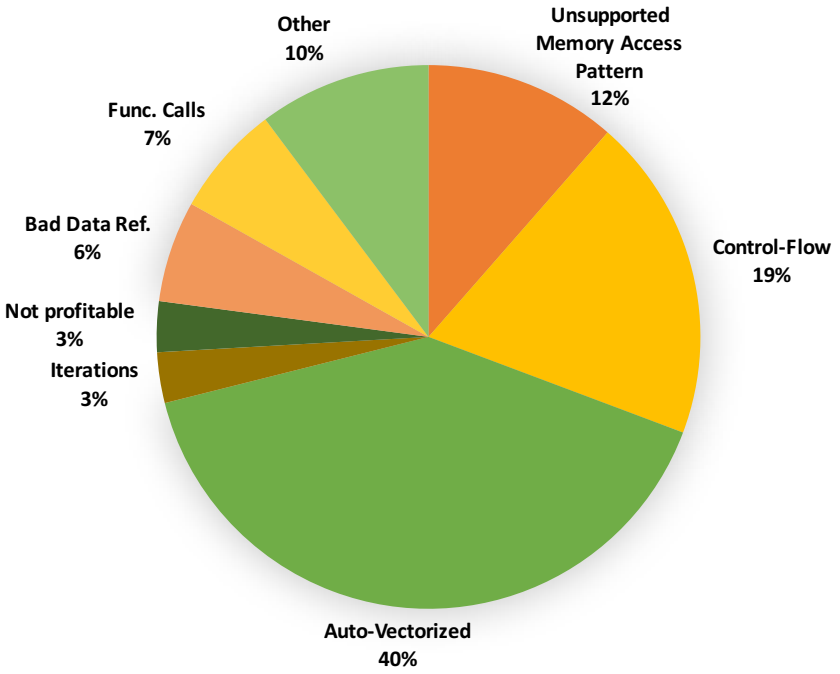


Figure 3.2: Primary issues encountered during automatic vectorization in GCC 6.1 on Spec CPU2006, Spec OMP2012, NAS Parallel Benchmarks 3.3.1, Rodinia Benchmark Suite 3.1 and SPLASH-2x. Numbers are for inner loops representing more than 5% of the benchmark execution time.

```
1 for(i = 0; i < N; i++) {
2     if(a[i] >= b[i])
3         c[i] = a[i];
4     else
5         c[i] = b[i];
6 }
```

(a) Example program.

a	b	mask
3	7	0
6	1	1
2	3	0
9	8	1

(b) Data and mask.

Figure 3.3: Vectorizable program with control flow and example data with accompanying mask.

3.2.1 Limitations due to Unsupported Control Flow

The biggest limitation we see in practice for GCC 6.1 is due to control flow. 19% of the loops could not be vectorized to control-flow in the loops. The main way of dealing with control-flow is if-conversion and use of the masking instruction in AVX or similar SIMD extensions.

As an example see Figure 3.3. The loop iterates each element in `a` and `b` and saves the largest value in `c`. This loop can be vectorized using predication, all paths of the loop is executed and then when saving the values, we use predication, also called masked stores. The mask is generated dynamically at run time and used when saving the results back to `c`. The mask is complemented and used for saving the else code path. In fact, the loop shown in Figure 3.3a cannot be handled by the if-conversion optimization in GCC 6.1. `c[i]` is misclassified as potentially trapping by GCC 6.1 as if-conversion optimization has the limitation that memory accesses present in both branches can be misclassified.

For the 19% of the loops of the loops that could not be vectorized, as seen in Figure 3.2, we categorize these limitations in more detail as seen in Figure 3.4. In the following sections we will go over each of the dominant categories using examples from the benchmarks.

3.2.1.1 Extra Edges in the Control-Flow Graph

Limitations related to extra edges in the CFG can be due to loops that have multiple entries at the top and a single exit at the bottom of the loop. These extra edges, or exits, in the loop can for C be due to `continue`, `break`, `exit`, `return` and `goto`. For C++ `throw` might add extra edges in the CFG. For

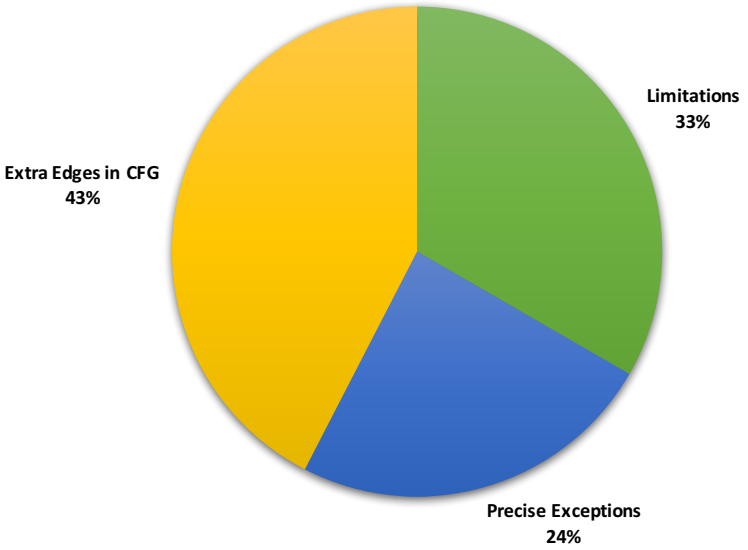


Figure 3.4: Primary issues encountered during if-conversion in GCC 6.1 on Spec CPU2006, Spec OMP2012, NAS Parallel Benchmarks 3.3.1, Rodinia Benchmark Suite 3.1 and SPLASH-2x. Numbers are for inner loops representing more than 5% of the benchmark execution time.

```
1 while( struct ) {
2     if(struct->orientation) {
3       struct->a = struct->b + struct->c;
4     } else {
5       struct->a = struct->c - struct->b;
6       check++;
7     }
8     struct = struct->child;
9 }
```

Figure 3.5: Hot inner loop code pattern similar to 429.mcf from the SPEC CPU2006 benchmark suite [Hen06] representing 29% of the execution time. Cannot be if-converted due to precise semantics.

Fortran `STOP` will add an edge. Even though these loops could technically be vectorizable, doing so will introduce extra operations for dealing with the extra edges and will in many cases not be profitable. 43% of the limitations due to control flow were due to such extra edges.

3.2.1.2 Precise Exceptions

For if-conversion we need to preserve the semantics of the program. Precise exceptions state that the processor state should be consistent whenever an exception or interrupt occurs [HP96]. If care is not taken, we can easily trap on memory instructions given that we execute all code paths in an if-converted program. If the compiler cannot prove that if-conversion is valid, it has to be conservative and not optimize. 24% of the limitations due to control-flow are due to precise exceptions.

One example of this is seen in the 429.mcf benchmark from SPEC CPU2006. It has a hot inner loop representing 29% of its execution time. The loop can be seen in Figure 3.5. Several accesses to a struct inside the loop are control flow dependent.

We have to assume that reading these might not be valid depending on the condition of the if statement and we cannot execute all code paths for all iterations of the loop. If-conversion is therefore illegal for this loop and we can therefore not automatically vectorize it.

```
1  for(i=0; i<reg->size; i++)
2  {
3    /* Flip the target bit of a basis state if both
        control
4    bits are set */
5    if(reg->node[i].state & ((MAX_UNSIGNED) 1 <<
        control1))
6    {
7      if(reg->node[i].state & ((MAX_UNSIGNED) 1 <<
          control2))
8      {
9        reg->node[i].state ^= ((MAX_UNSIGNED) 1 <<
            target);
10     }
11   }
12 }
```

Figure 3.6: Hot inner loop from 462.libquantum from the SPEC CPU2006 benchmark suite [Hen06] representing 62% of the execution time. Cannot be if-converted due to limitations in the compiler analysis. GNU General Public License 2.

3.2.1.3 Limitations in the Compiler Analysis

Unfortunately, there are also loops that could be vectorized, but the compiler is unable to handle them because of assumptions in the implementation. There are many good reasons for these limitations as it is unrealistic to cover all cases and many cases are not worth handling in the compiler.

In GCC 6.1 33% of the limitations due to control-flow are because of limitations in the analysis. As an example see the hot inner loop from 462.libquantum from SPEC CPU2006 in Figure 3.6. Each loop iteration accesses `reg->node[i].state`. Due to limitations in the supported code patterns in the analysis for if-conversion and the interplay between optimization, this loop cannot be optimized. The compiler calculates the base address of the `reg` variable outside the loop using loop invariant code motion, also known as hoisting [Aho+06]. However, the if-conversion pass assumes it is enough to calculate dependences only for variables defined inside the basic block of the loop. Therefore, the optimization pass fails to if-convert this loop.

3.2.2 Limitations due to Unsupported Memory Accesses

The majority of operations can easily be vectorized by substituting them with their SIMD counterparts, e.g. replace a scalar add with a vector add. Some operations are harder to vectorize, for example memory operations loading and saving data. These operates on addresses, but the SIMD extensions cannot operate on arbitrary vectors of addresses. We could duplicate the scalar memory operations and insert the results back into a vector, but doing so is inefficient.

The simplest, and common, case is consecutive address operations on a base address. Thus all accesses are of the form $base + offset$ where $offset = \{i, i + 1, i + 2, ..., i + N\}$. This access pattern can be implemented using the standard SIMD load and store operations common in SIMD extensions. More advanced access patterns, such as strided accesses where the strides can be determined at compile time are often also supported. Last, some extensions support scatter and gather operations, a form of indirect addressing.

From Figure 3.2 we see how 12% of the loops were not vectorized due to issues related to unsupported memory operations. To illustrate the kinds of issues experienced see Figure 3.7. The biggest issue encountered is when the group-size of accesses is not a power of 2 or 3. In the following paragraphs we will go over each of the dominant categories using examples from the benchmarks.

3.2.2.1 Unsupported Scatter Pattern

Some forms of indirection can be supported using gather and scatter memory instructions. This makes it possible to implement operations with indirection. A gather or scatter operation consist of a base address to which an index vector is added with multiple offsets.

One example highlighting this limitation is the Radix benchmark from SPLASH-2x that contains a loop requiring scatter instructions as shown in Figure 3.8. The loop contains several scatter operations, for example when the array `key_to` is updated using `tmp` as an index coming from array `rank_ff_mynum`. This access pattern is simply not supported in AVX2 or previous generations. To efficiently support this category of accesses we need further hardware support. Intel has support for scatter operations in their extensions AVX-512f for the many-core Intel Xeon Phi [Rei12].

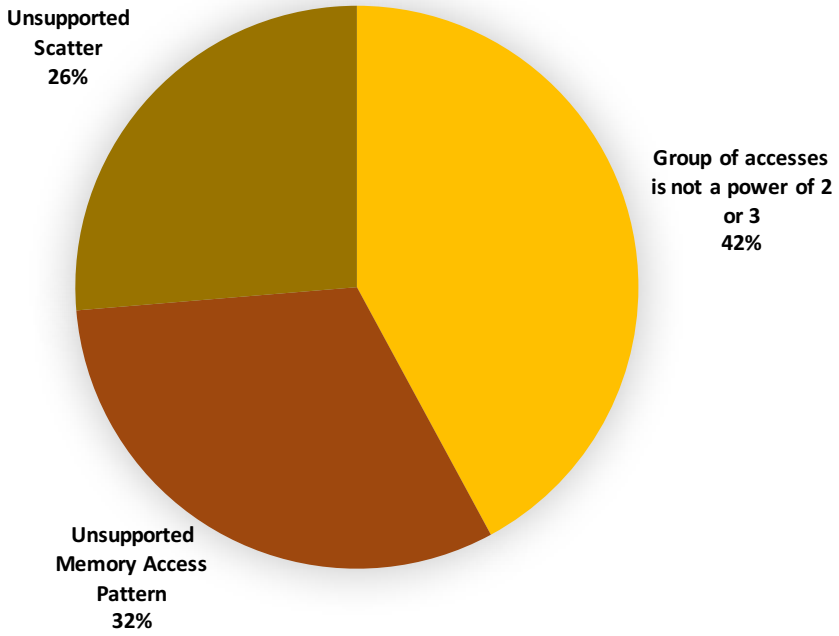


Figure 3.7: Primary issues encountered during automatic vectorization due to memory accesses in GCC 6.1 on Spec CPU2006, Spec OMP2012, NAS Parallel Benchmarks 3.3.1, Rodinia Benchmark Suite 3.1 and SPLASH-2x. Numbers are for inner loops representing more than 5% of the benchmark execution time.

```
1 for (i = key_start; i < key_stop; i++) {
2     this_key = key_from[i] & bb;
3     this_key = this_key >> shiftnum;
4     tmp = rank_ff_mynum[this_key];
5     key_to[tmp] = key_from[i];
6     rank_ff_mynum[this_key]++;
7 }   /*  i */
```

Figure 3.8: Loop nest from Radix from SPLASH-2x [Bie11]; [Woo+95] with this inner loop taking up 26% of the execution time. Cannot be vectorized due to unsupported scatter pattern.

```
1  for (iindex=firstcol;iindex<=lastcol;iindex++) {
2     indexp1 = iindex+1;
3     indexm1 = iindex-1;
4     for (i=firstrow;i<=lastrow;i++) {
5        ip1 = i+1;
6        im1 = i-1;
7        z[i][iindex] = factlap*(x[ip1][iindex]+
8                                x[im1][iindex]+
9                                x[i][indexp1]+
10                               x[i][indexm1]-
11                               4.*x[i][iindex]);
12    }
13 }
```

Figure 3.9: Loop nest from Ocean Non-Contiguous Partition from SPLASH-2x [Bie11]; [Woo+95] with this inner loop taking up 25% of the execution time. Cannot be vectorized due to group of accesses not being a power of 2 or 3.

3.2.2.2 Group of Accesses Not a Power of 2 or 3

Another class of limitations due to memory accesses is limited not by indirection, but by a statically known access pattern. These accesses are not sequential, but strided. Unfortunately, the stride is unsupported as we can only efficiently implement store and loads in with current SIMD extensions if the group of accesses has a power of 2 or 3. This class of limitations accounts for 42% of the limitations due to unsupported memory access patterns.

An example of an unsupported loop is the hottest loop in the Ocean from SPLASH-2x shown in Figure 3.9. The store to `z` has an outer indexing variable calculated in the outer loop leading to this common group of accesses that are unsupported by the AVX-2 SIMD extensions. Similarly, all the loads from `x` have an unsupported stride pattern.

3.2.2.3 Other Unsupported Memory Access Pattern

32% of the limitations are due to other unsupported memory access patterns. These limitations take different forms, but often consist of complicated loops indexes.

```
1 int i;
2 for( i=0; i<NUM_KEYS; i++ )
3 {
4    k = key_array[i];
5    key_buff2[bucket_ptrs[k >> shift]++] = k;
6 }
```

Figure 3.10: Loop nest from IS from NAS Parallel Benchmark Suite [Bai+91] with this inner loop taking up 36% of the execution time. Cannot be vectorized due to unsupported scatter pattern.

As an example of an unsupported memory access pattern see the example inner loop from the IS benchmark from NAS Parallel Benchmark Suite in Figure 3.10. Compiling with GCC 6.1 targeting AVX2 this loop is not automatically vectorizable. The store to `key_buff2` goes through several layers of indirection and address calculations. The analysis in GCC is not refined enough to handle such cases. Such indirection could potentially be supported using scatter operations.

Scatter operations are supported in the newest Intel extensions AVX-512f for Intel's many-core Xeon Phi. If targeting AVX-512f, when compiling the loop from the IS benchmark, GCC 6.1 does not support the loop as it does not support stores with a zero step in inner loop vectorization when performing data dependence analysis.

3.2.3 Limitations in Dependence Analysis

Each dependence test has its strengths as discussed in Chapter 2. Compilers apply multiple dependence tests to achieve an accurate and precise results within reasonable execution time. Dependence analysis therefore have limitations that show up in loops that cannot be optimized.

An example highlighting this limitation is the loop nest from the Needleman-Wunsch benchmark from the Rodinia Benchmark Suite in Figure 3.11. In this loop nest the dependences between read and write to `input_itemsets_l` cannot be determined by the dependence tests implemented in GCC 6.1.

Another example exposing the limitations in GCC is the loop nest from 359.botsalign from SPEC OMP2012 as seen in Figure 3.12. GCC 6.1 cannot automatic vectorize this loop as it both reads and stores to `inner[i*bots_arg_size_1+j]` in every iteration. But the indexing into `inner` is not based on the innermost

```
1 for ( int j = 1; j < BLOCK_SIZE + 1; ++j) {
2  input_itemsets_l[i*(BLOCK_SIZE + 1) + j] =
3  maximum(input_itemsets_l[(i - 1)*(BLOCK_SIZE + 1) +
       j - 1] + reference_l[(i - 1)*BLOCK_SIZE + j -
       1], input_itemsets_l[i*(BLOCK_SIZE + 1) + j - 1]
        - penalty, input_itemsets_l[(i - 1)*(BLOCK_SIZE
        + 1) + j] - penalty);
4 }
```

Figure 3.11: Loop nest from the benchmark Needleman-Wunsch from the Rodinia Benchmark Suite 3.1 [Che+09] with this inner loop taking up 7% of the execution time. Cannot be vectorized due to unknown dependences issues. Copyright (c) 2008-2014 University of Virginia. All rights reserved.

loop creating a dependence between iterations. Interchanging the two innermost loops could fix this issues.

3.2.4 Limitations per Benchmark Suite

Studying why GCC 6.1 could not automatically vectorize per benchmark shows how differently the benchmark suites are. Even though the benchmarks are collected from multiple sources, the benchmarks in a suite with a specific goal in mind can exhibit similar coding patterns. Each benchmark suite is different and has a different evaluation goal. The challenges are therefore going to be different between benchmark suites.

The limitations for each suite can be seen in Figure 3.13. The benchmarks are optimized very differently and the amount of automatically vectorized loops ranges from 20% to 63%. The Rodinia Benchmark suite has the lowest percentage of automatically vectorized loops and SPEC OMP2012 has the highest percentage of automatically vectorized loops.

The biggest difference between the two benchmark suites are the amount of issues arising due to control-flow. For the Rodinia benchmarks, 34% of the loops cannot be handled due to control-flow that the if-conversion optimization in GCC cannot transform into predicated execution. For SPEC OMP2012 only 6% of the limitations arise due to control-flow.

After Rodinia, in SPLASH-2X 40% of the loops can be automatically vectorized.

```
1 int i, j, k;
2 for (i=0; i<bots_arg_size_1; i++)
3    for (j=0; j<bots_arg_size_1; j++)
4       for (k=0; k<bots_arg_size_1; k++)
5          inner[i*bots_arg_size_1+j] =
6                          inner[i*bots_arg_size_1+j] -
7                          row[i*bots_arg_size_1+k] *
8                          col[k*bots_arg_size_1+j];
```

Figure 3.12: Loop nest from 359.botsalign from SPEC OMP2012 [Mül+12] and the Barcelona OpenMP Tasks Suite [Dur+09] with this inner loop taking up 89% of the execution time. Cannot be vectorized due to an unsupported access pattern. Copyright (C) 2009 Barcelona Supercomputing Center - Centro Nacional de Supercomputacion, Copyright (C) 2009 Universitat Politecnica de Catalunya. GNU General Public License 2.

Limitations are not in particular due to a specific issue, but due to issues with control-flow, unsupported memory patterns, etc.

For the SPEC CPU2006 benchmark suite 42% of the loops are automatically vectorized by GCC 6.1. The main obstacle for this benchmark suite is control-flow with 27% of the issues arising due to it.

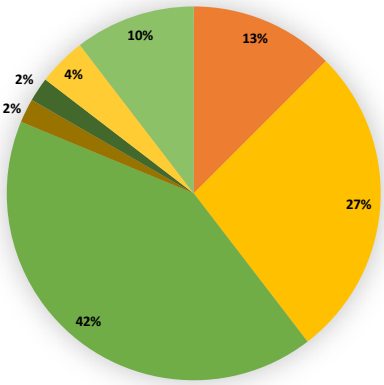
For the NAS Parallel Benchmarks, NPB, 47% of the inner loops can be automatically vectorized by GCC 6.1. The benchmarks in NPB have no issues with control-flow that GCC cannot handle in any of the benchmarks. Instead the benchmarks cannot be vectorized due to issues with unsupported memory access patterns. In particular, with groups of accesses that are not a power of 2 or 3 as required by GCC's analysis.

To see the limitations in greater detail per benchmark in each suite, please refer to Appendix 11.4. Here each individual limitation per loop is represented.

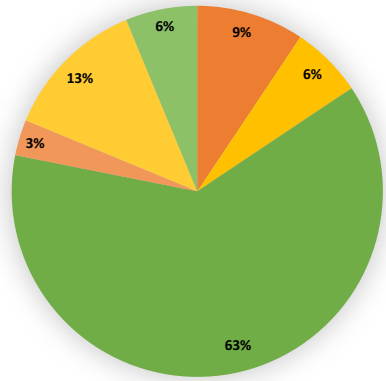
3.3 Discussions

This chapter shows that the majority of hot loops in several benchmarks suites are not automatically vectorized due to many different reasons.

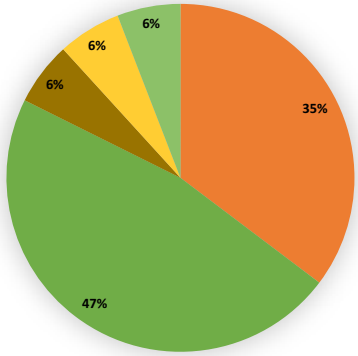
In comparison with the previously described study from Maleki et al. [Mal+11],



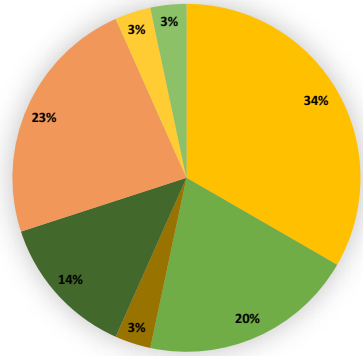
(a) SPEC CPU2006



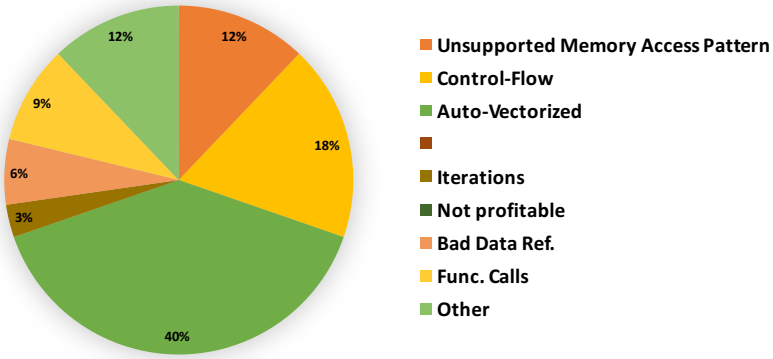
(b) SPEC OMP2012



(c) NAS Parallel Benchmarks



(d) Rodinia Benchmark Suite



(e) SPLASH-2x

Figure 3.13: Primary issues encountered during automatic vectorization in GCC 6.1 on Spec CPU2006 (a), Spec OMP2012 (b), NAS Parallel Benchmarks 3.3.1 (c), Rodinia Benchmark Suite 3.1 (d) and SPLASH-2x (e). Numbers are for inner loops representing more than 5% of the benchmark execution time

where they study automatic vectorization of loops from the Test Suite for Vectorizing Compilers, we identify some of the same limitations. The significance of the limitations is very different. Maleki et al. find that the biggest limitation is due to unsupported gather and scatter patterns. However, Intel has since their study introduced gather operations into AVX-2 and therefore many of these limitations are now removed. The next biggest limitation according to Maleki et al. is due to control-flow, for GCC 6.1 the biggest amount of limitations arises due to control-flow. Maleki et al. further finds that they can manually handle many of the issues arising due to control-flow.

The chapter does not study whether automatic vectorization was actually profitable, as we assume the compilers profitability analysis is accurate. Obviously, this is not always the case as demonstrated by Porpodas et al. [PJ15], they introduce further throttling into the LLVM SLP vectorization pass. They achieve a decrease in the execution time of 9% on average compared to SLP.

Identifying the current limitations of our tools is a good place to identify research challenges in the next chapter.

Chapter 4

Motivation

The previous Chapter 3 has shown and illustrated the gap between what the hardware supports and what applications require specifically for automatic vectorization. The issues are similar for many other hardware features and optimizations, for example automatic parallelization, loop fusion, loop interchange, loop-invariant code motion. Thus, even if we only highlight issues in automatic vectorization, the trends can be widened and more broadly applied to many types of optimizations.

In this chapter we identify research questions motivated by some of the challenges of parallel programming:

1. How do we decompose a problem into subproblems that can be executed efficiently in parallel?

2. How do we ensure that performance linearly scales with increased amount of parallelism in current and future systems?

3. How do we debug parallel applications? The challenges in debugging parallel programs is very different from sequential ones. The extra challenges include non-deterministic execution, parallel bugs such as deadlocks, and the sheer amount of collected information available at large scale.

These challenges lead to the research questions introduced next.

4.1 Improving Utilization of Existing Optimizations

Compilers can perform many impressive optimizations in a transparent way. As shown in Chapter 3, compilers have many limitations due to theoretical and practical reasons. Compilers therefore refrains from optimizing in many cases.

Chapter 3 also highlighted how compilers optimize the same program differently. If one compiler cannot apply an optimization, another compiler is often able to. In some cases, it is not a limitation in the underlying theory of optimizations, but an issue with assumptions and supported patterns in compilers. We therefore argue that it is possible for programmers to rewrite their programs slightly to enable more aggressive optimizations.

One common case is that program analysis in the compiler cannot determine an important program property needed for optimization. If the programmer were aware that the compiler has issues with proving a specific program property, it is possible for the programmer to refactor the code to make it more amenable to aggressive optimization. It is however unclear if compiler feedback systems are practical and to what extent they can be applied.

This leads to the following research questions:

Research Question 1: Details are lost due to abstractions during compilation where we lower from a program representation close to the source language down to an intermediate representation. When generating feedback for programmers on missed optimizations, there are advantages and disadvantages of generating compiler feedback using different representations of a program. What are these advantages and disadvantages, and under which circumstances are the different representations best suited?

Research Question 2: Recently production compilers have implemented better optimization reports. Obviously, these reports can be useful to produce compiler feedback to the programmer. Can we use the reports from multiple compilers to filter away false positive compiler feedback?

Research Question 3: Can compiler feedback guide programmers to higher memory hierarchy performance by improving the utilization of existing compiler optimizations?

Research Question 4: Can we guide the programmer using compiler feedback to improve the utilization of automatic parallelization and automatic vectorization optimizations in existing compilers?

Maintaining legacy codebases is hard. They are large, they may have been written many years ago and the people who understand them are perhaps no longer available. One challenge is to cost effectively maintain the performance of such codebases on new architectures. It is desirable to maintain these codebases with the smallest effort possible while still achieving high performance on modern hardware. For these legacy codebases it is not evident whether we can use compiler based feedback systems to achieve high performance for legacy codebases. If possible it could prove to be a very time efficient way of maintaining legacy codebases.

This leads to the following research question:

Research Question 5: Can we use compiler feedback to increase performance by taking advantage of modern architectures for legacy codebases?

4.2 Improving Compiler Optimizations

Chapter 3 showed the trend that many loops can already be handled by an existing state-of-the-art compiler. 40% of the important hot loops across several benchmarks were shown to already be automatically vectorized by GCC. On the other hand, this means that 60% of the important hot loops were unable to use SIMD, one of the most important hardware features for fast and power-efficient computation in modern computers.

The most transparent way of improving performance is through improving the compiler. When improving optimizations in the compiler, compiled code will transparently execute faster without any effort from the programmer. Mapping applications to efficiently use modern hardware features is hard and is still an open research challenge. The contributions of this thesis seeks to significantly improve performance by addressing this gap with improved compiler automatic vectorization. It is however unclear how much room for improvement exist. In

the end, for dependence analysis we can turn to symbolic analysis using automated theorem proving [Ske12], but doing so would be too slow to be practical for a compiler.

This leads to the following research questions:

Research Question 6: What method should we use to obtain and compare the precision and accuracy of an implementation of a static dependence analysis?

Research Question 7: To what degree are false loop-carried dependences recorded for loop nests in GCC during automatic vectorization?

Research Question 8: How can worksharing information improve the accuracy of dependence analysis of loop nests with false dependences?

Research Question 9: How can the improved loop dependence analysis enable vectorization of loops, also considering profitability, when false dependences are removed?

4.3 Improving Parallel Debugging

Debugging is a critical step in the development of any program. However, the traditional interactive debugging model, where users manually step through code and inspect their application, does not scale well and quickly overwhelms the human programmer with information. The amount of information can quickly become unsustainable as the system scales. First, the biggest systems today are in the order of millions of cores. Second, the amount of information gathered and information presented to the programmer is massive. While lightweight debugging models, which have been proposed as an alternative, scale well, they can currently only debug a subset of bug classes. We propose a new model, which we call prescriptive debugging. It is however unclear, whether the general-purpose nature of the new debugging model impede its scalability. We seek to demonstrate the scalability of the prescriptive debugging model.

This leads to the following research question:

Research Question 10: How does the prescriptive debugging model scale in terms of system size and information presented to the programmer?

4.4 Research Methodology

The thesis contributions aim to research these questions and achieve high performance for prescriptive optimization and debugging.

4.4.1 Compiler Feedback

Research Question 1 address how to generate compiler feedback and the most suitable representation for doing so in Chapter 6, Chapter 7 and Chapter 8. The research methodology used to study this question is building prototypes utilizing several representation and report on our experiences. We study generating feedback from a lowered representation suitable for optimization in Chapter 6 and a representation closer to the language being compiled in Chapter 7. Last, in Chapter 8 we study using existing compiler reports allowing integration with an existing toolchain without modifications. These studies highlight advantages and disadvantages of each method.

Research Question 2 address how to filter false positive compiler feedback, where the feedback is not effective. We study this question in Chapter 8 and show one solution to the problem. In our evaluation we report on the filters effectiveness and whether it removes important feedback on the Test Suite for Vectorizing Compilers [Mal+11].

Optimizing memory accesses is an essential challenge to achieve high performance. Research Question 3 address this by studying how we can improve the utilization of an existing memory optimization in GCC using compiler feedback. We study this aspect using two benchmarks from the SPEC CPU2000 Benchmark Suite [Hen00]. We apply our prototype on the benchmarks and report on the improvement in execution time.

Research Question 4 addresses improving the usage of automatic parallelization and automatic vectorization using compiler feedback. We study this feedback in Chapter 7 and Chapter 8 on several standard benchmarks. We apply the tool prototypes to enable automatic parallelization or automatic vectorization and report on the improvements in execution time. We study benchmarks from the

UTDSP benchmark suite [Lee], Java Grande C benchmarks [Bul+01], the Test Suite for Vectorizing Compilers [CDL88] and one industrial use case.

Research Question 5 addresses porting legacy code bases to newer architectures to maintain high performance over time. We argue that compiler feedback is a cost effective way to maintain the performance. Taking the place of legacy code, we apply our tool prototypes to optimize several benchmarks and one industrial use case in both Chapter 7 and Chapter 8.

4.4.2 Improved Compiler Optimizations

While compiler feedback can mitigate many limitations in compilers, there is also room for improvement in automatic compiler optimizations. Improving automatic vectorization through an improved dependence analysis is the subject of Chapter 9 studying Research Question 6-9.

Research Question 6 addresses how to evaluate a dependence analysis. It is desirable to compare with the ideal dependences. We propose to compare with the dynamically obtained ideal dependences given the complexity of obtaining the correct dependences. We obtain the dynamic ideal dependences by manually instrumenting the programs and creating a trace of all memory accesses. The trace is then processed offline through a custom tool that classify the loop dependences.

Research Question 7 addresses the precision of the existing dependence analysis and the amount of reported false dependences. We study this question by evaluating the original dependence analysis in GCC and our proposed dependence analysis on a set of benchmarks. For this study we use a set of benchmarks from the Rodinia benchmark suite [Che+09]. We then group the dependences according to their type and whether they are vectorizable.

To study the improvement of the suggested OpenMP aware dependence analysis we study Research Question 8 and report the improvement in vectorizable loop dependences.

After integrating the suggested dependence analysis into an automatic vectorization optimization, we study to which degree improvements in performance is achieved by addressing Research Question 9. We study a set of benchmarks from the Rodinia benchmark suite [Che+09] and find that due to the fragile nature of automatic vectorization we often do not achieve speedups, but we are still able to transform the improvement in dependence analysis into significant speedups.

4.4.3 Parallel Debugging

After studying ways to achieve higher performance, we also study the related challenge of how to debug and obtain correctly executing parallel programs in Chapter 10.

We propose the prescriptive debugging model, achieving scalability while also staying general-purpose.

We address the scalability Research Question 10 by using an analytic performance model to indicate extreme scalability. We have validated the model up to 8,192 cores and use it to predict the scalability at larger scale. The debugging model is user-guided and general-purpose. It is therefore possible for a user to specify debugging sessions that does not scale well. On the other hand, it is also possible to specify extremely scalable debugging sessions. We evaluate the scalability in terms of information filtering on a real world use case. The evaluation shows that the filtering of information results in a very condensed output to the programmer.

Chapter 5

Related Work

In Chapter 2 the background and terminology was defined. In Chapter 4 we saw some of the limitations existing today in the GCC compiler for automatic vectorization. In this chapter we will discuss some of the related work, how it improves state-of-the-art and briefly how it compares to the contributions of this thesis.

5.1 Memory Optimizations

Optimizing accesses to memory is also very important. Memory is organized into a hierarchy where access to the smaller caches is faster than access to the bigger last-level caches and main memory. Furthermore, when a given memory is accessed it is likely that the same memory location is accessed again in the immediate future. This aspect of caching is called temporal locality. We also have spatial locality, where data that is accessed together is placed closed to each other in memory. So when a single value is read from memory, access to the adjacent data in the same cache line will be fast.

Many programs are optimized manually for memory performance. There are many transformations that can be applied to loops to make sure that memory

```
1  a = (int**)malloc (N)
       ;
2  for (i=0; i<N; i++)
3   a[i] = (int *)
         malloc(M);
4  a[i][j]++;
```

(a) Dynamically allocated non flattened two-dimensional C array.

$\longrightarrow$

```
1  a = (int*) malloc(N*M);
2
3
4  a[i*M + j]++;
```

(b) Dynamically allocated flattened C array.

