Feedback Driven Annotation and Refactoring of Parallel Programs

Larsen, Per; Karlsson, Sven ; Madsen, Jan

Publication date:
2011

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

Link back to DTU Orbit

Citation (APA):
Larsen, P., Karlsson, S., & Madsen, J. (2011). Feedback Driven Annotation and Refactoring of Parallel Programs. Kgs. Lyngby, Denmark: Technical University of Denmark (DTU). (IMM-PHD-2011; No. 251).

Feedback Driven Annotation and Refactoring of Parallel Programs

Per Larsen

Kongens Lyngby 2011
IMM-PHD-2011-251

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

This thesis combines programmer knowledge and feedback to improve modeling and optimization of software. The research is motivated by two observations. First, there is a great need for automatic analysis of software for embedded systems – to expose and model parallelism inherent in programs. Second, some program properties are beyond reach of such analysis for theoretical and practical reasons – but can be described by programmers.

Three aspects are explored. The first is annotation of the source code. Two annotations are introduced. These allow more accurate modeling of parallelism and communication in embedded programs. Runtime checks are developed to ensure that annotations correctly describe observable program behavior. The performance impact of runtime checking is evaluated on several benchmark kernels and is negligible in all cases.

The second aspect is compilation feedback. Annotations are not effective unless programmers are told how and when they are beneficial. A prototype compilation feedback system was developed in collaboration with IBM Haifa Research Labs. It reports issues that prevent further analysis to the programmer. Performance evaluation shows that three programs performes significantly faster – up to 12.5x – after modification directed by the compilation feedback system.

The last aspect is refinement of compilation feedback. Out of numerous issues reported, few are important to solve. Different compilers and compilation flags are used to estimate whether an issue can be resolved or not. On average, 43% of the issues reported can be categorized as potentially resolvable (27%) or unresolvable (15%).

Resumé

Denne afhandling kombinerer computer programmørens ekspertise med feedback for bedre at modellere og optimere programmel. Forskningen er motiveret af to observationer. For det første er der et stort behov for automatisk analyse af programmel til indlejrede systemer. Programanalyse er nødvendigt for at identificere og modellere parallelisme i indlejrede programmer. For det andet så er programanalyse utilstrækkeligt til at udlede visse egenskaber af teoretiske og rent praktiske grunde – men programmører kan ofte beskrive disse.

Tre aspekter af denne problemstilling udforskes. Det første er annotationer som indsættes i kildekoden. To annotationer introduceres og det vises at disse tillader en mere præcis modellering af parallelisme og afhængigheder i kildekoden. Der blev implementeret funktionalitet som under programafvikling sikrer at annotationerne beskriver programmets afhængigheder på korrekt vis. Denne funktionalitets indflydelse på køretiden blev evalueret på en håndfuld test-programmer. Eksperimenterne påviste ikke en forringelse i ydeevnen.

Det andet aspekt er feedback fra oversættere. Annotation er ineffektivt med mindre det oplyses hvor og hvornår programmøren kan bidrage. En prototype af et feedback system blev udviklet i samarbejde med IBM Haifa Research Labs. Systemet rapporterer problemer som forhindrer programanalyse. Eksperimenter viste at tre test-programmer afvikles væsentligt hurtigere – op til 12.5 gange – efter modifikation vha. systemets anvisninger.

Det tredje aspekt er raffinering af feedback. Et fåtal af de mange problemer som rapporteres er værd at løse. Forskellige oversættere og oversætterflag blev dernæst anvendt til at estimere om et problem kan afhjælpes eller ej. I gennemsnit blev 27% af problemerne kategoriseret som potentielt løsbare og 15% blev kategoriseret som uløselige.

Preface

This thesis was prepared at DTU Informatics, Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D. degree in engineering.

Programmers gain a high-level understanding of programs after working with their source codes for some time. It was studied how such programmer knowledge can be leveraged to model and optimize software for embedded and parallel systems.

The thesis is based on a collection of five research papers written during the period 2009–2011, and elsewhere published.

Lyngby, June 2011

[illegible]

Per Larsen

Papers included in the Thesis

[94] Per Larsen, Sven Karlsson, Jan Madsen. Identifying Inter-Task Communication in Shared Memory Programming Models. *Proceedings of 5th International Workshop on OpenMP*, 2009.

[95] Per Larsen, Sven Karlsson and Jan Madsen. Expressing Inter-task Dependencies between Parallel Stencil Operations. *Proceedings of 3rd Workshop on Programmability Issues for Heterogeneous Multicores*, 2010.

[97] Per Larsen, Razya Ladelsky, Sven Karlsson and Ayal Zaks. Compiler Driven Code Comments and Refactoring. *Proceedings of 4th Workshop on Programmability Issues for Heterogeneous Multicores*, 2011.
Received Best Paper Award

[98] Per Larsen, Razya Ladelsky, Jacob Lidman, Sally A. McKee, Sven Karlsson and Ayal Zaks. Automatic Loop Parallelization via Compiler Guided Refactoring. Technical Report, *IMM-Technical Report-2011-12*, DTU Informatics, Technical University of Denmark, 2011. http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=6041

[96] Per Larsen, Sven Karlsson and Jan Madsen. Expressing Coarse-grain Dependences among Tasks in Shared Memory Programs *Special Issue of IEEE Transactions on Industrial Informatics*, 2011.
Accepted for publication

Acknowledgements

First of all, I thank my primary supervisor Sven Karlsson for his tireless efforts to make a scientist out of me. The task has certainly not been easy and I am deeply grateful for his unfailing confidence in me. His determination, patience and optimism were sources of encouragement and an invaluable help throughout the project.

I also thank my co-supervisor Jan Madsen. He encouraged me to apply for the DTU Informatics PhD programme. Throughout my studies, I have benefited greatly from Jan's comprehensive knowledge of embedded systems and his positive and pragmatic attitude towards any and all challenges.

In the spring of 2010, I applied for summer internship through HiPEAC, a European Network of Excellence. I spent the summer working as an intern at the IBM Haifa Research Labs in Israel. The collaboration with the compiler technologies group in Haifa provided a critical boost to my research. Hence, I am deeply indebted to IBM and HiPEAC for making this possible. A special thank to my hosts Ayal Zaks and Razya Ladelsky for their help and inspiration for the work on compilation feedback and, additionally, for making me feel truly welcome in Haifa. I also want to thank the good friends I made during my stay including Vladimir Cakarevic, Lois Orosa, Waldemar Hummer and Mircea Namolaru.

Many persons at DTU Informatics have been of great assistance to me. I am truly grateful for their helpfulness. These include Karin Tunder, Dina Berenstein and Ulla Jensen among others. I also thank Michael R. Hansen for his careful advice early in this study.

I thank Xavier Martorell for hosting me during my stay at Universitat Politèchnica de Catalunya, Barcelona and for treating me to some magnificent Tapas. I also thank Per Stenström for encouraging me to submit my work to the fourth MultiProg workshop. Others who have been very helpful to me include Laust Nannestad-Brock, Mason Chang, Jacob Lidman and Sally A. McKee.

Before I entered the PhD programme, I had the pleasure of working at a small company in the heart of Copehhagen. I got a little real-world experience. I would like to thank Lars Mensal, Jacob Lildballe, Hans Steenberg and countless others for being great mentors, colleagues and friends.

Working hard is not worthwhile unless one can sometimes rest and relax in the company of good friends. I am grateful for the many good times I had in the company of Charlotte Vilhelmsen, Peter Sørensen, Mads Johnsen, Christian Rank, Michael R. Boesen, Stavros Passas, Aske Brekling, Pascal Schleuniger and many, many more than can be listed here. Thank you all.

Most of all I thank my parents Else Marie and Flemming, my brother, Bo, and my wonderful girlfriend, Britta, for their unconditional encouragement, love and support. I could not have done this without you. Thanks!

The research in this thesis has been partially supported by HiPEAC and Artist-Design, both European Union Networks of Excellence. It has also been partially supported by the SMECY project, a part of the ARTEMIS Joint Undertaking.

My years as a PhD student contained several unexpected turns of events. Leonard Cohen penned the words that best describe my current sentiment. They are taken from a song which means a lot to me, and incidentally, to my friends in Haifa and elsewhere. Cohen sings *I did my best, it wasn't much.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computers form an integral part of the developed world. We use laptops, smartphones, cameras and tablet computers every day. Some computers can go where man cannot – deep underwater, in space, nuclear power plants, munitions and autonomous vehicles, to mention a few.

In many cases, the computers are embedded into other devices. Such devices are quite different from personal computers, servers and super-computers. Devices that integrate computing capabilities, or *embedded systems*, account for the vast majority of computer systems.

The research in this thesis is motivated by challenges in analyzing and optimizing programs for embedded systems. Many challenges are primarily due to limitations in our ability to analyze program source codes. This thesis demonstrates that several concrete limitations can be mitigated by programmer inserted annotations and code refactoring. Compiler feedback is used to report to the programmer where and why program analysis encounters issues. Many issues are reported but not all can be resolved by changing the code. The possibility of resolving an issue via annotation and refactoring is estimated by combining compiler feedback from multiple builds of the same source code. This prioritizes the annotation and refactoring efforts.

This chapter has three parts. The first part characterizes embedded systems and their development. It also explains where and why the analysis of embedded systems rely on abstract models of programs. Finally, the difficulties in generating such models are outlined. The second part focuses on the programming aspects of embedded systems development. Most importantly, it explains the challenges of exposing parallelism to the underlying hardware. This part also describes techniques to *analyze* programs, techniques to *annotate* programs and their interaction. The final part motivates the work in this thesis and outlines its contents.

1.1 Embedded Systems Design

Embedded systems are often battery powered, are manufactured in high volume and must provide real-time performance. As a result, they are highly constrained in terms of cost, size and power consumption. To meet performance and power-consumption constraints, critical parts may be realized as dedicated hardware blocks. The resulting systems often have a heterogeneous system architecture combining general-purpose units with dedicated hardware components. This complicates the system design [127]. Yet, design teams are expected to deliver increasingly complex designs within a short period of time. When combined, these circumstances create a need for automation of embedded systems design.

Typically, it is easier and less costly to make significant changes early rather than later in a project. It is therefore important to explore different choices early in the design process. Initially, the design team must determine what the embedded device will look like at the system level. For instance, the type and number of processing elements, memories and interconnects between these must be determined.

Devices having essentially the same function can be realized through a number of quite different system-level designs [144]. If two candidate designs can be evaluated and compared with respect to objectives such as power, cost and performance, a systematic search of the *design space* is feasible. This is called *design space exploration*, DSE. Early in the design process, simulators and hardware prototypes are typically unavailable. Alternatively, design teams can build analytical models of hardware components and relevant application workloads. Techniques using analytical models for evaluation are therefore important to identify corner cases early in the design process [68]. The *task graph* model is one, commonly used abstraction of application workloads.

1.1.1 Task Graphs

Task graphs or variants thereof model coarse grain computation and communication in parallel programs. Task graphs are directed and acyclic. Sequentially executed instruction sequences are called *tasks* and form the nodes of a task graph. Communication and synchronization creates dependencies among tasks. These constrain the execution order among tasks. The dependence relation is represented by directed edges in the graph.

As previously mentioned, DSE tools often use task graphs to model embedded application workloads [143, 52, 168, 108, 58, 167]. This enables design space exploration to happen before simulators or hardware prototypes become available. Another area which relies on task graphs is scheduling and schedulability analysis [145, 48]. The accuracy of the analysis results, however, is only as good as permitted by the input. Hence, it is important that the task graphs reflect the application workloads as accurately as possible.

1.1.2 Constructing task graphs

Programs are usually expressed in imperative programming languages such as C or C++. These languages partition programs into modules, functions and basic blocks that are connected by control-flow. The flow of data between program statements is implicit and must be found by analyzing the effects of statements. Task graphs, on the other hand, are partitioned into sequential sub-tasks connected by data dependencies. This makes it non-trivial to extract task graphs from source code.

Manual approach For very simple programs, it may be possible for the programmer to determine the tasks and their dependencies by hand [143]. Reasoning about data dependencies is known to be difficult and error prone. Therefore, a completely manual approach to task graph extraction is infeasible for all but the simplest programs.

Execution based approaches Instrumenting and executing the program is one alternative. This makes data dependencies occurring under a particular execution directly observable. There are two problems with this approach though. First, fine-grained instrumentation is required to find all dependencies. Such instrumentation slows down program performance by orders of magnitude. This means that profiling is only practical for small input sets. This is related to the second problem. A single profiling run does not necessarily expose all data dependencies [155]. Hence, a task graph constructed via profiling may not capture all dependencies. Dependencies must not be overlooked because they constrain the execution order of tasks. Violating this ordering may lead to incorrect results.

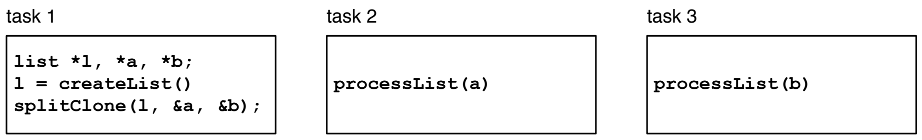
Program analysis approach A third approach to task graph extraction relies on program analysis. Program analysis approximates the effect of each source statement. The dependencies between program tasks can be found by analyzing the source code. Compared to execution-based approaches, program analysis is typically faster and finds all dependencies. While this makes program analysis the most attractive option so far, it does have a significant drawback. To remain computable, the results of program analysis are necessarily *approximate* [118]. In case of dependence analysis, program analysis over-approximates the number of dependencies. Superfluous dependencies reduce the apparent amount of parallelism in a task graph. When given such graphs as input, scheduling and DSE tools have fewer degrees of freedom and are therefore prevented from considering all feasible solutions.

The approaches outlined above lead to task graphs that either *over-approximate* or *under-approximate* the dependencies between program parts. Missed dependencies lead to correctness issues while over-approximation may cause task graph tools to miss useful solutions. Techniques to analyze embedded systems via task graphs can therefore benefit from a more precise approach to extract task graphs from the source codes of programs. The next section explains techniques that enable programmers to assist program analyzers.

1.2 Annotations and Compiler Feedback

Program analysis tools are built to analyze source code regardless of purpose or application domain. Programmers, on the other hand, typically spend most of their time working on relatively few programs. Programmers learn a great deal about programs this way. This knowledge is encoded in the choice of algorithms, data structures and symbol names.

As an example, consider the three tasks shown in figure 1.1 on the facing page. A task graph extraction tool may not be able to determine if *task 2* and *task 3* are independent. Without additional information, the tool must consider *task 2* and *task 3* as potentially dependent. This adds a superfluous edge to the corresponding task graph. The programmer, on the other hand, can determine



```
list *l, *a, *b;
l = createList()
splitClone(l, &a, &b);
```

```
processList(a)
```

```
processList(b)
```

Figure 1.1: Program fragment consisting of three tasks.

that tasks 2 and 3 are independent by interpreting symbol names and drawing on his or her understanding of the function calls involved. Program analysis on its own is therefore less potent than a combination of program analysis and programmer provided information. This makes *annotations* – declarative language constructs – attractive. In addition to annotations, code *refactoring* [62] can also make code more amenable to program analysis.

Annotations have traditionally been used by expert-programmers to allow a compiler or runtime system to perform additional optimization. This thesis explores source code annotation as a way to extract more precise task graphs from source code. Also, programmers traditionally use these annotations at their own peril. Erroneous use may cause the compiler to produce incorrect code without warning. The annotations proposed in this thesis are subject to runtime checks, which warn about incorrect use.

Annotation and refactoring take effort and must therefore be used judiciously. Ideally, annotation or refactoring is only performed:

1. when program analysis cannot determine an important program property; and

2. when annotation or refactoring helps program analysis determine said property.

Determining where programmers should modify programs was not studied in the context of task graph extraction. Rather, the focus was shifted to a related problem: finding parallelism in sequential code. The two problems are related as the analysis of data dependencies plays a central role in both instances. The change was necessary for practical reasons including the availability of tools.

The first of the above points was addressed by extending a compiler to report issues that prevent automatic parallelization. The idea is that the programmer applies a potential workaround and recompiles until the issue is no longer reported. The high volume of issues reported creates a challenge of its own.

Many of the reported issues cannot be resolved or do not improve program performance significantly.

To address the second point, the code was compiled several times in different contexts to estimate if annotation or refactoring is worthwhile. Compilers differ in their ability to automatically parallelize loops in sequential codes. Furthermore, the compilation options can greatly affect results of automatic parallelization. Multiple compilers and multiple compilation options were therefore used to compile the same code. This approach builds on two ideas. First, if one compiler reports failure to parallelize where another compiler is able to successfully optimize, the programmer may be able to resolve this issue. Similarly, if a compiler is able to parallelize a loop under potentially unsafe assumptions, then the programmer should determine whether the assumptions are safe to make or not. Together, the two techniques separate issues that may be resolved from issues that may waste programmer time.

Code can be made more amenable to analysis and optimization through improved interaction between programmers and program analyzers. The research in this thesis contributes to this important direction.

1.3 Thesis Outline

The following three chapters provide background information on areas related to the research in this thesis. Chapter 2 introduces the fundamentals of parallel computer architecture, programming abstractions and program optimization. Program analysis is also covered in more detail.

Chapter 3 concerns parallel programming. It covers the types of dependencies in programs and describes how to approximate these. Decomposition of sequential programs into tasks that may execute in parallel is also discussed. Finally, it covers manual and automatic ways to exploit parallelism.

Chapter 4 revisits embedded systems and models of embedded programs. It introduces task graphs and their role in the analysis of embedded systems. The last section of this chapter covers the correspondence between task graphs and programs parallelized with OpenMP – a parallel programming model.

Chapter 5 explains how the research in this thesis contributes to the state of the art and summarizes the major results presented herein.

Chapters 6 to 8 present the research upon which this thesis is based. Chapter 6 describes two annotations that transfer information from the programmer step to program analyzers. The impact of the directives are evaluated on benchmark kernels. Programmers need guidance on where and how to annotate. Hence, chapter 7 presents a system that provides the programmer with feedback from compiler analysis. This helps the programmer expose more parallelism to the compiler. The system is evaluated on three embedded codes. Finally, chapter 8 presents techniques to evaluate and filter the compilation feedback. This lowers the programmer effort required to address compilation feedback. The techniques are evaluated via feedback from compilation of an embedded benchmark suite. The final chapter summarizes the thesis and provides an outlook.