Figure 5.1: Dynamically allocated arrays in C showing both non flattened and flattened versions.

accesses are efficient. Many of these transformations can be applied automatically by the compiler to ease the burden of the programmer.

Compiler memory optimizations includes matrix-reorg [Lad06] where dynamically allocated matrixes can be automatically flattened and transposed. The optimizations can either be based on static predictions or on profiling data.

The framework consists of two optimizations: matrix flattening and matrix transposing. When a matrix is flattened an m-dimensional matrix is replaced with an n-dimensional matrix where $n < m$. This leads to fewer levels of indirection for matrix accesses. Part of such an optimization can be seen in Fig. 5.1. The matrix transposing optimization swaps rows and columns resulting in better cache locality depending on access patterns. Afterwards, the accesses are flattened. Profiling is used by GCC to make decisions on which matrixes to transpose.

There exist several other compiler-based data layout transformations including structure layout optimizations [GZ07]; [KF00]. In general, these techniques optimize structures by decomposing them into separate fields, substructures or reordering the fields in a structure.

The compiler can also improve memory performance through loop interchanging [BE04]. Here the compiler interchanges the order of loops in a loop nest. The legality of doing so is determined through dependence analysis and has to preserve the dependences. Data layout transformations and loop restructuring optimizations are all part of the normal compilation optimization flow.

It is also possible to automatically perform data layout optimizations. For example, Leung and Zahorjan have presented their implementation of array

restructuring at run time [LZ95]. They compare it with common forms of loop restructuring and find that it has comparable performance and in some cases even superior performance.

Beyls et al. present their tool Suggestions for Locality Optimizations, SLO [BD06a]; [BD06b]; [BD09]. SLO directly suggests program refactorings for improving temporal locality. It bases its refactorings on instrumented programs sampling the reuse distance. SLO will highlight the code where the largest reduction in reuse distance is possible and in this way suggest code regions the programmer should focus on.

5.2 Compiler Feedback

During compilation the compiler generates useful information about the program being compiled. This information is used in compiler driven feedback systems to take advantage of the powerful existing analysis in compilers. The information is used in systems for improving interactive feedback in integrated development environments [Ng+12], commentary about applied compiler optimizations describing what the compiler has done [Ora]; [Du+15] and optimization advice [Lar+12]; [Lar+11b]; [Lar+11a]; [Jen+12b]; [Int13]; [Du+15].

5.2.1 IDE Support

IDEs help the programmer with interactive feedback on syntax errors, type errors and code completion. Furthermore, they often provided a rich set of automatic refactorings for otherwise tedious coding tasks. These features require an understanding of the program and many IDEs have implemented technology which is also found in compilers. There is an opportunity for reusing code from compilers in the IDE, providing IDE support. One prominent example of this is the Microsoft Roslyn compiler [Ng+12]. It tackles the issue of serving intelligent integrated development environments with data for auto-completion information, refactorings and jumping around in source code such as finding definitions. This is all information where compilers could open up and through APIs to their internal data structures make these services available in IDEs.

5.2.2 Optimization Commentary

Another important compiler feedback feature is letting the programmer know which and where optimizations have been applied to the program.

The Oracle Solaris Studio [Ora] has a unique feature called Compiler Commentary. It annotates the source code with details on which optimizations were applied. Some comments are easy to understand, such as duplicate expressions or a loop being fissioned. Other comments only make sense to experienced developers or compiler engineers.

The tool cannot give advice on source code changes enabling further optimization. Such tools are introduced next.

5.2.3 Optimization Advisers

Programmers often prevent optimization with their coding choices, but small source code modification which does often not affect software engineering principles can help the compiler optimize further. Based on this realization a series of tools have been implemented, guiding the programmer in how to modify programs allowing the compiler to optimize more aggressively.

The Intel Performance Guide included in the Intel Composer XE Suite [Int13] can guide the programmer in profiling for hotspots, suggest optimization flags and give advice for source code modifications enabling automatic parallelization. It is a very user friendly tool, however it only gives suggestions when it is confident, thus often not showing any advice. Complementary IBM's XL C/C++ and Fortran compilers can generate XML reports describing applied optimization, and in some cases suggest modifications to enable further optimization [Du+15].

Larsen et al. also describes their tool, consisting of a modified version of GCC, outputting information on why a given optimization was not applied and displays this information in an IDE [Lar+12]; [Lar+11b]; [Lar+11a]. In this way their tool reuses existing aggressive compiler optimizations for feedback and help the programmer understand why a given optimization was not applied. In a parallelization study, cases with super-linear speedups of parallel code parts were reported due to positive side-effects of modifications [Lar+12]. The tool has issues with mapping internal GCC data structures to the real source code which is not always possible, and they generate many false positive comments due to GCC, requiring extensive comment filtering which is still an open research

problem [Lar11].

Bluemke et al. introduce their parallelization adviser tool ParaGraph [BF10]. The goal of ParaGraph is to use a combination of manual and parallel parallelization. The tool support parallelization of simpler worksharing loops, but is not focused on parallelization of general more complex loops.

SUIF Explorer [Lia+99] parallelize sequential Fortran codes using their automatic parallelization analysis and user interaction. It uses dynamic execution analysis to determine loops that can benefit from parallelization. SUIF Explorer then suggest annotations to be inserted to eliminate data-dependences and perform parallelization using the annotations approved by the programmer.

The Paralax infrastructure [VRD10] explores programmer knowledge to improve parallelization. The tool use dynamically obtained dependence to propose a set of annotations to remove data-dependences that cannot be handled by static dependence analysis. Their approach is input sensitive entailing that dependences observed with one program input is not guaranteed to be the same with another input. The dependences are therefore verified them at run time. The Paralax parallelizer focuses on parallelizing irregular pointer intensives codes.

Last, Aguston et al. shows their tool, which uses code skeletons for parallelization hints [ABH14]. Using skeletons allows Aguston et al. to make larger code transformation and leave the question of whether doing so is safe to the programmer. Applying their skeletons on a subset of SPEC benchmarks suggest a speedup of 30%, however whether doing so is safe is not clarified.

Most compilers include options to generate optimization reports. Both GCC and XLC includes optimization reports for several optimization passes. These mostly state only what was done, and not what could have been done. GCC reports often refer to intermediary representation and without any explanations.

5.3 Automatic Vectorization

Automatic vectorization is a well-researched area with many contributions [Cab+15]; [NRZ06]; [Nai04]; [Tri+09]; [RZB15]; [KH11]; [EWO04]; [Nai+03]; [NZ08]. Modern SIMD architectures have many limitations, such as predicated execution using masks, data alignment, and non-uniform memory accesses. Some of these limitations have been addressed. For example, Nuzman et al. shows how we can handle interleaved data accesses [NRZ06]. Eichenberger et al. show how to consider alignment when generating SIMD code. Diamos et al. proposes

a stack-based approach for handling arbitrary control flow on SIMD processors. Despite these significant contributions from industry and the research community, compilers will often fail to generate automatically vectorize codes to effectively take advantage of SIMD architectures.

Production quality compilers for C and C++, like the GNU Compiler Collection (GCC) [Fou], Clang [The] and Intel's ICC [Int], all have impressive automatic vectorization capabilities. Unfortunately, automatic vectorization of programs is fragile with many limitations in the analysis and supported transformations. Reasons for this include strided memory accesses, control flow complexity and unknown loop trip counts, making heuristics complicated,

We have studied GCC's automatic vectorization capabilities. GCC contains multiple automatic vectorizers. One focuses on loop vectorization, where the entire loop is vectorized. Another focuses primarily on vectorization of straight-line codes, the Superword Level Parallelism vectorizer as originally proposed by Larsen et al. [LA00].

The loop vectorizer has many limitations. For example, it can only vectorize loops with one basic block, complex loops are not supported and only simple inner loops can be vectorized.

The basic block vectorizer can be applied to sub-parts of loops and is therefore often more successful. Sadly, the performance gains are limited for complex codes and only a small fraction of the code base can be vectorized.

Intel ICC is by many regarded as the best automatic vectorizer for Intel processors. It contains a very aggressive vectorizer. Maleki et al. compared ICC, GCC and XLC [Mal+11]. They found that ICC will vectorize significantly more loops compared to GCC when studying synthetic benchmarks. For real applications, only few loops are vectorized by the compilers.

Another approach is the polyhedral model where the program is translated into a linear-algebraic representation. Afterwards, new execution orderings are found by applying a reordering function. The polyhedral model has been applied successfully for a wide range of optimization such as automatic parallelization [Bon+08] and automatic vectorization [Tri+09].

Chatarasa et al. propose an alternative approach for improving the polyhedral framework [CSS15]. They incorporate executions orderings from explicitly parallel programs and broaden the range of legal transformations for explicitly parallel programs.

5.4 Dependences

Improving the accuracy and speed of dependence analysis is a well-researched area with many contributions.

Goff et al. proposes a dependence testing scheme for fast yet exact tests [GKT91]. They implement several dependence tests in combination. They start out with simpler tests, and in the cases where the simple tests are not applicable, apply more expensive tests. This is the same strategy as implemented in GCC. GCC implements `ZIV`, `SIV`, `MIV`, `GCD` and the two advanced frameworks Lambda [LP94] and GRAPHITE [Tri+10]. Lambda is based on integer non-singular matrices. GRAPHITE is a polyhedral framework that is based on a linear-algebraic representation. GCC 6.1 uses the Lambda framework by default.

There are several ways to improve the precision of dependence analysis. Pugh suggests the omega test [Pug91], applying integer linear programming to solve the dependence analysis. Omega tests have previously been implemented in GCC.

Viitanen et al. compares six different data dependence analysis and their performance in terms of execution time and accuracy. The tested algorithms are the GCD test, extreme value test, Fourier-Motzkin elimination, the Omega test and the data dependence analysis implemented in the SUIF compiler [Wil+94]. Each analysis can determine a different number of independencies with varying execution time. The most accurate of the evaluated analyses is the Omega test and the least accurate is the GCD test. The GCD test has an accuracy of 58.4% on the selected benchmarks compared with the Omega test accuracy of 61.3%. The omega test takes approximately 1.3 longer to execute than the GCD test.

5.5 Models in Parallel Debugging

Debugging is important for a multitude of reasons, both diagnosing incorrect behavior and figuring out performance issues.

Many tools and techniques exist for finding and correcting errors in parallel programs. The same tools and techniques can be used for performance debugging. These range from user code instrumentation (i.e.,`printf`) to traditional interactive debuggers to proving correctness with formal methods. Each tool has its benefits and no one tool is sufficient for identifying all bugs. These tools can have high overheads, due to the cost of instrumenting or probing the application,

analyzing the debug information and communicating amongst the tool components. Applying these tool has an overhead, in the cost of instrumentation and communication overhead as you scale the system. For example, Valgrind can impose a slowdown of 10x-100x [NS07]. Many bugs only manifest themselves at large scale, and therefore we need tools that works at extreme scale.

The current state of the art in parallel debugging is focused on four main models: traditional; lightweight; semi-automatic; and automatic. In this section, we discuss the pros and cons of these models and the implications of the architectural and application trends in HPC.

5.5.1 Traditional Model

The traditional debugging model seeks to aid programmers in interactively diagnosing an error. It enables them to view the detailed program state and to modify it at any point in execution. The tools that offer this paradigm include GDB [SS14], DDT [All14] and TotalView [Rog14]. They must provide features that support nearly *all* debugging facets. Commonly, these features include various data displays, lock stepping of execution, breakpoints or evaluation-points, and process or thread group control, such as barriers, if the tool is parallel-aware.

This model is highly effective in isolating the root cause of many classes of errors at low to moderate scales. However, in the face of large concurrency and high application complexity, its effectiveness starts to decrease sharply. More importantly, manually viewing and managing detailed state information at various points of execution is becoming increasingly unwieldy at only a few thousands of processes and on complex modern design patterns. The recent efforts of several parallel debugger vendors [All14]; [Rog] have improved software scalability by supporting the model through innovative communication and display mechanisms. But its fundamental scheme of having to enable all idioms for interactive use and the central point of control, i.e., the user, clearly limits its scalability.

At very small scales, applying this serial debugging paradigm to parallel debugging is manageable, but becomes unwieldy at even moderate scales, let alone extreme scales. This pitfall is shared by another common strategy, debugging using `printf`. Even at today's scales gathering debug information from each core is intractable. For example, on the Sequoia machine [Law], with its over 1.6 million cores, gathering or printing just one byte from each core leaves the programmer nearly 1.6 megabytes of data to analyze. This is value is multiplied when analyzing larger data types, such as 8-byte double precision floating point variables, or arrays of variables, and is further multiplied by the 4 threads of execution per core on Sequoia.

5.5.2 Lightweight Model

The lightweight model has recently emerged in specific response to the challenges at scale. This model pursues trade-offs between debugging scalability and capabilities. Thus, the tools of this paradigm, such as the Stack Trace Analysis Tool, STAT [Arn+07]; [Lee+08]; [Ahn+09]; [Lee+07], IBM's Blue Gene Coreprocessor debugger [LK32] and Cray's Abnormal Termination Processing, ATP [Cra13], drastically limit the types and amount of execution state information that are fetched, collected and displayed. They also limit the level of user interaction and provide only coarse mechanisms to select points in execution to analyze. This paradigm has proven to be extremely scalable, for example, STAT has successfully isolated certain classes of errors that only emerge over one million processes. However, due to its coarseness, it can leave programmers with no actionable information on some other classes of errors.

5.5.3 Semi-Automatic Model

The semi-automatic model is based on user-guided automation of finding errors in large data arrays. Relative debugging [Abr+96] allows users to select corresponding data arrays and points in two slightly different versions of the same program, and at runtime automatically differentiates the arrays at those points. The idea is that the region of code where corresponding arrays of different versions diverge is likely to be where the error originated. Because the bulk of analysis is *computed*, this model significantly reduces user interaction and saves the user from having to examine individual data array elements manually. However, its main debugging mode that requires two different simultaneously running jobs can hamper debugging those errors that emerge only at large scales, due to resource requirements.

Further, recent work, such as assertion-based parallel debugging [Din+11], has advanced this model to be used in more diverse scenarios, including comparing processes within a single job, and also to be more scalable. However, it targets data-centric errors, as opposed to being general purpose.

5.5.4 Automatic Model

The automatic model has burgeoned in recent years with a promise to isolate general programming errors with no user intervention. Specific work in this area includes AutomaDeD [Bro+10]; [Lag+11]; [Mit+14], which applies statis-

tical techniques on a semi-markov model representation of MPI processes to automatically detect and localize software bugs. The model also strives to automate code instrumentation and hence avoids opening the execution state and selection mechanisms to the users. While the automatic identification of erroneous tasks and code regions is the ideal approach, and recent work [Lag+11] has shown that this model can scale easily to a very large number of tasks, the level of isolation is only as good as the analysis and instrumentation techniques, which are currently still lacking. Often, a user's debugging intuition can allow a model to overcome the lack of analysis accuracy and precision, but the automatic nature makes it hard to take advantage of this intuition for error isolation. Other than these approaches, more popular forms of automated tools tend to target specific types of errors, such as memory leaks [GS06] or MPI coding errors [Kra+03]; [Lue+03]; [Che+10]; [VS00].

Chapter 6

Compiler Feedback for Higher Memory Performance

In the previous Chapter 4 we saw how the compiler often refrains from optimizing due to the complexity of automatic optimizations. It is often a problem for programmers to understand how source code should be written to enable optimizations. Interactive tools that guide programmers to higher performance are very important. This chapter present one aspect of such a tool we have developed. The tool helps programmers modify their code to allow for aggressive optimization. In this chapter, we describe our extension for high level memory optimizations such as matrix reorganization in the tool. We evaluate the tool using two benchmarks and four different compilers. We show that it can guide the programmer to 22.9% higher performance.

Optimizing compilers are complex and difficult for programmers to understand. Programmers often do not know how to write code that the compilers can optimize well. This work extended a previously developed tool that helps non expert programmers write code that the compiler can better understand [Lar+11a] in an interactive way. It uses feedback generated by the production compiler the GNU Compiler Collection, GCC [Fou]. We have modified GCC to improve the quality and precision of the feedback. Our tool then interprets the feedback

and presents it in a human readable form directly into the Eclipse integrated development environment, the Eclipse IDE. In principle, the techniques could have been applied to any compiler and IDE.

We have extended our tool to support high level memory optimizations such as matrix reorganization. These optimizations change the way matrixes are accessed resulting in improved memory hierarchy performance. We perform a performance evaluation using two benchmarks from SPEC CPU2000 [Hen00] and four different compilers. We show that our enhanced tool can guide the programmer to 22.9% higher performance.

Although one can argue that compilers should be better at optimizing code, programmers required to write performance sensitive code often cannot wait for a more advanced compiler to be released. Our tool helps non expert programmers change their code to fully utilize all the optimizations an existing production compiler can offer. Our work is thus complementary to the work on more precise analysis methods and aggressive compiler optimization passes.

In short, this chapter contributes the following:

- We have extended our tool significantly. It can now handle the matrix reorganization memory optimization.
- We have developed a code refactoring wizard that helps programmers apply changes directly to their code.
- We evaluate the extended tool and show substantial performance improvements on two SPEC CPU2000 benchmarks and with several compilers.

This chapter extends our previous work [Lar+11a] in that we have significantly expanded the tool to provide feedback on high level memory optimizations.

The chapter is laid out as follows. Our tool and how it can improve memory optimizations is introduced in Section 6.1 Experimental results are analyzed in Section 6.2. We discuss the work, thesis research questions and conclude the chapter in Section 6.3.

6.1 Memory Optimization

Compilers have to generate correct code. To this end, compilers often conservatively decide not to apply optimizations in ambiguous cases where the intent

```
1 a = (int**) malloc (N)
2 for (i=0; i<N; i++)
3     a[i] = (int *) malloc
          (M)
4 a[i][j]++
```

(a) Dynamically allocated non flattened two-dimensional C array.

$\longrightarrow$

```
1 a = (int*) malloc (N * M
      )
2
3
4 a[i*M + j]++;
```

(b) Dynamically allocated flattened C array.

Figure 6.1: Dynamically allocated arrays in C showing both non flattened and flattened versions.

of the programmer is not clear. The general idea behind our tool is to help programmers fully utilize modern advanced compiler optimizations. We propose that programmers will work actively with the performance critical sections of their code. Our tool interactively provides the programmers with hints and suggestions for changing the code so that source code ambiguities are removed thereby facilitating the application of additional compiler optimizations. The tool consists of several parts. First, we link GCC [Fou] with a special library that we have developed. The library will augment GCC's diagnostic dump files with information on the optimizations performed but also on optimizations not performed. These dump files are then read by an Eclipse plugin that we have developed as well. The plug-in interprets and displays the information. The plug-in also suggests refactoring changes based on the extracted information [Fou]. Refactoring is a technique for modifying source code in a structured way, without changing the program's external behavior [99].

We have extended our work to support GCC's matrix reorganization [Lad06] framework, which optimize dynamically allocated matrixes. The optimization pass can perform two optimizations: matrix flattening and matrix transposing. When a matrix is flattened an m-dimensional matrix is replaced with an n-dimensional matrix where $n < m$. This leads to fewer levels of indirection for matrix accesses. As an example of the optimization see Figure 6.1. The matrix transposing optimization interchange rows and columns followed by flattening resulting in better cache locality depending on access patterns. Profiling is used by GCC to make decisions on what matrixes to transpose.

Reorganizing matrixes is an intrusive operation: declarations, allocation, matrix access and deallocation sites have to be updated. The GCC compiler will therefore refrain from using the optimization unless it can analyze exactly how the matrix is used. In many cases, the compiler cannot fully analyze how a matrix is accessed. The matrix escapes analysis. This may happen if the matrix

is an argument to a function, even if it would be safe to optimize. Here the analysis is conservative and chooses not to analyze even local functions. Only global dynamically allocated arrays are optimized. Manual vector expressions could lead to errors. Therefore, matrixes are not reorganized when vector or assembler operations exists. Additional restrictions exist. For example, GCC assumes only one matrix allocation and one or more accesses, therefore matrixes with multiple allocation sites will not be optimized. In general, many matrices are not optimized.

Our tool can help the programmer when the compiler refrains from optimizing. It does so by giving reasons why the compiler did not optimize and suggests changes if applicable. There exist many reasons why the optimizations do not apply so our tool prioritizes the information so that the most useful hints are shown first. One example is missing compiler options. If the correct options are not used, it will present a hint in the Eclipse IDE suggesting the programmer to change options.

As mentioned, if the matrix escapes the analysis it will often not be optimized. One common scenario is when a matrix is an argument to an external function or some other function for which the source code cannot be analyzed. Here our tool points out all escaping matrixes to the programmer. It will also describe why each matrix escapes as well as suggestions for how code can be rewritten. This may include using annotations to get the compiler to inline functions. More invasive refactorings can also be attempted. Instead of passing the matrix to a function, it might be possible to pass a temporary variable if individual matrix elements or their addresses are passed. This will often help the compiler analysis understand the source code better. Another solution, not implemented, is to use data copying in places where it is known where a matrix escapes and where it returns. In the escaping region the original version of the matrix can be used. This adds some overhead synchronizing data.

Many programmers have issues using the profiling features of GCC used by the matrix reorganization framework to determine whether matrixes should be transposed and how. It is easy to detect in the compiler whether profile guided optimization is disabled. Our tool shows messages to the programmer explaining the required steps to use profile guided optimization. This is seen in Figure 6.2.

We have developed an automatically refactoring wizard. The wizard helps the programmer apply matrix flattening and transposing directly to the source code. This makes it possible for the programmer to turn off matrix reorganization in the compiler. This is sometimes necessary. For example, we have discovered at least one bug in GCC that forced us to manually rewrite the code. The wizard also allows us to apply code transformations using other compilers. This might be useful as many compilers do not have matrix reorganization optimizations.

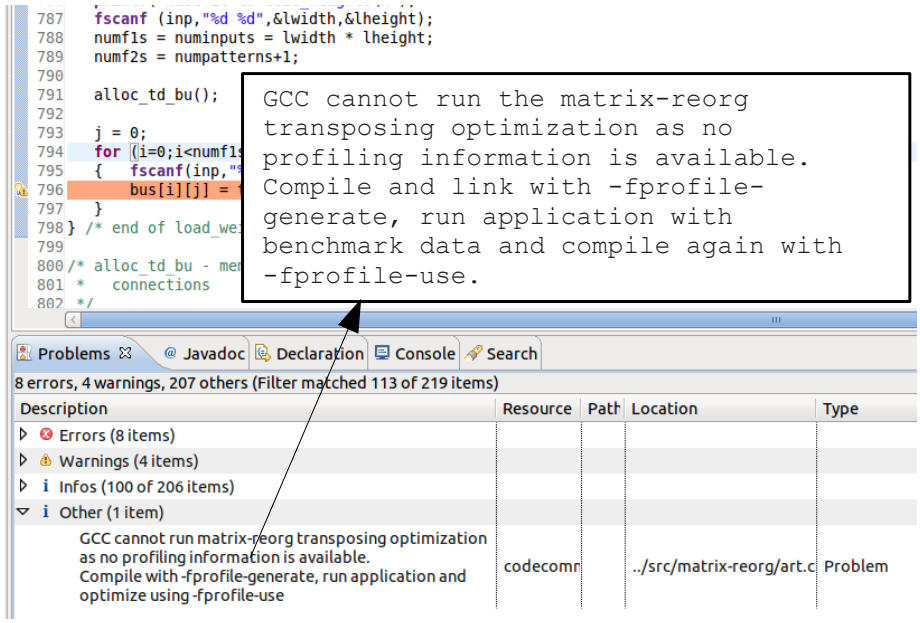


Figure 6.2: Marker helping the programmer use profiling.

The wizard is implemented using Eclipse's refactoring framework and so have a familiar look and feel. The wizard will be offered to the programmer if the compiler found that the matrix could be optimized. Both full and partial flattening and transposing are supported. An example of the wizard can be seen in Figure 6.3.

6.2 Experimental Evaluation

We have applied our tool on two kernel benchmarks from the SPEC CPU2000 benchmark suite: 179.art [RD] and 183.equake [OK]. The benchmarks have previously been used to evaluate the matrix reorganization capabilities in GCC [Lad06]. Good results were shown. A total of 35% improvement on 179.art and 9% on 183.equake. This indicates that the standard optimizations in GCC already are effective.

We will use our tool to further optimize the benchmarks. First by mitigating issues preventing optimizations and then by applying the optimization directly at the source code.

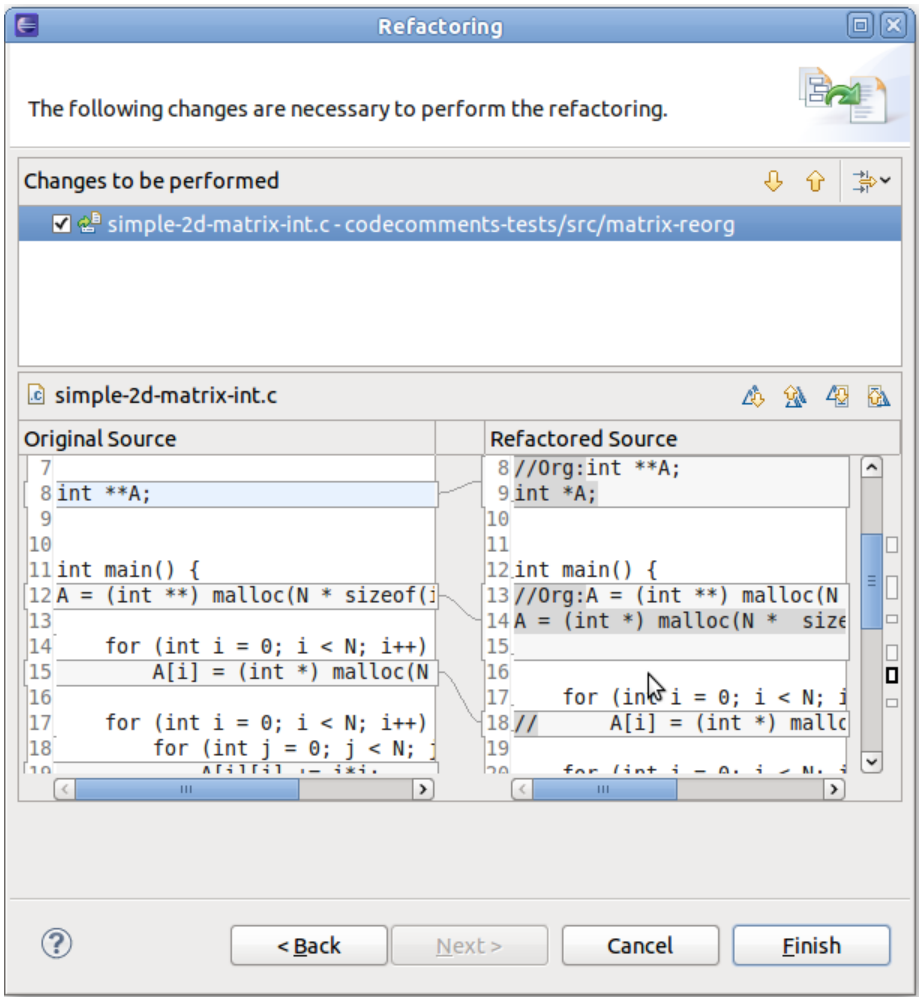


Figure 6.3: The refactoring as it appears in the Eclipse IDE.

Table 6.1: Name, version and host machine of used compilers.

Compiler	Version	Host
GCC [Fou]	The GNU Compiler Collection 4.5.1	xeonserver
ICC [Int]	Intel C++ Composer XE 2011 for Linux 12.0.4	xeonserver
SUNCC [Ora]	Oracle Solaris Studio 12.2 Linux	xeonserver
XLC [IBM15]	IBM XL C/C++ for Linux 11.1	g5server

Table 6.2: Compiler options used.

Compiler	Options
GCC [Fou]	-O3 -fipa-matrix-reorg -fwhole-program -std=c99
ICC [Int]	-fast -ipo
SUNCC [Ora]	-fast -xc99
XLC [IBM15]	-O2 -qhot -qipa=level=2

We have run experiments on two machines. We have used a Dell Poweredge 1900, with one Quad-core Intel Xeon E5320 processor at 1.86GHz and a level two cache of 2x4MB. The machine, called xeonserver, runs Debian 6.0.2.1. The second machine is an iMac G5 with a 1.8GHz PowerPC 970fx processor. It is called g5server and runs Red Hat Enterprise Linux 5.5.

We have used four different compilers. An overview of compilers, versions and host systems are shown in Table 6.1. The compiler flags used are shown in Table 6.2. All experiments have been run 50 times and the average benchmark execution time has been used.

6.2.1 Case study: 179.art

The 179.art kernel benchmark is using neural networks to recognize objects in thermal images. It consists of a training process where the neural network learns from test images and an analysis process where it is matching a thermal image against the training images [RD]. The training process is short. We will only use the execution time for the analysis process when evaluating performance.

The program consists of 1042 lines of C code as measured using SLOCCount [Whe13]. It contains a number of matrixes but only three global multidimensional dynamically allocated matrixes are candidates for optimization.

GCC can optimize two of them, the `tds` and `bus` matrixes, automatically but the `cimage` matrix escapes. The matrix is not an argument to a function and

Table 6.3: Global multidimensional dynamically allocated matrixes in 179.art and whether they are optimized automatically by GCC and XLC.

Declaration	GCC	XLC
`unsigned char **cimage`	Not optimized, matrix escapes	Flattened
`double **tds`	Flattened	Flattened and transposed
`double **bus`	Flattened and transposed	Flattened and transposed

therefore it should have been optimized by GCC. We found that the escape analysis in GCC has problems with char arrays. This issue is due to an internal limitation in the compiler's analysis, which could be overcome by changing the datatype, e.g. to an integer.

When profiling the program, the `bus` matrix was transposed meaning that the dimensions of the matrix were swapped. We also used the IBM XL C/C++ compiler, XLC, to compile the kernel. Here all three matrixes were flattened and two were chosen for transposing. XLC does not need profiling data to determine whether transposing is useful and statically evaluates this.

The optimizations each compiler performed automatically are shown in Table 3. One thing that stands out from this experiment is that XLC has a more precise program analysis. It will probably optimize more aggressively.

Our tool could not guide the programmer to a version where the `cimage` matrix in 179.art is optimized by the GCC compiler. Therefore, results are only presented for the unmodified original program and a version where two of the matrixes are optimized at the source code level obtained using the wizard. The wizard optimizes the matrixes that GCC can automatically optimize as seen in Table 6.3.

This, however, did not give any significant speedup over GCC's own optimization. In Figure 6.4 it can be seen that using GCC a speedup of 1.16 was possible. This is only due to the wizard utilizing the profiling data to transpose a matrix. Using profile guided optimization, the compiler does an equally good job.

We also have made experiments with multiple compilers. The performance results for 179.art on xeonserver are shown in Figure 6.4 and the speedup achieved are shown in Figure 6.5. Using SUNCC a speedup factor of 1.36x was achieved over the original version. ICC produced the fastest program but did not benefit from the optimization with a speedup of one. The last compiler is XLC. It has the matrix reordering optimization. However, it performed worse with the

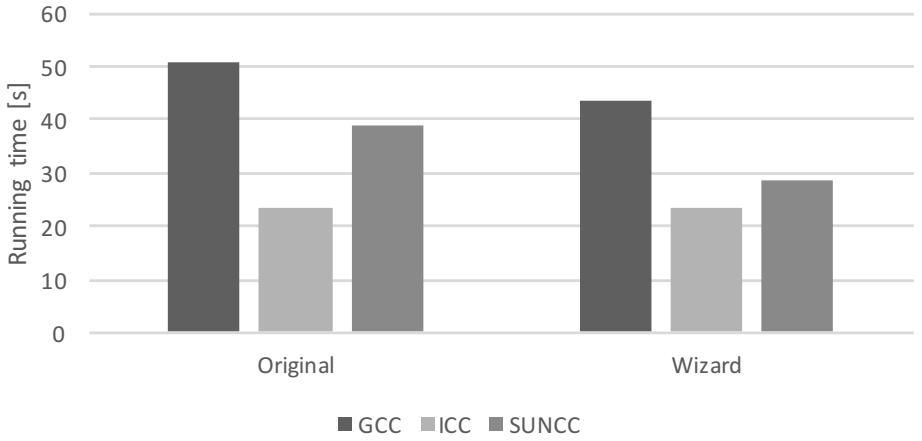


Figure 6.4: Execution time of original and wizard optimized 179.art compiled with GCC, ICC and SUNCC. The most aggressive optimization options have been used, but not profile guided optimization.

optimization as seen in Figure 6.5 with a speedup of 0.8. One reason might be that GCC chooses only to transpose one of the matrixes and XLC transposes two of them. As the wizard applies optimization at the source code it prevents XLC from further optimizing matrixes.

6.2.2 Case study: 183.equake

The 183.equake kernel is a program that simulates seismic waves propagation [OK]. It contains ten global multidimensional dynamically allocated matrixes. All are candidates for optimization. Six out of the ten are automatically recognized as optimizable and the refactoring wizard is offered.

Two matrixes `ARCHcoord` and `ARCHvertex` escape as they are input to a function. If a matrix escapes, we cannot always analyze all access sites. The compiler tends to choose to be conservative and back off when meeting a matrix passed as an argument to a function. However, it may be possible to inline the functions that will solve the problem. Using our tool, it was identified that the `fscanf` C library function was the problem. Functions from the C library cannot be inlined using annotations. However, only the address of a single element in the matrix was passed as an argument. We chose to handle this situation by inserting a temporary variable as seen in Figure 6.6. This is possible as only a single element of the matrix is passed as an argument. This allowed the compiler to optimize both `ARCHvertex` and `ARCHcoord` matrixes.

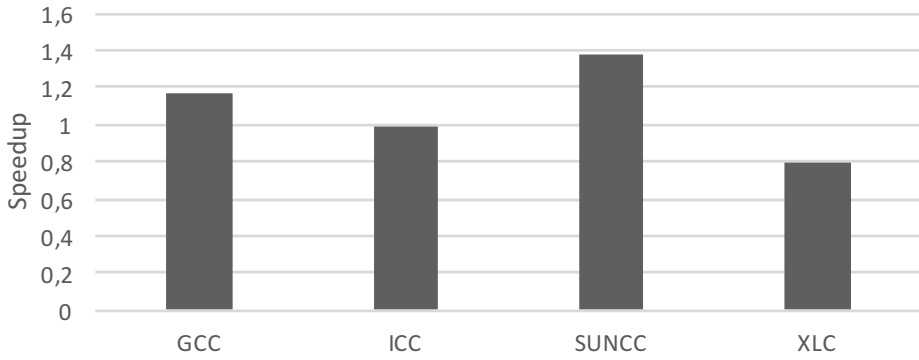


Figure 6.5: Speedups of the wizard optimized 179.art compiled with GCC, ICC, SUNCC and XLC. The most aggressive optimization options have been used, but without profile guided optimization.

```
1 fscanf(packfile, "%d",
2   &ARCHvertex[i][j]);
```

(a) Original source.