CHAPTER 2

Technical Background

This chapter serves as a foundation for the topics discussed in subsequent chapters. Each section covers a fundamental aspect of programming, namely: *what* we program, *how* we program and finally *where* and *how* we can improve the programs. The former two aspects are governed by the computer architecture and the programming model respectively. The latter aspects are addressed by program analysis and program optimization.

2.1 Computer Architecture

Computer architecture is concerned with the organization and design of hardware components. This includes processing elements, memories and the interconnects between these.

Until recently, processor designs were focused on sequential programs and high clock frequencies. In the period between 1978-2002, sequential performance increased approximately 50% annually. Since then, the annual growth in sequential performance has been significantly lower [125]. This slowdown can be attributed to the following observations, dubbed *walls*, in computer architecture:

Power wall Power has become a limiting factor [66]. More transistors can be put on a chip than there is power to turn on.

Memory wall Memory limits the performance of processor cores. Computation is fast but loads and stores are slow [165].

ILP wall Finally, the gains from finding more parallelism at the instruction level are diminishing [125].

Moore's law [113], which predicts a doubling of transistors on a single chip every eighteen months, is still in effect [81]. However, the transistor budget is no longer spent on optimizing sequential performance. Rather, processor designers are integrating multiple, simpler and lower frequency cores on a single chip. While the move to multicore chips addresses the power wall, it also requires programmers to move from sequential to parallel programming models.

2.1.1 Processing Elements

Flynn's taxonomy classifies computers according to the number of instruction and data streams that can be processed in parallel [61]. There are four possible combinations. Single instruction, single data stream, SISD describes a sequential computer. Single instruction, multiple data streams, SIMD, performs the same instruction on multiple data streams in parallel. Multiple instruction, multiple data streams, MIMD, is the most general class as the instruction and data streams are be fully independent. The final combination – multiple instruction, single data stream, MISD – is mostly of interest in fault tolerant processors and irrelevant to this discussion.

Figure 2.1 on the next page shows the relation between several types of *processing elements* – processors, processor cores and vector units respectively. At the outermost level, multiple processors can be combined to build a parallel system. If the processors are similar, the system is called a *symmetric multiprocessor*. Each processor can operate on a stream of instructions and data independent of other processors in the system. The system therefore supports MIMD parallelism.

Each individual processor may also support MIMD parallelism. Such processors duplicate the processor *core*, while sharing peripheral functionality such as IO and memory controllers. Such processors are called *multi-cores* and also support MIMD parallelism.

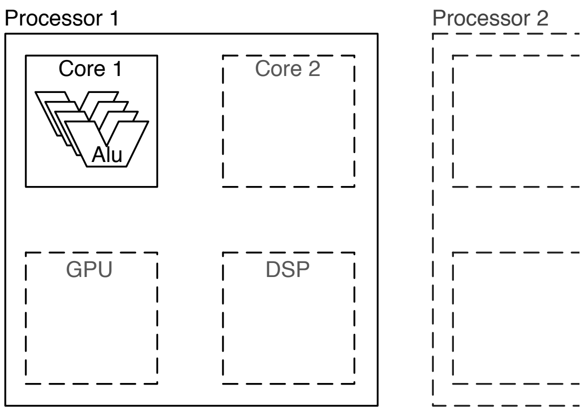


Figure 2.1: Processing elements supporting MIMD and SIMD parallelism.

Each general purpose processor core may also support SIMD parallelism via vector units. Scalar instructions operate on a single data element at a time. SIMD instructions operate on a short vector of data elements. The vector length is dependent on the size of each data element and are typically between 2 and 32 elements.

Notice that *Processor 1* in figure 2.1 contain two cores as well as two other types of processing elements. One is a *graphics processing unit*, GPU and the other is a *digital signal processor*, DSP. GPU's and DSP's are highly optimized for certain tasks but unsuitable for many others. Embedded processors typically have a heterogeneous configuration of processing elements.

2.1.2 Memory Organization

Memory runs at a fraction of the processor speed. Hence, processors do not operate directly on the contents of main memory. Rather frequently used data is kept in a hierarchy of smaller and faster memories closer to the processor. This creates multiple copies of shared data. These copies must be consistent to give each processor a coherent view of the memory. The memory coherence problem must be addressed by computer architects, compiler writers and sometimes programmers.

Caches, scratchpads and registers A *cache* is a local memory that transparently stores data so requests to the same or a nearby memory address can be sent to the processor faster. A coherence protocol can be used to maintain consistency of cached data. The exact type of memory consistency depends on the memory consistency model [1] implemented by the protocol. A *scratchpad* is another type of high-speed local memory. In contrast to caches, data is moved to and from scratchpads via data transfer instructions. They must therefore be managed by a runtime system or explicitly by the programmer. Finally, the memory closest to the processor – *registers* – is managed automatically by the compiler or manually by assembly programmers.

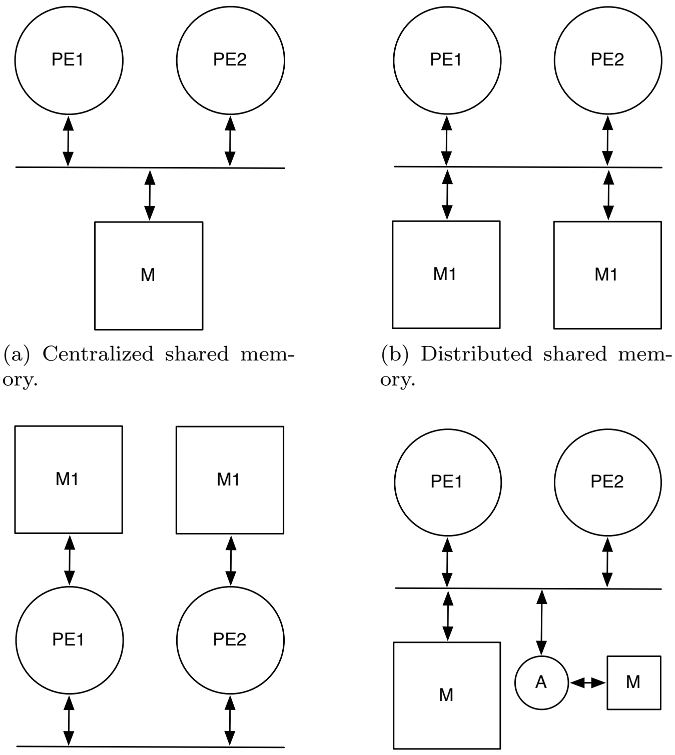
Main memory Multiple processing elements must be able to communicate among each other and with main memory. This raises the question whether the main memory should remain centralized or be distributed with the processing elements. MIMD systems is therefore further classified according to the memory architecture.

A system having a one or more shared memories is called a *shared memory multiprocessor*. There are two possible access policies when memory is distributed. With a private access policy, the memory can only be directly accessed by its associated processing element. This is known as *distributed memory*. With a shared access policy, a global address space makes the distributed memories accessible to all processing elements. This is called a *distributed shared memory*.

Processor/memory bandwidth is typically a limiting factor for the performance of shared memory multiprocessors. When memory is distributed with the processing elements, the memory bandwidth grows with the number of processing elements. Distribution of memory therefore allows systems with more processors.

The access time from a processor to a memory location depends on how close the processor is to the memory. Shared memory multiprocessors have uniform memory access times, UMA to main memory. With distributed memory, processors have non-uniform memory access times, NUMA. To mitigate NUMA effects, processors in a distributed-shared memory system have caches and a protocol to keep them coherent. These are known as cache coherent non-uniform memory access systems, ccNUMA.

Hybrid memory architectures are also possible. A system may consist of a set of homogeneous general purpose cores connected to one or more accelerators. The cores may access centralized shared memory whereas each accelerator may have its own memory. Figure 2.2 on the facing page shows four distinct memory architectures. One architecture is fully centralized, two are distributed and one is a hybrid between a centralized and a distributed architecture.



(a) Centralized shared memory.

(b) Distributed shared memory.

(c) Distributed memory.

(d) Hybrid between centralized and distributed memory.

Figure 2.2: Four different memory organizations.

2.2 Programming Models

The architecture of a parallel computer defines an interface between the software stack and the hardware that executes it. Similarly, a programming language and the libraries available define an interface between the programmer and the system being programmed.

The *programming model* is the abstraction or conceptualization of the underlying system that is presented to the programmer. The hardware/software interface presents a programming model for developers of hardware specific software such as of compilers, assemblers, operating systems and device drivers. The choice of programming language and libraries defines the programming model for application programmers. This section is about the latter kind.

Programming models differ greatly in terms of programmer productivity, safety, availability, compatibility and execution speed. The interplay between these attributes is intricate. For instance, parallel programming models must carefully balance the opacity and visibility of the system architecture – non-essential details and idiosyncrasies should be hidden while features to enable the full computational power of the hardware must be exposed. The programming model also determines how program parts executing in parallel communicate and which synchronization constructs are available.

Programming models typically support one of the following memory abstractions – *shared memory* or *distributed memory*. The memory abstraction usually reflects the memory organization of the system but it is not necessarily so. For instance, a programming model can provide the programmer with the abstraction of a distributed memory on a shared memory multiprocessor. Software distributed shared memory middleware can provide the opposite: the abstraction of shared memory on a system with distributed memory [132].

Communication through shared memory has been compared to the use of a bulletin board [50]. Information is exchanged by posting data to shared locations which are agreed upon by the sender and the receiver.

In shared memory programming models, parallelism is introduced by running multiple threads inside a single process so threads can communicate directly using ordinary loads and stores. This is possible since threads share the address space of the parent process. Typically, a single *master* thread executes the sequential tasks in the program. When several tasks can execute in parallel, the master distributes them to a set of *worker* threads including itself. This is the fork step. The master then waits before all worker threads have run to completion before executing the next sequential task. This is the join step. This is known as *fork-join* or *master-worker* parallelism.

With distributed memory programming models, communication is commonly implemented as point-to-point transfers between two or more processing elements. This is known as *message passing*. It is conceptually similar to the exchange of letters that explicitly name the sender and receiver of the information [50]. Since there is no shared address space, all exchange of information and synchronization must happen via exchange of messages.

During execution, multiple instances of the same message passing program are run with different parameters that cause each process to operate on different data. This approach is known as *single program, multiple data*, SPMD. The exchange of messages is typically done via system call or by calling a library function. Thus, a single information exchange has much higher overhead than executing a pair of load and store instructions. The programmer can amortize the communication overhead by sending a few large messages rather than many smaller ones.

The shared memory programming model is a simple extension of the sequential programming model. It allows gradual parallelization of sequential programs and no special primitives are required to communicate among tasks. The message passing programming model requires the programmer to accept more responsibilities. In return, the programmer is given more flexibility to schedule the communication and synchronization among processes.

An important difference between programming models based on message passing and shared memory is that communication among tasks is *explicit* in the former and *implicit* in the latter. With message passing, communication happen via well known function calls that send or receive messages. With shared memory programs, communication happens when accessing shared memory. Since memory which is shared is typically indistinguishable from memory that is private to each thread, the accesses that transfer data between tasks are equally hard to identify.

2.3 Program Analysis

Program analysis refers to techniques that predict safe and computable approximations to the values or behaviors observable during program execution. This enables program optimization. Program analysis is also used to detect functional defects and security vulnerabilities. To remain computable, all variants of program analysis can only provide approximate answers [76, 118]. Correctness of compiled programs is more important than efficiency. When a precise answer cannot be produced, program analysis must therefore err on the safe side. The relation between a program property and the answers computed by program analysis is illustrated in figure 2.3 on the next page.

Figure 2.3 highlights another important point. Different techniques may be used to implement the same program analysis. The implementer must therefore balance the precision delivered by an approach with its computational complexity. Modest increases in precision often comes at the cost of significantly increased computational complexity [75]. The following section introduces data-flow analysis and design choices that affect its precision and speed.

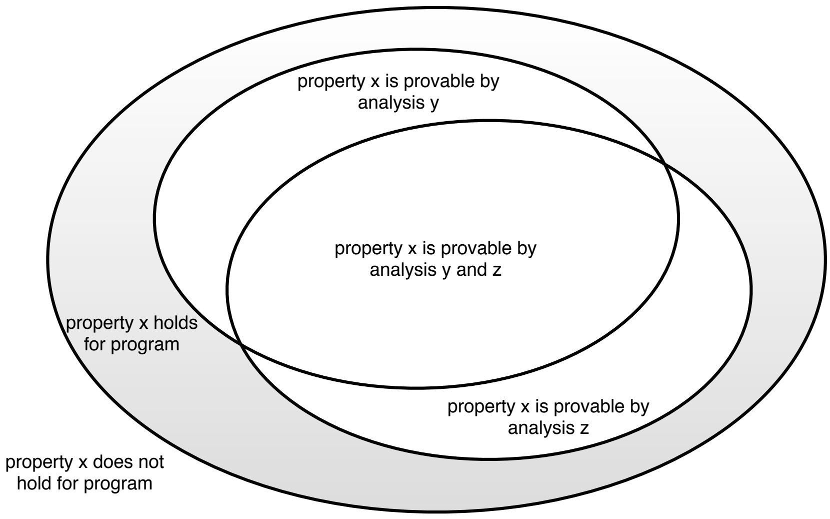

property x is provable by analysis y
property x is provable by analysis y and z
property x holds for program
property x is provable by analysis z
property x does not hold for program

Figure 2.3: Program analysis computes approximate answers regarding program properties. The level of precision and computational complexity varies with the type of analysis. A gray zone exists where all types of program analysis provides imprecise but conservatively correct answers.

2.3.1 Data-Flow Analysis

Basic blocks are sequences of instructions that do not contain branches except for the last instruction in each block. A fundamental type of program analysis computes the data-flow between basic blocks. In a forward flow analysis, the *exit state* – the program state after exiting the basic block – is formulated as a function of the state at entry to the block. This is the transfer function. The *entry state* of a basic block is a function of the exit states of its predecessor blocks. A *join* operator is used to combine multiple exit states. This creates a pair of data-flow equations for each basic block *bb*.

$$\begin{aligned} entry_{bb} &= join(\{exit_p : p \in predecessors(bb)\}) \\ exit_{bb} &= transfer_{bb}(entry_{bb}) \end{aligned}$$

The transfer function and join operation depend on the type of data-flow analysis. In a backward analysis, the transfer function and join operation translates from the exit to the entry state and the join – or meet – operator computes exit state from entry states of successor basic blocks.

An iterative algorithm is often used to solve the data-flow equations. In a forward analysis, the entry state of each basic block is approximated and the block itself is enqueued in a work list. The exit states are then computed via the transfer functions and the entry states are updated. The process continues iteratively until a *fix-point* is reached. This happens when additional iterations do not generate any new information and therefore drain the work list.

It must be guaranteed that a data-flow algorithm eventually converges on a fix-point solution. The domain of the entry and exit states must therefore be a lattice – i.e. a partially ordered set having an unique least upper bound and greatest lower bound for any two elements. Further, the transfer function and join operation must be monotonic with respect to the lattice [118].

A *reaching definitions* analysis is the textbook example of a forward analysis. It computes where definitions (assignments) of a variable are used. A *live variables* analysis is an example of a backward data-flow analysis. It detects assignments to variables that are never used. This allows dead assignments to be removed.

2.3.2 Control-Flow Approximation

Data-flow analysis is built on a representation of the program control-flow. The control-flow may be represented by the *control-flow graph*, CFG, or the *loop hierarchy tree* which itself is based on the CFG. Basic blocks form the nodes in a CFG. Each basic block is connected to its successor and predecessor blocks by directed edges that represent the flow of control.

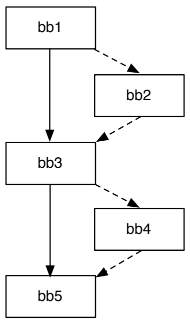
The CFG approximates the actual control-flow. Some edges represent control-flows which will never occur under execution. This happens because control-flow analysis does not analyze the conditional expressions controlling conditional branching. Figure 2.4 on the following page illustrates over-approximated control-flow edges.

2.3.3 Precision of Program Analysis

An advanced compiler performs a hundred or more optimization passes on each file. Each optimization in turn may require one or more types of program

```
if((x % 2) == 0) {
  ...;
}
...
if((x % 2) != 0) {
  ...;
}
```

(a) Source code.



(b) Paths through the code fragment.

Figure 2.4: 2.4a Simple example to illustrate over-approximation of of control-flow. 2.4b Execution can only follow paths containing both dashed and solid arrows. Program analysis will report that all the shown paths possible.

analysis. To keep compilation times acceptable, a program analysis must balance precision and execution time. This section introduces aspects of this trade-off.

Analysis scope The analysis described in the previous section is typically run on basic blocks inside a single procedure. Such types of analysis are called *intra-procedural*. The alternative is to operate on the scope of a translation unit – a set of files being compiled together – or the entire program. In such cases, the scope of the analysis may cross procedure boundaries and is therefore *inter-procedural*.

Analysis sensitivity Another dimension is the *sensitivity* of a program analysis. An analysis is said to be *sensitive* or *insensitive* to a program property if it provides distinct or summarized information about the property. An analysis is *flow*-insensitive if it summarizes information across all points in the program. Similarly, an analysis is *context*-insensitive if it summarizes information about a procedure across all calls to that procedure. Finally, an analysis is *field*-insensitive if it does not distinguish between the individual fields of aggregate objects. The time required to perform inter-procedural, context and flow-sensitive analysis is often unacceptable. Summarizing program properties lowers the analysis time at the cost of precision.

Analysis time Another important dimension is the time at which the analysis happens. Analysis can take place at compile-time, link-time, run-time or post runtime. The analysis time interacts with the scope of the analysis. For instance, an analysis of the whole program cannot happen at compile time and must be deferred to link-time. If the program analysis happens at run-time it is said to be *dynamic* and otherwise it called *static*. Dynamic program analysis directly adds to the execution time of the program. Static approaches, on the other hand, can afford to spend more time on analysis since it does not add to the execution time. Also note that program analysis at compile or link-time happens before the program is executed in contrast to analysis at run-time and post-runtime. The former two approaches are therefore unable to take advantage of information obtained during program execution. The latter two approaches can do so. Both the program source code and profiling provides information to program analysis. Additionally, the source code may contain programmer inserted annotations that assist the analysis.

2.4 Program Optimization

The primary goal of program optimization is to improve a programs use of computational resources while preserving its functionality. Some optimizations seek to *minimize* the use of resources such as memory, power, communication and execution time. Others seek to *maximize* the use of caches and the available processing elements. Finally, many optimizations – especially those transforming loops and inlining functions – have an important secondary effect: they create additional opportunities for other types of optimizations. Loop unrolling, for instance, lowers the control overhead in frequently executed loops by duplicating the loop body. This increases the basic block representing the loop body and in turn benefits the instruction scheduling optimization. Loop unrolling also illustrates the time-space tradeoff involved in program optimization. When applied successfully, loop unrolling *decreases* the execution time of the loop but it also *increases* the code size.

Hotspots and Liveness Temperature is used as an metaphor for the execution frequency of code. If an instruction or sequence of instructions are executed frequently, it is considered *hot*. Analogously, instructions which are infrequently executed are *cold*. Similarly, liveness is used to describe whether instructions are redundant or not. Instructions computing a result that is never used are said to be *dead*. Instructions that are unreachable in the control-flow of the program are also considered dead.

As a rule of thumb, programs follow the 90/10 rule [125]. For instance, 90% of the execution time may be spent executing just 10% of the code. These frequently executed parts are known as *hot spots*. Obviously, the optimization effort should target these to be effective. Hot spots can be found by profiling the program and re-optimizing the program based on the profiling data. Alternatively, hot spots can be estimated via program analysis. Basic blocks nested inside several loops, for instance, are likely to be hotter than basic blocks outside loops. Optimization can also happen at runtime and this allows optimization of instruction sequences as they become hot.

Roles of compilers and programmers Optimization can be done by the compiler, the programmer or both. While programming, the programmer can manually optimize the source code. Unfortunately, manual optimization tends to make the source code less readable. Rather than optimizing the source code directly, the programmer can insert annotations. These direct how the compiler should optimize the source code. Finally, optimization can be performed in a mostly transparent manner during compilation, link or runtime. Generally speaking, the automatic approaches do not obscure the source code and require only modest or no programmer effort. The likelihood that errors are introduced during optimization is therefore much smaller with automatic techniques.

The need for manual optimization is reduced but not removed by automatic optimizations. Automatic optimization relies on program analysis. Opportunities for optimization are necessarily missed since analysis results are approximate. In such situations, the programmer can either optimize by hand or annotate and refactor the source code to make it amenable to program analysis. Annotations are most attractive since they do not obscure the source code. Further – unlike manual optimization – it is not completely left to the programmer to preserve program correctness. Finally, annotated code is transformed automatically so errors are not introduced by the programmer at this step.

Chapter 3

Exploiting Parallelism

As mentioned in the previous chapter, multi-core processors are now commonplace. Software written for sequential processors need to be modified to to expose parallelism to multi-core processors. This chapter is concerned with the conversion of sequential programs into parallel ones. Programs contain parallelism at different levels of granularity and regularity. Some types of parallelism can be exploited automatically while others must be addressed by the programmer.

Manual and automatic approaches to exploit parallelism are covered by this chapter. It builds on the material introduced in the background chapter. Section 3.1 revisits program analysis and optimizations with a focus on automatic parallelization. Section 3.2 expands the background on programming models by introducing three concrete parallel programming models.

3.1 Automatic Parallelization

Automatic parallelization is attractive since the compiler or runtime system must shoulder the burden of finding and exploiting parallelism – rather than the programmer. Writing sequential code is hard in the first place. Programmers must address several concerns such as correctness, maintainability, flexibility, portability, readability and testability [111]. Parallel programming adds a new dimension to many of these concerns. This may explain the substantial research efforts to lower the burden of parallel programming.

With automatic parallelization, the programmer writes sequential code. A compiler or runtime system then analyzes the code to discover parallelism. This is in contrast to manual approaches where the parallel tasks are explicitly marked in the source code.

Programs can contain recognizable, implicit parallelism in one of two ways. First, programs may be constructed out of high-level primitives whose operations are known to be independent. A built-in function that sums a list is an example of such a primitive. Many existing codes, however, are not written this way. Second, many programs contain loops having no inter-iteration dependencies or which can be transformed to remove such dependencies. Many optimizing compilers are able to recognize and generate parallel code for these loops. This section covers program analysis and program optimizations that enable automatic loop parallelization and vectorization.

3.1.1 Program Analysis for Automatic Parallelization

Before a loop can be automatically parallelized by the compiler, it must undergo several types of program analysis. These include alias analysis, induction variable analysis and dependence testing. The sophistication of each analysis step varies from one compiler to another which leads to varying strengths and weaknesses among different compilers.

After introducing dependencies, this section describes three types of program analysis. When combined, they allow automatic loop parallelization. These are dependence testing, alias analysis and induction variable analysis.

Dependencies The notion of dependence is central to both automatic and manual parallelization [14, 15]. Parallel execution causes instructions to be reordered. Dependencies among instructions prevent reordering. Programs contain two kinds of dependence: *data dependence* and *control dependence*.

An operation o_1 is *control dependent* on operation o_2 if the latter determines whether o_1 should execute or not. Control dependencies originate from conditional statements and loops. Control dependence can be transformed into data dependence [123, 15]. This can increase the scope of data dependence analysis.

Two operations are *data dependent* when they access the same storage location *and* at least one of the accesses is a write. Three types of data dependence exist.

Read-after-write This dependence transfers data which typically prevents reordering. This is called a *true* or *flow* dependence. Special cases exists where reordering is still possible. For instance, true dependencies can be reordered when the sequence of updates forms a recurrence relation.

Write-after-read No data is transferred among the operations. The operations cannot be reordered since the read could then receive the wrong value. The dependence can be removed by renaming the target of one of the accesses. This is also called an *anti* dependence.

Write-after-write Like an anti-dependence, there is no transfer of data. Reordering of the operations is prevented unless the target of one of the accesses is renamed. This is also called an *output* dependence.

Anti and output-dependencies can be classified as *name* dependencies since they are arise from name (resource) conflicts rather than data transfer.

The above classification assumes that the storage locations accessed are internal to the program. When accessing variables that are visible outside the program, reordering is not legal. For instance, accesses to storage locations used for IO must be performed in program order.

Data dependencies in loops The execution time of many programs is concentrated in iterative computations such as loops. Parallelization of loops may therefore result in a significant performance increase over sequential execution. Being able to reorder the execution of loop iterations is a requirement for automatic loop parallelization. Dependencies between instructions in loops are therefore important.

Loops are different from straight line code in two respects. Loops execute the statements in their bodies repeatedly. Typically, they also access subscripted variables such as pointers and arrays. The subscripts used to access arrays are often linear functions of the loop counters. Such loops form a regular pattern of computation.

A statement can be nested inside n loops. A statement *instance* refers to the execution of the statement in a given loop iteration. Statement instances can be identified by a n-dimensional *iteration vector* where each loop corresponds to a dimension in the vector. For example, $S(1, 2)$ refers to the execution of a statement S in a doubly nested loop during the first iteration of the outer loop and the second iteration of the inner loop.

Dependencies in loops can be divided into *intra-iteration* or *inter-iteration* dependencies. The former occur between statement instances with identical iteration vectors. The latter describe dependencies between statement instances with different iteration vectors.

Dependencies between two memory accesses in a loop may also be characterized by *distance* vector. Distance vectors must be lexicographically positive. For instance, the dependence vector between a read from an array `arr[i,j,k]` and a write `arr[i+1,j,k-1]` is $(1, 0, -1)$. The *direction vector* is an abstraction of the dependence vector, which only indicates the direction of the dependence in each dimension. The direction vector corresponding to the distance vector $(1, 0, -1)$ is $(<, =, >)$.

Dependencies can be analyzed by a dependence tester during compilation or manually by the programmer [110]. They can also be analyzed via runtime layer [136, 140], hardware support [125], or a combination thereof [71, 138, 128, 161].

Dependence testing When array subscripts are affine functions of the loop iteration, dependence testing can be done by the compiler. Such a loop is shown in listing 3.1.

Listing 3.1: Example loop

```
for(i = 0; i < N; i++)
  for(j = 0; j < M; j++)
    /*  (1)           (2)        */
    data[i][j] = data[i][j-k] + 1;
```

The loop nest contains a statement with two memory accesses labeled 1 and 2 respectively. Let i_1 and j_1 be the values of variables `i` and `j` in array reference 1. Similarly, let i_2 and j_2 be the values of `i` and `j` in reference 2. Dependence testing of the loop is then equivalent to finding integer solutions to the following set of linear equalities and inequalities:

$$\begin{aligned} i_1 &= i_2 \\ j_1 &= j_2 - k \end{aligned}$$

$$\begin{aligned} 0 \leq \; & i_1, i_2 \leq N \\ 0 \leq \; & j_1, j_2 \leq M \end{aligned}$$

This is also known as a *dependence problem*. Exact solutions to such problems can be found with integer linear programming [60]. Unfortunately, integer programming is NP-complete [37]. Many approximate or specialized dependence tests have therefore been developed [16, 25, 57, 133, 126, 105].

Compilers may take a divide-and-conquer approach to dependence testing. Specialized and inexpensive tests are applied first. If any data references are unresolved after these tests, a more general and expensive test is applied. Measurements have shown that this strategy is effective since simple cases are most common [67].

The per-iteration increment or decrement of the loop counter is called the *loop stride*. Dependence tests are often based on the assumption that loops have unit strides [67]. Loops with non-unit strides must therefore be *normalized* – at least during dependence testing.

Alias analysis Programming languages lets programmers assign symbol names to memory locations. The naming method allows multiple symbols to name – or *alias* – the same memory location. Given a pair of memory accesses, an alias analysis attempts to determine their aliasing relation. There are three possibilities: *no alias*, *must alias* and *may alias*. When the alias information cannot be determined, the alias analysis returns may alias, which is conservatively correct. Other types of program analysis rely on alias analysis to disambiguate symbol names [115]. For automatic parallelization, dependence testing must use alias analysis to determine if references to pointers and arrays may alias.

The aliasing problem is undecidable [134, 137] and alias information is necessarily approximate. As a result, many different types of alias analysis algorithms exist. They differ in their level of precision and computational complexity [75]. Compilers must be able to compile millions of lines of source code in reasonable time. Hence, they often use a fast and less accurate alias analysis by default [74].

Induction variable analysis Symbolic analysis seeks to express the value of variables as functions of program input and other – *reference* – variables. Induction variable analysis is such an analysis. Induction variable analysis is an important analysis for automatic loop parallelization [163, 130] – as well as many other optimizations such as strength reduction, loop nest transformations and bounds check elimination.

The purpose of induction variable analysis is to approximate the evolution of values inside loops. Given a set of constants $c_0, c_1, \ldots, c_n$ and reference variables $r_1, r_2, \ldots, r_n$, the expression $c_0 + c_1r_1 + c_2r_2 + \cdots + c_nr_n$ is *affine*. The array subscript expressions are often affine. Representing these values as affine expressions of reference variables is therefore of special interest. Induction variables are those whose values can be expressed as a function of the iteration count of the closest surrounding loop [7]. Such functions are called *closed form expressions.*

Consider the following loop: `for(i = 0; i < N; i++) { j++; ... }`. It contains inter-iteration flow and name dependencies on `j`. The dependencies can be removed by expressing `j` as a function of i (the reference variable). This can allow automatic parallelization. Replacing induction variables with their closed form expressions is called *induction variable substitution.* Induction variable analysis is a prerequisite to this transformation.

To parallelize a loop, the number of iterations must be *countable.* A loop is countable when the number of iterations can be expressed as i) a constant; ii) a loop invariant expression; or iii) a linear function of surrounding loop counters. For countable loops, the number of iterations can be determined by analyzing the loop exit conditions. To compute the evolution of some scalars, induction variable analysis also needs to compute the number of loop iterations. This eliminates the need for a separate analysis to compute loop iterations.

A reduction is a set of updates to a single location of the form `var = var` *op* ... where *op* is a binary associative operation such as arithmetic addition and multiplication and `var` is a scalar or array. Like induction variables, reductions form a recurrence relations that must be transformed to allow parallelization. A symbolic analysis similar to that which recognizes induction variables is used to recognize reductions [9, 131].

Like alias analysis, induction variable analysis may be unable to properly analyze the code. For instance, when induction variable analysis happens, the code is already transformed by prior optimizations. This may jumble the code seen by induction variable analysis. An additional challenge when transforming induction variables is to preserve the effects of types and the behavior of overflows [130].

3.1.2 Code Transformations for Automatic Parallelization

Automatic parallelization relies on code transformation to increase performance and to facilitate further analysis.

Once program analysis has proved that parallelization of a loop nest is legal, the code can be transformed to increase performance. Parallelization can target multiple threads or a vector unit. Auto-parallelization normally refers to multi-threaded code generation. The more precise term *threadization* is used by some. *Vectorization* refers to code generation for vector units. Auto-parallelization and vectorization of the same loop or loop nest is sometimes possible.

Auto-parallelization Compilers typically transform loops using fork-join style parallelism. The compiler must insert code to start multiple threads before the loop and additional code to join the threads after the loop. The compiler must also arrange for loop iterations to be distributed among threads. OpenMP already includes such functionality. Hence, many compilers reuse parts of the OpenMP runtime for auto-parallelization.

Vectorization Two types of vectorization are possible: loop vectorization [24, 56] and vectorization of basic blocks [99, 90]. Only the former type will be discussed here.

Loop vectorization exploits SIMD parallelism by executing multiple loop iterations in parallel. Vectorization has traditionally targeted innermost loops but outer loop vectorization is also possible [120]. The number of iterations executed in parallel is called the *vectorization factor*. It depends on the width of the vector units as well as the size of the data types being operated on. For instance, a 128-bit vector unit and a loop that operates on 32-bit data types yields a vectorization factor of 4. The higher the vectorization factor, the higher is the potential speedup over sequential execution.

Vector units execute specialized vector instructions. Several loop transformations are necessary before vector instructions can be generated. Vector instructions are subject to memory alignment constraints. Accessing data not aligned on a natural vector boundary is either prohibited or carries a performance penalty. Alignment issues can be handled by special instructions to reorder or shuffle data once it has been loaded but degrades performance. Instead, alignment can be analyzed statically or dynamically and loop peeling can remove unaligned loads from the loop to be vectorized [100].

Vector loads and stores must also access consecutive, vector-sized storage locations. However, loops that operate on images or complex numbers often lead to strided access patterns. This must be addressed via platform specific instructions for data manipulation – i.e. packing and unpacking – or additional loop transformations [119].

Two additional loop transformations are applied before vector instructions can be generated for the loop. These are *strip mining* and *loop distribution* [88]. Strip mining transforms a single loop into two nested loops. The inner loop iterates over *strips* whose length equals the vectorization factor. Loop distribution splits the body of the newly created inner loop into several simpler loops so each corresponds to a single vector operation.

Facilitating transformations Control-flow in loops can prevent automatic parallelization and vectorization. Dependence tests can only analyze a limited class of loops with control-flow. Furthermore, loop vectorization requires that the flow of control does not diverge between loop iterations executed in parallel.

If-conversion transforms control dependence into data dependence [123]. It removes conditional branches and propagates the conditions to the successor basic blocks. In the successor blocks, assignment statements are changed to conditional assignment statements using the propagated condition. Finally, control-flow is removed by merging the affected basic blocks. *Loop unswitching* [115] is a related transformation. It hoists loop-invariant branches out of a loop body and replicates each execution path as an independent loop in each branch. This increases the code size but decreases the instructions executed and facilitates parallelization.

Loop normalization was mentioned in the previous section. It changes loop bounds, strides and expressions involving the loop counter so that the loop counter counts from 0 (or 1, depending on the language) in steps of one [88]. This simplifies dependence testing since it does not have to account for variations in loop bounds and strides.

The *inline expansion* or *function inlining* [115] transformation replaces a function call with the body of the callee. It lowers the overhead associated with a function call. It removes function call and return instructions as well as function prologues and epilogues. Inline expansion also enables the caller and callee to be analyzed as a single entity. This is important to allow parallelization of loop bodies containing function calls. When a dependence tester encounters a function call in a loop body, it will assume that inter-iteration dependencies are possible since the function call can have side-effects. Functions that are known to have no side effects – *pure* functions – do not prevent parallelization. With inline expansion, the function body rather than the function call becomes visible during dependence testing. It can therefore analyze the code as if there had been no function call.

The above mentioned transformation enable dependence testing. If dependencies are identified, additional loop transformations can in sometimes remove these. These transformations include loop reversal, skewing and peeling [122, 115, 88].

Profitability Auto-parallelization and vectorization adds additional computation to programs. The overhead sometimes outweighs gains from parallel execution. Hence, these optimizations are not guaranteed to be profitable. Compilers may therefore refrain from optimizing even when doing so is legal. For auto-parallelization, the overhead originates from forking and joining threads and from distributing work among threads which may require synchronization. For vectorization, if-conversion, loop transformations and vector instructions to rearrange unaligned data also create additional computation.

Compilers use a *cost model* to estimate if an optimization is profitable. Some cost models are static and work by analyzing the code at compile time and making informed guesses. Static cost modeling can benefit from program profiling information. Cost models can also be dynamic. With a dynamic cost model, the compiler may generate a sequential and a parallel version of a loop for instance. This is called loop versioning. A runtime check selects a loop version during execution. The runtime checks may analyze iteration counts, aliasing, alignment and features of the processing elements. If the time to evaluate the dynamic cost model outweighs the benefits of parallelization, performance is degraded nevertheless. As with other types of program analysis, programmers can supply information about the profitability of an optimization. The following section shows how programmers can transfer information to program analyzers.