$\longrightarrow$

```
1 int tmp;
2 fscanf(packfile, "%d", &
    tmp);
3 ARCHvertex[i][j] = tmp;
```

(b) Modified source.

Figure 6.6: Modifications needed for GCC to optimize `ARCHvertex`. In (a) the compiler backs off but in (b) it is clear to the compiler that the matrix can be optimized.

The matrixes `K` and `disp` also escape. Using our tool, which shows the function calls causing the matrixes to escape, it was possible to identify the function `smvp` as the main problem. In Figure 6.7, we show how information is presented in the IDE. To resolve the problem, our tool proposes to inline the function if possible. We did that by using the `__attribute__((always_inline))` annotation that we applied to the function prototype. If only inline was used, the compiler determined that it is not advantageous to inline the function. However, with `always_inline` the compiler is forced to inline it. After the function was inlined the matrixes are now chosen for optimization and it was thus possible to help the compiler when its analysis is too primitive or conservative.

All global matrixes can be optimized after applying the aforementioned changes suggested by our tool. We changed three source code lines and added four lines. The modifications are minor and do not affect readability.

Unfortunately, we found a bug in GCC when using the optimized source code.

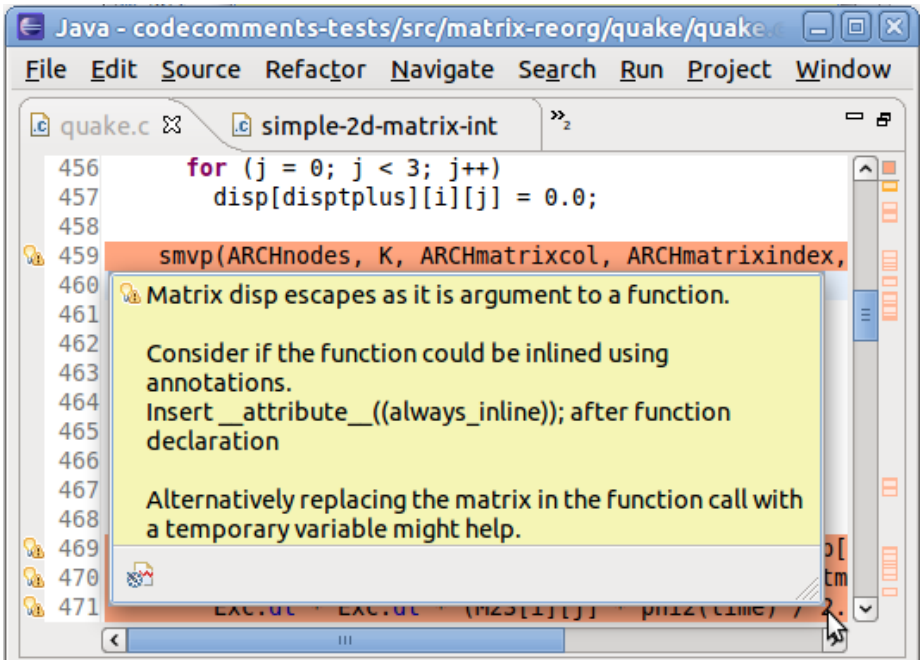


Figure 6.7: IDE output showing the function call that makes the matrix escape and suggested changes.

Table 6.4: Global multidimensional dynamically allocated matrixes in 183.equake and whether they are optimized automatically by GCC and XLC.

Declaration	GCC	XLC
`double **ARCHcoord`	Flattened with help	Flattened with help
`int **ARCHvertex`	Flattened with help	Flattened with help
`double **M`	Flattened	Flattened and transposed
`double **C`	Flattened	Flattened and transposed
`double **M23`	Flattened	Flattened and transposed
`double **C23`	Flattened	Flattened and transposed
`double **V23`	Flattened	Flattened and transposed
`double **vel`	Flattened	Flattened and transposed
`double ***disp`	Flattened with help	Not flattened
`double ***K`	Flattened with help	Not flattened

Under certain circumstances the matrix reorganization optimization might introduce a wrong malloc allocation size. This means that the generated executable will return with an error and stop execution. This is not a bug in our tool, but purely in the optimization pass of the GCC compiler. When flattening the matrix allocation sites, it will not include the allocation statement for all dimensions. This bug has been reported to the GCC Bugzilla [Jen12]. We changed the benchmark so that the affected matrix, `disp`, is not optimized. Therefore, in the presented results, the `disp` matrix has not been flattened.

The 183.equake kernel was also optimized using XLC. It can optimize local matrixes and not just global. In our case, it could optimize two matrixes that GCC could not. Both GCC and XLC could optimize the same global dynamically allocated matrixes in the original source code. However, after our tool has been used, GCC performs better. Now eight out of the ten matrixes are optimized. The reason for XLC not optimizing the last matrixes, like GCC did, appears to be that the compiler will only optimize two dimensional matrixes. GCC chooses not to transpose any matrixes with profile guide optimization. However, XLC chose to transpose some of the matrixes. Table 6.4 shows how each matrix were optimized by each compiler.

Our tool could guide the programmer using code comments to rewrite the code in a simple way that led to more matrixes being optimized. We saw a speedup of 1.30 when GCC was used. A speedup of 1.6 was achieved, using GCC, if the optimizations were applied on the source code level using the wizard. This speedup is not only the result of the compiler optimizing more matrixes. In this case, the changes to the source code made it possible for the compiler to apply more aggressive optimizations. The results, seen in Figure 6.8, show that with

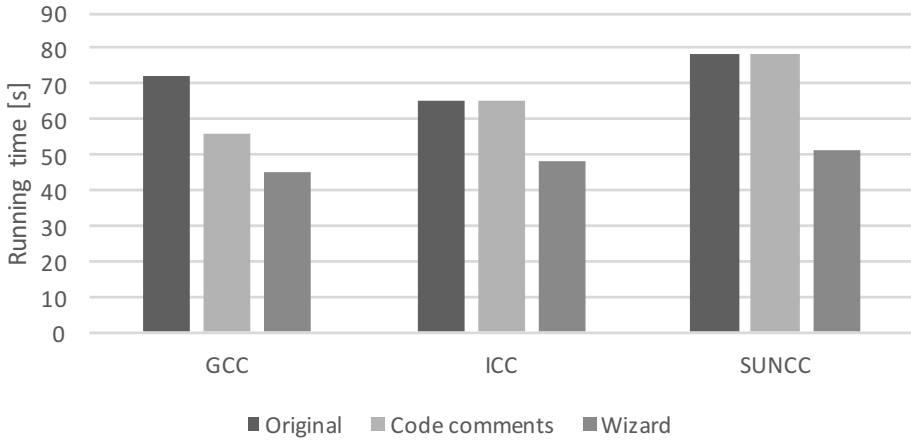


Figure 6.8: Execution time of the original, comments and wizard optimized 183.equake compiled with GCC, ICC and SUNCC. The most aggressive optimization options have been used, but not profile guided optimization.

minor programming effort significant improvements in performance are possible.

We also tried the modified code with multiple compilers. The speedups are shown in Figure 6.9. Only GCC could benefit from the code changes suggested by our tool and they did not affect the other compilers. All compilers did benefit from the changes made by the wizard with improvements in performance.

6.3 Conclusions

We have extended an interactive compilation tool to support high level memory optimizations – matrix reorganization. The tool presents information in the Eclipse IDE guiding programmers to write source code that can be aggressively optimized. The tool uses feedback generated by the GCC compiler. These aspects are related to Research Question 1, showing how the existing compilation infrastructure can be used to generate useful feedback despite the loss of details when lowering from a representation close to the language being compiled to a language more suitable for optimization.

We have furthermore complemented the tool with a refactoring wizard that applies the compiler optimization directly into the source code. This allows for optimized source code to be used with multiple compilers. This show an

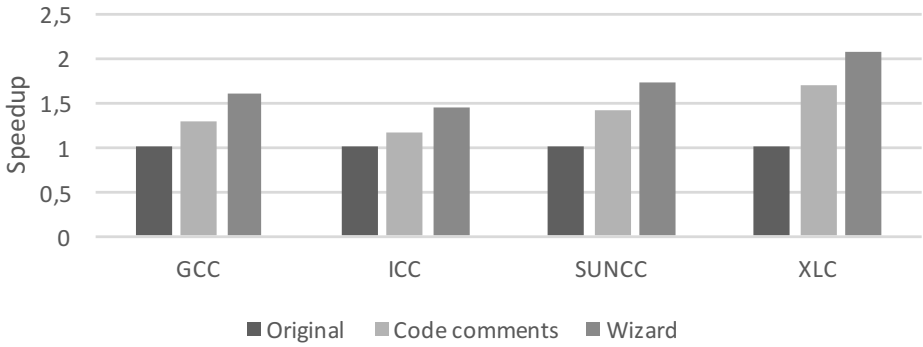


Figure 6.9: Speedups of the comments and wizard optimized 183.equake compiled with GCC, ICC, SUNCC and XLC. The most aggressive optimization options have been used, but without profile guided optimization.

approach for how to apply an optimization from one compiler to other compilers. This allows us to use an optimization in a compiler that does not implement the optimization or where it is not applied due to limitations.

We have evaluated our tool using two SPEC CPU2000 benchmarks. Our results show that it is not always possible to present good hints to the programmer. This was noticed for the 179.art benchmark, where the compiler has issues with a code pattern and no good solution is available. Modified source code was evaluated with four different compilers. We could show that two compilers benefited from the optimized source and two did not.

Better results were possible for the second benchmark 183.equake. Here the tool helped us rewrite the code, in a very simple manner, to allow all possible matrixes to be optimized. The modified source code yields a speedup of 1.3 using GCC. Using the refactoring wizard to make more invasive changes resulted in a larger speedup of 1.6. The refactored code was also compiled with multiple compilers resulting in speedups for the executable code of 1.36 to 1.6. These results contribute to the answer of Research Question 3. The feedback was not always able to cope with the limitations and assumptions of the optimization for the benchmark being compiled. In general, we are however able to significantly improve how benchmarks are optimized, by fully utilizing the existing optimizations without compromising the quality of the source code.

Chapter 7

Continuous Compiler Feedback during Development

In the previous Chapter 6 we saw how we can extend a compiler to emit more data during compilation. After compilation we then collected this data and used it to generate feedback to the programmer. The feedback shows how it is possible to refactor the source code to achieve higher performance by fully utilizing the existing compiler optimizations.

In this chapter we will study a similar tool, but where the feedback is generated continuously during development for faster feedback to the programmer and easier tool setup attempting to address Research Question 1.

Arguably we all rely on optimizing compilers for performance. When they optimize well, they can often generate code that outperform hand optimized code. However, compilers ability to optimize aggressively is limited in some cases as Chapter 3 highlighted.

Programmers are often not aware how their programs are optimized and how they should be written to allow the compiler to optimize well. We have created a compiler, that guides programmers in modifying their programs, potentially

making them more amenable to optimization.

To address this limitation, we have developed a compiler guiding the programmer in making small source code changes, potentially making the source code more amenable to optimization. This tool can help programmers understand what the optimizing compiler has done and suggest automatic source code changes in cases where the compiler refrains from optimizing. We have integrated our tool into an integrated development environment, interactively giving feedback as part of the programmer's development flow.

Optimizing compilers and integrated development environments both perform analysis of code, regrettably usually with no code sharing between these. Therefore, we have embedded a small optimizing compiler into an IDE. The compiler is part of the programmer's development flow continuously giving feedback to the programmer as it is integrated with the IDE.

We open up the black box the compiler is today, exposing valuable information to the programmer using compiler driven feedback. In contrast, other related tools can also report on optimization issues encountered during compilation. The programmer then has to understand why a given optimization was not performed. Our tool is integrated into the IDE, dynamically giving quick feedback to the programmer during development. The integration improves the automatic refactoring we provide.

The benefit of the feedback is twofold, programmers know how their code how been optimized and why the compiler did not optimize. Compilers often refrains from optimizing, especially due to limitations of program analysis. Programmers can unintentionally prevent optimization by unfortunate coding choices. If the programmer is aware of the problems, they can be mitigated by modifying the source code, often without affecting other software engineering principles.

An early implementation of the introduced compiler has been evaluated on 12 kernel benchmarks and we show that the feedback can lead to speedups of up to 153%.

In short, we make the following contributions:

- We show that the traditional compiler structure can be redesigned by reusing existing IDE technology in the compiler.
- We show that we can provide better automatic refactoring by integrating the IDE and the compiler. We provide feedback on a broader range of issues and present the feedback directly in the IDE.

- We have evaluated the tool and show that compiler driven feedback can lead to performance speedups. For example, the suggested automatic refactorings can improve performance with up to 153% when used with a production compiler.

The chapter is laid out as follows. The tool is presented in Section 7.1 and Section 7.2. Experimental results are analyzed in Section 7.3. We discuss the work in Section 7.4 and conclude the chapter in Section 7.5.

7.1 Compiler Infrastructure

We have designed and implemented an optimizing compiler infrastructure to address the issue of giving precise compiler feedback on the original source code. The compiler is embedded inside the Eclipse IDE. The compiler reuses existing Eclipse technology in the front-end by using the infrastructure already implemented for code analysis and supporting the programmer during development. In this way, the implemented compiler integrates into the normal Eclipse development flow, continuously giving feedback to the programmer during development. Furthermore, internal Eclipse data structures constructed for existing source knowledge tools can be reused. This decreases the load of executing our compiler during development in contrast to running an entire production compiler.

The compiler is constructed as a series of passes where most consume a single intermediary representation. The architecture and flow through the compiler is presented in figure 7.1. The infrastructure is organized as a series of modules with clear interfaces between them. The C front-end is constructed using Eclipse language tooling. The front-end takes the Eclipse representation of the source code and generates the internal representation inside the compiler for optimization. The optimizer consumes the internal intermediary representation, IR, and produces IR. The optimizer interfaces with the feedback-visualization Eclipse plugin by generating data on applied optimization and when an optimization was not applied. Last the code generator consumes the IR and emits assembly code compatible with GCC [Fou] calling convention, which can be assembled and linker by GNU Binutils [Fou13].

Only the C language is supported and the C front-end uses the Eclipse CDT C/C++ language tooling [CDT13]. The idea behind the compiler infrastructure is not restricted to C and there exists Eclipse language tooling for many other languages.

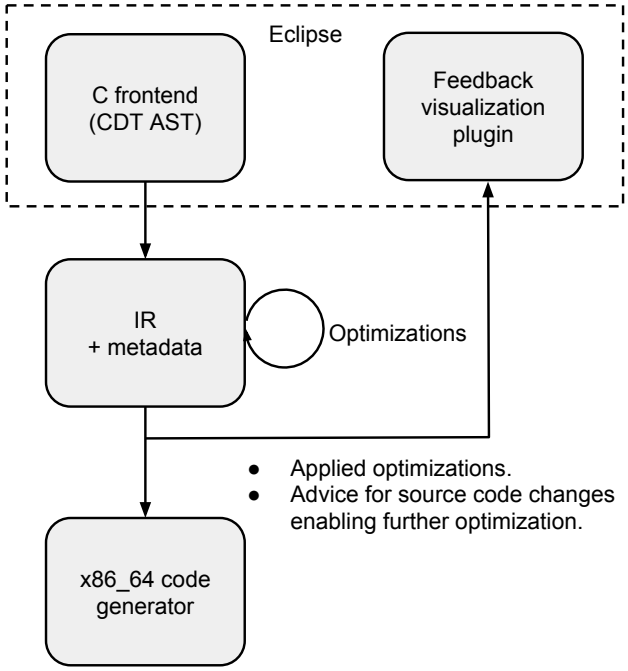


Figure 7.1: Compilation framework architecture.

The intermediary representation used for feedback and optimizations is based on Static Single Assignment, SSA, form [Cyt+91]. It has the special property that variables are only assigned at most once. This form is argued to improve many types of optimization and is found in many optimizing compilers. The IR is on three address form and consists of blocks with sequential instructions and special blocks for control flow.

7.1.1 Optimizer

The compiler is intended to be executed during the development, constraining the type of optimizations that can be executed fast enough without disturbing the responsiveness of the IDE. However, arguably the more optimization passes we implement the better feedback we can provide. We have implemented a series of scalar optimizations on the SSA form: Copy propagation [Ske12], constant folding, sparse conditional constant propagation [WZ91], arithmetic simplifications, partial redundancy elimination [Ken+99], global value numbering [TC11], operator strength reduction [Ske12], loop interchange and inlining. All optimizations are performed in the order suggested by Muchnick [Muc97].

7.1.2 Backend

The backend of a compiler is very important for its performance. The implemented backend is simple and does not generate optimal code. The backend targets the x86_64 instruction set. It does instruction selection using the greedy maximal munch algorithm [App97] and instruction scheduling using a simple list scheduling implementation [Ske12]. The register allocator is based on the simple Linear Scan working on SSA form [WF10]. It allocates registers for one linear block at a time, linearly assigning register by choosing a register not used in the period where the register is alive. The allocator only looks a single block at a time, thus the allocation is not optimal globally. Furthermore, no low level optimization is performed during code generation. For simplicity all the features of the x86_64 instruction set is not used.

7.2 Feedback Infrastructure

To display feedback to the programmer and suggest automatic refactorings, we extend and use the infrastructure in Eclipse. This includes markers, yellow

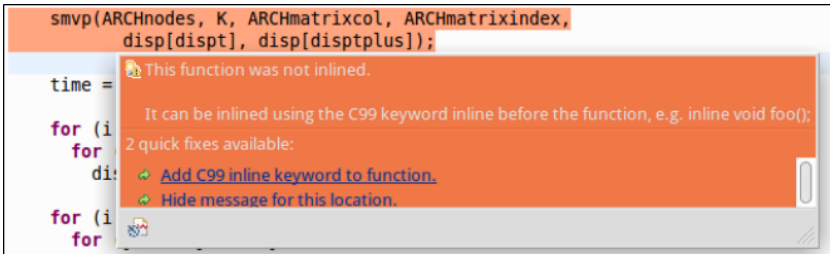


Figure 7.2: Simple marker suggesting adding the C99 `inline` keyword to potentially enable inlining of the `smvp` function. Executing the suggested automatic refactoring gives a 22.6% improvement with GCC 4.8.1.

sticky notes in Eclipse used to present tasks, problems or general information about a source code construct [GM01]. Eclipse contains views used for navigating or displaying a hierarchy of information. We have implemented a view, giving an overview of all the feedback presented to the programmer. This allows the programmer to get an overview of all the feedback given by the compiler.

An example of the kind of feedback we provide to the programmer is seen in figure 7.2. Here we inform the programmer that a function was not inlined and suggest an automatic refactoring. This simple refactoring lead to a 22.6% improvement using GCC 4.8.1.

All optimizations in the compiler generate data on applied optimizations and data on missed optimizations. These show to the programmer how the compiler has optimized. Furthermore, the compiler gives details on why a given optimization was not applied and suggest an automatic refactoring. The modifications can be applied automatically, simply by accepting the compilers suggestion.

The tool can currently provide automatic refactorings for:

- Removing dead code
- Applying the `inline` keyword to functions
- Permuting the loop order
- Applying the `restrict` C99 keyword to pointers, even if matrix notation is used.

Mapping the internal representation used by the optimizer to the source code is important. Therefore, we have integrated with the IDE and keep Eclipse

metadata in the IR during compilation, relating the IR to the Eclipse internal data structures.

7.3 Experimental Evaluation

7.3.1 Setup

We evaluate our implementation on a platform based on a dual core 1.9Ghz Intel Core i7-3517U and a total of 6GB of DDR3 ram. The operating system is Linux with kernel version 3.8.0.

Furthermore, the performance is compared to GCC 4.8.1 with no optimizations `-O0` and with the aggressive optimization level `-O3`.

The compiler has been evaluated on 12 kernel benchmarks. Three very simple kernels and nine kernel benchmarks from the Spec2000 benchmark suite [Hen00] and the Java Grande C benchmarks [Bul+01]. The benchmarks are of varying size, art and equake are the largest with 1042 lines of C code as measured using SLOCCount [Whe13]. The kernel benchmarks are chosen to represent resource demanding programs testing different elements of the system as they have high processing, I/O or memory demands.

In addition, the power of the tool as an optimization adviser for a production compiler is evaluated. This have been evaluated on an edge detecting program from the University of Toronto DSP Benchmark Suite [Lee]. This benchmark has only been evaluated using GCC due to a bug in the compiler.

7.3.2 Compiler Performance

First we consider the performance of the compiler used as a static offline compiler, similar to how GCC normally would be used. I.e. we do not consider the compiler feedback in this section, only the quality of the generated code. The compiler has been evaluated using all 12 kernel benchmarks. The performance is reported first without any optimizations and second with all the implemented optimizations.

The performance data is presented in figure 7.3, where the running time of each benchmark is reported using our compiler and GCC. GCC outperforms our

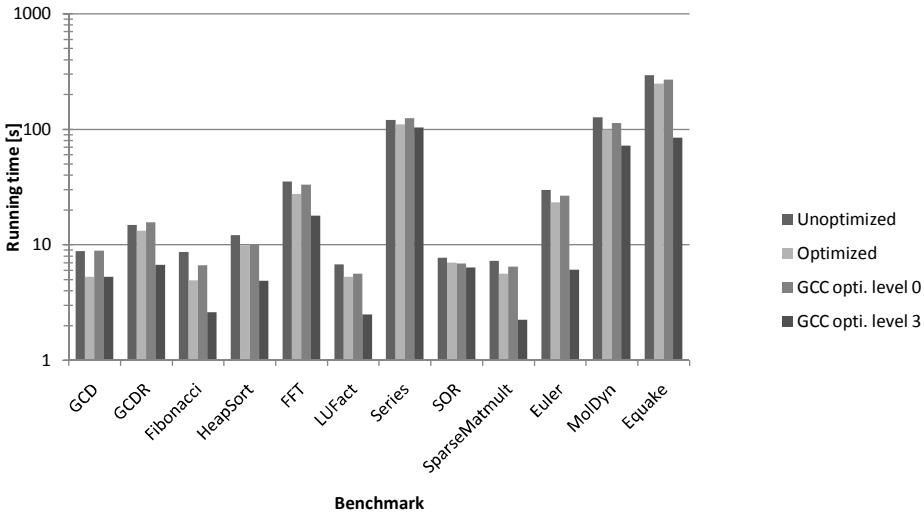


Figure 7.3: Raw performance of our compiler and GCC 4.8.1. Lower is better.

compiler. Even for the simple Fibonacci benchmark, GCC is 1.8 times faster. In the Fibonacci benchmark GCC performs many optimizations on the source code. It inlines the recursive calls and use tail recursion optimization to turn the call into a loop. This decreases the amount of recursion significantly. In contrast we can only do simple scalar optimizations on the Fibonacci kernel.

Our compiler is designed to be lightweight and executed during development, for giving quick feedback to the programmer. In contrast GCC use optimizations that are significantly slower, but optimize more aggressively.

7.3.3 Feedback

The compiler can guide the programmer with automatic refactorings, potentially making the code more amenable to optimization.

An example of the feedback advice provided is that a given function was not chosen to be inlined. This might be due to the size of the function or if the function is recursive. This can be seen in figure 7.4 on a recursive greatest common divisor implementation. In C99 the `inline` keyword exists, which can be applied to functions to force that the function is inlined. GCC implements three semantics for the `inline` keyword in programs, one for GNU C89, another for C99 or C11 or GNU C99 or GNU C11 and a third semantics for C++. In C99

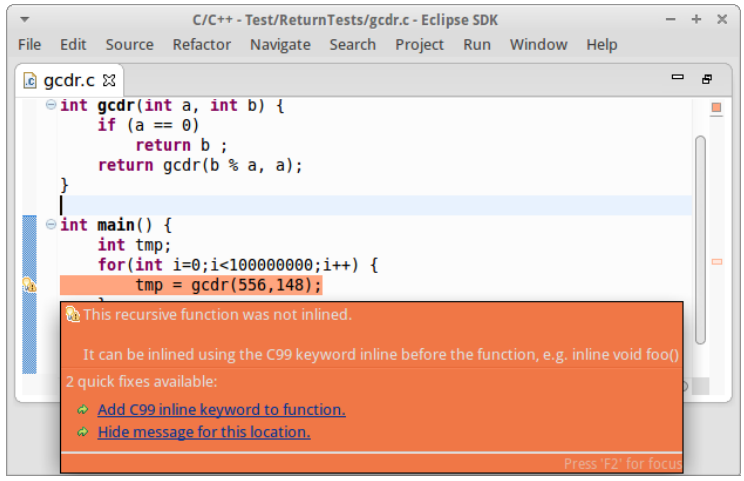


Figure 7.4: Simple marker suggesting adding the C99 inline keyword to enable inlining of the recursive function named gcdr.

the `inline` keywords intend that the function definition is only provided for inlining. Thus if it is not inlined the compiler will rely on another implementation of the function without the `inline` keyword somewhere else in the program. In GNU C89 the `inline` keyword is treated as a hint to the optimizer. In C++ the `inline` keyword is a linkage directive. To this end, we support adding either `static inline` or the GNU specific attributes.

The feedback adviser can only handle a specific number of cases where the compiler refrains from optimizing. Thereby, we cannot generally optimize all programs. Two benchmarks, the simple Fibonacci and GCD kernels, did not benefit from the feedback advice. The other 10 kernels yielded small speedups. The speedup obtained, by using the feedback advice for small source code changes is shown in figure 7.6. The baseline is the optimized results from figure 7.3.

For the LUFact and Euler benchmarks the tool suggest automatic refactorings for applying the `inline` keyword to loop intensive functions and interchanging the order of some loops. However, the compiler cannot automatically analyze whether the transformation is safe due to limitations in the dependency analysis. Therefore, we suggest an automatic refactoring and leaves the task of whether doing so is safe to the programmer as seen in figure 7.5. Performing the loop interchange refactoring yields a speedup of 12% for LUFact and 8% for Euler.

For the Equake benchmark the feedback suggests adding the `inline` keyword a function call to the `smvp` function. This both reduce the function call overhead,

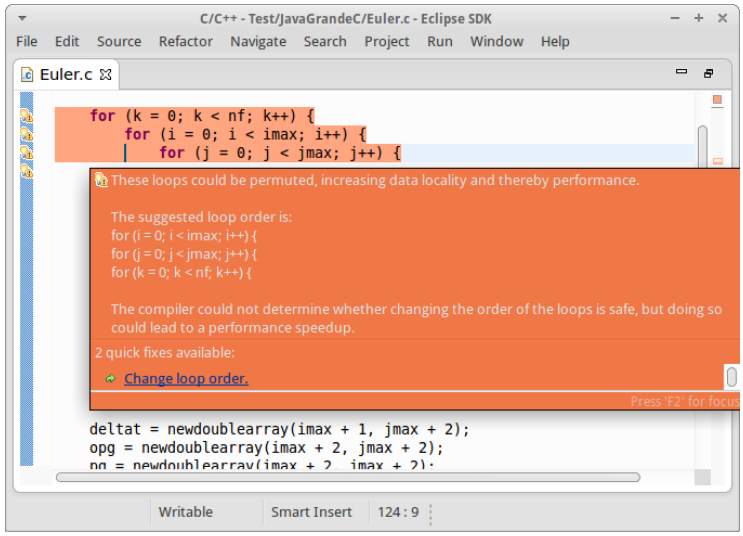


Figure 7.5: Marker suggesting permuting the order of the loops, if the programmer determines doing so is safe.

but also makes some optimizations more effective. Inlining the `smvp` function call yields a speedup of 7.6%.

7.3.4 Evaluate Feedback on Production Compilers

The quality of the feedback when used by a production compiler is also evaluated. In this use case we can use the small optimizing compiler during development and the refactored source code is then subsequently used with a production compiler. The focus of our compiler is however not its performance.

The automatic refactorings suggested by our tool are also evaluated using the GCC compiler. The kernel benchmarks have been optimized by applying the suggested automatic refactorings described in section 7.3.3 and then afterwards compiled with GCC 4.8.1 using the aggressive optimization level `-O3`.

The results are seen in figure 7.7. The baseline is the original source code, aggressively optimized with `-O3` using the same version of GCC.

Three of the kernels shows decent speedups, namely LUFact, Euler and Equake of 11%, 5% and 23% respectively. More inlining have widened the scope of GCC's optimizer, allowing it to optimize the compute intensive parts of the

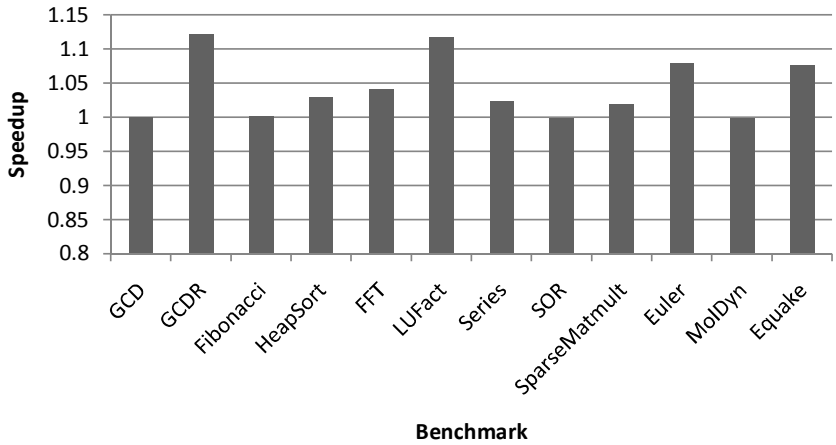


Figure 7.6: Speedup using the feedback advice on each benchmark.

kernels more aggressively. The manually interchanged loops also yield small speedups, but as the loops are not part of the compute intensive part of the kernels the speedup is moderate.

7.3.4.1 Parallelization Adviser

Earlier research has shown the benefit of applying the `restrict` keyword to C pointers, mitigating issues preventing optimization [Lar+11a]. The `restrict` keyword states that only the pointer itself, or a value derived directly from the it, is used for all accesses to that object. This makes the programmers intent clearer to the compiler, mitigating many issues preventing optimizations. It can benefit automatic parallelization, vectorization and loop transformations.

For evaluating the feedback when performing automatic parallelization, we study an edge detection application from the University of Toronto DSP Benchmark Suite. Our compiler can provide optimization advice on the edge detection kernel as seen in figure 7.8. Applying the C99 `restrict` keyword is simple on pointers, but may require more work on arrays. For example, the array arguments can be rewritten as `(*restrict input_image)[N]` or `int input_image[restrict N][N]`. We prefer the second notation, but they are for the GCC compiler optimized in the same way.

We can also annotate the inner dimension of the array with the `static` and `const` keywords. `static` entails that there will be at least the defined number of elements (`N` in the previous example) also implicate that the pointer will not

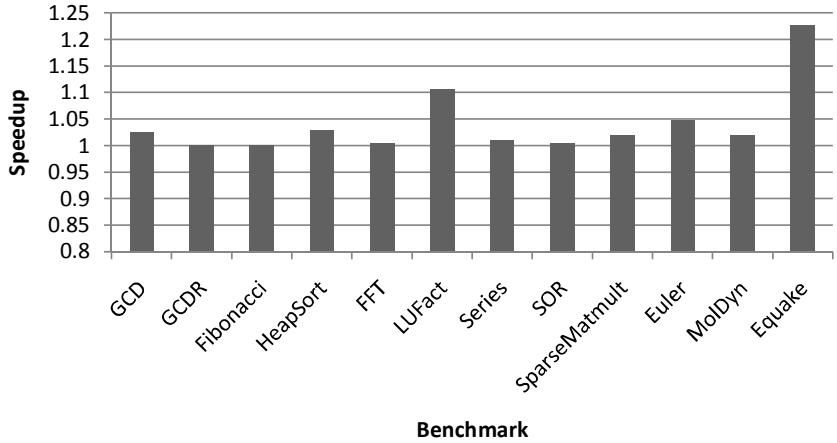


Figure 7.7: Speedup using the feedback advice on each benchmark using GCC.

be null if more than 0 elements and for `const` that the array pointer does not change in the function. We support the `restrict` and `static` keywords. The suggested refactoring can be applied automatically by the tool.

The result has been evaluated on the system described in section 7.3 using GCC 4.8.1 and the compiler flags `-O3 -ftree-parallelize -loops=#threads -fopenmp` as seen in figure 7.9. The reported time is excluding the I/O activity in the beginning and end of the program. Three numbers are reported for the Intel i7 platform one and two threads respectively; the sequential execution, the speedup of GCC's automatic parallelization and the speedup after performing the automatic refactoring.

The best speedup is achieved with two threads, where the modified version is 16% faster than the automatically parallelized unmodified version.

We have also analyzed the edge detection application on a 2.66 GHz quad-core Intel Xeon X5550 system, with 24GB of ram running Linux kernel 2.6.32. The available compiler on this system was GCC 4.7.2. The evaluation is shown in figure 7.9. All loops were parallelized in the modified version, but most of the speedup comes from automatic vectorization. GCC chose to parallelize the inner loop and therefore the full potential of the parallelization is not achieved. The best speedup was 2.62 with eight threads compared to the sequential version.

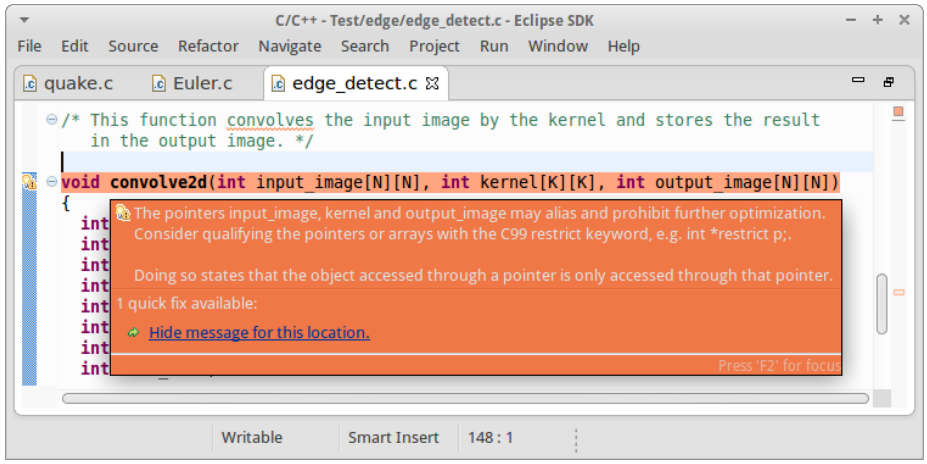


Figure 7.8: Marker suggesting adding the `restrict` keyword to the three array arguments.

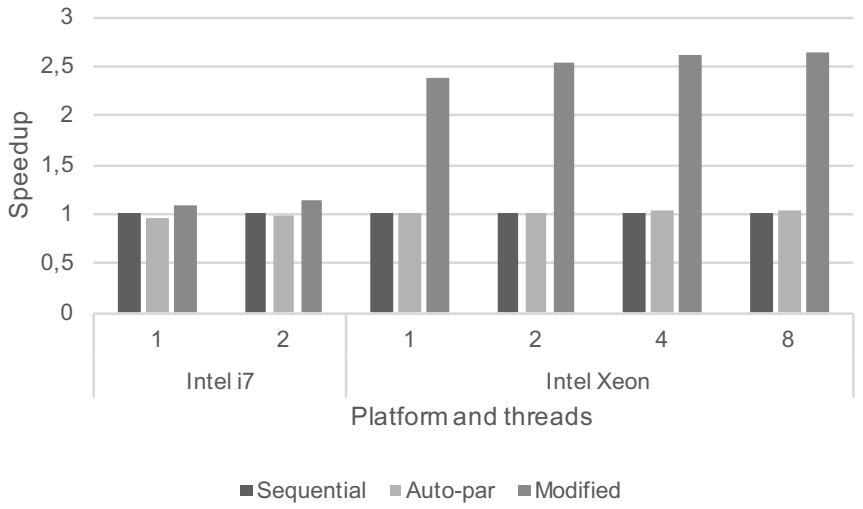


Figure 7.9: Speedup and executing time of the edge-detection kernel on the (a) Intel i7 platform and (b) Intel Xeon platform. In the modified version using the feedback suggestions, all loops were parallelized. In the unmodified the outer loop were not parallelized.

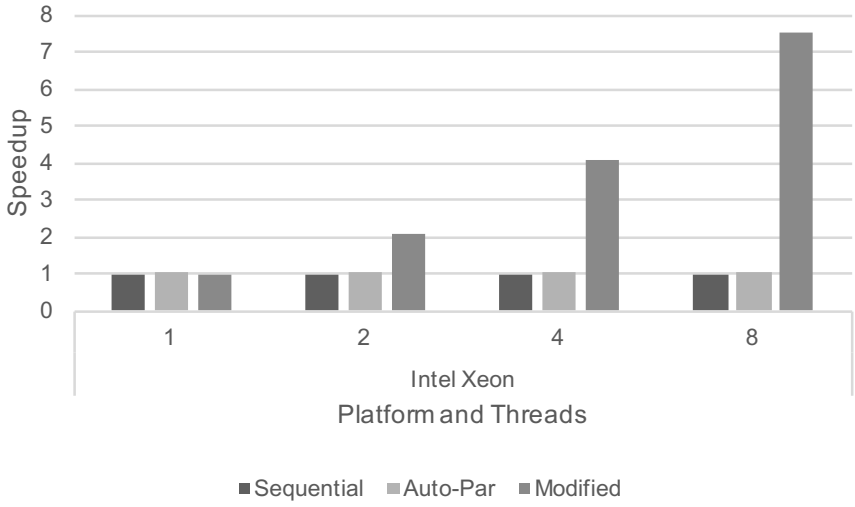


Figure 7.10: Speedup and execution time achieved with `-O2` and disabled vectorizer on the Intel Xeon Platform.

7.4 Discussion

One big challenge for feedback systems is when and what to display to the programmer. Feedback system can often generate many comments, but many are false positives and only a few will lead to a speedup. Therefore, one limitation is how we do not validate whether the feedback is a false positive and remove it. One way to do it is to filter the feedback by automatically applying the suggested automatic refactorings and validate whether the compilation benefited. The generated assembly can also be used to statically approximate the running time and use this information for filtering.

Another limitation is that even when we enable the compiler to optimize aggressively, it might not make the best decision for how to do so. Compiling with a production compiler can be complicated, for example GCC 4.8.1 has over 250 optimization flags with many possible combinations. On the edge detection program, if we use the `-O2` optimization flag and disable the vectorizer, we can parallelize all loops giving a speedup of up to 7.46 as shown in figure 7.10. From a programmer's perspective this combination of automatic refactorings and optimization flags is far from obvious. This speedup indicates a great potential for also guiding in combination of compiler flags.

7.5 Conclusions

Many applications rely on automatic compiler optimization for optimal performance. Regrettably, programmers are often not aware of how their programs are optimized and especially which small refactorings can lead further optimization by the compiler. To address this issue, we have developed our dynamic feedback compilation system, which can give feedback to the programmer during development directly in the IDE. Compared to other compiler driven feedback systems the advantage of the presented tool is dynamically executing the embedded compiler, as part of the normal Eclipse development feedback system and the IDE integration benefiting the automatic refactoring. This chapter contributes to address Research Question 1 as we show how it is possible to generate useful feedback without involving the compiler. We highlight the benefits of faster feedback and advanced refactorings made possible by working on a representation closer to the source code level. We also highlight the limitations with false positives generated.

The implemented compiler has been evaluated on 12 C benchmarks, but compared to GCC does not yield good performance. This is expected, as the tool is designed to be fast and be executed continuously as the programmer works. The optimization advise system have also been evaluated on the benchmarks. The feedback on inlining of function calls and automatically changing the order of loops lead to simple source code modifications, where 8 out of 12 benchmarks yield a speedup. The system can guide to a 12% improvement over the compiler optimized version.

The tool can also be used as a feedback adviser for a production compiler. Here 9 out of 12 of the benchmarks benefited from the feedback. On the Equake benchmark the tool can guide to a 23% improvement. Given the little work required by the programmer to obtain the speedup, the compiler driven feedback is cost effective. Furthermore, it was also investigated on an edge detection program, how the feedback could improve the automatic parallelization performance using a production compiler. Here the tool could guide to a 153% performance improvement. This is part of addressing Research Question 4 as we show how we can generally provide feedback improving optimizations such as automatic parallelization and automatic vectorization.

Chapter 8

Compiler Feedback using Multiple Compilers

In this chapter we study another approach and another tool for compiler feedback. In the previous Chapter 7 we saw how we can generate feedback inside the IDE as the programmer is developing. Further, we saw how we have extended one compiler in Chapter 6 to emit further information during compilation and show this feedback in the IDE guiding the programmer.

In this chapter we describe a tool using information from multiple unmodified compilers. The tool is named Kommentator. It assists programmers in understanding how the compilers optimize and give advice changes to the source code allowing more aggressive optimizations in an attempt to address Research Question 1.

Optimizing compilers are essential to the performance of parallel programs on multi-core systems. It is attractive to expose parallelism to the compiler letting it do the heavy lifting. Unfortunately, it is hard to write code that compilers are able to optimize aggressively and therefore tools exist that can guide programmers with refactorings allowing the compilers to optimize more aggressively. We target the problem with many false positives that these tools often generate, where the amount of feedback can be overwhelming for the programmer. Our approach is to use a filtering scheme based on feedback from multiple

compilers and show how we are able to filter out 87.6% of the comments by only showing the most promising comments. This addresses Research Question 2.

Writing programs that performs well on modern multi-core systems is a major challenge. Many aspects influence the performance, especially how well the optimizing compiler has transformed the code into a faster version. As parallel programming is hard it is attractive to expose the parallelism to the compiler. With automatic parallelization and automatic vectorization, the compiler can do the heavy lifting. Sadly, it is hard to write code that compilers can optimize well given the large complexity of compilers.

We have developed a tool Kommentator that can assist programmers in understanding how compilers optimize and even give advice on source code changes that could allow for more aggressive optimization. Many such tools exist, but we believe that one key difference will make our tool more usable, namely the number of false positives and true positives. Programmers will only use a tool if it is cost effective and a good use of their time. If only one out of many comments can successfully be applied it is not effective. In contrast to other similar tools, we use input from multiple compilers, allowing us to produce fewer false positives while still producing many true positives. We believe this is key for a wider adoption of optimization advice tools.

Kommentator works by parsing the optimization reports from multiple compilers and use insight that if one compiler succeeded in optimizing, the others could potentially as well. In this way, if one compiler succeeds in optimizing, even just partially, we might be able to modify the original source code such that more optimizations can be applied. We are able to filter the number of comments generated by three compilers with 87.6%, resulting in an amount of comments that is easier to handle and focus on for the programmer.

Our current implementation supports the optimization reports from three major production compilers ICC [Int], GCC [Fou] and Clang [The]. Each of these has optimization reports, which describe the applied optimizations and the missed optimization. We analyze these reports in our plugin, built into the Eclipse Integrated Development Environment. Combined with our own analysis we can suggest automatic source code refactorings to the programmer. We also visualize how the different compilers have optimized the code by coloring the source code, giving a very quick overview of where the programmer's time is best spent.

To summarize, this chapter makes the following contributions:

- We propose a novel compiler driven feedback model based on input from multiple compilers.

- An implementation supporting optimization reports from ICC, GCC and Clang.
- Show how we are able to filter 87.6% percent of the compiler generated comments.
- Last, we study an industrial use case and achieve a speedup factor of 1.54 over an OpenCV implementation executing on a GPU by optimizing the version of the use case targeting the CPU.

The chapter is laid out as follows. The tool is described in section 8.1. Experimental results are analyzed in section 8.2. Last, section 8.3 concludes the chapter.

8.1 Multi-Compiler Feedback Tool

Our tool is based on the optimization reports from production quality compilers. These can report applied optimizations and missed optimization.

We currently support input from three compilers, namely ICC from Intel and the two production quality open source compilers GCC and Clang as seen in Table 8.1. The versions are the newest at the time of writing. The optimization report feature is new in GCC and Clang and thus has limited support for the number of optimization passes it can produce feedback from. Therefore, we focus on automatic vectorization of loops, an optimization that is very important for good performance on modern processors.

There are many limitations to automatic vectorization as it involves numerous advanced analysis steps. Every compiler performs roughly the same steps, however as the implementations vary they each has strengths and weaknesses. Some of the types of analysis that needs to succeed are: identification of loop bounds and stride, induction variable analysis to determine dependencies between loop iterations and alias analysis to again to determine dependencies within and between loop iterations. These three analyses are used as input into the actual data dependency analysis. For each of these analyses there exists many weaknesses, some of which can be addressed at the source code level. We propose automatic refactorings to the programmer to mitigate these limitations. One example is how we can help alias analysis by specifying that two memory locations are distinct using the C99 `restrict` keyword. We reuse many of the automatic refactorings shown to be effective in earlier research [JKP14b]; [Lar+12]. The tool process is twofold as seen in Figure 8.1:

Table 8.1: Supported compiler versions and compiler flags

Compiler	Version	Flags
ICC	Intel Composer XE 2015 [Int]	-opt-report
GCC	GCC 5.3 [Fou]	-fopt-info-optall
Clang	Clang 3.5.0 [The]	-Rpass=.*

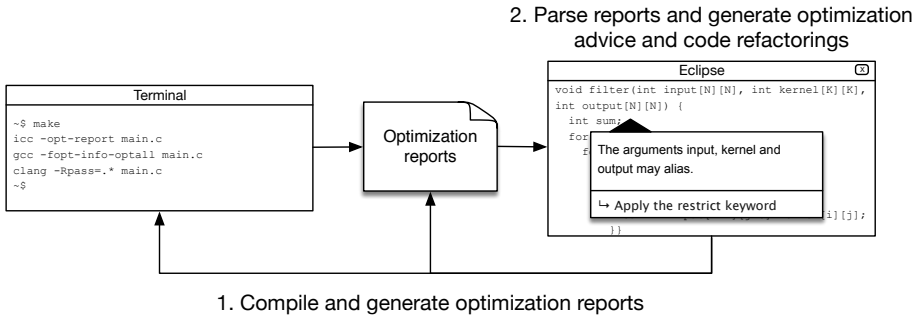


Figure 8.1: Overview of tool process.

1. First the programmer has to manually change the build system, such that the program is compiled with multiple programs and with the additional compiler flag for producing optimization reports. We are working on automating this step. The flags used for all later examples are shown in Table 8.1.

2. Second we parse the optimization reports in our Eclipse plugin, aggregate all the comments based on loops, filter them based on how each compiler optimized and display the overview and proposed automatic refactorings to the programmer.

The plugin is based on Eclipse Kepler 4.3. The first step is to parse the optimization reports generated by the compilers. Correlating the comments presents a challenge in itself, for multiple reasons. Different coding styles must be handled as seen in Figure 8.2. Comments from different compilers may refer to same loop, but different source code lines. We handle this by relating each comment to a loop instead of a source line. This is based on a simple algorithm that first finds loops and loop nests, and their corresponding source code line ranges. This implementation assumes that we only have one loop per source code line. One last issue handled is how comments for inlined function calls are handled. Depending on the compiler, these may be described as corresponding to the call site or the function itself.

```
for (int i=0; i < N; i++)
```

(a) No opening brackets

```
for (int i=0; i < N; i++)
  {
```

(b) Brackets on the subsequent lines

```
LOOP BEGIN
  at file.c(1)
  remark #15300:
  LOOP WAS VECTORIZED
LOOP END
```

(c) ICC comment referring to line 1

```
file.c:3: loop vectorized
```

(d) GCC comment referring to line 3

Figure 8.2: Issues encountered when correlating compiler comments. In (a) and (b) different coding styles must be handled. In (c) and (d) comments from different compilers refers to the same loop, but different lines numbers in the source code.

After aggregating comments, we classify how each compiler has optimized into three categories: not vectorized, partially vectorized or fully vectorized. We present this classification directly in the IDE to the programmer using colored source code lines. We color the Eclipse marker bar either green, orange or red depending on how many compilers optimized. In this way we do not overwhelm the programmer with too much information and if more information is desired, hovering over a marker bar will present the classification and the individual compilers comments as seen in Figure 8.3.

For producing advise to the programmer we use a set of rules, detecting issues in

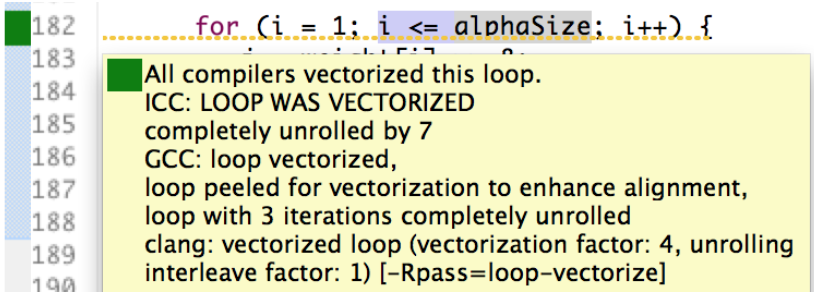


Figure 8.3: Screenshot of the visualization of how the different compilers vectorize. By hovering over the source line the compiler feedback will be displayed.

the compilation output and suggesting a refactoring potentially leading to more aggressive optimization. We only directly support targeting the GCC compiler, but advise for any compiler can be generated by overcoming the challenge in different optimization output and supported directives. The following limitations can be mitigated with the supported automatic refactorings:

- Aliasing by suggesting static and global arrays or automatic refactoring for restrict keyword.
- Data alignment by suggesting adding alignment attribute.
- Data dependency by compiler specific pragmas making the compilers assume no loop carried data dependency.
- Profitability by suggesting pragmas for forcing vectorization.
- Suggesting linking against a math library with vector implementations.
- Suggesting permuting loop order.
- Suggest inlining using the C99 keyword.
- Side effects of function call using the pure attribute.

Besides these, the tool will show issues with optimizing that the programmer can combine with his knowledge of the application.

8.2 Experimental Evaluation

We have studied the C benchmarks from the SPEC CPU2006 benchmark suite [Hen06] in total 11 benchmarks: 401.bzip2, 403.gcc, 429.mcf, 433.milc, 445.gobmk, 456.hmmer, 458.sjeng, 462.libquantum, 464.h264ref, 470.lbm and 482.sphinx3. These benchmarks consist of 494.709 lines of C code and contain 30.370 loops that can potentially be optimized. We use the compilers ICC, GCC and Clang with the versions seen in table 8.1. Each compiler targets the Intel Haswell platform. Out of the total 30.370 loops in the benchmarks, only 8829 loops produce any comments by some compiler. This is mainly due to the large amount of optimization done in the compilers, e.g. full loop unrolling will eliminate a loop.

The number of loops each compiler vectorized is shown in Figure 8.4. We see how ICC from Intel is clearly dominating with 805 vectorized loops, where GCC and Clang vectorized 335 and 260 loops respectively. We also see how GCC and

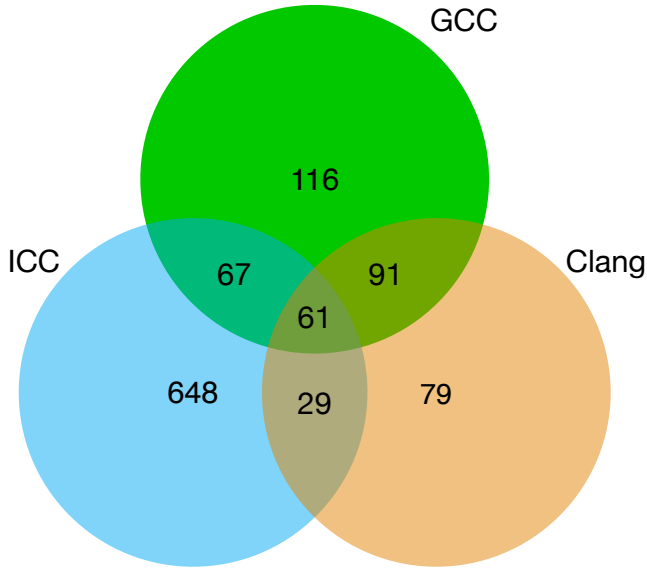


Figure 8.4: Venn diagram of how many loops the three compilers, ICC, GCC and Clang have vectorized among the loops in the 11 C benchmarks from SPEC CPU2006.

Clang optimize many loops that ICC does not, it may be a missed optimization in ICC or that it is actually not beneficial to do so.

We intend to use this data in a loop ranking mechanism that ranks all loops based on how likely it is a suggested refactoring is going to succeed and improve performance. We use profiling data to rank hot loops and answer how worthwhile a closer inspection is. We use compiler reports to rank loops on how likely it is a refactoring is going to succeed. This means we can focus the programmer's attention on hot loops that are not optimized fully by his chosen compiler, but by another compiler. These loops are good candidates for making the suggested automatic refactorings cost effective.

In this way see how for example ICC could have vectorized 286 additional loops, GCC could have vectorized 756 additional loops and last Clang could have vectorized 831 additional loops. If the programmer only target one specific compiler, the amount of compiler feedback is possible to handle given that these are derived from 11 benchmarks. For all compilers the loops with feedback have reduced the number of loops with feedback from 8829 loops to 1091, an 87.6% reduction.

In the previous Chapter 6 and Chapter 7 we have shown the benefit of automatic refactorings that allows the compilers to optimize more aggressively. One result is the speedup achieved by adding the `restrict` keyword from C99 to an edge detection kernel from the UTDSP Benchmark Suite [Lee]. This comment is not filtered out by our tool, and can help GCC with an automatic refactoring allowing it to vectorize one loop giving a 140% speedup [JKP14b]. Multiple related works presents other use cases with good speedups [Jen+12b]; [Lar11]; [Lar+12].

We have applied our tool to the extended Test Suite for Vectorizing Compilers, TSVC [Mal+11]. We were able to vectorize an additional 30 loops by introducing source code directives for forcing vectorization due to profitability and ignoring data dependencies, permuting the loop order and introducing further alignment and aliasing hints. These changes led to an improvement in performance of 5.7% for GCC on an Intel Core i7-3517U with AVX @ 1.90GHz.

To give more precise data it would be very relevant to add more compilers and platforms. This could include XLC from IBM on the Power architecture. With input from more compilers it would be possible to give extra priority to loops that are optimized by multiple other compilers.

8.2.1 Case Study: Kolektor's RoughnessMeasurements Application

We also study applying our tool to an industrial use case from Kolektor. The use case is called RoughnessMeasurements and is one of the programs used by Kolektor to evaluate a texture using machine learning as part of their quality control in the production line for commutators.

The use case has two main computationally intensive parts: one recalculates a regression tree model with a set of images using supervised machine learning. The other processes an incoming image and classifies it as "OK" or "NOK". The costliest part is recalculating the model so we have focused on this part. A set of 300 images are used to recalculate the regression tree model. The program is written in C++ using OpenCV [Bra00] library calls. The OpenCV routines are optimized using CUDA to accelerate it for executing on a GPU. The program contains several image processing steps for extracting properties later fed to the machine learning library. We focus on the extraction part of the program as it is the costliest part. The machine learning is performed using the WEKA machine learning suite [Hal+09].

To evaluate DTU Kommentator within COPCAMS, we report on our experi-

Table 8.2: Experimental Machine Specification

Processor	ARM Cortex-A15 32bit
Frequency	2.3 GHz
Cores	4
Caches	32 KiB L1D, 4 MiB L2
Processor vector capabilities	NEON 128-bit SIMD
OS	Linux 3.10-40
Compiler	GCC 5.3

ences applying Kommentator to the Roughness Estimation use case provided by Kolektor and JSI within T5.2. This use case is one of the main COPCAMS project demonstrators.

The Roughness Estimation program runs on a NVIDIA Tegra TK1 embedded development kit. The NVIDIA Tegra TK1 combines a performant embedded ARM processor and a modern GPU. See Table 8.2 for the system specification. The host processor is a quad-core ARM Cortex-A15, each core has a 128-bit NEON SIMD vector processing capabilities making it very capable. We do not report power usage of the NVIDIA Tegra TK1 as the board does not have any power measurements facilities, neither dedicated power rails for measuring or energy sensors built into the board. We could have measured the power consumption of the entire board, but the power consumption of the other components on the board would dominate a significant part of the power consumption. We focus on optimizing the costliest part of recalculating the regression tree model. The dataset of 300 images are loaded from either an internal eMMC, which in speed is comparable to an SD card. The entire recalculation takes around 110 seconds.

The RoughnessMeasurements application applies a box filter a size of 1 x 50 pixels on each grayscale image. This filter by far takes up the most significant part of the computation time and we therefore focus on optimizing it. A box filter is a linear filter used for blurring, it computes the average of all pixels in a rectangle around a pixel.

The original unmodified program used the OpenCV implementation of box filter. The OpenCV implementation uses the NVIDIA Performance Primitives (NPP). NVIDIA has released NPP as a collection of GPU accelerated processing functions. The box filter implementation in NPP only supports `NPP_BORDER_REPLICATE`, but OpenCV and RoughnessMeasurements require the `BORDER_REFLECT_101` setting. The first setting clones the outermost pixel, whereas the other reflects several pixels as a border. Before calling the NPP library call, OpenCV creates a copy of the image with the required reflected

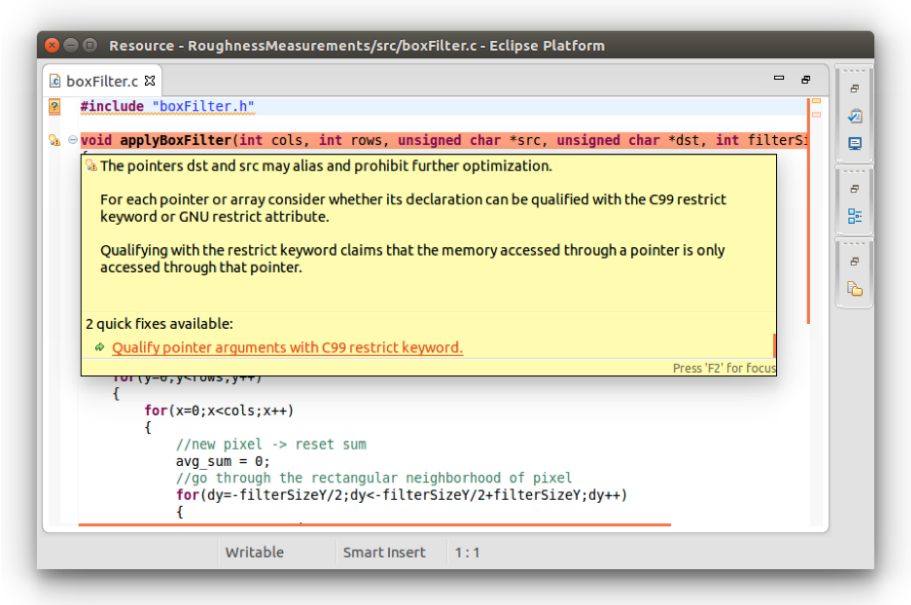


Figure 8.5: Example feedback from Kommentator, here we can apply the C99 `restrict` keyword.

borders to produce the same results.

The first step for applying Kommentator to the use case was to replace the OpenCV box filter, using Nvidia NPP GPU optimized libraries to perform the filter, with a sequential C version. This allows us first to modify the source and target the host CPU instead of the GPU. The sequential version only works on gray scale images, whereas the OpenCV box filter operates on multiple types of images.

Next, we analyzed the naïve implementation of a box filter using Kommentator and found that the arguments `src` and `dst` potentially could alias. This was blocking several optimizations, most importantly automatic parallelization. The feedback from Kommentator can be seen in Figure 8.5. As suggested we apply the C99 `restrict` keyword to the arguments using the suggested automatic refactoring. After applying the C99 `restrict` keyword GCC can automatic parallelize the box filter. Using four threads, we obtain an execution time of 17 seconds. This is slightly slower than using OpenCV optimized for GPU.

Finally, even though the box filter has been parallelized, several other important optimizations cannot be applied. For example, automatic vectorization is

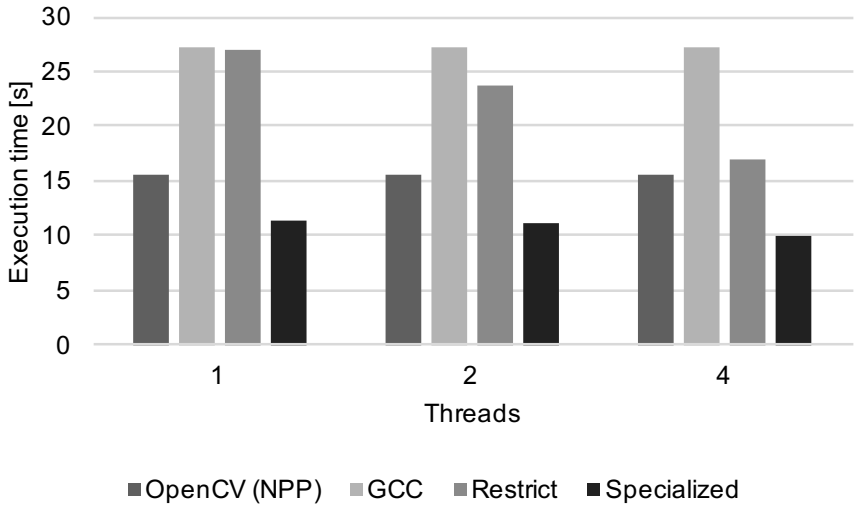


Figure 8.6: Box filter execution time on the Jetson TK1.

not applicable because of an unsupported reduction. Kommentator display a comment describing how the reduction was not supported. To mitigate this, we use fact that the applied filter is one dimensional. This allows us to remove one loop nest. This transformation allows GCC to unroll an inner loop and vectorize parts of the loop. This specialized version of the box filter leads to an execution time of 10 seconds using four threads. This is a speedup factor of 1.54 over the OpenCV GPU version. The speedup of the different versions can be seen in Figure 8.6.

8.3 Conclusions

Many applications rely on optimizing compilers for performance. Unfortunately, they are often not written in a way that allows compilers to optimize aggressively. Tools that help programmers write code in a way that the compilers can understand are important. However, tools that do this often have many false positives leading to programmers not using them, as they are simply not cost effective.

To this end we introduce how the feedback of multiple compilers can be used as a filtering mechanism, reducing the number of false positives by only showing

the most promising comments. Using a simple filtering we are able to achieve an 87.6% reduction in comments. This is a significant step in the direction of making compiler driven automatic refactorings cost effective. This address Research Question 2 by showing that it is possible to use the existing compiler optimizations reports for generating feedback to programmers. Furthermore, contributing to Research Question 4 we see on a case study how we are able to both automatically parallelize and automatically vectorize the use case. These changes lead to high performance yielding a speedup factor over the OpenCV based GPU implementation of 1.54.

Chapter 9

Improving Loop Dependence Analysis

In the previous chapters we have seen how compiler feedback can help programmers refactor their source code to allow more aggressive compiler optimizations. In this chapter we focus on improving the underlying dependence analysis as it is the heart automatic vectorization.

Loop dependence analysis is the basis for many compiler optimizations. We explore how we can improve the loop dependence analysis by using the information given by the programmer for parallelization of worksharing loops. We have implemented our worksharing aware dependence analysis and an automatic vectorization pass making use of it in GCC. We evaluate our work on a subset of the Rodinia benchmark suite, and improve the amount of correctly classified dependences with 133% and show an average speedup of 1.46 compared to GCC's current automatic vectorization pass. We thus seek to answer Research Question 7 by studying the accuracy of the existing loop dependence analysis in GCC. We seek to answer Research Question 8 and Research Question 9 in our evaluation of our improved dependence analysis.

Since 2005, the historically exponential clock frequency increase of processors has leveled off. Programmers can no longer expect new generations of processors to have a higher clock frequency than previous generations. This also means

that we need to look at other means than the clock frequency to improve single thread performance.

One of the most transparent ways of improving performance is to improve the compiler. The compiler can automatically analyze, optimize and transform our programs into versions that execute faster and use less energy. The basis of many important compiler optimizations is data dependence analysis.

There is a data dependence between statements if they access a shared memory location and at least one of the statements writes to the location. Data dependences impose a constraining execution ordering on the program, as it may not be legal to reorder dependent statements or execute them in parallel. Unfortunately, data dependence analysis is hard and often fails in practice. The analysis fails due to unanalyzable program constructs where static program analysis cannot give a precise enough approximation to determine whether an optimization is legal and profitable. In general, solving the dependence problem statically is intractable [KA02].

Many optimizations rely on data dependence analysis. For example, loop interchange, automatic parallelization and automatic vectorization [KA02]. Automatic parallelization is the process of automatically converting a sequential program into a multi-threaded version. Similarly, automatic vectorization is the automatic conversion from a sequential program to a vector implementation.

In this chapter, we focus on improving data dependence analysis and its impact on automatic vectorization. We supplement the data dependence analysis with OpenMP worksharing information to improve the analysis precision. This improves the amount of correctly classified dependences from 29% currently in GCC 6.1 to 69% with our analysis. Our improved data dependence analysis enables compilers to improve vectorization of many benchmarks, yielding speedups of up to 6.4.

9.0.1 Contributions

To summarize, we make the following contributions:

- An improved loop dependence analysis utilizing worksharing loop information.
- A prototype implementation in GCC 6.1 complementing the existing automatic vectorization optimization passes.

- Thorough evaluation using a set of benchmarks from the University of Virginia Rodinia 3.1 benchmark suite showing speedup factors of up to 6.4.

The chapter is laid out as follows. Automatic vectorization is introduced in Section 9.1, followed by an introduction of OpenMP in Section 9.2. Our approach and prototype is presented in Section 9.4. Experimental results are analyzed in Section 9.5. We discuss how we specifically are improving upon state-of-the-art in Section 9.6 and conclude the chapter in Section 9.7.

9.1 The Automatic Vectorization Problem

Most of the major modern processors have adopted single instruction multiple data, SIMD [Rus78], extensions to their instruction set. For example, Intel processors support the MMX, SSE and AVX SIMD extensions [Int15b], with AVX-512 extensions supported on their newest processors. The current trend is to have wider vector execution units, more vector registers and richer instruction extensions.

SIMD have high theoretical computational throughput but using the vector units also incurs several additional overheads. SIMD operations can only access data that is packed correctly in vector registers. To load and store vector registers, conventional SIMD platforms incorporate mechanisms for memory operations of both contiguous accesses, strided accesses, indirection accesses and conditional masked accesses. These different memory access types are illustrated in Figure 9.1.

Furthermore, SIMD extensions include shuffle operations for data in vector registers. These shuffle operations can be expensive if required for each iteration in a loop.

Several forms of automatic vectorization that complement each other are normally implemented in compilers: loop-level automatic vectorization and basic-block level automatic vectorization. Loop-level automatic vectorization analyzes a loop to determine all dependences within and across iterations and to determine whether the form of the loop is supported by the compiler and the underlying target hardware. Last, it also determines whether vectorization is profitable before generating the vector instructions. An example of a loop and its vectorized and sequential versions can be seen Figure 9.2. In each iteration of the vectorized loop, eight scalar iterations are executed. The potential speedup

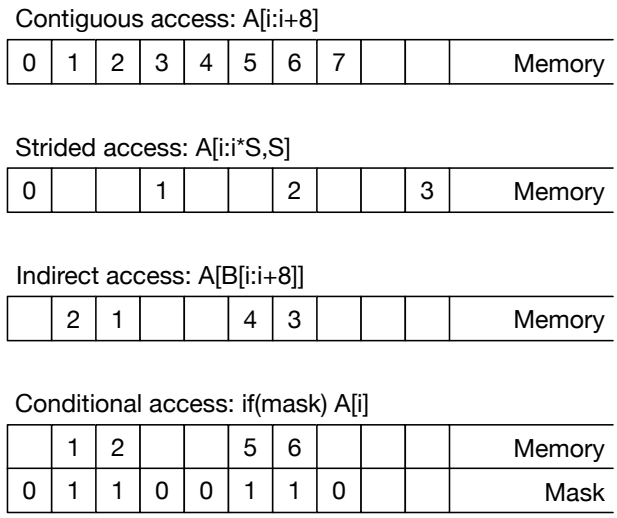


Figure 9.1: Memory access types for the SIMD units.

of this transformation is thus eight. In practice, the theoretical speedup is rarely achieved as vectorization adds overhead for many operations.

Analyzing the dependences of memory operations between loop iterations is hard when automatically vectorizing sequential loops. Depending on the type of the loop carried dependence, its direction and distance vector, it may be illegal to vectorize. For example, for a negative direction vector it is not possible to vectorize. For a zero direction vector it is legal to vectorize for self-dependences and lexically backward dependences. For a positive direction vector all types of dependences can be vectorized.