3.1.3 Annotations and Compiler Options

There are two commonly used mechanisms with which the programmer can provide information to program analyzers. These are annotations and compiler options. Options state facts that hold for across a source file or across an entire code base. Annotations are put directly in the source code where they provide fine-grain information about individual functions, loops or variables.

The syntax used for annotations is dependent on the programming environment. Programming languages such as Java, Python and C# support annotation of classes, functions and function parameters as part of their core language syntax. Some languages also allow special source code lines – called *pragmas* in C, Objective-C and C++ – to be embedded at arbitrary places in the source code. If the compiler does not know how to interpret a pragma, it is simply ignored.

The C99 language standard – formally ISO/IEC 9899:1999 – added the `restrict` pointer qualifier[84]. Semantically, if memory addressed by a `restrict` qualified pointer is modified, no other pointer is used to access that memory [41]. The compiler can therefore assume that no dependencies exist between two accesses to `restrict` qualified pointers. Dependencies can still exist between two accesses to the *same* `restrict`'ed pointer. Programmers can use the `restrict` qualifier to rule out data dependencies that would otherwise prevent automatic parallelization. Only pointers can be qualified although objects with other data types may also alias. Several compilers also recognize the `__restrict__` syntax outside C99 mode.

Compilers also define non-standard extensions to the language syntax. In contrast to the annotation constructs defined by the programming language, the use of such extensions lead to non-portable code. The extensions, however, may prove sufficiently useful to be included in revised language standards. Several non-standard annotations help automatic parallelization. For instance, functions have side-effects if they modify global variables, perform IO operations, etc. Unless a dependence tester knows that the target of a function call is pure or constant it must assume that side-effects from the call prevent parallelization. The programmer may use non-standard annotations to specify that a function is *constant* or *pure*. A constant function may modify its parameters but no other program state. Pure functions are entirely free from side-effects.

Many compilers also support an annotation that overrides data dependence testing in part or altogether. For instance, an annotation may specify that loop iterations are independent, that no dependencies exist unless the dependence test can prove the opposite or that dependencies that exist can be ignored [46, 45, 42]. Compared to manual parallelization, these annotations let the compiler decide if and how to exploit the parallelism.

The last kind of annotation to be discussed here targets the profitability of parallelization. Compilers may recognize annotations of minimal and maximal iteration counts of loop. This helps static cost models estimate the benefit of automatic parallelization when profiling information is unavailable. Alternatively, the programmer may directly specify whether parallelization is profitable or not. Finally, the cost model may benefit from information about the target hardware. For instance, this makes runtime checks for the presence and type of vector units superfluous. Such information is passed via compilation flags.

3.2 Manual Parallelization

Manual parallelization is an alternative to automatic parallelization. To parallelize a program, the programmer must at least i) identify tasks that can be executed in parallel; ii) choose a parallel programming model; and iii) decompose the program into independent sub-tasks as prescribed by the programming model.

This section introduces sub-task decomposition and three programming models commonly used for manual parallelization: two shared-memory models and one based on messages passing.

3.2.1 Decomposition

Once parallelism has been discovered, it can be used to speed up program execution by performing independent operations in parallel. The best way to exploit parallelism depends on its type and granularity. Fine-grain *instruction-level parallelism*, ILP, can be exploited automatically. For instance ILP can be exploited by compilers via static instruction scheduling. Processors also exploit ILP via pipelining, multiple-issue, out-of-order execution and speculation [125].

Programs are naturally composed of instructions, so ILP can be exploited with no further decomposition. To exploit parallelism at the level of entire instruction sequences, the entire program must be decomposed into sub-tasks so work can be distributed among multiple processing elements. Several partitioning strategies exist. The best program decomposition varies from program to program. Often, programmers need to test several partitioning strategies to determine which works best. The partitioning strategy determines how many processing elements can be used simultaneously to speed up the computation. This is known as the *degree of concurrency*.

Programs are often decomposed using one or more of these partitioning strategies [91, 110]:

- **Data decomposition** This strategy works well when the application works on large data structures such as those found in image processing programs or scientific computing. Parallelization of loops is often based on data decomposition. This leads to a high degree of concurrency. The textbook parallel programming example – matrix multiplication – uses data decomposition.

- **Recursive decomposition** A recursive decomposition is based on the divide-and-conquer strategy. It is suitable when it is natural to decompose the overall problem into similar sub-problems which can be solved independently. Parallel sorting algorithms typically use this strategy.

- **Exploratory decomposition** This strategy is similar to recursive decomposition in its structure. In contrast to recursive decomposition, which is suitable for sorting, exploratory decomposition is often used for searching. The main difference is that with recursive decomposition, all sub-tasks must be solved. With exploratory decomposition, computation finishes once the first sub-task finds a solution.

- **Structural decomposition** Recursive and exploratory decompositions are both instances of structural decompositions and many more variants exist. Pipelining is another general approach that may be used to partition a program based on its overall structure.

A decomposition, which exposes plenty of parallelism, does not guarantee optimal program performance. The sequential code on which the parallel code was derived should be carefully optimized as well. In computers with a centralized-shared memory, parallelization increases the computing resources available to the application while bandwidth to main memory remains unchanged. It is therefore important that parallel applications are optimized to make good use of the memory hierarchy [110]. Minimizing the interactions between tasks is also important. The speedups from parallelization are easily eroded if tasks spend their time waiting to communicate with other tasks.

3.2.2 Threads

As mentioned in section 2.2, shared-memory programming models use threads to implement MIMD parallelism whereas message passing models use processes. Processes and threads differ several ways. A process can be independent while a thread is part of a process. Each process has a separate virtual address space whereas all threads in a process share its address space. This enables threads to communicate directly via shared variables. In contrast, processes must interact via system-provided inter-process communication mechanisms.

POSIX threads The *Portable Operating System Interface*, Posix, is a standardization of the interface between applications and operating system belonging to the Unix family. Most importantly Posix contains an API to create and manage threads, Posix.1c, which is also known simply as `pthreads` [78].

The `pthreads` API includes functions create, synchronize and deallocate threads. When starting a new thread, the programmer must specify its entry point by passing a function pointer to the `pthreads_create` function. To parallelize a program, the programmer must split parallel sub-tasks into new functions which serve as entry points for threads. The process of splitting a code region into a new function fn_{out} and replacing the region with a call to fn_{out} is called *function outlining*.

In parallel programs, tasks executed by different threads are not related by program order. Synchronization must therefore be used to enforce an ordering between the instructions in different tasks. The `pthreads` API provides several low-level synchronization primitives. These include mutexes, locks, semaphores and barriers [73].

3.2.3 OpenMP

A thread is a flexible but low-level primitive for parallel programming. It can be argued that threads should not be used directly by the programmer. Programmers should rather use higher-level abstractions that can be mapped to threads by an underlying runtime [103, 51].

OpenMP [152] follows that philosophy. OpenMP is a set of compiler directives and API's that lets programmers direct the compiler on how to parallelize the code. Like `pthreads`, OpenMP is a shared memory programming model based on fork-join parallelism. Unlike `pthreads`, which relies solely on library functions, OpenMP uses *directives* as the primary way to direct the parallelization. This raises the abstraction level. For instance, the programmer does not manage threads directly. For C and C++, these directives use the previously mentioned pragma mechanism.

Support for *incremental* parallelism is a key feature of OpenMP. Directives can be added gradually starting from a sequential program while retaining the possibility of sequential execution. In many cases, the compiler can generate a sequential program by simply ignoring the OpenMP directives.

Parallelization of loop nests is an area where OpenMP excels [51]. A loop is made parallel by inserting a single line – `#pragma omp parallel for`. Unlike `pthreads`, the programmer must not create a function outline for the loop. Rather, this is handled by the compiler.

The variables accessed in the loop can either be private to each thread or shared among them. The `omp parallel for` directive lets the programmer declare shared and private variables. Again, the declarative approach saves time compared to `pthreads`, which offers no assistance with variable scoping.

Finally, the programmer must schedule loop iterations among the available threads. OpenMP also handles scheduling declaratively. The `omp parallel for` directive has a `schedule` clause to select among five built-in schedules. This is also a timesaver for programmers since the schedules are non-trivial to implement.

3.2.4 Message Passing and MPI

Message passing is an important communication method since it allows the exchange of information between two logically or physically separate entities. Message passing is commonly used to write parallel programs for computers with physically distributed memories.

In its most basic form, data is transferred from a named sender to a named recipient in a one-to-one exchange. The sender performs a *send* operation and the recipient performs a *receive*. Some message passing implementations also offer one-to-many (broadcast and scatter), or many-to-one (reduction and gather) operations. The operations are typically available in both synchronous and asynchronous versions. A synchronous pair of send/receive operations define a synchronization point between the two entities. The operation requires no buffering since the sender remains blocked until the transfer completes. If the synchronous send/receive pair is not executed simultaneously, the sender or receiver must wait for the other party.

The *message passing interface*, MPI [149] is the de-facto standard API to implement message passing. MPI was developed for supercomputers and therefore emphasizes performance, scalability and portability. Several MPI implementations targets embedded systems [4, 142, 162].

While the message passing programming model allows very good scalability and flexibility, it also requires more programmer effort than shared memory models. Most importantly, the programmer must explicitly orchestrate communication. This takes substantial effort and increases the code size at the cost of readability [139, 32].

Chapter 4

Modeling Embedded Software

This chapter explains why and where task graphs are used to model embedded software. Section 4.1 motivates the need to model embedded systems. Two ways to automate embedded design are then discussed in section 4.2. Both techniques use task graphs to model the parallelism, communication, and synchronization in embedded software. Task graphs are discussed in section 4.3. Finally, section 4.4 explains the challenge of extracting task graphs from source code.

4.1 Embedded Systems

Embedded systems are seemingly similar to other types of computers. Most computers, for instance, are programmed using similar languages and tools. The focus on power efficiency and resulting transition toward parallel architectures is another similarity. Raw performance has always been less important than performance per watt for embedded systems. Now, power efficiency is a key concern for personal computers, servers and super-computers as well [11]. Yet, several aspects of embedded systems remain different from other computers.

Mobility Embedded systems are often mobile and battery powered. In contrast, stationary computers are typically connected to a power grid and can consume hundreds of watts. The power budgets of mobile devices are therefore orders of magnitude smaller than stationary devices. Moreover, the size and capacity of the battery contributes significantly to the weight and cost of the device.

Real-time requirements Embedded systems typically interact with their physical environment. Sensors collect data and actuators respond to stimuli. For instance, an embedded device can visually inspect items on a conveyor belt passing under a photo sensor. When detecting a faulty item, the embedded system triggers a mechanism to remove it from the belt. Embedded systems must therefore respond within limits imposed by their surroundings. Such systems are called *real-time* systems. For *hard* real-time systems, the response is worthless once the deadline has passed. With *soft* real-time systems, results delivered past the deadline are merely inconvenient. To ensure timely responses, upper bounds on the program execution times must be established. Such a bound is known as the *worst-case execution time*, WCET.

Specialization Embedded systems are often specialized to serve a single purpose in a highly efficient manner. In the conveyor belt example, the system needs only inspect items visually and remove faulty items. Media players, network appliances and digital cameras are other examples of highly specialized devices. In contrast, personal computers are sufficiently general and extensible to serve numerous purposes.

Cost sensitivity The number of embedded processors manufactured each year dwarfs the volume of desktop and server processors. Manufacturer of embedded chips, ARM Holdings, estimates that 4 billion ARM processors were sold during 2010. In comparison, the worlds largest maker of processors for personal computers, Intel Corporation, sold hundreds of millions desktop and server processors in the same year [160]. For high volume products, component costs contribute significantly to the overall manufacturing cost. These forces increase the specialization of processors for embedded systems. As a result, the processors used in embedded systems vary significantly in terms of cost and features.

Design window Consumer electronics have short product cycles and the competition is intense. HTC, a taiwanese maker of smart-phones, launched eighteen new hand sets on four different mobile operating systems in 2010 [158]. In such markets, shorter times to market equals higher returns on investment.

In summary, the design of embedded systems are subject to a wide range of design goals. These include energy efficiency, short and predictable response times and low component cost. Finding a good trade-off among these is a non-trivial task. As difficult as the design task may be, market forces requires that embedded systems have a short time to market. The next section, discusses ways to accelerate embedded systems design. The idea is to use heuristics to explore a search space and thereby find candidate designs faster.

4.2 Applications of Task Graphs

Researchers have sought to automate many aspects of embedded systems design. Design automation is attractive since it potentially saves time, removes human bias, generates reproducible results and scales to large problem sizes.

Analytical approaches to design automation often use task graphs as models of the coarse-grain computation and communication within embedded software. The following sections highlight two types of design automation which use task graphs as models of embedded software. The first is design space exploration at the system level and the second is static task scheduling.

4.2.1 Design Space Exploration

Embedded system designs are often quite complex. Cars, for instance, contain between thirty and a hundred microprocessors and tens of millions lines of source code to control the engine, brakes, airbags, windshields, etc. [34]. Within the domain of network processors, studies have shown that a wide diversity of designs have been used to implement the same type of system [144]. This variety may be explained by the use of experience gained from prior, recently completed project. This means that the range of feasible designs are trimmed based on previous, beneficial decisions and the designers preferences, even if they are sub-optimal for the current project. The quality of the design resulting from such ad-hoc design approaches are likely to be inferior to systematic search for – and evaluation of – feasible designs.

Evaluating a design point There are several ways to evaluate a candidate design. The evaluation can use benchmarking, simulation, analytical evaluation or a combination of these. Each approach offers distinct advantages and drawbacks. Benchmarks accurately show how select applications perform on a specific hardware platform. However, evaluation via benchmarking is only feasible when the target platform is available. Actual hardware is often unavailable early in the design process.

Simulators are typically available before the actual hardware. Simulators mimic the target platform with varying degrees of accuracy. Simulation, however, is orders of magnitude slower than hardware execution. This especially so for parallel hardware designs which are often simulated by serializing work which hardware would execute in parallel. Designers of embedded systems must choose an acceptable tradeoff between the speed and fidelity of the simulation.

Analytical methods are suitable when building a simulator is either too costly or impractical. They are also suitable when deterministic or worst-case behavior is the reasonable assumption for the system under design. When applicable, analytical methods can be used to make early design decisions by identifying corner cases of candidate designs [68].

Contrary to benchmarking and simulation, analytical approaches do not operate on the actual software or hardware. Instead, they use abstract *models*. Due to their high level of abstraction, the models can be evaluated quickly. The challenge is to make sure that the models accurately reflect the relevant characteristics of the hardware and software they represent – if not, the results of the evaluation will be unreliable.

Exploring the design space The inspiration for design space exploration originates from the field of logic synthesis. The idea is to search for solutions in a design space by systematically altering the design parameters. By altering synthesis constraints, for instance, hardware designers realized that the trade-off between area and delay could be plotted as a curve in the design space defined by area of silicon and monetary cost.

Design space exploration is used to solve high-level synthesis problems such as resource allocation and mapping of computation and communication to resources [148, 59]. During the exploration, design parameters are varied for a fixed problem description. To support decisions early in the design process, the exploration is often done at the system level [68]. Systematic exploration is often based on the *Y-chart* approach [89]. It is named after the Y-like appearance of its flow chart which is shown in figure 4.1 on the next page.

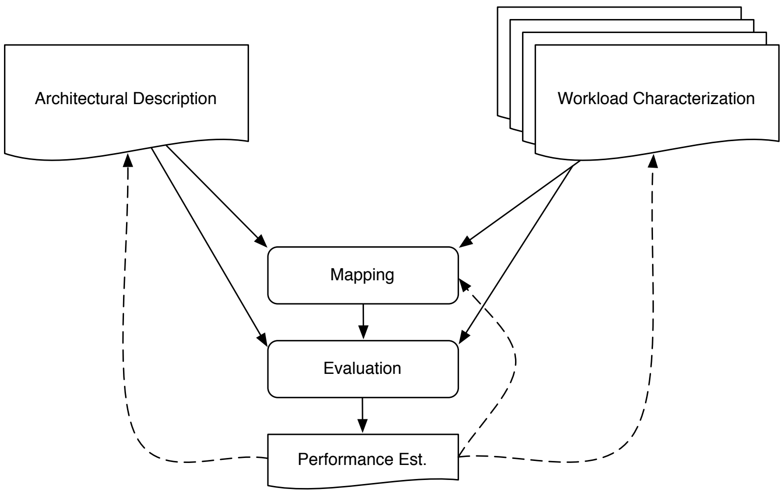


Figure 4.1: Y-chart approach to design space evaluation.

With an analytical approach to design space exploration, the description of architectural building blocks and models of application workloads are stored separately. Task graphs and variants thereof are often used to model the workloads [143, 52, 168, 108, 58, 167]. A mapping step binds tasks to the architectural building blocks defined in the architectural description. Next, the mapping is evaluated with respect to an objective function. This function may be a weighted sum of individual objectives or a true multi-objective function. Objectives include power dissipation, throughput, latency and cost. Constraints from the architecture and workload models may influence these steps. Results from the evaluation step may cause the process to be repeated. In each iteration, designers may adapt the workload descriptions, the mapping strategy and the allocation (selection) of architectural building blocks. The feedback paths are shown as dotted arrows in figure 4.1.

4.2.2 Task Scheduling and Feasibility Analysis

Allocating tasks to processors and determining their execution order is known as task *scheduling*. Task graphs are the primary model in task scheduling [145]. A task graph is *schedulable* by an algorithm A if A can produce a valid schedule for the task graph. Typically, scheduling algorithms tries to find a spatial and temporal assignment of tasks onto a target platform which minimizes the execution time of the corresponding program.

The efficiency of parallelization is measured as the speedup relative to the equivalent, sequential program. Multi-processor scheduling has a fundamental impact on the efficiency of parallelization. Finding the schedule which minimizes the execution time is NP-hard in its general form [157]. Consequently, heuristics that compute near-optimal schedules have been the subject of much research [5, 92].