Other reasons preventing automatic vectorization are aliasing issues where several pointers might point to the same memory, mixed data types, unsupported vector operations or an unknown number of iterations.

9.2 OpenMP Application Programming Interface

OpenMP is an Application Programming Interface, API, for developing parallel applications. OpenMP consists of a set of compiler directives, also called pragmas, and library routines. Worksharing loops annotated with OpenMP

```
1  /* Sequential  */
2  for(int i=0; i<N; i++) {
3    A[i] = B[i] + C[i];
4  }
5
6  /* Vectorized */
7  for(int i=0; i<N;i+=8) {
8    vmovapd (%rdx,%rax,1),%ymm0
9    vaddpd (%rsi,%rax,1),%ymm0,%ymm0
10   vmovapd %ymm0,(%rdi,%rax,1)
11 }
```

Figure 9.2: Example of a sequential and vectorized version of the same loop.

directives such as a for loop, allows the compiler and runtime to assume the absence of dependences between loop iterations, i.e., each iteration of these loops can execute in parallel.

A sequential program specifies a total execution ordering describing the happens-before relationship. For a loop with memory operations, the order of these has to comply with the memory model. For an OpenMP worksharing loop, the parallel semantics define a partial execution where each iteration of the loop can be executed independently. If the loops actually contain loop-carried dependences, and each iteration is not allowed to execute in parallel, the program is non-confirming and a compliant implementation may exhibit unspecified behavior [Boa15].

9.3 Using OpenMP Information in Compiler Optimizations

The information in the directives is key to overcome the limitations of static analysis present in state-of-the-art compilers today when it comes to dependence analysis. The directives allow the compiler to assume no dependences for worksharing loops and execute the individual iteration in any order.

Compliant OpenMP program will execute correctly with our proposed analysis, but might change the behavior of incorrect OpenMP programs. The OpenMP specification defines non-compliant programs as having undefined behavior [Boa15].

The OpenMP specification does not specifically mention the legality or allowed use of OpenMP information in compiler optimizations. OpenMP information is used in all OpenMP compliant compilers to perform the required transformations when implementing the OpenMP directives. Further, many compilers perform some optimizations as part of their OpenMP lowering where they map from OpenMP to their intermediate representation.

Pop et al. describes how explicitly parallel programs are implemented using lowering from the explicitly parallel format to a thread based implementation without maintaining the original information in the directives during optimizations as for example done in GCC [PC10]. Further, they describe how optimizing on this lower level representation is an obstacle for optimization. For example, we could imagine that a compiler could vectorize a regular loop, but if turned into an OpenMP worksharing loop the compiler cannot automatically vectorize it anymore due to the representation. To this end, Pop et al. suggest maintaining the original directive information through the compilation, and lower the representation much later in the compiler. In this chapter we adopt a hybrid approach where we let the compiler perform its original lowering, but also save key OpenMP worksharing information for use later in the compiler.

Chatarasi et al. also utilize information from explicitly parallel OpenMP programs in their optimizer [CSS15]. They incorporate executions orderings from explicitly parallel programs and broaden the range of legal transformations for explicitly parallel programs in a polyhedral optimizer.

Furthermore, unless the `monotonic` specifier is used, individual chunks of iterations can be executed in any order. The OpenMP 4.5 specification makes this clear:

> "Note —The next release of the OpenMP specification will include the following statement:
>
> Otherwise, unless the monotonic modifier is specified, the effect will be as if the nonmonotonic modifier was specified."

Furthermore, if the program depends on any execution order of the chunks then the behavior of the program is unspecified.

The information in the directives is key to overcome the limitations of static analysis present in state-of-the-art compilers today when it comes to dependence analysis. The directives allow the compiler to assume no dependences for worksharing loops and execute the individual iteration in any order.

```
#pragma omp parallel for
for(int i=0; i<N; i++) {
  foo(A,B,i);
  B[i] = A[i] + 1;
}

void foo(A, B, i) {
  A[i] = B[i-1] * B[i-1] - 1;
}
```

Figure 9.3: Example of an erroneous non vectorizable loop with a possible dependence between loop iterations. Considering the OpenMP `#pragma` a compliant OpenMP implementation is free to assume that `foo` does not add dependences.

For an inner loop, such as in the example in Figure 9.3, the compiler cannot figure out that the entire loop can be vectorized unless it also analyzes the call site. There are many ways to handle such an analysis, but it is often impossible, for example if `foo` is defined in another compilation unit and if link time optimization is not enabled. The compiler has to assume that the function call to `foo` can add loop-carried dependences preventing vectorization. We also show how `foo` could be defined, in Figure 9.3, adding an anti-dependence for the read and write of `B`.

OpenMP 4.0 [Boa13] added support for programmer-guided vectorization with the `#pragma omp simd` construct. We believe that OpenMP 4.0 is a significant step forward, but is still limited in supported loop constructs and puts a bigger burden on the programmer.

9.4 Our Approach to Dependence Analysis and Automatic Vectorization

To improve precision of data analysis, we supplement the analysis with OpenMP worksharing information. A worksharing construct divides the execution of a code region, for example a loop, among a team of threads. Worksharing information allows the compiler and runtime to assume an execution ordering. The compiler is allowed to assume no loop carried dependences between the iterations of a loop.

The novelty in our approach is how we utilize the parallel worksharing information the programmer has specified for parallelization. Our analysis uses the existing worksharing constructs to improve the precision of dependence analysis and automatic vectorization.

9.4.1 Overview

Our dependence analysis and automatic vectorization transformation are implemented as a GCC optimization pass. The pass is placed right before the existing loop vectorizer, as we utilize a significant part of the infrastructure from the existing loop vectorization pass. We have implemented our work as a new pass. The supported loop forms are limited are in the existing GCC automatic vectorizer. We support Intel SSE4 and AVX SIMD extensions.

OpenMP defines confining worksharing loops as canonical loops [Boa13]. For these loops, the loop index induction variable "must not be modified during the execution". Furthermore, the loop nest must consist of structured blocks. A structured block is a series of statements with a single entry and a single exit or another OpenMP construct. The entry to the block cannot be due to a `setjmp`.

For an inner loop in the canonical loop form with an OpenMP worksharing annotation, our analysis works by determining the following:

1. Dependences on memory references
2. Vectorization factor
3. Loop-induction variables
4. Operations

9.4.2 Dependence Analysis

We have extended the existing loop-dependence analysis in GCC and made it OpenMP workshare aware.

The existing dependence analysis in GCC works in two steps. First, it goes over all pairs of memory references and initializes book keeping data structures for each. Afterwards, it calculates actual dependences. Each pair is initialized to either `chrec_known`, `chrec_dont_know` or `NULL_TREE`. A `chrec_known` means

it has been determined that no dependences exist between a pair of references, i.e. not aliased. A `chrec_dont_know` means that the analysis was not able to determine any useful dependences, i.e. may alias. Finally, `NULL_TREE` means that there exists a dependence between a pair, i.e. must alias. The dependence is represented as a distance vector and direction.

When initializing the dependence data structures, we supplement the analysis with the information from the OpenMP worksharing annotations. If a reference is only dependent on an outer worksharing loop, and not the inner loop, we can determine that the pair does not have any dependences. This is due to the OpenMP semantics where all iterations can be executed independently.

In a nested OpenMP worksharing loop, where the inner loops are not worksharing loops, the inner loops have a sequential total execution ordering semantics. This execution ordering is in general not vectorizable, unless an analysis can prove that we can change the order of operations without affecting the result of the computation.

The implementation and vectorizer prototype described in Figure 9.4 can also be used for non OpenMP loops. The prototype is based on the existing vectorizor [Nai04]. Compared to the existing GCC automatic vectorizers, we improve the data dependence analysis for OpenMP worksharing loops. We are able to use the partial execution ordering semantics for OpenMP outer loops and eliminate undecided dependences in inner loops.

9.4.3 Prototype Overview

The vectorization factor is the number of scalar iterations of the loop performed in a single vectorized iteration of the loop. We determine the vectorization factor as having the largest data length appearing in the loop as a multiple of the vector length. Thus, if the largest data type is a 32-bit integer and the supported vector length is AVX2 with 256 bits, the vectorization factor will be 8.

As loops are in the canonical form as described in OpenMP [Boa13], we assume in our analysis that the loop will have at least one induction variable. We determine the induction variable using a series of reaching definition steps. As part of the transformation, we make sure to increment or decrement the induction variable according to the vectorization factor.

We also iterate over all operations and determine for each whether the operation is supported by the target. Finally, when generating the vectorized loop, our

```
1  transform_loop(struct loop *loop)
2  {
3    FOR_EACH_STMT(loop, stmt)
4      transform_stmt(stmt);
5    transform_loop_iter(loop)
6  }
7
8  vect_analyze_loop (struct loop *loop)
9  {
10   loop_vec_info loopinfo;
11   if(!is_openmp_loop(loop)) return;
12   if(!analyze_VF(loopinfo)); return;
13   if(!analyze_data_refs(loopinfo)) return;
14   if(!analyze_data_ref_accesses(loopinfo)) return;
15   if(!analyze_operations(loopinfo)) return;
16   if(!analyze_profitability(loopinfo)) return;
17   transform_loop(loop);
18 }
```

Figure 9.4: Overview of our automatic vectorizer prototype.

optimization pass ensures the prologue and epilogue loops, as already emitted by OpenMP, cover the necessary loop bounds.

An overview of our prototype optimization pass can be seen in Figure 9.4. We start out by analyzing the loop nest. We determine whether it is an OpenMP loop and what information was in the `#pragma`. Then we determine the vectorization factor, dependence analysis, using an OpenMP aware version of GCC's dependence analysis. Finally, we make sure the required operations are possible and that it is actually profitable to vectorize the loop.

When GCC performs lowering of the OpenMP directives early in the compiler, it translates the directives into a thread based version using the GCC intermediate tree representation. In this lowering, we save the original semantic of the OpenMP directive and attach it to the loop nest for later use during compilation as shown in Figure 9.5.

```
1  void foo() {
2    #pragma omp parallel for
3    for(i=0; i<N; i++) {
4      for(j=0; j<N; j++) {
5        A[I[i]][j] = A[I[i]][j] +
6        B[I[i]][j];
7      }
8    }
9  }
```

(a) Original sequential program.

```
1  foo._omp_fn.0 (struct .omp_data_s & restrict .
       omp_data_i) {
2    int num_threads= __builtin_omp_get_num_threads();
3    int thread_num = __builtin_omp_get_thread_num();
4    #pragma omp parallel for
5    for(i=omp_lower(); i < omp_upper(); i++) {
6      for(j=0; j<N; j++) {
7        omp_data_i->A[I[i]][j] = omp_data_i->A[I[i]][j]
             +
8                    omp_data_i->B[I[i]][j];
9      }
10   }
11 }
12
13 void foo() {
14   .omp_data_o.1.I = I;
15   .omp_data_o.1.B = B;
16   .omp_data_o.1.A =
17   __builtin_GOMP_parallel (foo._omp_fn.0, &.
         omp_data_o.1, 0, 0);
18 }
```

(b) Lowered OpenMP program.

Figure 9.5: Expansion of OpenMP program early in the compiler, the highlighted pragma indicates that we store the pragma for later use in the compiler.

9.4.4 Control Flow

As was seen in Chapter 3, control flow was the biggest obstacle to automatic vectorization across the benchmarks.

Control-flow is a significant hurdle to vectorization of loops. Divergence in control flow between iterations pose a challenge when multiple iterations of a loop is executing concurrently. If-statements can be replaced with a sequence of predicated instructions, i.e. conditionally executed instructions, in a method called if-conversion.

There are many obstacles to if-conversion. It can introduce faulting traps changing the program behavior and have to maintain the precise exception semantics. Precise exceptions assert that the state of the processor is consistent before and after a faulting instructions. Traps can be caused by writes in a read only memory, accessing out-of-range memory, invalid pointers and division by zero. To prove correctness of if-conversion the compiler relies on alias analysis and dependence analysis.

Intel AVX2 SIMD extensions support masked loads and stores. If we have a program with only small control-flow divergences, we will generate mask predicates for the different paths. We use the built-in if-conversion optimization pass in GCC, improve its dependence analysis. The if-conversion pass assumes it is enough to calculate dependences only for variables defined inside the basic block of the loop.

9.4.5 Memory Operations

Memory operations are important for vectorized loops. Intel AVX2 SIMD extensions incorporate mechanisms for memory operations with both contiguous and strided accesses. We support either of these and use the existing GCC loop analysis to determine stride sizes of memory operations.

9.4.6 Profitability

Vectorization is not always profitable. We model the cost of the original scalar loops and the new vectorized loop.

Targets in GCC can implement cost functions for different types of statements;

scalar and vectorized. We query these and compare the values. We do not transform the loop if it is not profitable.

9.5 Experimental Evaluation

We evaluate our proposed loop dependence analysis, automatic vectorization approach and prototype implementation in GCC 6.1 to demonstrate its effectiveness. We evaluate it by compiling, running and analyzing a set of C and C++ benchmarks.

The main questions we seek to answer are:

- How well does dependence analysis in production compilers perform?
- Can our OpenMP aware loop dependence analysis, which utilizes worksharing information, be used to remove all dependences for worksharing loops and many dependences even for inner loops with a worksharing super loop?
- Will removing dependences lead to more loops being vectorized and more vector instructions being executed dynamically?
- What is the impact of our approach on performance and the total energy consumption?

We answer the first point by analyzing benchmark loop dependences with static analysis and comparing it to results obtained with dynamic analysis. We study the second point by comparing our OpenMP aware dependence analysis with the ideal dependences obtained with dynamic analysis. Last, we study the execution time of the benchmarks and evaluate the speedup obtained. To study the achieved performance, we analyze the cache performance and the dynamically executed instructions.

The benchmarks all make use of OpenMP worksharing. Some benchmarks make use of explicit vectorization using OpenMP. We use these explicitly vectorized benchmarks as a baseline to compare the performance of our automatic vectorization optimization.

9.5.1 Experimental Setup

We use benchmarks from the University of Virginia Rodinia 3.1 Benchmark Suite [Che+09]. The benchmark suite contains a collection of parallel programs for heterogeneous platforms with multi-core CPUs and GPUs. We evaluate our loop dependence analysis and vectorization optimization on the multi-core CPU benchmarks that use OpenMP worksharing.

We use seven benchmarks in our analysis and evaluation. They represent a wide range of applications in fields from fluid dynamics to grid traversal. The benchmarks show cases where our analysis can improve the results and cases where existing approaches already give good results.

CFD Solver is a fluid dynamics application that calculates compressible flow for a finite volume using three-dimensional Euler equations. The main computational kernel in the solver is an unstructured grid computation.

HotSpot is a physics simulation used to estimate processor temperature based on a floorplan and simulated power estimates. The main computation is a structured grid kernel.

LavaMD is a molecular dynamics application that calculates particle potential and relocation due to forces between particle. The main computation in LavaMD is an N-body simulation.

LU Decomposition is a linear algebra application that calculates the solutions of a set of linear equations.

Needleman-Wunsch is a bioinformatics application implementing a global optimization method for DNA sequence alignment. It represents a dynamic programming type of computation.

Pathfinder is a grid traversal application for finding the shortest path in a grid based on a dynamic programming computation.

SRAD is a diffusion method for ultrasonic and radar imaging applications. It performs image processing, image extraction and image compression.

The inputs used when executing each benchmark is displayed in Table 9.2. For most application we have used the standard inputs. For some we have scaled it up to have at least seconds of computational time.

We analyze the vectorized benchmarks on an Intel® Xeon® CPU E3-1276

Table 9.1: Experimental Machine Specification.

Processor	Intel® Xeon® CPU E3-1276 v3
Frequency	3.6 GHz
Cores	4
Caches	256 KiB L1D, 1 MiB L2 and 8 MiB L3
Memory	16GB DDR
SIMD	AVX2 256 bit
OS	Debian 8.4, Linux kernel 3.16.7
Compiler	GCC 6.1, ICC 16.0.1

Table 9.2: Input sizes for the benchmarks.

Benchmark	**Input**	**Seq. exe. time**
CFD Solver	193 K Elements	11.5 s
HotSpot	16384 x 16384 grid, 2 iterations	0.9 s
LavaMD	100^3 boxes	3.3 s
LU Decomposition	8000 x 8000 matrix	72.4 s
Needleman-Wunsch	8192 x 8192 matrix	9.6 s
PathFinder	1000000 x 1000 steps	9.4 s
SRAD	502 x 458 image, 1000 iterations	41.7 s

Haswell processor. The Haswell processor has 4 physical cores, capable of executing 8 threads with Intel Hyper-Threading. The processor supports AVX2 as well as previous SIMD extensions such as SSE4.2. The specification for the processor and its memory hierarchy is described in Table 9.1.

Our prototype optimization pass is implemented in GCC 6.1. All benchmarks are compiled with the `-O3` optimization level flag. For comparison we also compile and analyze the benchmarks with Intel's ICC 16.0.1. Here we use the `-O3` optimization level flag as well.

We measure loop dependences, execution time, cache performance, energy and dynamically executed instructions.

Accurate determination of the dependences for pairs of memory operations in loops is important for many loop optimizations. We study the result of GCC's analysis and compare it with the ideal dependences. The ideal dependences are obtained using dynamic analysis of all memory operations. This analysis is input dependent, but for the benchmarks and loops we are studying, we have manually determined that the dependences are not input dependent. The loop bounds do not affect the loop dependences. We have instrumented the benchmarks to

create traces of all memory operations. We study the traces offline to determine and classify the dependences. We use the same inputs as used in all the other experiments as described in Table 9.2.

Execution times are obtained by running the benchmarks 30 times using the native input sets as provided in Table 9.2. The reported execution times are the average means. All results are stable with negligible standard deviations of less than 0.1% and absolute 95% confidence intervals within ±0.1 seconds. The experimental machine is isolated and not connected directly to external networks to reduce noise.

We conduct cache measurements using performance counters present in the Intel processor. The counters are accessed through the Performance Application Programming Interface, PAPI [Bro+00]. We obtain numbers for level 1 data cache references and cache misses for loads and stores.

To measure energy, we use the Intel RAPL energy counters [Int15a] exposed through Linux `perf`. We report power consumption by the entire processor package including the cores and caches, and the processor core alone.

Finally, we have studied how many vector instructions are dynamically executed. We instrument the applications using Intel's Pin [Luk+05] with a custom Pin tool counting how many times each basic block is executed. From this we can derive counts for SIMD instructions and total executed instructions in the parallel region. We do not use performance counters as the Haswell processor does not include a suitable performance counter.

As obtaining cache numbers and measuring energy introduce a small overhead proportional to the sampling rate, we obtain these numbers in separate runs for each metric. Similarly, we obtain dynamically executed vector instructions in separate runs as the Pin based instrumentation adds a large overhead.

9.5.2 Analysis of Dependences

First, we study the result of loop dependence analysis. It is very important for many loop optimizations to determine the loop dependences between iterations. For automatic vectorization it is furthermore important to determine whether vectorization is legal when dependences exist.

We compare our OpenMP aware loop dependence analysis pass with the dependence analysis in GCC. We look at dependences for inner OpenMP loops

considered for vectorization. Dependences are classified according to GCC's classification.

Each pair of dependences between iterations is classified as either not vectorizable or vectorizable. Furthermore, these are divided into:

- Not vectorizable due to unsupported loop form.
- Unknown reason due to limitations in the analysis.
- Loop-carried dependences.

The vectorizable dependences are divided into no loop-carried dependences or loop-carried dependence but vectorizable. An unknown dependence means the analysis was not able to determine any useful dependences and there may exist a dependence. For loop-carried dependences we further classify whether the dependence can be vectorized, both with our vectorizer and the GCC loop vectorizer. The dependence analysis in GCC will return early for unsupported loop forms, e.g. loops containing function calls with unsupported side effects. For this reason, we have modified the dependence analysis to report the number of dependences for unsupported loop forms. However, we are not able to classify whether some of the dependences in these unsupported loop forms could actually be vectorized due to limitations in the analysis. Loop forms that are not valid cannot be vectorized and correctly classifying dependences in these loops would not improve vectorization.

The pairs of dependences, in inner loops of nested worksharing loops, can be seen in Figure 9.6 for each benchmark. The dependences classified in percentages can be seen in Figure 9.7. We present numbers for GCC's dependence analysis, our proposed worksharing aware dependence analysis (named improved) in the Figure and the ideal dependences. The ideal dependences have been obtained dynamically by analyzing all memory operations dynamically.

We define the precision as the fraction of correctly predicted vectorizable dependences. As dependence analysis by design is conservative both methods achieve a precision of one. This means that the analysis methods do not classify non vectorizable dependences as vectorizable. To describe how many of the true ideal vectorizable dependences that each method correctly classifies as vectorizable, we calculate the recall. We define recall as the amount of correctly predicted vectorizable dependences compared to the ideal. The original dependence analysis in GCC achieves a recall of 29% for all benchmarks, whereas our worksharing aware analysis achieves a recall of 69%. This is a big improvement without sacrificing the precision and thus correctness of the analysis.

Overall, if we study dependences that are vectorizable and those that are not, we go from 146 vectorizable dependences to 340. This is an increase of 133% more dependences that can be vectorized. For non-vectorizable dependences we go from 346 to 152 dependences, this is a decrease of 56%. We thus conclude that for OpenMP worksharing loops our dependence analysis is very efficient. Even when the worksharing annotation is not applied directly to the inner loop we are studying, it still provides a significantly improved precision of the analysis.

There are several reasons behind the results depending on the benchmarks and loops in the benchmarks:

PathFinder PathFinder contains one parallel loop dwarfing the execution time. The loop has OpenMP worksharing annotation specified directly and we can thus ignore dependences. For each pair of references, the compiler only has to determine whether the access patterns can be vectorized efficiently. This allows us to use the worksharing `#pragma omp parallel for` to enable automatic vectorization.

CFD Solver The CFD Solver application contains three inner loops with worksharing super loops. One loop in CFD can be unrolled completely. For the statements in this loop, the worksharing annotation can be applied directly. This aids the dependence analysis as OpenMP worksharing semantics can now be applied directly.

One of the two other loops in the benchmark has been turned into a memory copy. The last loop in the benchmark have a form that GCC does not support, we can thus not analyze the data dependences in the loop.

HotSpot HotSpot contains two inner loops inside a worksharing loop. Our dependence analysis can improve the amount of correctly classified data dependences for both loops as parts of these loops independent of the inner loop they are inside. Unfortunately, there are still many unknown dependences that neither our analysis nor the existing analysis can handle the dependences.

LavaMD LavaMD contains one worksharing loop with several nested loops. We analyze its inner loop. It contains several subscripted memory references. We are able to apply the outer loop worksharing semantics to several variables. The existing dependence analysis is then able to expand upon this and improve the amount of correctly classified vectorizable dependences significantly. A few

dependences are only related to the inner loop and have a complicated access pattern that is not vectorizable.

LU Decomposition The LU Decomposition benchmark contains two worksharing loop. Each worksharing loops contain several inner loops.

Out of the five inner loops, the existing dependence analysis in GCC can already handle three of them. GCC determines that they contain dependences, which are all vectorizable. With our improved analysis we can in many of the cases go further and convert vectorizable dependences into no loop carried dependences. In our case, this makes vector code generation straightforward.

Unfortunately, neither analysis methods can prove that all dependences in the remaining two loops are vectorizable.

Needleman-Wunsch Needleman-Wunsch contain two worksharing loops each with four loop nests inside, i.e., in total eight inner loop nests. Two of these cannot be handled by the existing analysis in GCC due to their access patterns. For the remaining loops, we apply the worksharing information from the outer loops to several pairs of dependences. With this information, GCC is able to prove that all the remaining pairs are vectorizable.

SRAD The SRAD benchmark contains two worksharing loops. Each worksharing loop contains one inner loop. GCC can handle many of the dependences between iterations of the inner loops, but cannot determine if the many arrays accessed are aliased. Due to the high number of aliased arrays GCC decides not dynamically test these. But, the programmer's assertion that these are worksharing loops prove to the analysis that these aliases are not possible.

9.5.3 Automatic Vectorization Results

We compile and run each benchmark with Intel ICC 16.0.1 [Int] and GCC 6.1. We compare the execution time of the following versions:

- Original automatic vectorization optimization in ICC.
- Original automatic vectorization optimization in GCC.

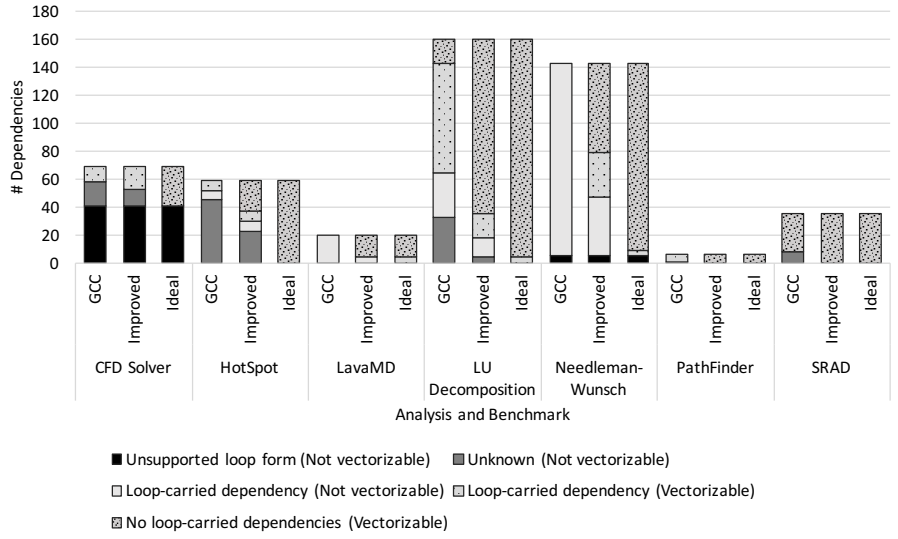


Figure 9.6: Number of pairs of dependences and their classification using the original GCC dependence analysis, our improved analysis and the ideal dependences.

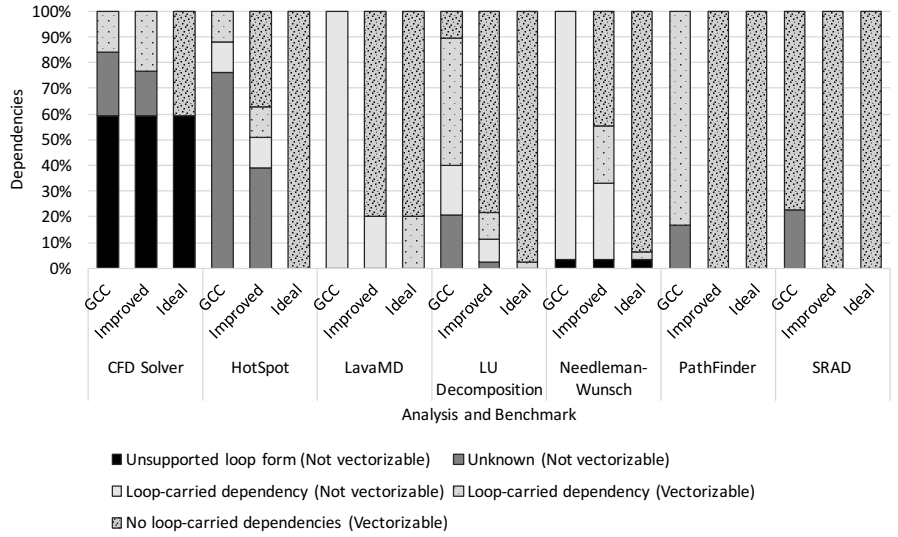


Figure 9.7: The percentages of the categories of dependences using the original GCC dependence analysis, our improved analysis and the ideal dependences.

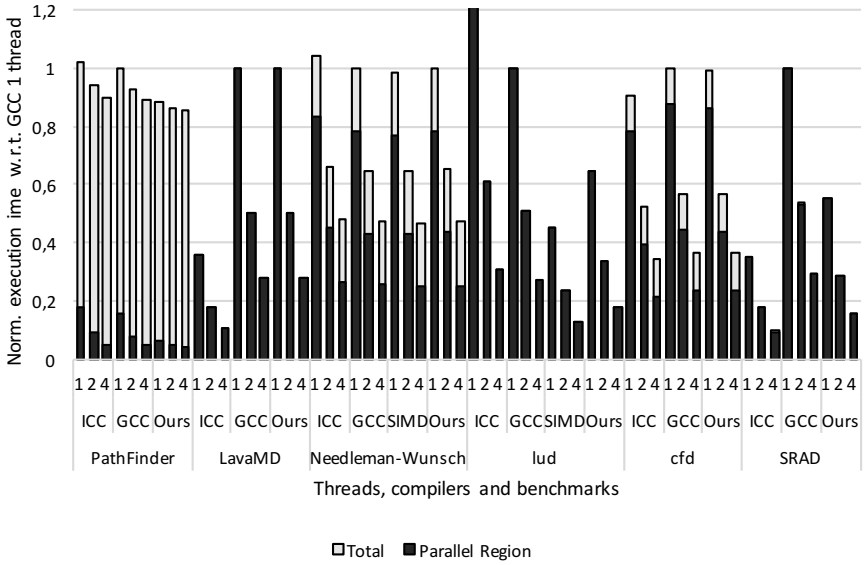


Figure 9.8: Normalized execution time of each benchmark with Intel ICC, GCC, hand-vectorized and our vectorization pass.

- Hand-vectorized using OpenMP SIMD pragmas for the benchmarks in Rodinia that have been hand-vectorized.
- Our automatic vectorization optimization utilizing the improved loop dependence analysis.

We execute each benchmark with one, two and four threads. The machine has four physical cores. We do not take advantage of simultaneous multi-threading. We measure the total execution time of the program and the execution time of just the parallel region. The normalized execution time results are presented in Figure 9.8. For some benchmarks ICC has the shortest execution time, for some it is the hand-vectorized version and for some it is our automatically vectorized version. The speedups of the respective parallel regions are presented in Figure 9.9. We achieve speedup using one thread of up to 1.54.

For PathFinder our approach is the fastest. The improved dependence analysis enabled vectorization as we also hoist out first iteration to the OpenMP prologue loop. Neither ICC nor GCC are able to vectorize the benchmark. The parallel region of the program only account for 5% to 17% of the execution time. Allocating data structures and random number generation takes up a significant

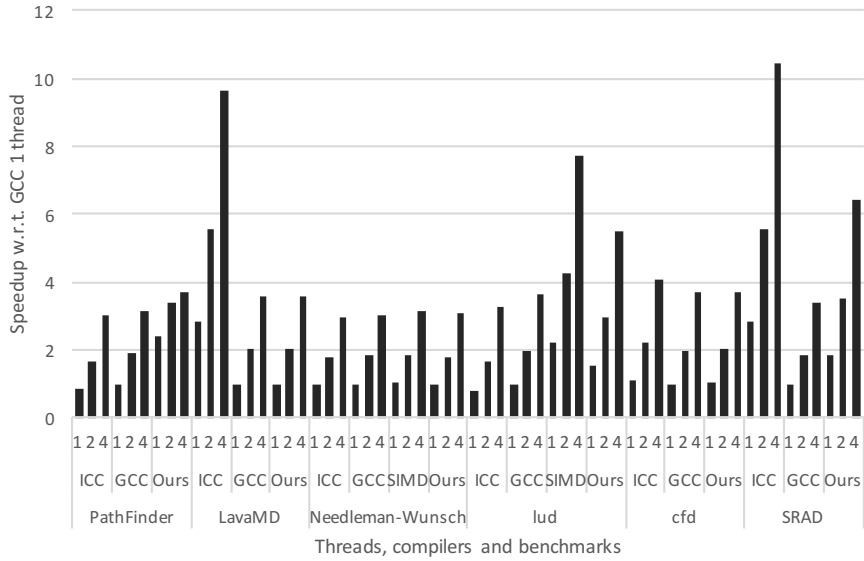


Figure 9.9: Speedup of each benchmark with Intel ICC, GCC, hand-vectorized and our vectorization pass.

part of the execution time. Using Intel ICC and one thread, PathFinder exhibits a slowdown factor of 0.85 compared to GCC. Our vectorized version achieve a speedup of 2.4 compared to GCC using one thread. When we scale the number of thread to two and four, the speedup becomes smaller.

For LavaMD we can vectorize the innermost loop of its worksharing loop except a function call to `exp`. However, doing so is not profitable as the function call adds too much overhead and we do therefore not vectorize. Intel ICC and GCC cannot fully vectorize the loop. We achieve the same performance with our approach as GCC, as we decide to not vectorize this loop. Intel achieve a significant speedup over GCC. This speedup is not due to vectorization. Even at the lower optimization level (`-O1`), ICC produces faster code. This indicates that ICC applied some basic optimizations more successfully than GCC. Studying the assembly source code indicates that ICC avoids some rematerialization compared to GCC.

We compare four versions of the Needleman-Wunsch benchmark as a hand-vectorized version of it exist. We observe how the four versions exhibits similar execution times even though the four versions are optimized quite differently. For the original GCC version the compiler can automatically vectorize several of

the loops in the benchmark. However, mostly inner loops moving data around are optimized. The hand-vectorized version of the Needleman-Wunsch benchmark has been annotated with `omp simd` pragmas on some of the inner loops in the benchmark. All the loops that are hand-vectorized are converted to unrolled memory copies as GCC can completely unroll the hand-vectorized loops. Our improved analysis can also vectorize the same loops as were hand-vectorized. Similar to the hand-vectorized version, these loops are later completely unrolled and converted into memory operations in the compiler. This indicates that some versions of the loops are simpler to analyze for the compiler and thus optimized more aggressively in later stages. The computational part of the benchmark is unfortunately not vectorizable as also determined in our analysis of the dependences earlier.

For LU Decomposition, we also compare four versions including the hand-vectorized version. For this benchmark, ICC generates executable code slower than GCC with a slowdown of 0.8 for one thread. GCC is able to vectorize three out of six of the inner loops in the benchmark. The hand-vectorized version vectorize two additional loops and achieves a speedup of 2.2 over GCC with one thread. Our improved analysis and optimization achieves a speedup of 1.54 over GCC using one thread. The difference between the hand-vectorized and our automatically vectorized version is due to one loop, where our analysis cannot prove that the dependences are vectorizable.

In the CFD application, ICC achieves a lower execution time resulting in speedup of 1.1 over GCC with one thread. GCC is already capable of vectorizing two out of five inner loops in the benchmarks. Out of the three remaining loops, GCC cannot analyze them because they contain many function calls and limitations in the dependence analysis. The loops that it can vectorize are later optimized to memory operation intrinsics. Our improved data dependence analysis can improve the recall, but does not prove that all of any of the additional loops are completely vectorizable making vectorization not profitable. Furthermore, the strides of this loop makes the required memory operations expensive.