Task scheduling can either be *dynamic* or *static*. With dynamic scheduling, a runtime system decides which task should execute next each time a processing element finishes executing its current task. Static scheduling, in contrast, performs processor allocation and ordering prior to execution. Static scheduling has two advantages. First, the mapping and scheduling decisions can take dependencies and communication between tasks into account. Second, since scheduling is done ahead of time, the scheduling overhead does not affect the runtime. Dynamic scheduling on the other hand, does not require that all tasks are known a priori and allows the schedule to be changed during execution.

Feasibility analysis Task graphs or variants thereof are also used for *feasibility analysis* of hard real-time systems [48]. A schedule is *feasible* if the execution of all tasks will always meet their deadlines at runtime under all permissible circumstances. A task graph tg is feasible if there exists a scheduling algorithm which can compute a feasible schedule for tg. Many variations of this problem have been studied in the literature [19, 18, 20].

4.3 Task Graphs

Parallel programs are composed of sequential parts or *tasks*. As mentioned in section 1.1.1, a task graph is an abstract but accurate representation of parallelism, communication, and synchronization in such programs. It allows tasks to be described at many levels of detail - ranging from the execution time of a task to the executable code implementing the task. A single task can represent anything from a single instruction to long instruction sequences comprised out of entire loops, basic blocks, etc. The only restriction is that there is no parallelism *within* a task – except among individual instructions. Communication and synchronization form a dependence relation among tasks. The dependence relation defines a partial ordering of the tasks. Respecting this order is necessary to preserve the correctness of the program output.

Formally, a graph is a pair $G = (V, E)$ where V is the set of vertices and E is the set of edges. Tasks are associated with the vertices of the graph and dependencies with its edges. The tasks and dependencies form a directed, acyclic

Table 4.1: Sizes of non-synthetic task graphs. Sources [156, 143, 52]

Application	Number of tasks	Number of dependencies
GSM Encoder	53	80
GSM Decoder	34	55
MP3 Decoder	16	15
Robot control	88	131
Sparse matrix solver	96	67
SPEC fpppp	334	1145
Telecom transport sys.	1072	N.A.

graph, DAG [40]. Vertices and edges can be associated with computation and communication costs respectively.

Strictness of tasks Tasks are said to be *strict* with respect to their input and output [145]. This means that, conceptually, a task cannot begin its execution before its input dependencies are satisfied. Likewise, the output generated by a task is not available until all computation within the task has completed.

Unlike control-flow graphs, the basic task graphs model contains no notion of branching. This means that a task does not control which of its successors are part of the program execution. Control structures such as `if...else` statements must therefore be encapsulated within tasks or control dependencies must be converted into data dependence which is directly represented by edges.

4.4 From Source Code to Task Graphs

While useful, task graphs are not easy to obtain. As a testament to this, many authors use synthetic task graphs such as those generated by the *task graphs for free* tool [53] or from the *standard task graph set* [156].

Some authors argue that task graphs must be manually derived from application specifications [145, 117, 166]. This approach is impractical, however. Even task graphs extracted from simple programs contain a substantial number of tasks and dependencies – see table 4.1. Others have using program analysis to extract task graphs from source code [159, 47, 141, 6, 2].

Embedded programs are usually expressed in programming languages such as C or C++. Source codes in these languages partition programs into modules, functions and basic blocks connected by control-flow. Task graphs, on the other hand, are partitioned into tasks connected by edges representing dependencies. Hence, it is non-trivial to convert source code into task graphs.

Most task graph extraction tools take sequential code as input [159, 47, 141, 6]. These approaches rely on simple automatic parallelization. These tools are therefore restricted to programs where parallelism can be discovered via program analysis. Adve and Sakellariou [2] present a tool which accepts explicitly parallel programs. Specifically, the tool analyzes scientific code written with FORTRAN and MPI. Starting from parallel programs makes sense since it decouples the parallelization process from the task graph extraction process. This leaves the choice of parallelization strategy – automatic or manual – with the programmer.

While FORTRAN is popular in scientific computing, embedded software is mostly written in a combination of C, C++ and assembly [116, 17]. Hence, the research in this thesis is concerned with programs written in C or C++ and parallelized – manually or automatically – using OpenMP directives. The OpenMP parallel programming model is important for two reasons. First, high-end mobile devices, such as smartphones and tablets, are now symmetric multiprocessors. OpenMP is a natural fit for such devices and offers several advantages over hand-threading as explained in section 3.2 on page 31. Second, unlike pthreads [33], OpenMP specifies the memory semantics an OpenMP implementation must adhere to [152, sect. 1.4.2-1.4.3]. The importance of the second point is explained in the following.

Memory consistency A *memory consistency model* [1] is a formal specification of how the memory system appears to the programmer. Programmers may reasonably expect that reads return the value that was last written to that location. In sequential programs, last is defined by program order. In parallel programs, however, instructions are only *partially* ordered – instructions executing on different processors are not related by program order. The simplest extension to the single processor model requires that memory operations appear to execute one at a time - this is called *sequential consistency* [93]. Unfortunately, this model prevents hardware and compilers from reordering memory accesses to improve performance. Several *relaxed* memory consistency models exist which allows increasingly more memory reordering.

The relaxed memory consistency model provided by OpenMP is *weak ordering* [1]. This model divides memory operations into *synchronization* operations and *data* operations. Two memory operations are ordered with respect to each other if and only if one of the operations is a synchronization operation – or a `flush` in OpenMP terminology. The weak ordering model allows each OpenMP thread to have a temporary view of memory. Each threads view of memory is not necessarily consistent with the main memory between memory synchronization operations [152]. A thread's view of memory is made consistent with the main memory by a `flush`. Apart from explicit `flush`'es, many OpenMP sections imply a `flush` operation on entry and/or exit.

OpenMP programs as sets of tasks A properly written OpenMP program cannot rely on writes by one thread to become visible to another thread before both threads have executed a `flush`. In effect, OpenMP programs can be viewed as being partitioned into a set of of tasks, each of which is a sequence of instructions delimited by implicit or explicit `flush` operations. The write operations in a task does not become visible to its dependent tasks before it finishes computing and executes a `flush` operation. This fits in with the task strictness of the task graph model. It requires that task cannot execute before all of its inputs are available and no outputs are available before its computation has finished cf. section 4.3.

4.5 Summary

Embedded systems must be energy efficient, predictable, inexpensive among other things. Finding a good trade-off between these conflicting goals within a short time frame is challenging. Analytical approaches help designers evaluate trade-offs and identify corner cases early in the design process. Application workloads are often represented as task graphs with these approaches. Extracting task graphs from source codes is challenging however. Prior work have relied on simple detection of parallelism or assume a distributed memory programming model. OpenMP is the natural programming model for embedded systems that have symmetric multiprocessor architectures. Further, OpenMP programs can be viewed as a collection of tasks delineated by memory synchronization operations. One challenge remains, the dependencies among the tasks remains implicit and must be approximated via program analysis. The approximation leads to superfluous edges in the task graphs.

Chapter 5

Motivation

The preceding chapters introduced and provided background material. This chapter motivates the research presented in the three following chapters.

5.1 The Importance of Feedback

The scientific method use observation as a way to find things out. The central idea is that observations test predictions. Observations that do not agree with the predictions causes the latter to be refined or discarded. Observations that do agree with predictions increases our confidence in the latter. Repeated observation and refinement can be thought of as a feedback loop. Feedback loops work remarkably well as a way of finding out things about computer systems.

Figure 5.1 on the next page shows several feedback loops found in software engineering [150]. The activities involve observations to determine non-functional properties such as correctness, performance and utility to end users. The three outer loops are widely practiced since they are essential to improve software. The innermost loop represents annotation and refactoring of program code based on analysis feedback. Regrettably, this process is currently reserved for expert programmers.

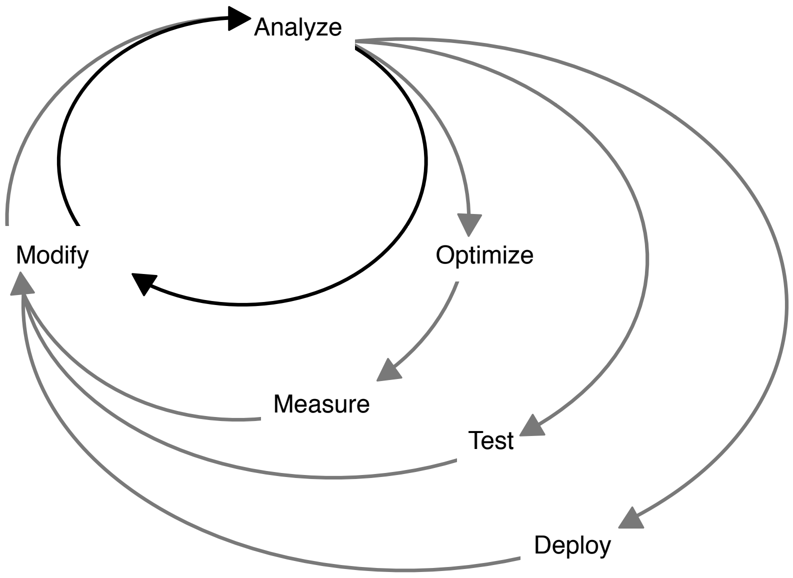


Figure 5.1: Feedback loops in software engineering.

There are multiple reasons why this is so. The programmer may not be familiar with annotations and their purpose. Also, incorrect use of annotations may lead to subtle errors without warning. Further, determining when and where to use annotations is tedious. The programmer may need to peruse lengthy compilation reports to understand analysis issues. Finally, the reports may require familiarity with compiler internals and intermediate code. This makes the reported issues difficult to understand. This prevents non-experts from removing obstacles to program analysis.

5.2 The Need for Program Annotation

The design of embedded systems is becoming increasingly complex. Similarly, programming has recently become harder since programmers must also expose parallelism. This thesis studies these two related problems for reasons given in the introduction. Both are based on program analysis. All other things being equal, increases in complexity leads to increases in development time. On the other hand, any task that can be automated by tools has the potential to shorten development time.

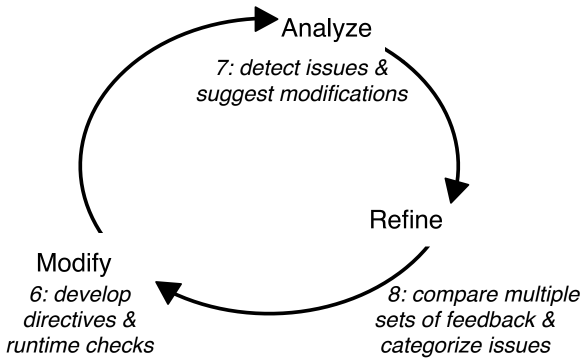

Analyze
7: detect issues & suggest modifications
Refine
8: compare multiple sets of feedback & categorize issues
Modify
6: develop directives & runtime checks

Figure 5.2: Analysis feedback loop. Italicized text shows what the following three chapters contributes to each step in the process.

Program analysis alone can only approximate properties of programs cf. section 2.3. Annotations and compiler options lets the programer mitigate limitations of program analysis. This can avoids an automation impasse but raises another problem: non-expert programmers will not use annotations effectively without proper guidance.

5.3 Making Annotation Effective

This thesis demonstrates new techniques that helps programmers annotate programs. It extends the analysis feedback loop with a new step: refinement of compilation feedback. It also improves the two existing steps. The extended loop is shown in figure 5.2. The italicized text shows the main topics of chapters 6, 7 and 8.

Chapter 6 studies how to extract task graphs from source codes of programs parallelized with OpenMP. Prior works on task graph extraction are based either on program analysis and sometimes program profiling. A hybrid approach where programmers also annotate programs is explored here. Annotation is done via two directives that assist task graph extraction tools. It is shown how annotation reduces assumed dependencies among tasks. The directives are subject to correctness checks at runtime. Catching errors early helps non-experts insert annotations.

Chapter 7 studies how issues preventing analysis can be reported to the programmer. No suitable tool to generate task graphs via program analysis was identified. Instead, issues preventing automatic loop parallelization and vectorization were studied. A production compiler was extended to report i) where issues prevent optimization; and ii) where optimization was successful. The reports or *code comments* also suggest code annotation to resolve issues. Code comments are presented as markers directly in the source code and do not expose the programmer to the compilers intermediate representation. This helps the programmer relate code comments to the relevant source code expressions.

Analyzing large amounts of source code generates a large number of code comments. Some code comments are false positives in the sense that programmer intervention cannot remove the issue. Hence, the code comments must be refined to avoid wasting programmer time. Chapter 8 refines code comments by categorizing each comment according to its chance of representing a missed opportunity for optimization. The first step is to generate multiple sets of feedback by varying the compilation options and compiling with different compilers. Code comments from different builds are then correlated using their source code locations. If a code comment reports an issue on a location where comment from another build reports an optimization, the reported issue is likely to represent a missed optimization.

In addition to the contributions that increase the accessibility of annotations and analysis feedback, the thesis is also demonstrates the efficacy of the programmer-in-the-loop approach. Chapter 6 shows that superfluous dependencies in a task graph fragment can be reduced by 69% on average. Chapter 7 demonstrates that resolving issues preventing automatic parallelization of two sequential kernels produced best-case speedups of 6x and 12.5x respectively. The performance of a JPEG decoding program was also increased by 12%. Finally, chapter 8 shows that up to 60% (43% on average) of the issues reported by a production compiler can be classified as potentially resolvable or unresolvable.

Chapter 6

Annotations for Task Graph Extraction

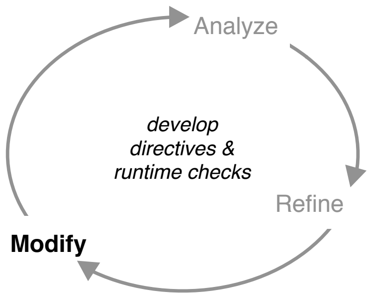
Designers of embedded systems face tight constraints on resources, response time and cost. The ability to analyze embedded systems is the key to timely delivery of new designs. As explained in chapter 4, design space exploration and scheduling tools often assume that application workloads are represented as task graphs.

Program analysis is essential to extract task graphs from program source codes. Ideally, one seeks analysis results which are precise and correct for all program inputs. Here, a task graph is considered to be correct if it has an edge between each pair of dependent tasks. A task graph is considered to be precise if it does not contain edges between independent tasks. As explained in section 2.3, *static* program analysis produces results which are *conservatively correct* but not *precise* for all program inputs. Alternatively, *dynamic* program profiling produces results which are *precise* but only correct for the program execution that generated the result. This motivated the study of a hybrid approach based on program analysis and programmer inserted annotations.

Section 1.2 described how it is possible to leverage the programmers understanding of the software annotation of the source code. Annotations provide information that improves the precision of tools to extract task graphs [159, 2].

When annotations are used correctly, runtime checks to verify this do not degrade program performance.

This chapter relates to the modification step in the analysis feedback loop. It introduces two compiler directives, which lets the programmer annotate source code with data dependencies among tasks. The directives reduce the number of assumed data dependencies that do not occur at runtime. This increases the precision of task graphs.



The correct use of the directives cannot be verified at compile time. Therefore, the correctness checks happen during execution. The intended workflow is shown in figure 6.1. Directives are validated by instrumenting and executing the program to determine if all runtime checks pass for relevant program inputs. The information in the directives can then be exploited by a task graph extraction tool.

The material in this chapter was published in three research papers. The first directive was presented at the 5th International Workshop on OpenMP[94] and the second at the 3rd Workshop on Programmability Issues for Heterogeneous Multicores [95]. Both directives are covered in an article to appear in a special issue of IEEE Transactions on Industrial Informatics [96]. This chapter adds a more precise evaluation of the number of dependencies excluded by the second directive.

The following section introduces a directive to exclude superfluous dependencies arising from approximate aliasing information. The second section, 6.2, introduces a more specialized directive to exclude dependencies in a common albeit more specific pattern of computation. Section 6.3 relates this research to prior work on task graph extraction.

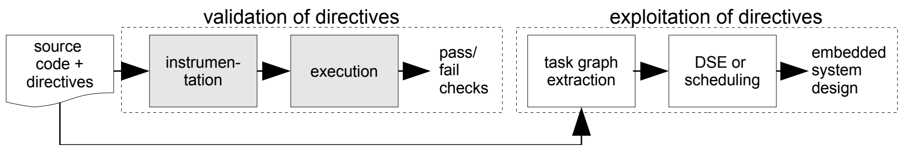


Figure 6.1: Validation and exploitation of annotations to extract task graphs.

```
int a, *b, *c, *d, *(*f)(void);
void first (int *, int *, int *);
void second(int *);
void third (int *);

void g(void) {
  /* b, c and d may alias */
  b = &a; c = f(); d = f();
#pragma omp parallel
  {
#pragma omp task
    first (b,c,d);
#pragma omp taskwait
#pragma omp task
    second(c);
#pragma omp task
    third (d);
  }
}
```

Listing 6.1: Example demonstrating potential pointer aliasing. Since the behavior of the function `f` can be arbitrarily complex, points-to analysis must conservatively assume that `b`, `c` and `d` may alias `a`. It is therefore unknown if the tasks `first`, `second` and `third` actually share data.

6.1 Dependencies among Tasks

This section starts with a motivating example to demonstrate how potential pointer aliasing leads to superfluous dependencies in a task graph. A directive which lets the programmer exclude such dependencies is then introduced in section 6.1.2. Runtime checks to detect incorrect annotations are then described in section 6.1.3 and measurements of the runtime checking overhead is presented in section 6.1.4.

6.1.1 Limitations of Alias-Analysis

Points-to analysis cannot, in general, determine if two tasks in a shared memory program communicate or not. Listing 6.1 illustrates a situation in which alias-analysis must pessimistically assume that pointers `b`, `c` and `d` may alias. This forces task graph extraction tools to generate task graphs which are imprecise in the sense that tasks `second` and `third` are assumed to be dependent even if they are not. The next section introduces a directive intended to reduce the number of these dependencies.

6.1.2 The `depends_t` Directive

Parallel programming models such as Jade [138] and SmpSs [55] let programmers declare data dependencies of a task. The declarations capture *what* data is needed by a task but not *which* task produced the data. The latter information is required to generate task graphs via program analysis.

This section introduces an annotation to declare not only which pointers provide input to a task but also which tasks produced the input. Similarly, a task that writes output to other tasks via pointers must declare which tasks can read the output.

In listing 6.2 on the facing page, the example has been updated by annotating lines 6, 10 and 14. The proposed directive is named `depends_t`. It is asserted that the task `first` provides output to `second` and `third`. This implies that that the task `second` does not provide output to `third` and that pointers `c` and `d` do not alias. Without the directives, communication between tasks `second` and `third` must be assumed due to potential aliasing of `c` and `d`.

The ability to determine which tasks communicate through pointers using the directive requires that it is used correctly. Verification via program analysis is infeasible by design so checking must be done at runtime. Pointers shared between tasks are subject to runtime checking and these will be called *checked pointers*. For efficiency reasons, certain restrictions are put upon checked pointers. As they are compared by their value, checked pointers should always hold the address of the first element of the object they point to, which is also called the *pointee*. Similarly, accesses via checked pointers must use indexing expressions rather than pointer arithmetic. Finally, pointees of checked pointers must not be or contain pointers themselves. This precludes the use of certain data structures such as trees and linked lists with the directive in its current form.

The `depends_t` directive must be placed on the line immediately before the beginning of the task which the directive applies to. The syntax of the `depends_t` directive is given below using the informal notation of the OpenMP 3.0 specification [152]:

`#pragma depends_t (`*name*`)` *[clause[[,] clause]. . .]*
task-boundary

```
 1  void g(void) {
 2    /* b, c and d may alias */
 3    b = &a; c = f(); d = f();
 4  #pragma omp parallel
 5    {
 6  #pragma depends_t first output(b,second) output(d,third)
 7  #pragma omp task
 8      first (b,c,d);
 9  #pragma omp taskwait
10  #pragma depends_t second input(c,first) \
11          output(c,fourth)
12  #pragma omp task
13      second(c);
14  #pragma depends_t third  input(d,first)
15  #pragma omp task
16      third (d);
17    }
18  }
```

Listing 6.2: Declaration of task dependencies using the proposed directive. The `output` clause on line 6 states that the task named `first` provides data for the `second` task via the pointer `b` while data for the `third` task is written via `d`. Similarly, line 10 declares that that `c` provides the `second` task with input from the `first` and output to the `fourth` (not shown).

where a valid *task-boundary* is at one of the following memory synchronization points as defined by the OpenMP specification [152, p. 291]: i) a `barrier`; ii) a `flush` with an empty flush-set; iii) an entry to or exit from `parallel`, `critical` or `ordered` regions; iv) an exit from a work sharing region without a `nowait` clause; v) an entry to or exit from a combined parallel work sharing region; vi) before or after an `omp task` scheduling point; vii) an entry to or exit from a region delineated by `omp_set_lock` and `omp_unset_lock` function calls.

name assigns a name to the task and *clause* is one of

`input` (*checked-ptr, task-name[,task-name]...*)

`output`(*checked-ptr, task-name[,task-name]...*)

`inout` (*checked-ptr, task-name[,task-name]...*)

The `input` clause takes two arguments. The first, *checked-ptr*, is a checked pointer. Each *task-name* is the name of a task that may have modified the pointee of *checked-ptr* prior to the execution of the task which the `input` clause applies to. The arguments of the `output` clause is similar except that each *task-name* identifies a task that may read the pointee of *checked-ptr* after the execution of the task which the `output` clause applies to. The `inout` clause is a syntactic shorthand and `inout(ptr,t1,t2)` is equivalent to `input(ptr,t1,t2) output(ptr,t1,t2)`.

Any pointer that points to data shared between tasks must appear in the `depends_t` directive of these tasks. A task t is only allowed to dereference a checked pointer p for reading when the following criteria are met:

- p appears in an `input` or `inout` clause in the `depends_t` statement of t; and

- any task t_{wr}, that writes to the shared pointee of p and can execute before or in parallel with t, appears in an `output` or `inout` clause in the `depends_t` directive of t. If control-flow analysis [8] cannot determine that t_{wr} may only execute after t it must appear in the `depends_t` directive of t.

The conditions for dereferencing a checked pointer for writing are analogous.

To detect when the above criteria are violated, the address held by the checked pointer is used as the index of a *permission table* which is initialized from all the `depends_t` statements in an annotated program. As previously mentioned, a checked pointer must therefore point to the beginning of the shared object and thus address arithmetic is disallowed on such pointers. An alternative solution would be to represent checked pointers as two regular pointers: one which holds first address of the task-shared object and one which can point to any part of the object. However, the use of such pointers to shared objects doubles the storage needed to hold a checked pointer and may cause problems when interoperating with program libraries.

The `depends_t` directive is assumed to be processed by a compiler like OpenMP directives are. In the experiments, however, the directives were translated to call runtime checking functions by hand. A full implementation of the necessary run-time operations was completed and benchmarked however.

Function	Output
$rd(p)$	the set of all tasks which may read shared objects pointed to by p
$wr(p)$	the set of all tasks which may write shared objects pointed to by p
$rd_{\text{deps}}(p,t)$	the set of tasks which are declared as readers of shared objects pointed to by p in the `depends_t` statement of t
$wr_{\text{deps}}(p,t)$	the set of tasks which are declared as writers of shared objects pointed to by p in the `depends_t` statement of t
$rd_{\text{pre}}(p,t)$	the set of tasks which may read shared objects pointed to by p but never execute after or in parallel with t
$wr_{\text{post}}(p,t)$	the set of tasks which may write shared objects pointed to by p but never execute before or in parallel with t

Table 6.1: Auxiliary functions to explain the functioning of the runtime checks.

6.1.3 Runtime Checking the `depends_t` Directive

The use of the directive must be runtime checked and this section describes this effort. The checks ensure that a task will not dereference a checked pointer which is not declared as dependence. This section will also discuss a set of library calls which can be emitted by a compiler to detect such incorrect use of the annotations.

Aliasing relationships between pointers can be inferred from the directives. In listing 6.2 on page 53, for example, lack of communication between tasks `second` and `third` requires that pointers `c` and `d` do not alias. For practical reasons, runtime checks of aliasing was not studied. However, detecting if two pointers alias is inexpensive as it amounts to a comparison of two scalars.

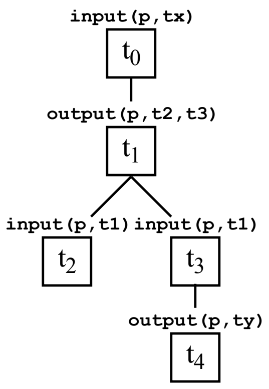
Let the function $rd(p)$ map a checked pointer p to the set of all tasks which declare that they may read input via p. Each task and pointer pair $\langle t, p\rangle$ is mapped to the set of tasks which are declared as readers of input via p in the `depends_t` statement of t by the function $rd_{\text{deps}}(p,t)$. Finally, $rd_{\text{pre}}(p,t)$ computes the set of tasks which are readers of shared objects through p but may only execute before the first execution of t. Analogous definitions for writes through checked pointers are shown in Table 6.1.

Before a task t reads the pointee of p it must be checked that

$$t \in rd(p) \wedge wr(p) = wr_{\text{deps}}(p,t) \cup wr_{\text{post}}(p,t) \tag{6.1}$$

and similarly, it must be checked that

$$t \in wr(p) \wedge rd(p) = rd_{\text{deps}}(p,t) \cup rd_{\text{pre}}(p,t) \tag{6.2}$$



a

$rd(p) = \{t_0, t_2, t_3\}$ $\quad$ $wr(p) = \{t_1, t_4\}$
$rd_{\text{pre}}(p, t_1) = \{t_0\}$ $\quad$ $wr_{\text{post}}(p, t_3) = \{t_4\}$
$rd_{\text{deps}}(p, t_1) = \{t_2, t_3\}$ $\quad$ $wr_{\text{deps}}(p, t_3) = \{t_1\}$

Output check of p before its pointee is written by t_1:
$t_1 \in wr(p) \wedge rd(p) = rd_{\text{deps}}(p, t_1) \cup rd_{\text{pre}}(p, t_1) \rightarrow$
$t_1 \in \{t_1, t_4\} \wedge \{t_0, t_2, t_3\} = \{t_2, t_3\} \cup \{t_0\}$

Input check of p before its pointee is read by t_3:
$t_3 \in rd(p) \wedge wr(p) = wr_{\text{deps}}(p, t_3) \cup wr_{\text{post}}(p, t_3) \rightarrow$
$t_3 \in \{t_0, t_2, t_3\} \wedge \{t_1, t_4\} = \{t_1\} \cup \{t_4\}$

b

Figure 6.2: a) Program fragment showing the control-flow between five tasks that share data through a pointer p. b) Example runtime checks which apply (6.1) to p in t_3 and (6.2) to p in t_1. Although both of tasks t_1 and t_4 write to the pointee of p, tasks t_2 and t_3 need only declare an input dependency on t_1 because t_4 is always executed after t_3. Similarly, the `output` clause of t_1 need not mention t_0 as it always executes before t_1.

before the pointee of p is written to by a task t. An concrete example is shown in figure 6.2. Since the second part of the conjunction is the same for all tasks which read or write p, the necessary dependency information is stored in a global hash table in which each pair of bit-vectors $\langle rd(p), wr(p)\rangle$ is indexed by p.

A runtime check is simply a lookup on p in the dependency hash table followed by one bit-vector membership test and one bit-vector comparison corresponding to the left and right-hand side respectively of conjunction (6.1) or (6.2). Bit-vectors are represented as machine word sized elements in a fixed size array meaning that operations on bit-vectors can be supported efficiently.

To determine where runtime checks should be performed, the OpenMP memory consistency model must be taken into account. The task-boundary points listed in the section 6.1.2 implies a `flush` operation. Consequently, given a conforming OpenMP program, a single runtime check inserted between each `flush` and dereference of a runtime checked pointer p is sufficient to detect violations of a task dependency declaration.

Besides the runtime checks themselves, the following operations, which update the dependency hash table, was implemented.

`register_input(p, input_tasks)` Sets the initial value of $rd(p)$ in the dependency table. Called at program start-up to register checked, file-scope pointers. It is also called at runtime to set dependencies of runtime checked pointers to dynamically allocated memory.

`register_output(p, output_tasks)` Sets the initial value of $wr(p)$. Otherwise it is similar to `register_input`.

`update_input(p, input_tasks)` Updates $rd(p)$ in the dependency table such that the new value equals $rd(p) \cup input\_tasks$. Called to update the input dependencies $rd'(p_1) \leftarrow rd(p_1) \cup rd(p_2)$ when a runtime checked pointer p_1 is assigned the value of a runtime checked pointer p_2.

`update_output(p, output_tasks)` Updates $wr(p)$. Otherwise it is similar to `update_input`.

`unregister(p)` Removes $rd(p)$ and $wr(p)$ in the dependency table. Called immediately after dynamic memory deallocation.

Accesses to the dependence table need to be synchronized to avoid data races. We currently use a read/write lock [49] provided by the POSIX threads [78] implementation on our platform. This means that the runtime checks can happen concurrently by acquiring the lock for reading. The register, update and remove operations may need to update the dependency table and must therefore acquire a write lock before doing so. The unregister operation always acquires the lock for writing. Since the time to acquire the lock for reading is several orders of magnitude slower than a lookup in the dependency table, the synchronization overhead is amortized by only acquiring the lock for reading once for all necessary runtime checks at a task-boundary.

6.1.4 Results for the First Directive

We evaluate three aspects of the first directive: i) the cost and scalability of each of the runtime operations ii) the programmer effort required to insert directives in a benchmark; and iii) the performance impact of adding runtime checks to the benchmark.

Experimental setup Minimum, maximum and average execution times are calculated from ten consecutive runs on an otherwise idle machine to compensate for variations between benchmark runs. Only the parts of the benchmark performing parallel work and which can contain runtime checks are included in the execution time. To determine if the cost of runtime checking has a statistically significant impact on the execution efficiency, the average execution times of programs with and without runtime checks is compared using two-sided, unpaired t-test using 95% confidence intervals.

The benchmarks were compiled with `llvm-gcc` version 2.5 whose front-end and OpenMP implementation are based on `gcc` version 4.2.1. Optimization level `-O2` was used for all experiments.

The quality of the code generated by `llvm-gcc` was compared to that of `gcc`. This was done to ensure that the use of the LLVM optimizer did not do a poor job thus lowering the impact of the instrumentation. The experiments show that the code generated by `llvm-gcc` is faster or as fast as code generated by `gcc` 4.2.1.

Experiments were performed on a dual-socket server having two quad-core 2.93 GHz Intel Xeon 5570 CPUs and a total of 12 GB DDR3 RAM. It had 256 KB L2 cache per core and 8 MB shared L3 cache per CPU. The operating system was Linux using the 2.6.36 kernel. The measurements we will present were obtained using eight threads but similar results were observed for experiments with one, two and four threads.

A secondary set of experiments were performed on a dual-core Intel Atom D525 powered machine with a clock frequency of 1.8 GHz and 2 GB DDR2 RAM. It had 512 KB L2 cache per core and used Linux with the 2.6.32 kernel. The Atom based measurements mirror the Intel Xeon based measurements. Only the measurements obtained on the Xeon based system are presented.

Runtime operations The runtime operations for the `depends_t` directive were benchmarked and the results are shown in figure 6.3 on the next page. Each operation was executed repeatedly to increase execution times well above the timer resolution.

The measurements of the operations that may insert a key into the dependence table are split into two cases: key *present* and key *not present*. To compare the cost of the runtime operations with the cost of entry and exit of a task-boundary, the performance of the OpenMP `parallel` construct is also included in the graph. It was measured using the EPCC micro-benchmark suite [27].

The `check_input` and `check_output` takes at most two microseconds. Since the cost is dominated by the time taken to acquire the lock for reading, this value is also a good indicator of the total cost incurred by all checks at a task-boundary

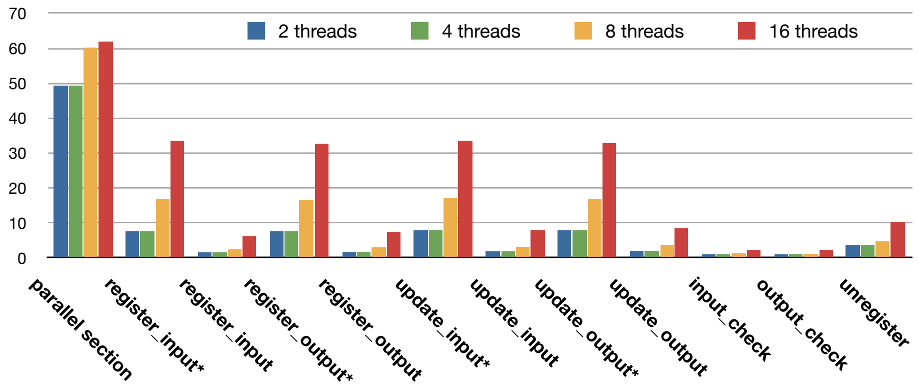


Figure 6.3: Time in microseconds taken by a single execution of each runtime operation for the `depends_t` directive and the OpenMP parallel directive as a function of the number of threads. All operations performing a hash table lookup are dependent on the lookup key. The measurements where the lookup key is not present in the hash table is marked with an asterisk (*).

such as entry to a `parallel` section. Thus, the added cost of performing runtime checks at task-boundaries is most likely insignificant.

Replacing the read/write lock with read-copy-update synchronization [112] would allow read operations to become lock-free. As read-only runtime checks typically execute much more frequently than the other operations, this could significantly lower the overhead and improve the scalability of the runtime checks.

The register and update operations which do not need to insert a key into the dependence table take at most eight microseconds and reflect the costs of modifying a checked pointer which is already registered with the runtime. Unregistering a shared object takes slightly longer but is only necessary when deallocating an object which itself is typically a costly operation.

Finally, the operations which need to insert a key into the dependence table under a write-lock have the highest execution time. Updating the dependence table takes up to thirty three microseconds when sixteen threads are used. However, these more costly operations occur only in context of memory allocation which itself is a costly and thus infrequent operation.

Integer sort The `depends_t` directive applied to the source code of the integer sort kernel, IS, in the NAS parallel benchmarks [13, 85] to determine the impact on programming effort and runtime performance. The IS program per-

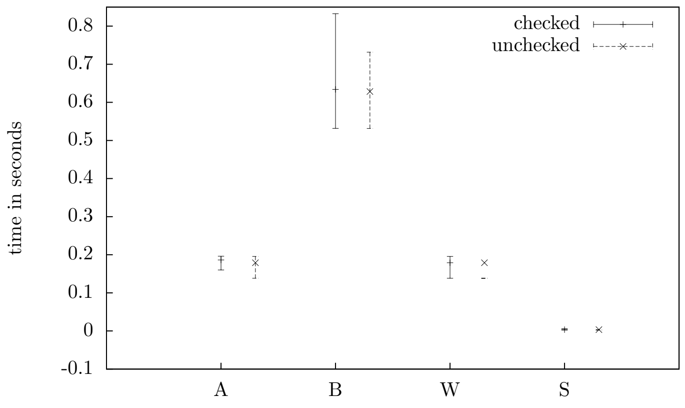


Figure 6.4: Comparison of the maximal, minimal and average running times of *checked* and *unchecked* builds of the NPB integer sort with eight threads and four different workload sizes as shown on the x-axis.

forms integer sort on a large array and tests integer computation as well as communication performance [13]. The workload contains data-level parallelism in several loops.

The set of task boundaries was first identified. Then the set of runtime checked pointers was identified and `depends_t` directives were added. Finally, the calls required to register, update and check task dependencies were added by hand as we have yet to automate this step.

The source contained 4 `parallel` sections, 5 work-sharing sections which corresponded to 13 task boundaries where `depends_t` directives were inserted. There were 8 runtime checked pointers. It was therefore necessary to insert 7 calls to `register_input`, 8 to `register_output`, 3 to `update_input`, 1 to `update_output` and finally 19 and 11 calls to `check_input` and `check_output` respectively. It was not necessary to call `unregister` as no memory deallocation occurs. By a rough estimate, the effort to annotate the integer sort required less than half a working day.