For the SRAD benchmark, the improved dependences in if-conversion and automatic vectorization have enabled vectorization in GCC. The dependences were hindering both if-conversion and automatic vectorization to succeed. The vectorized version achieves a significant speedup over the non vectorized version of 1.8 using one thread, and 6.4 using four threads. A OpenMP SIMD version does not exist for this benchmark, and applying the SIMD pragma is not enough to vectorize it due to limitations in the if-conversion used when enabling OpenMP SIMD. Intel ICC achieves the best performance, with a speedup of 2.8 using one thread. Both versions vectorizes the unaligned accesses, but the for ICC might be due to other scalar optimizations or superior thread based OpenMP support.

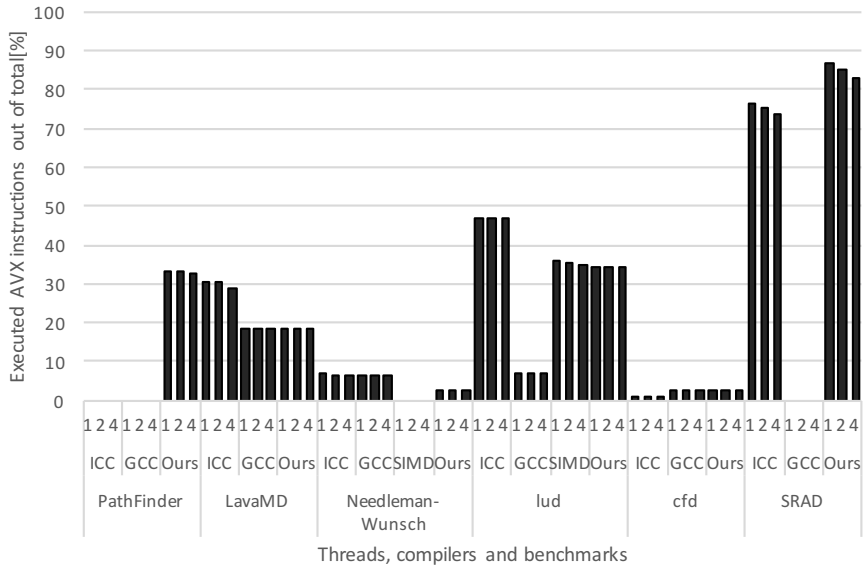


Figure 9.10: AVX2 instruction usage dynamically measured using Pin.

9.5.4 Instruction Mix

We study the dynamic instruction mix to answer the questions of how efficient vectorization is for the different benchmarks. Each benchmark has been instrumented with a custom Pin tool that counts basic blocks producing counts for each instruction. We quantify the amount of AVX2 instructions executed. We filtered the counts to remove the scalar AVX2 instructions. For example, scalar floating point operations can be implemented with AVX2 instructions but we are only interested in the SIMD instructions. These figures are presented in Figure 9.10.

We observe that for PathFinder, GCC and Intel ICC have almost zero percent SIMD instructions executed dynamically. For our vectorized version we have between 32% and 33% percent SIMD instructions. This is due to the improved dependence analysis that enabled vectorization of the benchmark as also indicated by the speedup achieved.

For LavaMD we observe dynamically executed SIMD instructions for all versions. However, neither ICC, GCC or our version decided to vectorize the inner loop. For all version the SIMD instructions are due to function calls to the

vectorized math libraries.

For Needleman-Wunsch, both ICC and GCC have some SIMD instructions. The SIMD instructions are due to calls to memory copy intrinsics and does thus not come from automatic vectorization. The only SIMD instruction ICC have generated is the `VMOVDQU` instruction. Therefore, the SIMD usage is due to an optimized move utilizing SIMD move instructions. In the hand vectorized version, GCC decided to completely unroll the SIMD loops and turned the loops into straight line memory copies. Similar, for our automatically vectorized version these loops are unrolled completely after the optimization pass. This optimization of unrolling is only applied to the hand-vectorized and our version by GCC.

ICC has the highest amount of SIMD instructions, above 45%, for LUD. The fastest version, the hand-vectorized, actually has fewer dynamically executed SIMD instructions than ICC. To understand why, we also look at the instruction mix for this benchmark. The SIMD instruction mix for LUD is presented in Figure 9.11. We study the outliers for the AVX2 SIMD instruction mix and observe the amount of `VINSERTPS` instructions in the ICC version. We also study the memory operations of the different version as presented in Figure 9.12. Here ICC and GCC have many 4-byte memory reads. For ICC, this is due to simulated gather memory read instructions for several loops. Instead of reading 16 bytes or 32 bytes with one SIMD instruction, ICC reads 4 bytes at a time into the SIMD registers. Exactly why ICC does this is not clear and it is not possible to determine this with optimization reports available in ICC.

We also observe how both the hand-vectorized of LU Decomposition and our implementation optimizing it have around 35% SIMD instructions. So even though the programmer by hand have vectorized more loops in the hand-vectorized version, the amount of executed SIMD instructions is within 2% between the hand-vectorized and our version.

All versions make use of AVX2 instructions for CFD. Large parts of the benchmark are vectorized. Unfortunately, we are not able to vectorize the costliest of the inner loops. For this reason, we do not see more SIMD instructions.

9.5.5 Cache Misses

We study the cache performance to analyze if vectorization affects the cache performance. We measure store and load misses for the first level data cache using hardware performance counters. The results are presented in Figure 9.13.

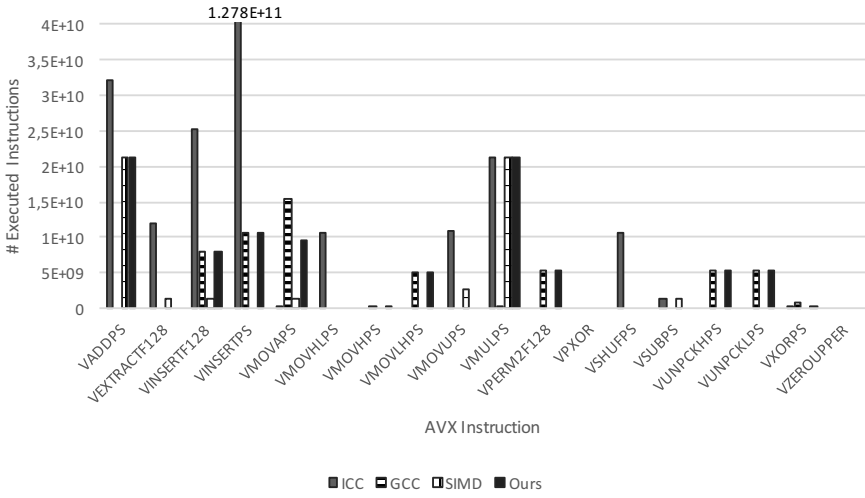


Figure 9.11: LUD AVX2 dynamically executed instructions mix.

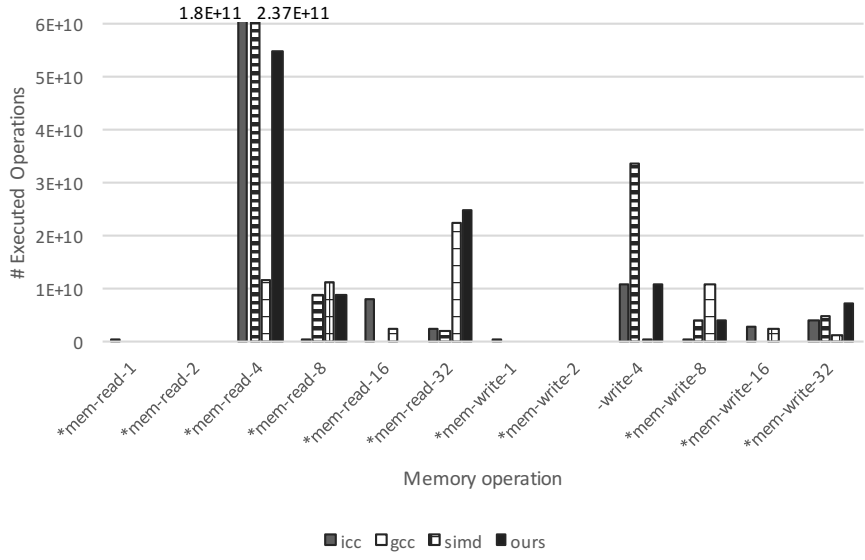


Figure 9.12: LUD AVX2 dynamically executed memory operations.

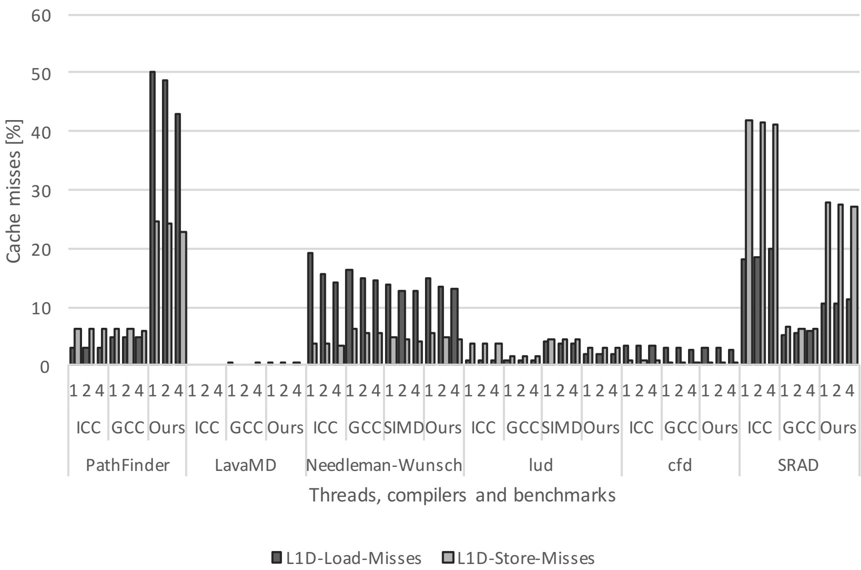


Figure 9.13: Cache miss rate for the first level data cache.

In general, we find that SIMD implementations can reduce the total number of memory accesses, but the number of cache misses remain approximately the same. Thus, vectorized implementations have a higher cache miss rate.

We notice this both for PathFinder and LU, where we get a significant increase in both load and store misses for our implementation compared to ICC and GCC.

9.5.6 Energy

We study the energy consumed to analyze if vectorization can reduce the energy consumed. Vector units consume a lot of energy but are also very performance efficient. With enough parallel efficiency, vector units can reduce the overall energy consumption.

We measure the energy consumption using the Intel RAPL interface. The measurements are presented in Figure 9.14. We observe that for benchmarks like PathFinder and LU decomposition where we have successfully vectorized significant parts of the benchmarks, we also reduce the energy consumption. For

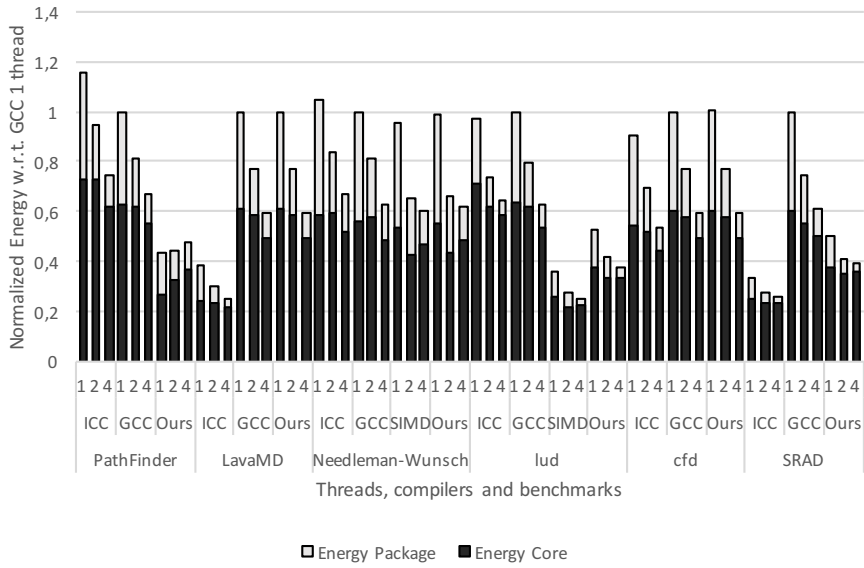


Figure 9.14: Energy measured using the Intel RAPL interface.

PathFinder, we decrease the energy consumption of the entire processor package with 57% over GCC using one thread. For LU Decomposition we decrease the power consumption of the entire package by 47%.

9.5.7 Summary

Overall, we find that we can improve the precision of the loop dependence analysis, improving the recall from 29% to 69%. Thus, improving the amount of correctly classified pairs of dependences. This improvement can enable several important optimizations to succeed. We have evaluated the effect on automatic vectorization and find that we can achieve speedups up to 6.4 using four threads. On average we achieve a speedup of 1.46. We also analyze why we do not always observe speedups when the improved dependence analysis has enabled automatic vectorization. Here we find multiple reason: from later optimization passes replacing vectorized code with intrinsics, loop unrolling and profitability.

We observe it is not necessarily a correlation between the amount of SIMD instructions executed dynamically and the execution time. For example, for LU Decomposition ICC generate an inefficient SIMD pattern when reading mem-

ory. As expected, when using SIMD instructions efficiently we see significant reductions in the power consumptions.

9.6 Discussion

OpenMP SIMD added support for manual vectorization using OpenMP directives [Boa13], being influenced amongst others by work from Caballero et al. [Cab+15] and Klemm et al. [Kle+12].

We compare in the experimental section with all benchmarks in the Rodinia Benchmark Suite where OpenMP SIMD has been implemented and do not achieve speedups higher than OpenMP SIMD. There are several reasons for this, but amongst other that the pragmas are only added to loops where the programmer has determined it will lead to a speedup. However, with our approach we lessen the burden on programmers as we automatically enable many of the optimizations otherwise requiring the programmer to reason about both thread-level parallelism and SIMD parallelism.

9.7 Conclusions

In this chapter, we have introduced an improved loop dependence analysis and evaluated it on a set of benchmarks. Using only static analysis, we increase the number of dependences that can be proven to be safe for vectorization. This improvement is due to applying the execution orderings made possible from parallel worksharing loops in automatic vectorization. This answer Research Question 6, Research Question 7 and Research Question 8 by comparing the ideal dependences with reported dependences. The original dependence analysis achieves a recall of 29% compared with our improved dependence analysis achieving a recall of 69% for the selected benchmarks.

Our results show that not all applications experience speedup. Optimization and speedups are limited by several factors: how vectorization friendly the benchmark is, the profitability of vectorization and remaining limitations in the analysis. On average, we achieve a speedup of 1.46 using one thread compared to the existing GCC automatically vectorized version. This answers Research Question 9 by showing the improvement made possible by the improved analysis.

As the data dependence analysis is the basis of many compiler optimization, we believe the idea behind our improved analysis could be applied to more loop

optimizations. Prime candidates are several loop optimizations such as loop interchange, loop fusion, loop skewing and other important loop optimizations

Chapter 10

Prescriptive Parallel Debugging

In the previous chapters we have seen how the compiler often refrains from optimizing programs as determining whether a transformation is legal is hard and it is often not possible to map applications efficiently to modern processors. We see a similar challenge in debugging, where a gap exist between how the programmer could most efficiently debug an issue and the existing debugging methodologies.

Optimized compilation and efficient optimized debugging is a critical step in the development of any parallel program. However, the traditional interactive debugging model, where users manually step through code and inspect their application, does not scale well even for current supercomputers due its centralized nature. While lightweight debugging models, which have been proposed as an alternative, scale well, they can currently only debug a subset of bug classes. We therefore propose a new model, which we call prescriptive debugging, to fill this gap between these two approaches.

This user-guided model allows programmers to express and test their debugging intuition in a way that helps to reduce the error space. Users are provided with the ability to define their own debugging probes, which are monitored by the system at runtime. The traditional, interactive debugging model, whereby

users manually step through and inspect their application, does not scale well even for current supercomputers. While lightweight debugging models scale well, they can currently only debug a subset of bug classes. The prescriptive debugging model fills the gap between these two approaches with a novel user-guided approach.

Based on this debugging model we introduce a prototype implementation embodying this model, the DySectAPI, allowing programmers to construct probe trees for automatic, event-driven debugging at scale. In this chapter we introduce the concepts behind DySectAPI and, using both experimental results and analytical modeling, we show that the DySectAPI implementation can run with a low overhead on current systems. We achieve a logarithmic scaling of the prototype and show predictions that even for a large system the overhead of the prescriptive debugging model will be small.

10.1 Motivation

Debugging is an important capability for large-scale simulations, but little has changed in how we debug applications. At the same time, high-fidelity simulations continue to drive strong demand for extremely large-scale machines while pushing application complexity to extremes. As a net result of this trend, machines with over a million cores are not uncommon today [Law]; [Oak]; and further, mission-critical applications often comprise a few million lines of code, coupling many scientific packages and libraries written in a wide range of programming paradigms and languages, e.g., C, C++, FORTRAN and Python. This sheer scale combined with application complexity has made debugging one of the most arduous tasks in code-development for high performance computing, HPC.

This situation will become even more challenging in the future [D+11]. Due to power and energy concerns, performance gains on HPC systems no longer come from increased single-thread performance, but rather, from increased core counts and the use of accelerators or co-processors. This trend increasingly requires programmers to rely on hybrid programming models, such as MPI + OpenMP or MPI + CUDA, to realize the full hardware potential, but this comes at significantly added coding complexity and to many unintended side effects between the various models. In short, programming complexity will rise, and with that so will the likelihood that bugs, particularly with respect to parallelism, will be introduced into codes; the current interactive (per thread/process) debugging techniques are not sufficient in helping programmers overcome these challenges. Without effective and scalable debugging models, and the tools that

embody these models, the cost of debugging will sharply increase due to the lost productivity of programmers and wasted compute cycles.

Unfortunately, while programming models designed to improve programmers' productivity are actively studied and proposed, there has been a lack of studies to understand the most capable debugging models with the same goal.

Traditional parallel debugging [Rog14]; [All14]; [SS14] is a several-decade-old model and does not scale even to today's core counts. Most importantly, manually stepping through source lines and inspecting the application state overwhelms the programmers with too much information. As a response to this problem, the lightweight debugging model [Arn+07]; [Lee+08]; [Ahn+09]; [LK32] arose, where information is sacrificed for scalability, but has the opposite problem – it scales well by design, but for many classes of errors, does not provide enough information. Further work has focused on fully automatic or semi-automatic debugging that helps automate the detection of suspicious behavior, e.g., through relative debugging [Abr+96] or automated anomaly detection [GQP07], but these approaches also limit themselves to particular classes of bugs and do not provide the generality of interactive debuggers.

To fill this gap, we propose a novel scalable debugging model called prescriptive parallel debugging that helps users automate the tedious parts of interactive debugging, while keeping its generality, and present our implementation that embodies this model: the *Dynamic Scalable Event-Chain Tracing API* or *DySectAPI*. This model offers a highly-scalable debugging paradigm by using an *engine* that *expresses a programmer's debugging intuition* to *automatically and progressively reduces the error search space* across both the task and source-code dimensions.

Our approach allows programmers to install debugging probes into a parallel application. These probes can gather data under user-specified conditions, in the form of either debugging procedures (e.g., trigger an action when a certain set of breakpoints is hit) or code behaviors (e.g., trigger an action when the code hangs). Probes are linked into a probe tree, automatically driving the prescribed debugging actions to reduce the error space. When the specified conditions are met, it presents highly condensed debug information to the programmer.

We demonstrate the scalability of our model by empirically evaluating the performance overheads of various probe-tree topologies of the DySectAPI and also by deriving a performance model. The evaluation suggests that using a probe tree that prunes out processes that are not of interest scales better than a flat topology analogous to the traditional debugging model. Further, we present results from a case study that show the effectiveness of our model: we used a DySectAPI probe tree to effectively debug a previously undiagnosed, real-world

bug in a scientific application, which manifested itself only when scaled to 3,456 MPI processes.

10.1.1 Contributions

To summarize, we make the following contributions:

- A novel prescription-based debugging model striking a balance between scalability and capability;
- The DySectAPI, an implementation embodying this model;
- Empirical experiments and an analytic model of overheads of the DySectAPI;
- A case study on a real world application showing its effectiveness.

10.2 Models in Parallel Debugging

As described in Chapter 2 the current state of the art in parallel debugging is focused on four main models: traditional; lightweight; semi-automatic; and automatic. In this section, we discuss the pros and cons of these models and the implications of the architectural and application trends in HPC.

In *the traditional debugging model* users insert breakpoints, step through their code and examine variable values. Example tools include GDB [SS14], TotalView [Rog14] and DDT [All14]. Traditional debuggers have cited operation on a large set of processes, for example DDT touts its ability to step and display 700,000 processes in 1/10 of a second [All14]. However, this time can increase with larger and more complicated applications and a typical debug session requires many step operations through the code. Typical debug sessions also involve performing more complex and expensive debug operations, such as examining and aggregating variable values.

But its fundamental scheme of having to enable *all idioms* for interactive use and the central point of control, i.e., the user, clearly limits its scalability.

The lightweight model has been created to address both the tool machinery and the human cognition scalability challenges. These tools aggregate debug information across a parallel application, thus reducing the amount of detail

presented to the user and making it possible to scale to existing and future systems. One example, the Stack Trace Analysis Tool (STAT), aggregates stack traces across the parallel application and groups processes that exhibit similar behavior into equivalence classes [Arn+07]; [Lee+07]; [Lee+08]; [Ahn+09]. This grouping often identifies a small set of outlier processes that led to erroneous execution and the stack traces can effectively pinpoint the precise location of the bug. In cases where more in-depth debugging is required, one process can be selected from each equivalence class, forming a small subset of processes that represents the full application's behavior. The user can then attach a traditional debugger to this manageable subset for root cause analysis.

Other approaches are the *automatic model* and the *semi-automatic model*, which include relative debugging. For example, Dinh. et. al. introduces scalable relative debugging [DAJ14], which requires running two versions simultaneously, one working and one failing, thus solving a special case and may not scale to extreme scales.

Similarly, DMTracker [GQP07] and FlowChecker [Che+10] solves special case automatic inference of root cause. Another approach to automatic debugging is with AutomaDeD [Bro+10] that probabilistically identify abnormal MPI tasks and find the least progressed task in many cases. AutomaDeD can efficiently solve many bugs at scale.

Other tracing techniques include systemtap, which has been used as a Linux trace/probe tool [Ibm+05]. systemtap has been extended with user-space probe capabilities using Dyninst [BH00]. The main difference between traditional tracing tools and DySectAPI is the dynamic nature of DySectAPI's probe-tree model and the fact that DySectAPI is targeted at massively-parallel systems. [EIg+05] [Jac+09]

10.3 In Search for Sweet Spots

At Lawrence Livermore National Laboratory (LLNL), one of the largest supercomputing centers, we have provided many tools embodying the debugging models described in Section 10.2. In recent years, however, it became apparent that a significant gap exists in models of parallel debugging on large HPC environments. The traditional model offers a general-purpose environment, but does not scale well. Other debugging models scale better, but can target only specific classes of errors, making them special purpose.

For example, STAT can be used to efficiently debug hangs at large scale, while

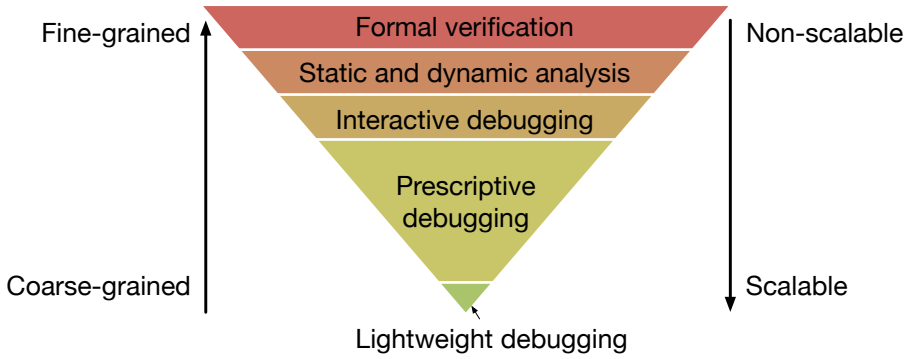


Figure 10.1: Debugging models.

doing so with a traditional debugger can be slow and complicated. STAT groups processes exhibiting similar behavior and can quickly expose why the hang arises, oftentimes due to a small set of outlier processes. With a traditional debugger, digesting the information from the many processes can overwhelm the programmer. On the other hand, a simple bug in the program logic that is exhibited even at small scale can be simpler to debug with a traditional debugger. STAT is a special purpose tool and can only find a bug if it visible by analyzing the stack traces of the application.

Therefore, we find significant needs for a new model that can be extremely scalable, yet capable enough, to serve general-purpose debugging. Figure 10.1 illustrates this notion.

To explore the trade-off space, we first identify several key principles that should drive the construction of a new scalable, general-purpose debugging model.

The new model must:

1. Be an engine to test the hypothesis behind a programmer's debugging intuition;
2. Enable users to express their intuition easily in terms of progressive reduction of error space;
3. Guide run-time debug actions with minimal interaction with the user;
4. Present condensed information after the error space is reduced.

The first principle states that any general-purpose debugging model must assist

users with mechanisms well-suited to test the programmer's debugging intuition, otherwise, the model is limited to be special purpose. The second principle dictates how to capture this intuition. Ultimately, the goal of any parallel debugger is to narrow down the root cause of a problem in several dimensions. One is to determine the particular process(es) or thread(s) that exhibit erroneous behavior. A second is to pinpoint the precise source-code location of the bug. Yet another dimension can be to isolate any contaminated data that led to the error. The new model must enable users to encode the notion of progressive reduction on the potentially huge error search space along all dimensions.

The third principle argues for the batch processing of debugging actions whereby the debugging expressions given by the programmer are evaluated in batches [Fel10]. This is necessary because the sheer volume of debug data may slow down a traditional model to the point where the tool would be intolerable to use. Even if a debugger could process that data with interactive latencies, it would be challenging to present that much data in a form that is scalable to the user. Computers, on the other hand, are much better suited for the task of sifting through massive amounts of data. Along with the third principle, the fourth one addresses the limitation in a programmer's information processing ability. Any information provided to the users must not be too much for a human to digest.

10.4 A New Model: Prescriptive Debugging

The traditional debugging paradigm has survived because it provides the rudimentary operations that a user needs to effectively reduce the error search space. In a typical debug session, a user first sets a breakpoint at a particular code location. Once that breakpoint is triggered, the user will evaluate the state of the application and subsequently set another breakpoint, perhaps on a subset of processes that satisfy certain conditions. This process is then repeated until the bug is isolated.

Our new prescriptive debugging model aims to capture the flexibility and generality of this interactive process, but allow users to codify individual steps and sequences in the form of debug probes that can then be executed without the need for individual interactions between debugger and user. Similar to Aspect-Oriented Programming, AOP [Kic+97], the prescriptive debugging model addresses the separation of concerns. AOP breaks down program logic into distinct concerns, where one concern could be debugging. AOP does not address scalability or debugging in general and does not satisfy the four guiding principles from section 10.3. Essentially, the prescriptive debugging model pro-

vides the means for a user to codify their debugging intuition into prescribed debug sessions. The application can then be submitted into the system's batch queue to be run under that debug session. At runtime, the debugger follows the user's intuition by executing the debug probes and, at the end, summary information is gathered that can be examined by the user during the execution or at their convenience after the job has completed.

While we argue for batch debugging instead of traditional interactive debugging, the end goal of our proposed model remains the same. Ultimately, the goal of any parallel debugger is to narrow down the root cause of a problem in several dimensions. One is to determine the particular process(es) or thread(s) that exhibit erroneous behavior. A second is pinpoint the precise source-code location of the bug. A potential third dimension is isolate any contaminated data that led to the error.

Our prescriptive parallel debugging model is built upon the notion of probes that can be linked together into a probe tree. A probe itself is composed of a domain, events, conditions, and actions as defined below.

$$Probe = \langle Domain, Events, Conditions, \{Actions\}, \{Probes\} \rangle$$

The *domain* is the set of processes to install a probe into. It also includes a synchronization operation that determines how long the probe should wait for processes in the domain before proceeding. More precisely, after the first process triggers a probe, the remaining processes have until some specified timeout to participate.

We define an *event* as an occurrence of interest. Events borrowed from traditional debuggers include breakpoints, which specify a code location (when reached, the debugger will stop the target process) and data watchpoints, which monitor particular variables, memory locations or registers. An event can also be a user-defined timeout that instructs a probe to be triggered after some elapsed amount of time. Events can also capture asynchronous occurrences such as a program crash, a signal being raised or a system-level event such as memory exhaustion.

These events allow programmers to express their debugging in terms of a set of procedures and in terms of code behaviors—e.g., on detecting a hang or slowness.

Further, individual events can also be composed together to enable advanced fine-grained event selection.

When an event occurs, its associated *condition* is evaluated. The condition is an expression that can be evaluated either locally on each backend or globally across the domain. A local condition may, for instance, check if a variable equals a particular value. A global condition can evaluate an aggregated value, such as minimum, maximum or average, across the entire domain. Conditions can also be composed to specify multiple variables of interest or to combine local and global evaluations.

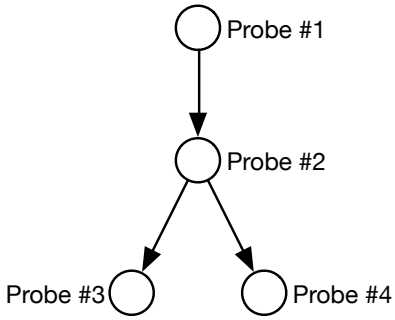
If the condition is satisfied, the probe is said to be triggered, and the specified *actions* are executed. Probe actions can be formulated by the user as an aggregation or a reduction, for example, aggregated messages, merged stack traces or the minimum and maximum of a variable.

A probe can optionally include a set of child probes, which is enabled upon the satisfaction of the parent probe's condition. In this manner, a user can create a probe tree. A probe tree naturally matches the control-flow traversal that is typical of an interactive session with a traditional debugger. This can effectively narrow down the search space across the source-code dimension.

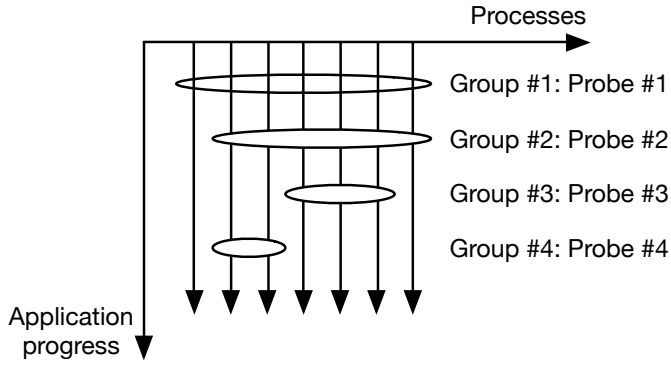
An example of a probe tree is shown in Figure 10.2a and the corresponding search-space reduction is shown in Figure 10.2b. As the application progresses, the probe tree effectively narrows down the search across the process space. As child probes are only installed in processes that satisfy the condition, processes that are not of interest are implicitly filtered out. Filtering not only helps narrow down the debug search space, but also reduces the number of subsequent probe installations, the amount of tool communication and the volume of data produced. These qualities are paramount to the scalability of the prescriptive parallel debugging model, both for the user's comprehension and for the tool's operation.

The process-space reduction has an additional benefit to debugging capabilities. While operations on the full application or a large set of processes should be lightweight in order to scale, operations on a small subset can be more complex. Thus, a well-formed probe tree would start with high-level summary information, such as a merged stack trace or a message displaying aggregated values, and get progressively complex, perhaps even gathering individual variable values across a subset of tasks.

A debugging session is then defined as a set of probe trees. Generic debug sessions can be created for common errors such as hangs, segmentation violations or other crashes. Furthermore, programmers can create application-specific de-



(a) Probe tree.



(b) Process search space reduction using a probe tree.

Figure 10.2: Example probe tree and corresponding processes being filtered out during the search space reduction.

bugging sessions for their users to employ when an error occurs. With inside knowledge, the programmer can write debug sessions that track known invariants for deviation, monitor control flow for abnormal behavior or specify conditions under which an application is considered to be hung. The aggregated debug log messages or merged stack traces can then be analyzed by the user or the programmer to aid in identifying the root cause.

10.5 DySectAPI: The Dynamic Scalable Event Tracing API

Based on the guiding principles from Section 10.3 and following the prescriptive debugging model introduced above, we developed the Dynamic Scalable Event-Chain Tracing API, DySectAPI. DySectAPI allows programmers to express and test their debugging intuition, executes with minimal user interaction and only presents condensed information once the error space has been reduced.

The programmer specifies a debugging session prior to the execution, based on their debugging intuition. They do so in a session file using our API for expressing probes and probe trees.

The workflow of DySectAPI has six steps:

1. Prior to execution, the programmer encodes a debugging session, which may contain several probe trees.
2. The session is compiled into a shared library.
3. The target application is launched by the DySectAPI runtime and the probe trees are installed in the specified domains.
4. The application executes.
5. When probes are triggered, the programmer-specified conditions and actions are performed.
6. The condensed diagnostic output is presented to the programmer.

Figure 10.5 illustrates this workflow.

```
Probe* p1= new Probe(
  Code::location("instrumentationHead"),
  Domain::world(10),
  Act::trace("probe 1: Location is instrumentationHead
     ()"));

Probe* p2 = new Probe(
  Code::location("instrumentPoint2"),
  Domain::world(10),
  Act::trace("probe 2: Location is
     instrumentationPoint2()"));

ProbeTree::addRoot(p1);
p1->link(p2);
```

Figure 10.3: Example probe tree debugging session.

10.5.1 Expressing Debugging Intuition

DySectAPI allows programmers to express their debugging intuition using probes and probe trees. Probes are represented as C++ objects, which can be linked into a probe tree. A debugging session snippet can be seen in Figure 10.3. This snippet is compiled into a shared library and launched together with the target binary under the tool's control. The two probes are triggered at the source code location `instrumentationHead` and `instrumentPoint2` in the target example program in Figure 10.4. The first probe will be installed in all target processes. When triggered it will send a message through the network with the number of triggered processes. This will result in a message being printed by the frontend. In addition, when the first probe has been triggered, the second probe is enabled in the processes that triggered the first probe.

The probe-tree debugging primitives allow programmers to specify their debugging intuition and to test their debugging hypothesis. The supported capabilities are:

Events can be either a code-centric classical breakpoint or an asynchronous event such as a signal. The latter includes a crash, an exit or a specific signal number.

```
void instrumentPoint2() {
  static volatile int _count = 0;
  _count++;
}

void instrumentHead() {
  static volatile int _count = 0;
  instrumentPoint2();
  _count++;
}

int main() {}
  MPI_Init(&argc, &argv);
  for(int i = 0; i < N; i++) {
    instrumentHead();
    MPI_Barrier(MPI_COMM_WORLD);
  }
  MPI_Finalize();
}
```

Figure 10.4: Example parallel application.

Conditions are evaluations based on data such as variables, e.g. `x > 0 && y < 0`.

Domain specifies in which processes to install the probe. This can be all processes or in a subset based on their MPI rank. An optional timeout can also specify the maximum time to wait for other participants before aggregating. The default is to wait infinitely long or until all processes have been triggered.

Actions can be formulated as aggregations or reductions, including messages that can aggregate the `min`, `max` and `desc`([min,max]) of variables, and the `function` or source-code `location` where triggered. There are no restrictions with respect to encapsulation and local variables. Further, the system can produce merged stack-traces both in text and graphical form.

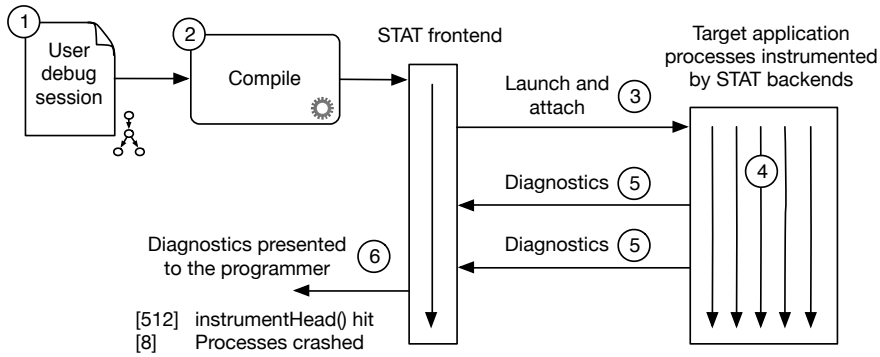


Figure 10.5: DySectAPI workflow: the programmer's debugging session is compiled and then launched. Probe trees are installed into the application and whenever probes are triggered, diagnostics are propagated up to the programmer.

10.5.2 Infrastructure

One important factor in DySectAPI is its scalability, which we achieve by basing communication on MRNet [RAM03], an efficient tree-based overlay network, TBON, for scalable tool communication and data processing. This system allows us to not only execute communication hierarchically following a tree structure, but also to embed processing operations into the tree, avoiding a central processing bottleneck at the tool frontend. In particular, our aggregation process has two steps:

1. Local data is aggregated on each backend. A backend is attached to a set of processes. When one process triggers a probe, other processes triggered within a timeout will form one packet to be sent through the MRNet tree network.
2. Packages from multiple backends are aggregated. Each MRNet node is also setup to wait for a specified timeout before forwarding packets through the network.

This process is illustrated in Figure 10.6. A set of events happens on each backend and is then efficiently aggregated using the MRNet network. DySectAPI uses Dyninst [BH00] to control and debug application processes. Dyninst is an API for binary analysis, binary instrumentation and process control. Dyninst allows us to debug unmodified application processes.

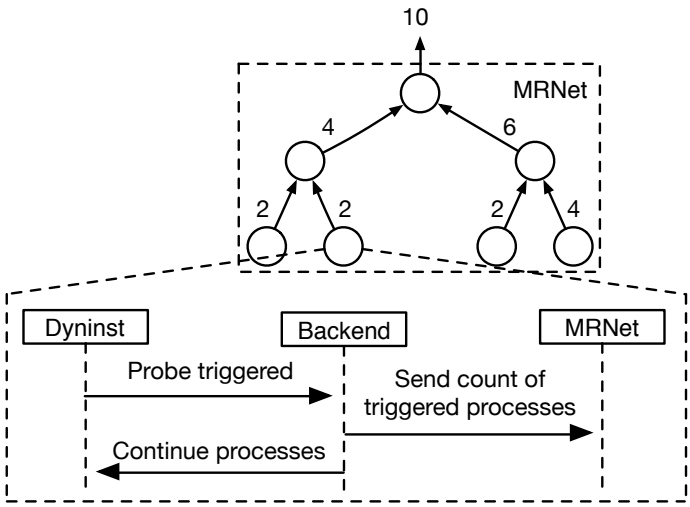


Figure 10.6: Communication architecture. The example probe counts the number of triggered processes and aggregates the information efficiently using the MRNet communication tree.

10.6 Evaluation

We evaluate the DySectAPI implementation to demonstrate the scalability and effectiveness of the proposed prescriptive debugging model. However, evaluating all aspects of a debugger is hard. It is possible to measure the performance of the debugging primitives in a traditional interactive debugger, but those numbers would not account for human interaction, which would require a larger psychological study. Similarly, it is hard to quantify the usability of a debugger, given the complexity of debugging.

We therefore want to show that the prescriptive parallel debugging model has the scaling characteristics of the lightweight debugging model and a sufficient set of capabilities from the traditional, interactive debugger to capture a wide range of parallel bugs. The main questions we seek to answer are:

- What is the scalability of the prescriptive debugging model?
- Can we use the capabilities of the prescriptive debugging model to debug real parallel bugs only manifesting themselves at scale?

We answer the first question by focusing on how the underlying implementation scales on large parallel machines using a performance study combined with modeling that predicts the scalability beyond current machine resources. This is valuable to determine if our debugging model is going to scale on large systems. As a baseline for the scalability of the traditional interactive parallel debugger we ignore the human factor and use a batch debugging session that reflects the operations in a traditional interactive session.

Second, we wish to demonstrate the prescriptive debugging model validity and usefulness. We do so by focusing on the usability of the debugging model with a use-case example. In this way we show how the prescriptive debugging model can be applied to an undiagnosed real-world bug that emerges at large scale and outline the steps involved.

10.6.1 Experimental Setup

All experiments were conducted on the Cab Linux cluster located at the Lawrence Livermore National Laboratory. This cluster consists of 1,296 nodes, each with 2 Intel Xeon E5-2670 processors for a total of 20,736 cores and 41.5 TB memory. Each node has a total of 16 cores. DySectAPI has been built on top of MRNet 4.1 and Dyninst 8.2.

10.6.2 Analytical Performance modeling

We derive an analytical performance model for an arbitrary probe tree to both reason about the underlying scalability and to predict scalability beyond current machine resources.

In each probe, we include a pruning factor, which is the fraction of processes that a probe filters out. A *ratio* in a probe of 0.5 means that 50% of the processes are filtered out by that probe. Figure 10.7 illustrates this notion.

Each compute node runs several application processes and one backend daemon that is responsible for debugging all processes on that node. We refer to the ancestor of a probe, in the probe tree as $ancs(probe)$. The $max(installs_{backend}(probe))$ is the maximum number of installations on any backend for that probe.

We first consider the number of probe installations, $installs(probe)$, for a single probe on a single backend. The root probe is installed into all specified processes,

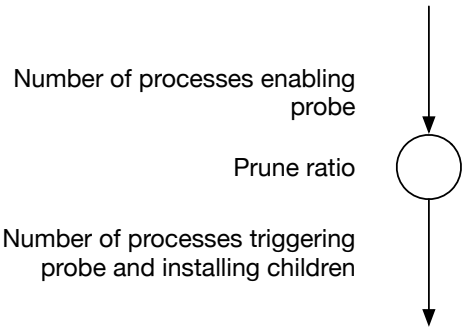


Figure 10.7: Pruning ratio in a probe.

while the number of children probes depends on how many processes satisfied the conditions associated with their ancestors:

$$installs(probe) = \begin{cases} \text{root}, max(installs_{backend}) \\ \text{otherwise}, invocs(ancs(probe)) \end{cases} \tag{10.1}$$

The number of invocations, $invocs(probe)$, defines the number of times a probe is triggered on a single backend:

$$invocs(probe) = (1 - ratio(probe)) \cdot installs(probe) \tag{10.2}$$

Probe installation are distributed across the backends and thus each backend install probes in parallel. For example, if a probe has to be installed in a total of 16 processes across two backends, and they each install in 8 processes, the cost of doing so should only account for one backend, as the installations will happen simultaneously.

The cost of installation might be slightly higher on one backend due to variations in the load in each backend, therefore we use max to represent the maximum value that we encountered across a large number of runs.

The model is not restricted to how many installations happen on each backend, where one backend could perform all the installations. Therefore, we use the backend that has the highest number of installations. The cost of installation for one probe is:

$$cost_{inst}(probe) = max(cost_{inst}) \cdot max(insts_{backend}(probe)) \tag{10.3}$$

The cost of invoking the probe has a sequential part and a communication part. The latter assumes that the depth of the MRNet topology increases as the number of processes increases, making network cost a logarithmic function as it depends on the MRNet tree depth [APM06]. We define $N_{processes}$ as the number of processes that a probe has triggered in the application.

$$\begin{aligned} cost_{invoc}(probe) = & max(cost_{invoc}) \cdot \\ & max(invocs_{backend}(probe)) + \\ & max(cost_{network}) \cdot log(N_{processes}) \end{aligned} \tag{10.4}$$

The total cost of all probes in the tree is the sum of the costs for installing and invoking each probe.

$$cost_{total} = \sum_{i=0}^{|probes|} cost_{install}(i) + cost_{invoc}(i) \tag{10.5}$$

We are interested in deriving what limits the scalability of our model. Therefore, we study what happens when $\lim_{N_{processes} \to \infty}$. In this case the logarithmic network term dwarfs the other factors. The cost of installation, $cost_{inst}(probe)$ becomes a constant as both $max(cost_{inst})$ and $max(insts_{backend}(probe))$ are constants. Similarly, for the cost of invocation $cost_{invoc}(probe)$, both $max(cost_{invoc})$ and $max(invocs_{backend}(probe))$ are constants. This leaves the logarithmic network term as the dominating factor:

$$\lim_{N_{processes} \to \infty} cost_{total} = max(cost_{network}) \cdot log(N_{processes}) \tag{10.6}$$

Therefore, we conclude that our model is able to achieve logarithmic scalability with our implementation. This is critical to enable scaling to extreme system sizes.

10.6.3 Performance Results

The model predicts logarithmic scalability $\mathcal{O}(\log n)$. In practice, many details can limit the scalability and we therefore seek to validate our model and to model scalability using experimental data.

Optimally we would compare the performance directly to a traditional debugger, however doing so would require a large psychological study. Therefore, we ignore the human factor and use a flat probe tree without any pruning as the baseline. In the flat probe tree, the root probe installs four children probes without any pruning. Each child probe aggregates a single message across all processes that satisfy the probe's condition. An example of the probes can be seen in Figure 10.3, containing the first two probes in the tree, and the target application in Figure 10.4.

During execution of a DySectAPI debugging session, a reduction in the task search space naturally occurs as probes are dynamically enabled only when a specified condition is met, which leads to reductions in the amount of instrumentation and in the amount of debug information generated. A chained probe tree with each probe tree having a pruning ratio of 50% represents the prescriptive debugging model. The pruning ratio is chosen as a representative pruning ratio. Actual pruning ratios in real scenarios will depend on the prescribed debugging session, the program being debugged and the inputs to the program. We will later study how important the pruning ratio is. The pruning of processes will be spread out equally over all the backends in our experiment.

Using microbenchmarking of DySectAPI we have obtained the following costs $cost_{invoc} = 0.72ms$, $cost_{install} = 0.28ms$ and $cost_{network} = 4.6ms$.

Figure 10.8 shows the actual and modeled overhead of the two probe trees with a pruning factor of 50% in the deep probe tree. We see that the modeled execution time for the deep probe tree more closely resembles the actual execution time, while for the flat tree there is a small difference. This is due to overlapping communication that results in a smaller communication overhead than modeled. In both cases, though, the model predicts an upper bound and therefore matches the observed scaling behavior, which is, as modeled, logarithmic.

The filtering of processes also reduces the amount of information presented to the programmer. For example, in the chained deep tree consisting of four probe 87.5% of the original processes are filtered out.

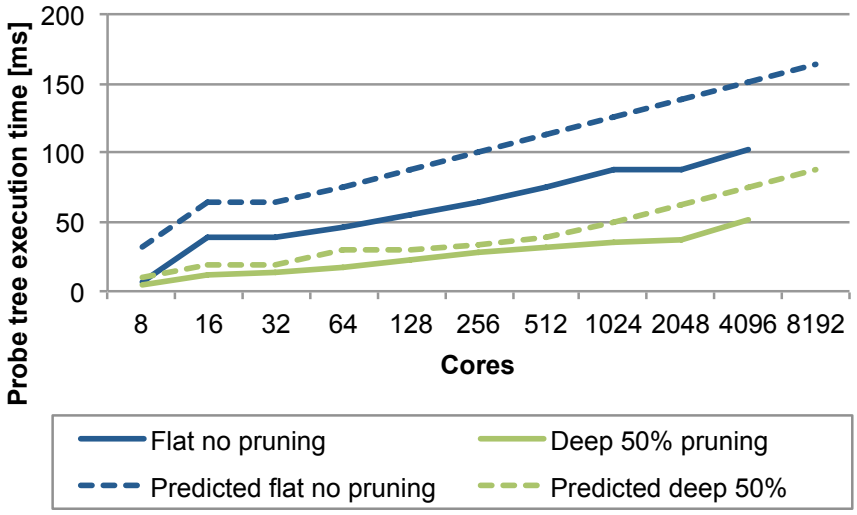


Figure 10.8: Actual and modeled execution time on Cab with 16 cores per node for a flat and a deep probe tree with 50% pruning.

10.6.4 Predicting Large Scale Performance

The introduced performance model is a good estimate of the worst-case runtime as exhibited by DySectAPI. We can use the model to predict the performance beyond current machine resources if we assume that the Cab cluster were an order of magnitude larger and exhibited the same performance.

Figure 10.9 show what the scalability would be like on a very large system. The modeled probes are the same ones as in Figure 10.8, consisting of four probes organized either in a flat tree or a deep chained tree with 50% pruning. The predicted overhead for a system with over 1,000,000 cores is just 180 ms for the four probes. This is consistent with the traditional interactive debugger DDT, which claims in the best case being able to step and display 700,000 cores in 100 ms [All14].

We also model the performance of the DySectAPI implementation for multiple pruning ratios and for cases when the pruning is not spread out equally over the backends. The pruning ratio affects the number of probe invocations per backend and has a demonstrable impact on overhead. At 1,048,576 cores the modeled overhead for the four probes organized in a deep chained tree, with 25% pruning in each of the probes, is 212.5 ms. With a pruning ratio of 50%

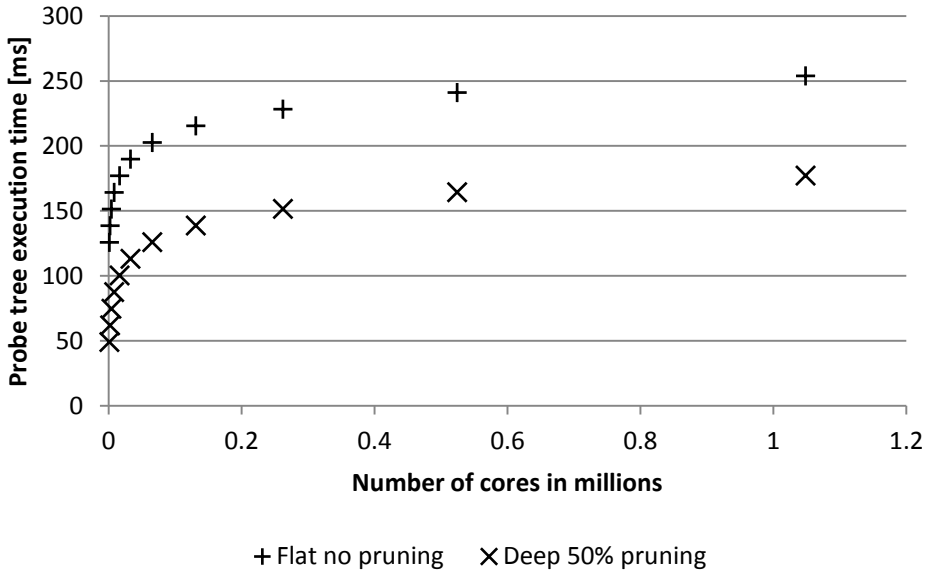


Figure 10.9: Modeled execution time on a larger version of Cab with 16 cores per node.

the overhead is 177.2 ms and with a pruning ratio of 75% the overhead is only 135.7 ms. Thus, overhead can be reduced by expressing probe trees in a way that prunes out many processes.

Intuitively, and according to the performance model, the cost of installing and invoking a probe depends on the backend with the highest cost. If we assume that the pruning of processes is unbalanced such that none are pruned on one backend, at 1,048,576 cores for the four probes organized in a deep chained tree, with a 25% pruning ratio, the modeled execution time overhead is 240.6 ms. With a pruning ratio of 50% the overhead is 221.9 ms and with a pruning ratio of 75% the overhead is 189.8 ms. Thus even if pruning of processes is unbalanced, the amount of pruning has a big impact on the execution time overhead. This can be explained by the difference in the overall amount of debug information that needs to be processed.

10.6.5 Case Study

We have evaluated DySectAPI on an MPI bug that only manifested itself at or above 3,456 MPI processes with BoomerAMG, a high-performance precondi-

```
Probe* mpiError =
  new Probe(Code::location("MPI_Err_str"),
  Domain::world(100),
  Acts(Act::stackTrace(),
       Act::trace("MPI_Err_str probe")));
```

Figure 10.10: MPI bug probe example.

tioner library developed at the Lawrence Livermore National Laboratory [Fal+14].

A specific configuration led to failures at scale and we have used DySectAPI to investigate the issue. An error was emitted by a method within MPI called `MPI_Error_string`. Therefore, we started with the simple probe seen in Figure 10.10 to figure out the call-path that led to the error message being printed.

The probe resulted in the debug output shown in Figure 10.11. From the output we see how a small subset of 6 processes called the `MPI_Error_string` routine and had equivalent stack traces. We can also see that the issue could be related to the call to `MPI_Allgather` and its call to `MPIR_ToPointer`, which resolves MPI communicator identifiers into pointers of the corresponding internal communicator structure. To this end, we created a more advanced probe to give more detailed information about the error, as shown in Figure 10.12. This probe tree captures the callpath from the previous stack trace and prints a trace message with the communicator identifiers. Further, a more detailed segmentation fault detector is setup to show precisely where it happens and to print the communicator identifier.

Figure 10.13 shows the debugging output of the probe. We see how three processes trigger the segmentation fault exception at the same location. By inspecting the call site at `intra_fns_new.c:2885` we see that this bug only appears in a shared memory feature, sporadically causing floating values to be resolved into pointers. We looked into the communicator initialization code and found a member field that was uninitialized. The problem was an issue in a recently upgraded MVAPICH 1.2.7 MPI library. Based on this diagnosis, a work-around was quickly identified and a bug was reported to the MVAPICH developers.

With this use case, we demonstrate how the prescriptive debugging model can be used as an engine to allow programmers to test their debugging intuition and that the expressiveness of the model helps reduce the error space. The use case shows how the prescriptive debugging model is capable of reducing to very condensed debugging information even for a complex debugging scenario.

```
DysectAPI Frontend:Info > [6] Trace: MPI Error string
   probe
DysectAPI Frontend:Info > [6] Stack trace
DysectAPI Frontend:Info > |-> [6] _start >
   __libc_start_main > main > HYPRE_PCGSetup >
   hypre_PCGSetup > HYPRE_BoomerAMGSetup >
   hypre_BoomerAMGSetup > hypre_seqAMGSetup >
   hypre_BoomerAMGSetup > hypre_BoomerAMGCreateS >
   hypre_MatvecCommPkgCreate >
   hypre_MatvecCommPkgCreate_core >
   hypre_MPI_Allgather > MPI_Allgather >
   intra_Allgather > MPIR_ToPointer > MPIR_Error >
   MPIR_Errors_are_fatal > PMPI_Error_string
```

Figure 10.11: MPI bug probe example output.

10.7 Conclusions

Prescriptive debugging is a novel debugging model that can scale without sacrificing key debugging information presented to programmers, thus filling the gap between traditional and lightweight debuggers. It allows programmers to codify their debugging intuition and to test their hypothesis with minimal user interaction during run-time. This allows the error search space to be reduced such that the information presented to the programmer is very condensed.

Using both experimental results and analytical modeling we show that our prototype implementation, DySectAPI, has logarithmic scaling on current systems. We also predict performance beyond current machine resources, and our model predicts good performance results even for very large system scales.

To this end we answer Research Question 10 by showing how the prescriptive debugging model achieves logarithmic scaling and when applied to a real use condensed the information

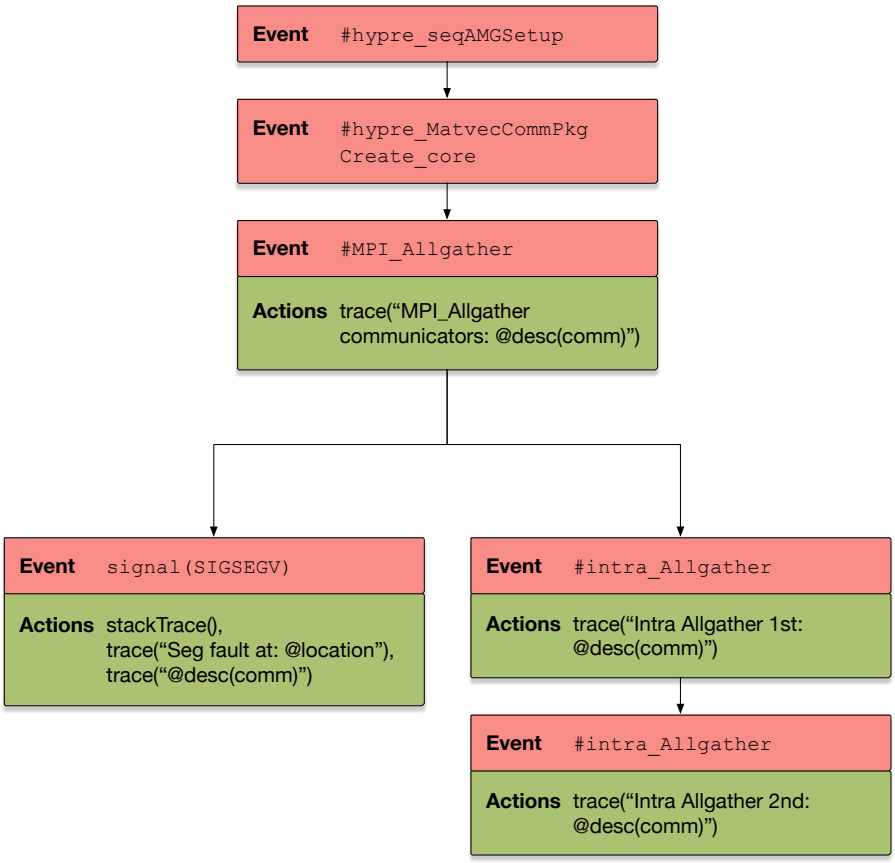


Figure 10.12: MPI bug advanced probe example.

```
DysectAPI Frontend: Info > [369] Trace: MPI_Allgather
   communicators: comm = [92:92]
DysectAPI Frontend: Info > [368] Trace: Intra
   allgather 1st comm = [92:92]
DysectAPI Frontend: Info > [363] Trace: Intra
   allgather 2nd comm = [92:92]
DysectAPI Frontend: Info > [3] Stack trace:
DysectAPI Frontend: Info > |-> [3] _start >
   __libc_start_main > main > HYPRE_PCGSetup >
   hypre_PCGSetup > HYPRE_BoomerAMGSetup >
   hypre_BoomerAMGSetup > hypre_seqAMGSetup >
   hypre_BoomerAMGSetup > hypre_BoomerAMGCreateS >
   hypre_MatvecCommPkgCreate >
   hypre_MatvecCommPkgCreate_core >
   hypre_MPI_Allgather > MPI_Allgather >
   intra_Allgather > intra_Allgather
DysectAPI Frontend: Info > [3] Trace: Segmentation
   fault at location: /usr/ local/tools/mvapich-intel-
   debug-1.2/src/src/coll/intra_fns_new.c:2885
DysectAPI Frontend: Info > [3] Trace: comm = [0:0]
```

Figure 10.13: MPI bug advanced probe example output.

Chapter 11

Conclusions

The thesis has contributed solutions to achieve high performance with prescriptive optimization and debugging. Achieving high performance is important to capitalize on using the specialized features offered in processors today.

11.1 Compiler Feedback

Chapter 6, Chapter 7 and Chapter 8 all contribute to showing how compiler based feedback to the programmer can lead to significant speedups without placing a big burden on the programmer. The feedback is based on the observation that small modifications to the source code can enable more aggressive compiler optimizations.

We investigate three ways of generating feedback:

- By extending an existing optimization to emit more information during compilation in Chapter 6.
- By embedding static analysis inside an IDE, allowing for dynamic feedback during compilation, i.e. it is not required to rebuild the whole project.

- By using existing compiler optimization reports from production compilers.

Each of these methods have advantages and disadvantages and are most suitable for specific feedback.

The first way, by modifying an existing compiler optimization, has the advantage of generating feedback that is easy to relate to the optimization being performed. On the other hand, it is not always possible to infer the exact statement in the original source code for which the limitation arises leading to fragile feedback. This method is best suited for an optimization pass with some implementation specific limitations that you like to overcome.

The second way, where the feedback is not generated from a specific compiler, but inside an IDE has the benefit of being fast and enabling very precise automatic refactorings. The downside is that this method can generate many false positives, where the feedback is not applicable. It is a suitable method when you like to enforce certain good coding patterns, like annotating with the C99 `restrict` keyword whenever suitable.

The third method, using optimization reports, is suitable to check that compiler optimizations are applied as expected given how we can compare the optimization performed by multiple compilers. It does not require a special toolchain, making initial setup easier, and allows us to perform a filtering with the feedback from multiple compilers. The downside is clearly the extra compilation time for compiling with more than one compiler. The feedback can be fragile, as correctly locating the statement in the original source code for which the feedback arises can be difficult, as it has been produced on the intermediary representation used for optimization.

In these studies, we see how this applies to legacy code bases, here in the form of benchmarks, as it good way to optimize these for modern architecture features not, perhaps not dominant when the program was written.

For memory optimizations we saw how slightly rewriting the memory accesses and applying the inlining feedback could lead to a speedup of 1.3 by making certain that accesses being optimized did not escape the compilers analysis. We also showed how it is possible to apply the memory optimization directly at the source code using our wizard leading to a speedup of 1.6 for 179.art from SPEC CPU2000 [Hen00] for GCC.

We enable automatic parallelization and automatic vectorization of several benchmarks and one industrial use case. For the use case, applying the C99 `restrict`

keyword to a filter and slightly rewriting the application using the feedback we achieve a speedup of 2.4 over the sequential version for the filter. The use case also contains an alternative implementation of the filter using OpenCV executing on a GPU. Here we achieve a speedup factor of 1.54 using the CPU over the GPU.

11.2 Improved Compiler Optimizations

In Chapter 9 we saw how we could improve dependence analysis for explicitly parallel programs leading to improvements in the accuracy of the dependence analysis. We have implemented our prototype in GCC 6.1. We evaluated our proposed analysis on set of benchmarks from the Rodinia Benchmark Suite [Che+09]. The OpenMP worksharing aware dependence analysis improved the number of correctly classified vectorizable dependences with 133%.

These improvements lead to more loops being vectorized, on average we achieve a speedup of 1.46 over the default dependence analysis and vectorizer in GCC. These incremental improvements in automatic vectorization allow more loops to be vectorized automatically. This lessens the burden placed on the programmer as loops are transparently vectorized without requiring vectorization specific pragmas or manually vectorization.

11.3 Prescriptive Debugging

Last, in Chapter 10 we introduced a new debugging model that is both general and scalable. Debugging is essential when creating programs to achieve high performance, find bugs and ensure correct execution. For parallel programs these challenges are even bigger due to non-deterministic execution, a whole new class of parallel bugs and the amount of information available.

The prescriptive debugging model is based on codifying the programmer's intuition in a way that allow the communication in the debugger to be scalable. Further, it focuses on condensing the information presented to the programmer to make it scalable for humans.

We evaluate our prototype using both experimental results and analytical modeling. The overhead is shown to be low and its scalability demonstrated to be logarithmic in terms of system size. This entails that the debugging model can scale to extreme sizes.

We also successfully apply our prototype to a real world bug test case and demonstrate how it was possible to quickly diagnose and fix the bug using our prototype.

11.4 Outlook

Unfortunately, automatic optimizations are not always applied due to limitations in compilers. When optimizations are not applied it is today often reserved for experts to study why and apply a fix. Similarly, parallel debugging quickly becomes overwhelming even for experts.

This thesis has demonstrated how we can transparently improve automatic optimization, by improving the underlying analysis and information the analysis has available. Fully automatic optimization is desirable, but as we show automatic vectorization is fragile with many limitations. When automatic optimizations fail, including the programmer is necessary to achieve high performance. We propose and contribute to the feedback approach where the programmer is guided in best utilizing the existing compilers optimizations. This enables non experts in applying more advances optimizations such as vectorization in more cases. One can argue that these methods cannot cover all programs. While we agree, it is still desirable to automatically optimize as many programs as possible and only rely on manual optimizations when required.

When it comes to the challenge of parallel debugging, we propose a user-guided approach that include the programmer. Fully automatic debugging methods are desirable, but not yet mature enough to be generally applicable. Therefore, we also here argue that for debugging we need to use the programmer's intuition and insight to effectively debug parallel programs.

We believe this trend could become more mainstream in the coming years, as more compilers start generate feedback and suggestions to the programmer and new debugging tools emerge.

Bibliography

[99] *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-48567-2 (Cited on page 67).

[ABH14] Cfir Aguston, Yosi Ben Asher, and Gadi Haber. "Parallelization Hints via Code Skeletonization". In: *Proceedings of the Symposium on Principles and Practice of Parallel Programming*. PPoPP. 2014 (Cited on page 59).

[Abr+96] David Abramson, Ian T. Foster, John Michalakes, and Rok Sosic. "Relative Debugging: A New Methodology for Debugging Scientific Applications". In: *Communications of the ACM* (1996) (Cited on pages 63, 139).

[Ahn+09] Dong H. Ahn, Bronis R. de Supinski, Ignacio Laguna, Gregory L. Lee, Ben Liblit, Barton P. Miller, and Martin Schulz. "Scalable Temporal Order Analysis for Large Scale Debugging". In: *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*. SC. 2009. DOI: 10.1145/1654059.1654104 (Cited on pages 63, 139, 141).

[Aho+06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2nd ed. 2006. ISBN: 0321486811 (Cited on page 38).

[AK87] Randy Allen and Ken Kennedy. "Automatic Translation of FORTRAN Programs to Vector Form". In: *ACM Transactions on Programming Language Systems*. TOPLAS (1987). DOI: 10.1145/29873.29875 (Cited on page 26).

[All14] Allinea Software Ltd. *Allinea DDT: The Global Standard for High-impact Debugging*. `http://www.allinea.com/products/ddt/features`. Accessed 17 October 2014. 2014 (Cited on pages 62, 139, 140, 156).

[All83] John Randal Allen. "Dependence Analysis for Subscripted Variables and Its Application to Program Transformations". PhD thesis. Rice University, 1983 (Cited on page 22).

[Amd67] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS. ACM, 1967. DOI: 10.1145/1465482.1465560 (Cited on page 14).

[APM06] D.C. Arnold, G.D. Pack, and B.P. Miller. "Tree-based Overlay Networks for Scalable Applications". In: *Proceedings of the International Parallel and Distributed Processing Symposium*. IPDPS (2006) (Cited on page 154).

[App97] Andrew W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. New York, NY, USA: Cambridge University Press, 1997. ISBN: 0521583896 (Cited on page 83).

[Arn+07] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. "Stack Trace Analysis for Large Scale Debugging". In: *Proceedings of the International Parallel and Distributed Processing Symposium*. IPDPS. 2007 (Cited on pages 63, 139, 141).

[Bai+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. "The NAS Parallel Benchmarks — Summary and Preliminary Results". In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. SC. 1991. DOI: 10.1145/125826.125925 (Cited on pages 32, 42).

[Ban93] Utpal K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, 1993. ISBN: 079239318X (Cited on pages 21, 23).

[BD06a] Kristof Beyls and Erik H. D'Hollander. "Discovery of Locality-Improving Refactorings by Reuse Path Analysis". In: *Proceedings of the International Conference on High Performance Computing and Communications*. HPCC. 2006. DOI: 10.1007/11847366_23 (Cited on page 57).

[BD06b] Kristof Beyls and Erik H. D'Hollander. "Intermediately Executed Code is the Key to Find Refactorings That Improve Temporal Data Locality". In: *Proceedings of the 3rd Conference on Computing Frontiers*. CF. 2006. DOI: 10.1145/1128022.1128071 (Cited on page 57).

[BD09] Kristof Beyls and Erik H D'Hollander. "Refactoring for data locality". In: *IEEE Computer* 42.2 (2009). DOI: 10.1109/MC.2009.57 (Cited on page 57).

[BE04] Daniel Berlin and David Edelsohn. "High-Level Loop Optimizations for GCC". In: *Proceedings of the GCC Developer's Summit*. 2004 (Cited on page 56).

[BF10] Ilona Bluemke and Joanna Fugas. "A Tool Supporting C code Parallelization". In: *Innovations in Computing Sciences and Software Engineering*. 2010. DOI: 10.1007/978-90-481-9112-3_44 (Cited on page 59).

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. "An API for Runtime Code Patching". In: *International Journal of High Performance Computing Applications* (2000). DOI: http://dx.doi.org/10.1177/109434200001400404 (Cited on pages 141, 150).

[Bie11] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, Jan. 2011 (Cited on pages 32, 40, 41).

[Bik+02] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. "Automatic Intra-register Vectorization for the Intel Architecture". In: *International Journal Parallel Programming* (2002). DOI: 10.1023/A:1014230429447 (Cited on page 26).

[Boa13] OpenMP Architecture Review Board. *OpenMP Application Program Interface (Version 4.0)*. OpenMP Specification. 2013 (Cited on pages 13, 113–115, 135).