No statistically significant differences between the average running times of the binaries with and without runtime checks were observed.

6.2 Dependencies among Stencil Computations

The previous section introduced a directive to eliminate assumed dependencies among tasks which arise from the approximate nature of alias analysis. This section presents a directive which eliminate superfluous dependencies among tasks representing a data-parallel pattern of computation known as a *stencil computation*.

The section is organized as follows. Stencil computations are introduced in section 6.2.1. An example stencil computation then described in section 6.2.2. Generating task graphs from stencil computations and the challenges of doing so is the subject of section 6.2.3. An annotation to addresses these challenges is introduced in section 6.2.4 and the runtime support required to verify correct use of this annotation is described in section 6.2.5. Section 6.2.6 presents experimental work that shows that the runtime checks have a neglible impact on the execution time.

6.2.1 Stencil Computations

A *stencil computation* is a kernel – or computational pattern – which updates a grid of elements by sampling neighbor elements according to a fixed pattern, called a stencil. Several stencil computations are often performed one after another. Each stencil computation is a sweep across a target grid whose non-boundary elements are updated by combining neighboring values in the source grid. The grid is commonly represented as a single or multi-dimensional array. Each array element can be updated independently of other elements in the array. Hence, stencil computations contain significant amounts of regular, data-level parallelism.

Sequences of stencil computations are commonly found in embedded signal and image processing codes. They typically apply a number of filters to raw or previously encoded data streams. For instance, stencil computations are found in four out of six kernels and in four out of twelve applications in the University of Toronto Digital Signal Processing benchmark suite, UTDSP [102]. Stencil computations are also of importance in scientific computing. For instance, they are used to construct solvers of varying complexity and with a wide range of applications. Examples include solvers of partial difference equations which model heat diffusion, electronmagnetic or fluid dynamics problems [114].

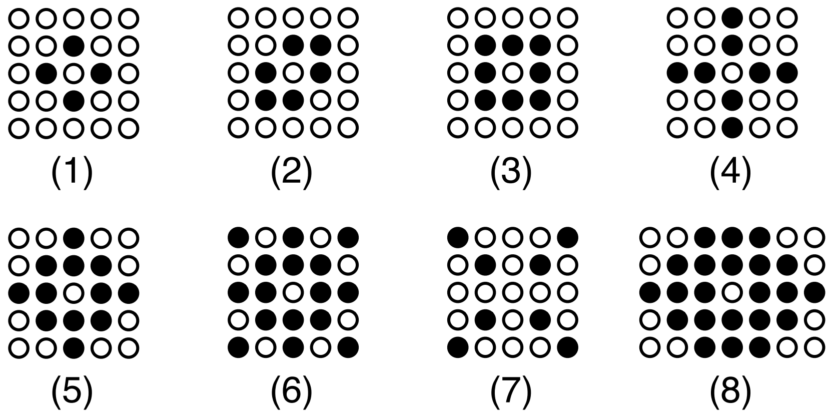


Figure 6.5: Stencils commonly encountered in embedded image processing codes. The patterns were reported by Ramanujam et al. [135] except from (7) which is from Bouchebaba et al. [26].

Stencil patterns The pattern of the stencil determines which elements are included in the nearest neighbor-computation and the size of the grid boundaries. Figure 6.5 shows eight two-dimensional stencils which are commonly encountered in source codes for embedded systems [135].

Stencil width A one-dimensional stencil accesses array elements in the order of increasing array indices. In the following, the *width* of a one-dimensional stencil is defined as the difference between the lowest and highest numbered index of the elements accessed in a single iteration. Multi-dimensional stencils sweep over grids that have rows, columns, planes, and so forth. The width of a multi-dimensional stencil is defined as the width of the one-dimensional stencil obtained by i) projecting the outermost dimension array dimension onto a line; and ii) computing the width of the resulting, one-dimensional stencil.

The most narrow stencils are three elements wide. Such stencils are said to be *compact*. In contrast, stencils having widths greater than three are classified as *non-compact*. The source code of the UTDSP Benchmark Suite contains stencils that are one and two-dimensional and symmetric. The stencil widths are 3, 4, 8, 15, 32 or 256 elements.

6.2.2 An Example Stencil Computation

The challenge of extracting task graphs for codes consisting of stencil computations will be studied in the context of a concrete example. The example that will be used is due to Mattson et al [110, p. 166]. The example and how it is parallelized will be explained in this section.

The example stencil computation models temperature changes along an infinitely narrow pipe. The two ends of the pipe have different temperatures which remain fixed throughout the computation. The program simulates the gradual change in temperature over time for the rest of the pipe. The temperature change in the pipe is captured by differential equation (6.3).

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} \tag{6.3}$$

To solve the problem efficiently, the problem space is discretized by representing the pipe, U as an array of values and simulating the progress of time as a sequence of discrete time steps. Consequently, the program only needs to store the state of U at time steps k and $k+1$. These arrays are called `uk` and `ukp1` respectively. A stencil computation is used to compute the change between two successive time steps.

Each iteration in the stencil computation updates the value of a point in `ukp1`. The updated value is computed by sampling five neighbor elements in `uk` in the following way

```
ukp1[i]=uk[i]+(dt/(dx*dx))*
  (-1/12*uk[i-2]+4/3*uk[i-1]-5/2*uk[i]+4/3*uk[i+1]-1/12*uk[i+2]);
```

where `dt` and `dx` are the differences between two discrete steps in time and space respectively. The stencil width is five. The relevant parts of the heat diffusion example are shown in listing 6.3 on the following page. The inner loop on lines 14-18 implements the stencil computation.

Parallelizing the stencil computation The heat diffusion example has been parallelized using OpenMP directives to distribute work among multiple threads. The iterations of the outer loop on line 8 represents time steps in the simulation and must execute in sequence. The iterations of the inner

```
 1 double *uk   = malloc(sizeof(double) * nx);
 2 double *ukp1 = malloc(sizeof(double) * nx);
 3 double *tmp, dx, dt;
 4 /* further initialization omitted. */
 5
 6 #pragma omp parallel
 7 {
 8   for (int k = 0; k < nsteps; ++k) {
 9     /* instruct compiler to distribute
10      * iterations among the available threads
11      * according one of the five schedule types.
12      */
13     #pragma omp for schedule(SCHEDULE)
14     for (int i = 2; i < nx - 2; ++i) {
15       ukp1[i] = uk[i] + (dt/(dx*dx)) *
16         (-1/12 * uk[i-2] + 4/3 * uk[i-1] - 5/2 *
17         uk[i] + 4/3 * uk[i+1] - 1/12 * uk[i+2]);
18     }
19     /* "copy" ukp1 to uk by swapping pointers */
20     #pragma omp single
21     {
22       tmp = ukp1;
23       ukp1 = uk;
24       uk = tmp;
25     }
26   }
27 }
```

Listing 6.3: Heat diffusion example. The code was adapted from Mattson et al. [110]

loop on line 14, however, are independent and may execute in any order. The parallelization is based on data decomposition such that each thread operates on separate parts of the iteration space. Each such part is called a *chunk* in OpenMP parlance. As a thread executes a chunk of the iteration space, it accesses consecutive elements of the source and destination arrays and these are also called chunks.

The `#pragma omp parallel` directive on line 6 causes the OpenMP runtime to create a team of threads which then collaborate to execute the work enclosed in the parallel region. Specifically, the runtime will create as many threads as there are logical processors in the system unless a specific thread count was requested by the programmer.

When the team of threads reach the `#pragma omp for` directive on line 13, the OpenMP runtime will distribute the loop iterations from 2 to `nx-2` among the available threads according to the thread identifiers and the schedule clause. The choice of schedule is controlled via a preprocessor definition in the example. The decision could also have been hard coded or deferred to the `OMP_SCHEDULE` environment variable by specifying `schedule(runtime)` on line 13.

Finally, the `#pragma omp single` protects a region that should only be executed on a single thread. The directive causes the first thread to execute the region swapping the pointers to the arrays; the remaining threads will simply skip the region.

Parallelization of multi-dimensional stencil computations The heat diffusion stencil computation in listing 6.3 on the preceding page is one-dimensional. When the stencil computation has n dimensions, the source and target grids are typically swept via n nested loops. For this work, it is assumed that only the outermost loop of stencil computations parallelized. This is a reasonable assumption whenever the outermost loop contains enough iterations to distribute work to all processing elements.

6.2.3 Modeling Stencil Computations as Task Graphs

This section discusses i) how to compute the number of data dependencies between two successive stencil computations; ii) why a program analysis may assume that more dependencies exist than may occur at runtime; and iii) how that affects task graphs generated via program analysis.

Figure 6.6 shows two successive stencil computations named sc_1 and sc_2 respectively. The loop iterations of each stencil computation are partitioned into a set of tasks. When parallelized with the OpenMP `for` directive, the partitioning is controlled via the `schedule` clause [152, p. 43]. A task t in sc_2 writes a chunk of elements. To do so, it reads elements which were written by tasks in sc_1. The task t is therefore dependent on one or more predecessor tasks in sc_1.

The number of predecessor tasks in sc_1 a task t in sc_2 may depend on are calculated as follows. Assume that sc_2 performs nearest neighbor computations using a stencil whose width is $stencil\_width_2$. Each time a task t writes a single value, it reads $stencil\_width_2$ elements from the source array. If it writes two

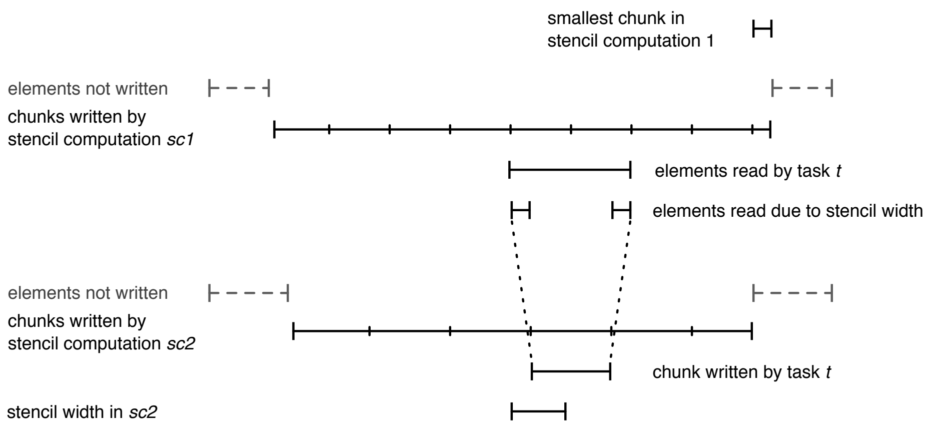

smallest chunk in stencil computation 1
elements not written
chunks written by stencil computation *sc1*
elements read by task *t*
elements read due to stencil width
elements not written
chunks written by stencil computation *sc2*
chunk written by task *t*
stencil width in *sc2*

Figure 6.6: A stencil computation sc_1 followed by another sc_2. During execution of sc_2 a task t writes a chunk of elements. To do so, it must read chunks which were written during the execution of sc_1.

elements it reads $stencil\_width_2 + 1$ elements from the source array. Hence, if t writes $chunk\_size_2$ elements, it will read $chunk\_size_2 + stencil\_width_2 - 1$ elements.

The question is then how many predecessor tasks could have written the elements read by t. In the worst case, the first element was written by one predecessor task while the remaining $chunk\_size_2 + stencil\_width_2 - 2$ elements were written by other predecessor tasks. If each predecessor task in sc_1 wrote at $chunk\_size_1$ elements, a task t depends on at most
$1 + \lceil (chunk\_size_2 + stencil\_width_2 - 2)/chunk\_size_1 \rceil$ predecessor tasks. The ceiling operator is necessary to round up when division yields a fraction. The result is rounded up since a task t in sc_2 is data dependent on a task t_{pred} in sc_1 even if the output of t_{pred} is only partially read by t.

To compute the *maximal* number of dependencies from tasks in sc_1 to a task t in sc_2, the equation must be updated to use the *maximal* chunk size in sc_2 and the *minimal* chunk size in sc_1. Equation (6.4) calculates the maximal number of data dependencies for a task t in sc_2.

$$max\_dep = 1 + \left\lceil \frac{max\_chunk\_size_2 + stencil\_width_2 - 2}{min\_chunk\_size_1} \right\rceil \tag{6.4}$$

The following discussion is concerned with the calculation of the minimal and maximal chunk sizes needed to evaluate equation (6.4). It will be assumed that two successive stencil computations, sc_1 and sc_2, are parallelized such that the loops have unit strides, use the same OpenMP schedule and will be executed on the same number of threads. These assumptions are reasonable as the workload characteristics of the two stencil computations are similar. Both the maximal and minimal chunk size depends on the `schedule` clause.

OpenMP schedules The OpenMP 3.0 specification defines five schedules [152, p. 43–44]. They are: `static`, `dynamic`, `guided`, `runtime` and `auto`. The first three schedule types allows the user to request a chunk size. For instance, the clause `schedule(static,4)` causes the OpenMP runtime to distribute iterations among threads in chunks of four if enough iterations are available.

Each OpenMP runtime may implement the schedules differently. Hence, the following discussion is necessarily implementation specific. It is based on `libgomp` which is the OpenMP runtime library distributed with `gcc`.

In `gcc`, `schedule(auto)` is handled as `schedule(static)` internally and it will therefore not be treated separately in the following. The schedules distribute work as outlined in the following where thr is denotes the number of threads, n the number of iterations and c is the requested chunk size:

`static` Distributes iterations among threads such that each thread receive a single chunk. All chunks are of roughly the same size. If tr divides n evenly, each chunk will contain n/thr iterations. If this is not the case, the first $thr - 1$ chunks contain $n/thr + 1$ iterations and the last chunk contains the remaining iterations which are $n - ((n/thr + 1) * (thr - 1))$.

`static`, c When a chunk size is requested for the `static` schedule, the runtime sizes each chunk accordingly as long as there are enough iterations to do so. If c does not evenly divide n, the last chunk contains $n \bmod c$ iterations.

`dynamic`, c Whereas the `static` schedule distributes work according to thread IDs, the `dynamic` schedule distributes the iterations in the order they are requested by the threads. If no chunk size is requested, the default is one. The chunk sizes are calculated similarly to the static schedule.

`guided`, c The goal of this schedule is to divide the work into fewer chunks than the `dynamic` schedule while retaining the ability to allocate work to threads dynamically. Each chunk size is proportional to the number of iterations remaining divided by the number of threads executing the parallel region. If a chunk size c is requested, each chunk except the last will contain at least c iterations. If no chunk size is requested, the default is one.

Unlike the preceding schedules, the size of each chunk computed by the `guided` schedule is dependent on the size of the previous chunk. This means that the minimal chunk size can only be found by computing the size of all chunks distributed by the `guided` schedule.

`runtime` This is a pseudo-schedule that defers the choice of schedule until runtime. The actual schedule must one of the preceding three or `auto`. It is set using an environment variable a call to the OpenMP runtime. If no schedule is specified via either of these mechanisms, the default is to use a `static` schedule with chunk size one.

Maximal chunk size Equation (6.5) calculates the maximal chunk size $max\_chunk\_size_2$ in sc_2. It is assumed that the loop iterates n_2 times and that iterations are distributed among thr threads according to a schedule $sched$.

$$max\_chunk\_size_2 = \begin{cases} \frac{n_2}{thr} + 1 & \text{when } sched \text{ is } \texttt{static} \text{ or } \texttt{auto} \\ c & \text{when } sched \text{ is } \texttt{static,c} \text{ or} \\ & \text{when } sched \text{ is } \texttt{dynamic,c}. \\ max(\frac{n_2 + thr - 1}{thr}, c) & \text{when } sched \text{ is } \texttt{guided,c} \end{cases} \tag{6.5}$$

Equation (6.5) was obtained by inspecting how the `libgomp` runtime implements the OpenMP schedules. When a chunk size is specified and the schedule is not `guided`, no chunk is bigger than what was requested. When the schedule is `guided` or `static`, the maximal chunk size is roughly in inverse proportion to the number of threads.

Minimal chunk size Similar to the previous discussion, it is assumed that the parallel loop in sc_1 iterates n_1 times and that these are distributed among thr threads according to a schedule $sched$. Equation (6.6) computes the minimal chunk size when $sched$ is either `static` or `dynamic`. When $sched_1$ is `guided`, the chunk sizes depends on the order in which threads request iterations. The minimal chunk size is therefore not known until sc_1 completes execution.

Like the formula to compute the maximal chunk size, equation (6.6) was obtained by inspecting the OpenMP schedule implementations in `libgomp`.

$$min\_chunk\_size_1 = \begin{cases} n_1 - (\left\lceil \frac{n_1}{thr} \right\rceil \times (thr - 1)) & \text{iff } sched \text{ is } \texttt{static} \\ c & \text{iff } sched \text{ has a chunk size } c \text{ which divides } n_1 \\ n \bmod c & \text{otherwise} \end{cases} \tag{6.6}$$

Why dependencies are approximated Equation (6.4) on page 66 calculates the maximal number of data dependencies for a task in a stencil computation following a previous stencil computation. The source code properties required to evaluate equation (6.4), (6.5), and (6.6) are:

- $stencil\_width_2$ The stencil width of the second stencil computation.
- n_1 and n_2 The number of loop iterations in the first and second stencil computation.
- thr The number of threads that loop iterations should be distributed among.
- $sched$ The value of the OpenMP `schedule` clause – including the chunk size parameter `c`.

Each of these values are required to evaluate equation (6.4). However, the values may or may not be known statically, i.e. at analysis time. What follows is a list of reasons why one or more values are likely to be unavailable to a task graph extraction tool:

Stencil width The stencil width may be dynamically adjustable unlike the heat diffusion example where the stencil width was fixed. If the stencil is used to blur an image for instance, it is practical to allow the user to choose the amount of blurring. In such cases, the stencil width is likely controlled by a variable whose purpose is only understood by humans. A task graph analysis must conservatively assume that the stencil may be arbitrarily wide. In that case, each task at step $k + 1$ will read all chunks written in step k. Since these chunks were written cooperatively by the set of tasks at step k each task at step $k + 1$ may depend on each tasks at step k.