[Boa15] OpenMP Architecture Review Board. *OpenMP Application Program Interface (Version 4.5)*. OpenMP Specification. 2015 (Cited on pages 12, 111).

[Boh07] Mark Bohr. "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper". In: *IEEE Solid-State Circuits Society Newsletter* 12.1 (2007). DOI: 10.1109/N-SSC.2007.4785534 (Cited on page 7).

[Bon+08] 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 International Conference on Compiler Construction*. CC. 2008. DOI: 10.1007/978-3-540-78791-4_9 (Cited on page 60).

[Bra00] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000) (Cited on page 102).

[Bra88] Thomas Brandes. "The Importance of Direct Dependences for Automatic Parallelization". In: *Proceedings of the 2nd International Conference on Supercomputing*. ICS. 1988. DOI: 10.1145/55364.55404 (Cited on page 20).

[Bro+00] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. "A Portable Programming Interface for Performance Evaluation on Modern Processors". In: *International Journal of High Performance Computing Applications* (2000). DOI: 10.1177/109434200001400303 (Cited on page 122).

[Bro+10] Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi, Bronis R. de Supinski, Dong H. Ahn, and Martin Schulz. "AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks". In: *International Conference on Dependable Systems and Networks*. 2010. DOI: 10.1109/DSN.2010.5544927 (Cited on pages 63, 141).

[Bul+01] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. "Benchmarking Java against C and Fortran for Scientific Applications". In: *Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande*. 2001 (Cited on pages 52, 85).

[BZS10] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. "Efficient Selection of Vector Instructions Using Dynamic Programming". In: *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO. 2010. DOI: 10.1109/MICRO.2010.38 (Cited on pages 26, 27).

[Cab+15] Diego Caballero, Sara Royuela, Roger Ferrer, Alejandro Duran, and Xavier Martorell. "Optimizing Overlapped Memory Accesses in User-directed Vectorization". In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS. 2015. DOI: 10.1145/2751205.2751224 (Cited on pages 13, 59, 135).

[CDL88] D. Callahan, J. Dongarra, and D. Levine. "Vectorizing Compilers: A Test Suite and Results". In: *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*. SC. 1988 (Cited on pages 30, 52).

[CDT13] CDT. *Eclipse C/C++ Development Tooling*. http://www.eclipse.org/cdt/. Accessed on 18/5/2013. 2013 (Cited on page 81).

[Che+09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. "Rodinia: A Benchmark Suite for Heterogeneous Computing". In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*. IISWC. 2009. DOI: 10.1109/IISWC.2009.5306797 (Cited on pages 32, 43, 52, 120, 165).

[Che+10] Zhezhe Chen, Qi Gao, Wenbin Zhang, and Feng Qin. "FlowChecker: Detecting Bugs in MPI Libraries via Message Flow Checking". In: *Proceedings of the 2010 ACM/IEEE Conference on Supercomputing*. SC. 2010 (Cited on pages 64, 141).

[Coh73] W. L. Cohagen. "Vector Optimization for the ASC". In: *Conference on Information Sciences and Systems*. 1973 (Cited on page 22).

[Cra13] Cray. *CrayDoc - ATP 1.7 Man Pages*. Accessed 22 December 2014. 2013 (Cited on page 63).

[CSS15] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. "Polyhedral Optimizations of Explicitly Parallel Programs". In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 2015. DOI: 10.1109/PACT.2015.44 (Cited on pages 60, 112).

[Cyt+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Transactions on Programming Languages and Systems*. TOPLAS (1991) (Cited on pages 18, 83).

[D+11] Jack Dongarra, Peter H. Beckman, et al. "The International Exascale Software Project Roadmap". In: *International Journal of High Performance Computing Applications*. IJHPCA (2011) (Cited on page 138).

[DAJ14] Minh Ngoc Dinh, David Abramson, and Chao Jin. "Scalable Relative Debugging". In: *IEEE Transactions Parallel Distributed Systems* (2014). DOI: 10.1109/TPDS.2013.86 (Cited on page 141).

[Den+74] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. Leblanc. "Design of Ion-Implanted MOSFETs with Very Small Physical Dimensions". In: *IEEE Journal of Solid-state Circuits* (1974) (Cited on page 6).

[Din+11] Minh Ngoc Dinh, David Abramson, Donny Kurniawan, Chao Jin, Bob Moench, and Luiz De Rose. "Assertion Based Parallel Debugging". In: *International Symposium on Cluster, Cloud and Grid Computing*. CCGrid. 2011 (Cited on page 63).

[Du+15] Yong Du, Kobi Vinayagamoorthy, Kevin Yuen, and Yan Zhang. *Explore Optimization Opportunities with XML Transformation Reports in IBM XL C/C++ and XL Fortran for AIX Compilers*. IBM developerWorks. 2015 (Cited on pages 57, 58).

[Dur+09] 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. 2009. DOI: 10.1109/ICPP.2009.64 (Cited on page 44).

[EIg+05] Frank C. EIgler, Vara Prasad, Will Cohen, Hien Nguyen, Martin Hunt, Jim Keniston, and Brad Chen. *Architecture of systemtap: a Linux trace/probe tool*. https://sourceware.org/systemtap/archpaper.pdf. 2005 (Cited on page 141).

[Esm+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark Silicon and the End of Multicore Scaling". In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA. ACM, 2011. DOI: 10.1145/2000064.2000108 (Cited on page 7).

[EWO04] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. "Vectorization for SIMD Architectures with Alignment Constraints". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. 2004. DOI: 10.1145/996841.996853 (Cited on pages 26, 59).

[Fal+14] Rob Falgout, Tzanio Kolev, Jacob Schroder, Panayot Vassilevski, and Ulrike Meier Yang. *Scalable Linear Solvers: Software*. https://computation-rnd.llnl.gov/linear_solvers/software.php. Accessed 17 October 2014. 2014 (Cited on page 158).

[FC07] Wu-chun Feng and Kirk Cameron. "The Green500 List: Encouraging Sustainable Supercomputing". In: *Computer* 40.12 (2007) (Cited on page 1).

[Fel10] Bob Feldman. *Debugging Exascale: To heck with the complexity, full speed ahead!* Sept. 2010 (Cited on page 143).

[Fly72] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071 (Cited on pages 6, 9).

[For15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard (Version 3.1)*. MPI Specification. 2015 (Cited on page 14).

[Fou] Free Software Foundation. *GNU Compiler Collection*. http://gnu.gcc.org. Accessed on 24/9/2014 (Cited on pages 17, 60, 65, 67, 71, 81, 96, 98).

[Fou13] Free Software Foundation. *GNU Binutils*. http://www.gnu.org/s/binutils/. 2013 (Cited on page 81).

[GKT91] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. "Practical Dependence Testing". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. 1991. DOI: 10.1145/113445.113448 (Cited on page 61).

[GM01] Dejan Glozic and Jeff McAffer. *Mark My Words*. http://www.eclipse.org/articles/Article-MarkMyWords/mark-my-words.html. Accessed on 7/4/2013. 2001 (Cited on page 84).

[GQP07] Qi Gao, Feng Qin, and Dhabaleswar K. Panda. "DMTracker: Finding Bugs in Large-Scale Parallel Programs by Detecting Anomaly in Data Movements". In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. SC. 2007. DOI: 10.1145/1362622.1362643 (Cited on pages 139, 141).

[GS06] Satish Chandra Gupta and Giridhar Sreenivasamurthy. *Navigating "C" in a "Leaky" Boat? Try Purify*. www.ibm.com/developerworks/rational/library/06/0822_satish-giridhar/. Accessed 17 October 2014. 2006 (Cited on page 64).

[GZ07] Olga Golovanevsky and Ayal Zaks. "Struct-reorg: current status and future perspectives". In: *Proceedings of the GCC Developer's Summit*. 2007 (Cited on page 56).

[Hal+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. "The WEKA Data Mining Software: An Update". In: *SIGKDD Explorations Newsletter* 11 (2009). DOI: 10.1145/1656274.1656278 (Cited on page 102).

[Hed+14] Ward Van Heddeghem, Sofie Lambert, Bart Lannoo, Didier Colle, Mario Pickavet, and Piet Demeester. "Trends in worldwide ICT electricity consumption from 2007 to 2012". In: *Computer Communications* 50 (2014). DOI: 10.1016/j.comcom.2014.02.008 (Cited on page 1).

[Hen00] John L. Henning. "SPEC CPU2000: Measuring CPU Performance in the New Millennium". In: *IEEE Computer* 33.7 (2000), pp. 28–35. ISSN: 0018-9162 (Cited on pages 51, 66, 85, 164).

[Hen06] John L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *ACM SIGARCH Computer Architecture News* (2006) (Cited on pages 32, 37, 38, 100).

[HGG06] Sebastian Hack, Daniel Grund, and Gerhard Goos. "Register Allocation for Programs in SSA-Form". In: *Proceedings of the International Conference on Compiler Construction*. CC. 2006 (Cited on page 19).

[Hin+14a] Andreas Erik Hindborg, Pascal Schleuniger, Nicklas Bo Jensen, and Sven Karlsson. "Hardware Realization of an FPGA Processor – Operating System Call Offload and Experiences". In: *Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 2014. DOI: 10.1109/DASIP.2014.7115604 (Cited on page x).

[Hin+14b] Andreas Hindborg, Pascal Schleuniger, Nicklas Bo Jensen, Maxwell Walter, Laust Brock-Nannestad, Lars Bonnichsen, Christian W. Probst, and Sven Karlsson. "Automatic Generation of Application Specific FPGA Multicore Accelerators". In: *The Asilomar Conference on Signals, Systems, and Computers*. 2014. DOI: 10.1109/ACSSC.2014.7094700 (Cited on page x).

[Hin+15] Andreas Erik Hindborg, Nicklas Bo Jensen, Pascal Schleuniger, and Sven Karlsson. "State of the Akvario Project". In: *Workshop on Architectural Research Prototyping (WARP)*. 2015. URL: http://www.csl.cornell.edu/warp2015/abstracts/hindborg-akvario-warp2015.pdf (Cited on page x).

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture (2nd Edition): A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1996. ISBN: 1-55860-329-8 (Cited on pages 6, 37).

[Ibm+05] Vara Prasad Ibm, Frank Ch, Eigler Red Hat, and Jim Keniston Ibm. *Locating System Problems Using Dynamic Instrumentation*. https://sourceware.org/systemtap/systemtap-ols.pdf. 2005 (Cited on page 141).

[IBM15] IBM. *XL C/C++ for Linux*. http://www-01.ibm.com/software/awdtools/xlcpp/linux/. Accessed on 27/1/2016. 2015 (Cited on pages 30, 71).

[Int] Intel. *Intel Composer XE 2015*. http://software.intel.com/en-us/intel-composer-xe. Accessed on 24/9/2014 (Cited on pages 30, 60, 71, 96, 98, 125).

[Int13] Intel. *Intel Composer XE 2013*. http://software.intel.com/en-us/intel-composer-xe. Accessed on 17/5/2013. 2013 (Cited on pages 57, 58).

[Int15a] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2015 (Cited on page 122).

[Int15b] Intel. *Intel Architecture Instruction Set Extensions Programming Reference*. Tech. rep. 2015. URL: http://download-software.intel.com/sites/default/files/319433-014.pdf (Cited on pages 26, 109).

[Jac+09] Bart Jacob, Paul Larson, Breno Henrique Leitao, and Saulo Augusto M Martins da Silva. *SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems*. IBM Redpaper Publication. 2009 (Cited on page 141).

[Jen+12a] Nicklas Bo Jensen, Per Larsen, Razya Ladelsky, Ayal Zaks, and Sven Karlsson. "Guiding Programmers to Higher Memory Performance". In: *Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)*. 2012 (Cited on page ix).

[Jen+12b] Nicklas Bo Jensen, Per Larsen, Razya Ladelsky, Ayal Zaks, and Sven Karlsson. "Guiding Programmers to Higher Memory Performance". In: *Proceedings of 5th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-12)*. 2012 (Cited on pages 57, 102).

[Jen+14] Nicklas Bo Jensen, Sven Karlsson, Niklas Quarfot Nielsen, Gregory L. Lee, Dong H. Ahn, Matthew Legendre, and Martin Schulz. "DySectAPI: Scalable Prescriptive Debugging". In: *Proceedings of the 2014 ACM/IEEE Conference on Supercomputing*. SC. Poster and extended abstract. 2014. URL: `http://sc14.supercomputing.org/sites/all/themes/sc14/files/archive/tech_poster/tech_poster_pages/post237.html` (Cited on page ix).

[Jen+15a] Nicklas Bo Jensen, Niklas Quarfot Nielsen, Gregory L. Lee, Sven Karlsson, Matthew LeGendre, Martin Schulz, and Dong H. Ahn. "A Scalable Prescriptive Parallel Debugging Model". In: *Proceedings of the International Parallel & Distributed Processing Symposium*. IPDPS. © 2015 IEEE. Reprinted, with permission. 2015. DOI: `10.1109/IPDPS.2015.15` (Cited on page ix).

[Jen+15b] Nicklas Bo Jensen, Pascal Schleuniger, Andreas Hindborg, Maxwell Walter, and Sven Karlsson. "Experiences with Compiler Support for Processors with Exposed Pipelines". In: *IEEE International Parallel & Distributed Processing Symposium: Reconfigurable Architectures Workshop*. IPDPSW. 2015. DOI: `http://dx.doi.org/10.1109/IPDPSW.2015.9` (Cited on page x).

[Jen12] Nicklas Bo Jensen. *GCC Bugzilla bug 49916*. `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=49916`. 2012 (Cited on page 76).

[Jen16] Nicklas Bo Jensen, ed. *D2.5 — Final Results for Exploration Tools*. COPCAMS. Deliverable for the Cognitive and Perceptive Camera Systems project. 2016 (Cited on page x).

[JK16] Nicklas Bo Jensen and Sven Karlsson. *Improving Loop Dependency Analysis*. Journal manuscript submitted for publication. 2016 (Cited on page x).

[JKP14a] Nicklas Bo Jensen, Sven Karlsson, and Christian W. Probst. "Compiler Feedback using Continuous Dynamic Compilation during Development". In: *Workshop on Dynamic Compilation Everywhere (DCE)*. 2014 (Cited on page ix).

[JKP14b] Nicklas Bo Jensen, Sven Karlsson, and Christian W. Probst. "Compiler Feedback using Continuous Dynamic Compilation during Development". In: *Workshop on Dynamic Compilation Everywhere*. DCE. 2014 (Cited on pages 97, 102).

[JPK14] Nicklas Bo Jensen, Christian W. Probst, and Sven Karlsson. "Code Commentary and Automatic Refactorings using Feedback from Multiple Compilers". In: *Swedish Workshop on Multicore Computing (MCC)*. 2014 (Cited on page ix).

[JTC99] JTC 1/SC 22/WG 14. *ISO/IEC 9899:1999: Programming languages – C*. International Organization for Standards. ISO C99 Standard. 1999 (Cited on page 24).

[KA02] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002. ISBN: 1-55860-286-0 (Cited on page 108).

[Kar+13] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary Devito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles H. Still. "Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application". In: *Proceedings of the International Symposium on Parallel and Distributed Processing*. IPDPS. 2013. DOI: 10.1109/IPDPS.2013.115 (Cited on page 16).

[Ken+99] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. "Partial redundancy elimination in SSA form". In: *ACM Transactions on Programming Languages and Systems*. TOPLAS (1999) (Cited on page 83).

[KF00] Thomas Kistler and Michael Franz. "Automated Data-member Layout of Heap Objects to Improve Memory-hierarchy Performance". In: *ACM Transactions on Programming Languages and Systems*. TOPLAS (2000). DOI: 10.1145/353926.353937 (Cited on page 56).

[KH11] Ralf Karrenberg and Sebastian Hack. "Whole-function vectorization". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO. 2011. DOI: 10.1109/CGO.2011.5764682 (Cited on page 59).

[Kic+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-Oriented Programming". In: *Proceedings of the European Conference on Object-Oriented Programming*. ECOOP (1997) (Cited on page 143).

[KL00] Andreas Krall and Sylvain Lelait. "Compilation Techniques for Multimedia Processors". In: *International Journal of Parallel Programming* (2000). DOI: 10.1023/A:1007507005174 (Cited on page 26).

[Kle+12] Michael Klemm, Alejandro Duran, Xinmin Tian, Hideki Saito, Diego Caballero, and Xavier Martorell. "Extending OpenMP* with vector constructs for modern multicore SIMD architectures". In: *International Workshop on OpenMP*. IWOMP (2012). DOI: 10.1007/978-3-642-30961-8_5 (Cited on pages 13, 135).

[Kra+03] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. "MARMOT: An MPI Analysis and Checking Tool". In: *International Conference on Parallel Computing*. ParCo. 2003 (Cited on page 64).

[LA00] Samuel Larsen and Saman Amarasinghe. "Exploiting Superword Level Parallelism with Multimedia Instruction Sets". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI (2000). DOI: 10.1145/349299.349320 (Cited on pages 26, 27, 60).

[Lad06] Razya Ladelsky. "Matrix flattening and transposing in GCC". In: *Proceedings of the GCC Developer's Summit*. 2006 (Cited on pages 56, 67, 69).

[Lag+11] Ignacio Laguna, Todd Gamblin, Bronis R. de Supinski, Saurabh Bagchi, Greg Bronevetsky, Dong H. Ahn, Martin Schulz, and Barry Rountree. "Large Scale Debugging of Parallel Tasks with AutomaDeD". In: *Proceedings of the 2011 ACM/IEEE Conference on Supercomputing*. SC. 2011 (Cited on pages 63, 64).

[Lar+11a] Per Larsen, Razya Ladelsky, Sven Karlsson, and Ayal Zaks. "Compiler Driven Code Comments and Refactoring". In: *Workshop on Programmability Issues for Heterogeneous Multicores*. MULTIPROG. 2011 (Cited on pages 57, 58, 65, 66, 89).

[Lar+11b] Per Larsen, Razya Ladelsky, Jacob Lidman, Sally A. McKee, Sven Karlsson, and Ayal Zaks. *Automatic Loop Parallelization via Compiler Guided Refactoring*. Technical Report. Technical University of Denmark, 2011 (Cited on pages 57, 58).

[Lar+12] Per Larsen, Razya Ladelsky, Jacob Lidman, Sally A. McKee, Sven Karlsson, and Ayal Zaks. "Parallelizing More Loops with Compiler Guided Refactoring". In: *International Conference on Parallel Processing*. ICPP. 2012 (Cited on pages 57, 58, 97, 102).

[Lar11] Per Larsen. *Feedback Driven Annotation and Refactoring of Parallel Programs*. PhD Thesis. Technical University of Denmark, 2011 (Cited on pages 31, 59, 102).

[Law] Lawrence Livermore National Laboratory. *Advanced Simulation and Computing Sequoia*. `https://asc.llnl.gov/computing_resources/sequoia`. Accessed 17 October 2014 (Cited on pages 62, 138).

[LCD91] David Levine, David Callahan, and Jack Dongarra. "A Comparative Study of Automatic Vectorizing Compilers". In: *Parallel Computing* 17 (1991) (Cited on page 26).

[Lee] Corinna G. Lee. *UTDSP Benchmark Suite*. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`. Accessed September 19, 2016 (Cited on pages 52, 85, 102).

[Lee+07] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Barton P. Miller, and Martin Schulz. "Benchmarking the Stack Trace Analysis Tool for BlueGene/L". In: *Proceedings of the International Conference on Parallel Computing*. ParCo. 2007 (Cited on pages 63, 141).

[Lee+08] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Matthew P. LeGendre, Barton P. Miller, Martin Schulz, and Ben Liblit. "Lessons Learned at 208K: Towards Debugging Millions of Cores". In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC. 2008 (Cited on pages 63, 139, 141).

[Lia+99] Shih-Wei Liao, Amer Diwan, Robert P. Bosch Jr., Anwar Ghuloum, and Monica S. Lam. "SUIF Explorer: An Interactive and Interprocedural Parallelizer". In: *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP. 1999. DOI: `10.1145/301104.301108` (Cited on page 59).

[LK32] Gary Lakner and Brant Knudson. *IBM System Blue Gene Solution: Blue Gene/Q System Administration*. IBM Redbooks. 20132 (Cited on pages 63, 139).

[LMM85] Olaf Lubeck, James Moore, and Raul Mendez. "A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S810/120, and Cray X-MP/2". In: *IEEE Computer* 18.12 (1985). DOI: `10.1109/MC.1985.1662769` (Cited on page 26).

[LP94] Wei Li and Keshav Pingali. "A singular loop transformation framework based on non-singular matrices". In: *International Journal of Parallel Programming* (1994). DOI: `10.1007/BF02577874` (Cited on page 61).

[Lue+03] Glenn R. Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. "MPI-CHECK: a tool for checking Fortran 90 MPI programs". In: *Concurrency and Computation: Practice and Experience* (2003) (Cited on page 64).

[Luk+05] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. 2005. DOI: 10.1145/1065010.1065034 (Cited on page 122).

[LZ95] Shun-tak Leung and John Zahorjan. *Optimizing Data Locality by Array Restructuring*. Tech. rep. University of Washington, 1995 (Cited on page 57).

[Mal+11] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. "An Evaluation of Vectorizing Compilers". In: *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*. PACT. 2011. DOI: 10.1109/PACT.2011.68 (Cited on pages 30, 44, 51, 60, 102).

[Mit+14] Subrata Mitra, Ignacio Laguna, Dong H. Ahn, Saurabh Bagchi, Martin Schulz, and Todd Gamblin. "Accurate Application Progress Analysis for Large-scale Parallel Debugging". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. 2014 (Cited on page 63).

[Moo75] Gordon E. Moore. "Progress in Digital Integrated Electronics". In: *International Electron Devices Meeting*. 1975 (Cited on page 6).

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA, 1997. ISBN: 1-55860-320-4 (Cited on pages 16, 83).

[Mül+12] Matthias S Müller, John Baron, William C Brantley, Huiyu Feng, Daniel Hackenberg, Robert Henschel, Gabriele Jost, Daniel Molka, Chris Parrott, Joe Robichaux, Pavel Shelepugin, Matthijs Waveren, Brian Whitney, and Kalyan Kumaran. "SPEC OMP2012 – An Application Benchmark Suite for Parallel Systems Using OpenMP". In: *Proceedings of the International Workshop on OpenMP*. IWOMP. 2012. DOI: 10.1007/978-3-642-30961-8 (Cited on pages 32, 44).

[Nai+03] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. "Vectorizing for a SIMdD DSP architecture". In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES. 2003. DOI: 10.1145/951710.951714 (Cited on page 59).

[Nai04] Dorit Naishlos. "Autovectorization in GCC". In: *Proceedings of the GCC Developer's Summit* (2004) (Cited on pages 59, 115).

[Ng+12] Karen Ng, Matt Warren, Peter Golde, and Anders Hejlsberg. *The Roslyn Project: Exposing the C# and VB compiler code analysis.* Whitepaper. Microsoft. 2012 (Cited on page 57).

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Secaucus, NJ, USA, 1999. ISBN: 3540654100 (Cited on page 17).

[NRZ06] Dorit Nuzman, Ira Rosen, and Ayal Zaks. "Auto-vectorization of Interleaved Data forc SIMD". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI. 2006. DOI: 10.1145/1133255.1133997 (Cited on pages 27, 59).

[NS07] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary I]nstrumentation". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI. 2007 (Cited on page 62).

[NZ08] Dorit Nuzman and Ayal Zaks. "Outer-Loop Vectorization". In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques.* PACT. 2008. DOI: 10.1145/1454115.1454119 (Cited on page 59).

[Oak] Oak Ridge National Laboratory. *Introducing Titan - Advancing the Area of Accelerated Computingß.* https://www.olcf.ornl.gov/titan/. Accessed 17 October 2014 (Cited on page 138).

[OK] David R. O'Hallaron and Loukas F. Kallivokas. *183.equake SPEC CPU2000 Benchmark Description File.* Accessed on 13/5/2016 (Cited on pages 69, 73).

[Ora] Oracle. *Oracle Solaris Studio.* http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html. Accessed on 17/5/2013 (Cited on pages 57, 58, 71).

[Pat15] David Patterson. "Past and Future of Hardware and Architecture". In: *SOSP History Day 2015.* SOSP. 2015. DOI: 10.1145/2830903.2830910 (Cited on page 6).

[PC10] Antoniu Pop and Albert Cohen. "Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers". In: *Proceedings of the 15th Workshop on Compilers for Parallel Computers.* CPC. 2010. URL: https://hal.inria.fr/inria-00551518 (Cited on page 112).

[PD80] David A. Patterson and David R. Ditzel. "The Case for the Reduced Instruction Set Computer". In: *ACM SIGARCH Computer Architecture News* 8.6 (1980). DOI: 10.1145/641914.641917 (Cited on page 6).

[PJ15] Vasileios Porpodas and Timothy M. Jones. "Throttling Automatic Vectorization : When Less Is More". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. PACT. 2015. DOI: 10.1109/PACT.2015.32 (Cited on page 46).

[Pug91] William Pugh. "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis". In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. SC. 1991. DOI: 10.1145/125826.125848 (Cited on page 61).

[RAM03] P.C. Roth, D.C. Arnold, and B.P. Miller. "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools". In: *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. SC. 2003 (Cited on page 150).

[Ram94] G. Ramalingam. "The Undecidability of Aliasing". In: *ACM Transactions on Programming Languages and Systems*. TOPLAS (1994). DOI: 10.1145/186025.186041 (Cited on page 24).

[RD] Charles Roberson and Max Domeika. *179.art SPEC CPU2000 Benchmark Description File*. Accessed on 13/5/2016 (Cited on pages 69, 71).

[RD14] Pethuru Raj and Ganesh Chandra Deka. *Handbook of Research on Cloud Infrastructures for Big Data Analytics*. English. Ed. by Pethuru Raj and Ganesh Chandra Deka. IGI Global, 2014. ISBN: 9781466658646. DOI: 10.4018/978-1-4666-5864-6 (Cited on page 2).

[Rei12] James Reinders. *An Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors*. Intel Corporation. 2012 (Cited on page 39).

[Rog] Rogue Wave Software. *TotalView Achieves Massive Milestone Towards Exascale Debugging*. http://www.roguewave.com/company/news-events/press-releases/2012/scalability-milestone-for-totalview-debugger.aspx. Accessed 17 October 2014 (Cited on page 62).

[Rog14] Rogue Wave Software. *TotalView® Graphical Debugger*. http://www.roguewave.com/products/totalview.aspx. Accessed 17 October 2014. 2014 (Cited on pages 62, 139, 140).

[Rus78] Richard M. Russell. "The CRAY-1 Computer System". In: *Communications of the ACM* (1978). ISSN: 0001-0782. DOI: 10.1145/359327.359336 (Cited on pages 26, 109).

[RZB15] Gil Rapaport, Ayal Zaks, and Yosi Ben-Asher. "Streamlining Whole Function Vectorization in C Using Higher Order Vector Semantics". In: *IEEE International Parallel and Distributed Processing Symposium Workshop*. IPDPSW. IEEE, 2015. DOI: 10.1109/IPDPSW.2015.37 (Cited on page 59).

[SHC05] Jaewook Shin, Mary Hall, and Jacqueline Chame. "Superword-Level Parallelism in the Presence of Control Flow". In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO. 2005. DOI: 10.1109/CGO.2005.33 (Cited on page 27).

[Ske12] Jonas Skeppstedt. *An Introduction to the Theory of Optimizing Compilers*. 2012. ISBN: 978-91-977940-1-5 (Cited on pages 21, 24, 50, 83).

[SS14] Richard M. Stallman and Cygnus Support. *Debugging with GDB : The GNU source-level debugger*. Free Software Foundation, 2014. ISBN: 1-88211-409-4 (Cited on pages 62, 139, 140).

[TC11] Linda Torczon and Keith Cooper. *Engineering a Compiler*. 2nd. San Francisco, CA, USA, 2011. ISBN: 012088478X (Cited on page 83).

[The] The LLVM Foundation. *clang: a C Language Family Frontend for LLVM*. http://clang.llvm.org. Accessed on 24/9/2014 (Cited on pages 60, 96, 98).

[Tri+09] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. "Polyhedral-Model Guided Loop-Nest Auto-Vectorization". In: *18th International Conference on Parallel Architectures and Compilation Techniques*. PACT. 2009. DOI: 10.1109/PACT.2009.18 (Cited on pages 59, 60).

[Tri+10] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. "GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation". In: *GCC Research Opportunities Workshop*. GROW. 2010 (Cited on page 61).

[VRD10] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. "The Paralax Infrastructure: Automatic Parallelization with a Helping Hand". In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT. 2010. DOI: 10.1145/1854273.1854322 (Cited on page 59).

[VS00] Jeffrey S. Vetter and Bronis R. de Supinski. "Dynamic Software Testing of MPI Applications with Umpire". In: *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. SC. 2000 (Cited on page 64).

[Wal+15] Maxwell Walter, Pascal Schleuniger, Andreas Erik Hindborg, Carl Christian Kjærgaard, Nicklas Bo Jensen, and Sven Karlsson. "Experiences Implementing Tinuso in gem5". In: *Second gem5 User Workshop*. 2015. URL: `http://www.m5sim.org/wiki/images/f/f5/2015_ws_16_gem5-workshop_mwalter.pptx` (Cited on page x).

[Wal91] David W. Wall. "Limits of Instruction-level Parallelism". In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS. 1991. DOI: `10.1145/106972.106991` (Cited on page 7).

[WF10] Christian Wimmer and Michael Franz. "Linear Scan Register Allocation on SSA Form". In: *Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization*. CGO. 2010 (Cited on page 83).

[Whe13] David Wheeler. *SLOCcount*. 2013. URL: `http://www.dwheeler.com/sloccount/` (Cited on pages 71, 85).

[Wil+94] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. *The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler*. Tech. rep. 1994 (Cited on page 61).

[WM95] Wm. A. Wulf and Sally A. McKee. "Hitting the Memory Wall: Implications of the Obvious". In: *ACM SIGARCH Computer Architecture News* (1995). DOI: `10.1145/216585.216588` (Cited on page 7).

[Woo+95] 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 Architectures*. ISCA. 1995 (Cited on pages 32, 40, 41).

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. "Constant Propagation with Conditional Branches". In: *ACM Transactions on Programming Languages and Systems*. TOPLAS (1991) (Cited on page 83).

[Zak15] Ayal Zaks. "Compiling for Scalable Computing Systems – the Merit of SIMD". The 5th Annual Henry Taub International TCE Conference. 2015. URL: `http://tce.webee.eedev.technion.ac.il/wp-content/uploads/sites/8/2015/11/AyalZaks.pdf` (Cited on page 10).

Limitations per Benchmark

Primary and first encountered limitation for each individual benchmark, in the five benchmark suites, Spec CPU2006, Spec OMP2012, NAS Parallel Benchmarks 3.3.1, Rodinia Benchmark Suite 3.1 and SPLASH-2x is shown in Figure 1.

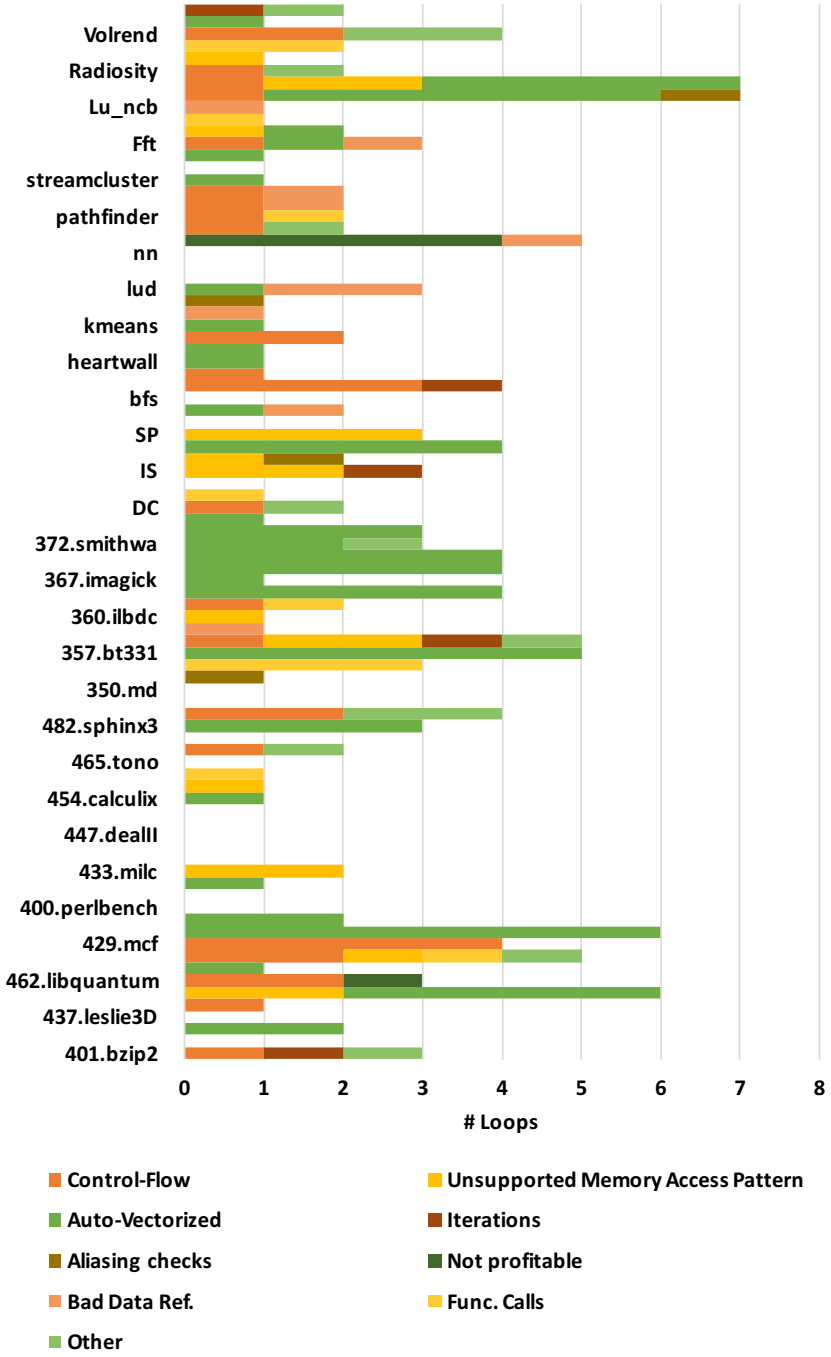


Figure 1: Primary issues encountered during auto-vectorization in GCC 6.1 for each benchmark. Numbers are for inner loops representing more than 5% of the benchmark execution time