Iteration counts The program is very likely to parameterized to perform stencil computations on a grid with varying dimensions. In such cases, the upper bounds of the loops that sweep across the grid must be held by variables at runtime.

Thread count The `#pragma omp parallel` and `#pragma omp parallel for` directives let the programmer request a specific number of threads. Seemingly this clause can make thread counts available at compile time. However, a number of threads specified with this clause is just one of several factors that determine the size of the team of threads which will execute the parallel region [152, p. 35–36]. Other factors include the nesting level of the parallel region. Furthermore, it is possible to dynamically adjust the number of available threads or conditionally execute a parallel region on a single thread. So even when present, the number of threads requested does not necessarily reflect the actual number of threads.

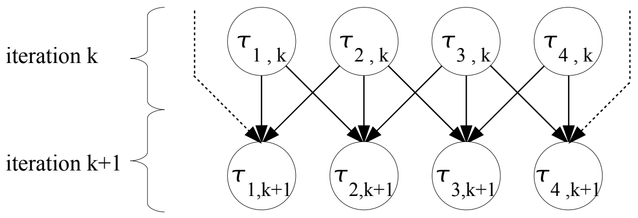
Second, it may not be a good idea to hard code the thread count in the source code as the number of available threads varies across hardware platforms. The number of the threads which maximizes program performance may also depend on the the program input.

Schedule It is possible to defer the choice of actual schedule by using the `runtime` pseudo schedule. Choosing a schedule at runtime enables the programmer to vary schedule choices for different hardware platforms without recompilation. It is also possible to leave the choice of schedule to the compiler by selecting the `auto` schedule.

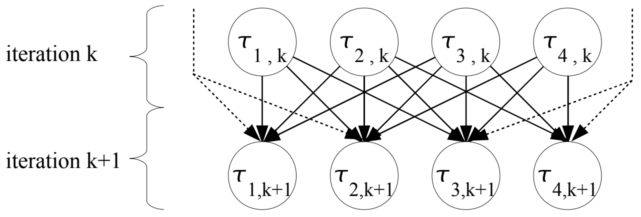
Chunk size Finally, some schedule types takes an optional chunk size parameter. This parameter can be symbolic and therefore unknown before runtime.

Task graph with and without approximation Figure 6.6 on page 66 showed how a task t in a stencil computation sc_2 reads elements written by tasks in a previous stencil computation sc_1 to write a chunk of elements. The task graph representing sc_1 and sc_2 therefore contains a number of dependencies from t to tasks in sc_1. Equation (6.4) computes the maximal number such dependencies. As explained in the previous section, at least one of the source code properties required to evaluate the equation is most likely unknown before run time. A task graph extraction tool must therefore conservatively over-approximate the number of dependencies.

The task graph which accurately reflects dependencies between two steps in a stencil computation is shown in figure 6.7a on the next page. The task graph resulting from over-approximation of dependencies among tasks and predecessor tasks is show in figure 6.7b on the facing page.



(a) Desired task graph for iterations k and $k + 1$ of heat diffusion example when executed on four threads numbered from 1 to 4 when using the `static` schedule. The dashed arrows are dependencies on the task that initialized the arrays (not shown).



(b) Same task graph fragment with over-approximation of dependencies among tasks.

Figure 6.7: Task graph fragment which shows actual and over-approximated dependencies among tasks. 6.7a: A fragment of the task graph corresponding to two successive stencil computation steps in the heat diffusion simulation. The dependencies shown are computed under the assumption that the number of iterations is at least twice as big as the number of threads. Program analysis, however, can usually not prove that each thread will receive more than one iteration of the inner loop because the values necessary to do so are unavailable at compile time. It must therefore produce the task graph fragment shown in 6.7b which over-approximates dependencies between each task and its predecessors.

6.2.4 The `depends_sc` Directive

The preceding section explained what source code properties must be known to calculate dependencies between tasks in two stencil computations and that these properties may not be known until runtime. However, the programmer can make additional information available at compile time via annotations.

One could imagine an approach where the allowable ranges of the stencil width, schedule choice, iteration and thread counts were specified individually. However, obtaining these values might be burdensome. The alternative is an annotation where the programmer specifies only a single number: the maximal number of data dependencies from a task to predecessor tasks. This number is dependent on the number of array chunks which a single task will read while writing a new array chunk. By capturing this information, the over-approximation of task dependencies shown in figure 6.7b on the previous page can be removed.

A variant of the `depends_t` directive introduced in section 6.1.2 on page 52 can be extended to capture the highest number of source chunks a task will access during a step of a parallelized stencil computation. The second directive is called `depends_sc`. The syntax of this directive in the informal notation of OpenMP [152] is:

`#pragma depends_sc`*[*`max_dep(`*expr*, *ndep*`)` ...*]*

The first argument, *expr*, must evaluate to an array or a pointer to an array and *ndep* is the maximal number of data dependencies on predecessor tasks – equal to the maximal number of chunks a task may read from *expr*. The directive can only be used with an `#pragma omp parallel for` or `#pragma omp for` directive and must be placed on a line directly before either of these directives.

6.2.5 Runtime Checking the `depends_sc` Directive

Like the `depends_t` directive, the use of the `depends_sc` directive should be checked to guard against human error. Section 6.2.3 argued that the source code properties to evaluate equation (6.4) on page 66 are unlikely to be available at compile time. Checks must therefore be performed at runtime. This section will explain how checks are performed.

The programmer asserts the maximal number of data dependencies from a task to predecessor tasks with the `depends_sc` directive. To check the assertion, equation (6.4), which computes the same number, is evaluated at runtime. To detect when the directive is used correctly, it is checked if the number of dependencies calculated via equation (6.4) is higher than the asserted number.

Incorrect assertions When the number of dependencies asserted with the directive is lower than what is observable at runtime, an error is raised. The program can be written to continue after an error is detected or simply terminate, which is the default behavior. To avoid termination on runtime errors, the programmer may register a callback function. It is invoked when an incorrect use of the directive is detected. The callback can then take appropriate action, e.g. the error can be logged and program execution can then continue.

```
 1 #pragma omp parallel
 2 {
 3   for (int k = 0; k < nsteps; ++k) {
 4     #pragma depends_sc max_dep(uk,3)
 5     /* instruct compiler to distribute
 6      * iterations among the available threads
 7      * according one of the five schedule types.
 8            #pragma omp for schedule(SCHEDULE)
 9     for (int i = 2; i < nx - 2; ++i) {
10       ukp1[i] = uk[i] + (dt/(dx*dx)) *
11         (-1/12 * uk[i-2] + 4/3 * uk[i-1] - 5/2 *
12         uk[i] + 4/3 * uk[i+1] - 1/12 * uk[i+2]);
13     }
14     /* "copy" ukp1 to uk by swapping pointers */
15     #pragma omp single
16     {
17       tmp = ukp1;
18       ukp1 = uk;
19       uk = tmp;
20     }
21   }
22 }
```

Listing 6.4: Parallel region of the heat diffusion code with the `depends_sc` directive added on line 4 such that it applies to the innermost loop on lines 10–14.

A runtime error implies that a task graph based on the information in the directives does not match observable program behavior. Subsequent runtime errors should be avoided by modifying the `depends_sc` directive which caused the runtime checks to fail. Any task graphs based on the incorrectly used directive should also be updated to match the new use of the directive. This scheme of dynamically checking a property of a program and raising an error on detected violations is similar to the implementation of bounds checking in Java, C# and many other languages. Likewise, an unhandled out-of-bounds exception will terminate such programs.

The following sections will discuss how to determine stencil width, maximal and minimal chunk size at runtime since these values are required to evaluate equation (6.4)

Determining stencil width The way to obtain the stencil width depends on how the program is written. In the heat diffusion code, the stencil width can be computed by analyzing the accesses to `uk`. The accesses are `uk[i-2]`, `uk[i-1]`, `uk[i]`, `uk[i+1]` and `uk[i+2]` respectively. Clearly, accesses are to five contiguous elements so the stencil width can readily be determined by analyzing the source code. In other cases, the stencil width is harder to determine from the source code. Consider the two following sets of array accesses in a two dimensional stencil computation:

1. `a[i-1, j]`, `a[i+1,j]`, `a[i,j]`, `a[i,j-1]` and `a[i,j+1]`
2. `a[idx-W]`, `a[idx+W]`, `a[idx]`, `a[idx-1]` and `a[idx+1]`
   where `idx = i * W + j`

The stencil patterns are identical but expressed in different ways. The first set of accesses is into a two dimensional array, and its clear that the stencil symmetric and has width is three in either direction. The second set of accesses is into a one dimensional array so the indexing expressions have been *linearized.* The programmer knows that the variable `W` is the width of a row, so the expressions `a[idx-W] a[i-1,j]` reference the same element. An automated analysis does not know that the variable `W` holds the row width and therefore cannot accurately compute the stencil width.

Furthermore, it may also be difficult to determine the stencil width when the stencil computation uses a stencil, which is computed dynamically based on user input. Listing 6.5 on the facing page shows a one dimensional stencil computation with a user selectable stencil width. There, the stencil width equals the number of iterations of the inner loop which is $center - (-center) + 1$ or simply $2 \times center + 1$. Whether this kind of stencils and their widths can be recognized and analyzed statically with reasonable effort is an open question.

There are two ways to ensure that stencil widths can be properly detected. One is to guide the programmer to write code such that stencil widths are analyzable. For instance the programmer could be asked to avoid linearizing array indexing expressions. An alternative solution is to add a `stencil_width` clause to the `depends_sc` directive. This also effectively sidesteps the difficulties of detecting which variable control the width of dynamically sized stencils.

Determining maximal and minimal chunk size The minimal and maximal chunk sizes are either observed or calculated using equations (6.5) and (6.6) depending on the schedule as follows:

```
#pragma omp for
for(r=0;r<rows;r++){
  for(c=0;c<cols;c++){
    dot = 0.0;
    sum = 0.0;
    for(cc=(-center);cc<=center;cc++){
      if(((c+cc) >= 0) && ((c+cc) < cols)){
        dot += (float)image[r*cols+(c+cc)] *
          kernel[center+cc];
        sum += kernel[center+cc];
      }
    }
    tempim[r*cols+c] = dot/sum;
  }
}
```

Listing 6.5: A one dimensional stencil computation which smoothes an image using a user selectable stencil width. The code is part of a Canny edge detector[29] implemented by M. Heath [72].

`static` min_chunk_1 and max_chunk_2 are observed since each thread is assigned a single chunk of iterations when no chunk size is requested.

`static`, c min_chunk_1 and max_chunk_2 are calculated based on iteration counts and chunk sizes.

`dynamic`, c Handled similar to the previous case.

`guided`, c In this case max_chunk_2 can be computed using equation (6.5) but min_chunk_1 cannot be calculated in a single step. Thus, each chunk size n_{cs} calculated by the OpenMP runtime is observed to find min_chunk_1.

`runtime` Both min_chunk_1 and max_chunk_2 are observed as chunk sizes are calculated requested by threads.

When the schedule is `static` or `dynamic`, the runtime check of a `depends_sc` directive takes constant time, whereas `guided` and `runtime` schedules incurs an overhead proportional to the number of chunks.

Inserting runtime checks during compilation A plug-in for the `llvm-gcc` compiler was developed to insert runtime checks. The `llvm-gcc` is a combination of the `gcc` front-end and the Low-Level Virtual Machine, LLVM, compiler back-end [101]. The LLVM compiler was selected since it was designed with extensibility in mind.

With `llvm-gcc`, a compilation unit is translated into object code as follows. First the source code is parsed and lowered into an intermediate representation, IR. Second, the IR is transformed as OpenMP directives are processed. Code in parallel sections is outlined and calls to the OpenMP runtime are inserted to distribute loop iterations to threads. Finally, optimization passes are run on the intermediate code and machine code is generated.

The runtime checks must use variables added to the code during the processing of OpenMP directives. Therefore, the plug-in to insert runtime checks is run after processing of the OpenMP directives and before subsequent optimization passes.

Figure 6.8 on the next page illustrates how the compiler transforms code annotated with an OpenMP `for` directive and how runtime checks are inserted. The modifications are shown as a simplified, source-to-source transformations for clarity, but are implemented as transformations of the LLVM IR.

6.2.6 Results for the Second Directive

We evaluate three aspects of the `depends_sc` directive: i) the programmer effort required to insert the proposed directives; ii) the performance impact of the runtime checks; and iii) the ability of the directive to exclude data dependencies that would otherwise have been assumed to exist by a task graph extraction tool. Three codes are used for the evaluation: the heat diffusion example from section 6.2.2 on page 63, and two embedded image processing benchmarks.

The directives ability to exclude superfluous dependencies is estimated as follows. The number of dependencies asserted with the directive is compared to a conservative estimate. The estimate is obtained by evaluating equation 6.4 under the assumption that min_chunk_1 is one – which is conservatively correct. max_chunk_2 is computed via equation (6.5) for non-`guided` schedules with requested chunk size `c`. Finally, $stencil\_width_2$, did not need to be approximated since the benchmark codes use fixed width stencils. The number of directives added in the source code is used to approximate the required programming effort.

The benchmarks were executed on the same two systems that were used to evaluate the `depends_t` directive – see 6.1.4. Once more, the graphs show the

```
#pragma omp for schedule(static)
for(i=lb;i<ub;i+=incr)
 /* <loop body> */
```

(a) Parallelized loop where iterations are distributed according to the `static` schedule and no chunk size.

```
#pragma omp for schedule(ls,c)
for(i=lb;i<ub;i+=incr)
 /* <loop body> */
```

(b) Parallelized loop where iterations are distributed according to any other schedule.

```
/* slice begin, slice end */
long sb, se;
if(loop_start(
 lb,ub,incr,"static",0,&sb,&se) {
 for(i=sb,i<se;i+=incr)
  /* <loop body> */
 }
}
loop_end();
```

(c) Code from figure 6.8a after the compiler processes an OpenMP directive with a `static` schedule.

```
/* slice begin, slice end */
long sb, se;
if(loop_start(
 lb,ub,incr,ls,c,&sb,&se)) {
 do {
  for(i=sb,i<se;i+=incr)
   /* <loop body> */
 }
 while(loop_next(&sb,&se));
}
loop_end();
```

(d) Code from figure 6.8b after the compiler processes an OpenMP `for` directive with any schedule except `static` with no chunk size.

```
long sb, se;
if(loop_start(
 lb,ub,incr,"static",0,&sb,&se) {
/* RUNTIME CHECK */
 _check_chunk(sb, se, ndep);
 for(i=sb,i<se;i+=incr)
  /* <loop body> */
 }
}
loop_end();
```

(e) Code from figure 6.8c instrumented with a runtime check. Each thread calls `_check_chunk` to check its chunk size.

```
long sb, se;
/* DYN. OR STATIC CHECK*/
if(loop_start_wrapper(lb,ub,
 incr,ls,c,&sb,&se, ndep)) {
 do {
  for(i=sb,i<se;i+=incr)
   /* <loop body> */
 } /* GUIDED OR RUNTIME CHECK*/
 while(loop_next_wrapper(&sb,
  &se, ndep));
} loop_end();
```

(f) Code from figure 6.8d with runtime checks. A call to either `loop_start` or `loop_next` was wrapped.

Figure 6.8: Processing of OpenMP directives and subsequent insertion of runtime checks. All schedules except `static` with no chunk size are handled in the same way so two different cases must be handled. Variables `sb` and `se` are OpenMP control variables and store the beginning and end of each chunk of iterations. `ndep` is the maximal number of dependencies asserted by the programmer via the `max_deps` clause of the `depends_sc` directive.

Table 6.2: Memory usage and speed-up of the benchmark applications when using the `static` loop schedule. All times are in seconds and memory consumption is in megabytes.

Name	Sequential time	Relative speed-up		Memory use
		2 threads	4 threads	
Heat diffusion	1.3781	1.9	3.1	2
Demosaicing	0.8980	1.9	3.5	221
Edge detection	0.0082	1.7	3.0	0.2
Corner detection	0.0047	1.7	2.6	0.3

performance for eight threads but similar results were observed for experiments with one, two and four threads.

Like the experiments with the first directive, only the parts of the benchmark performing parallel work and which can contain runtime checks were included in the execution time. The scalability of the parallelized parts of the benchmark applications is shown in Table 6.2.

Heat diffusion simulation The heat diffusion code was used as a micro-benchmark. It executes the same stencil operation iteratively for a number of time-steps so the inserted runtime checks are exercised repeatedly.

The inner loop of the heat diffusion example was annotated with a single `depends_sc` directive as shown in listing 6.4 on page 73. Using the directive rather than a conservative estimate reduces the number of assumed dependencies by 2 to 257 depending on the chunk size as shown in Table 6.3 on the facing page. This corresponds to a relative improvement between 33% and 99% with an average improvement of 74%.

The heat diffusion simulation was run for 2000 time steps with an array of 131072 `double` values. The array sizes were set such that all data structures fit in the last-level caches of the CPU's.

The running times of the heat diffusion simulation with and without runtime checks inserted are shown in Figure 6.9 on page 80. There were no statistically significant difference between the average running times of the binaries with and without runtime checks inserted.

The executions using the `dynamic` schedule and small values of the chunk size parameter have significantly higher execution times when compared to the other

Table 6.3: Actual dependencies, dependencies asserted via directive and conservative estimate of dependencies for the heat diffusion simulation with various OpenMP schedules.

Schedule	Dependencies			Difference	
	Actual	Asserted	Estimated	Abs.	Rel.
`static`,2 `dynamic`,2	3	4	6	2	33%
`static`,4 `dynamic`,4	3	3	8	5	62%
`static`,8 `dynamic`,8	3	4	12	8	67%
`static`,16 `dynamic`,16	3	3	20	17	85%
`static`,64 `dynamic`,64	3	3	68	65	96%
`static`,256 `dynamic`,256	3	3	260	257	99%
Average	3.0	3.3	62.3	59.0	74%

executions. This is because the `dynamic` schedule incurs a synchronization overhead each time a slice of iterations is mapped to a thread and the number of slices is in inverse proportion to the chunk size for the `dynamic` schedule.

Figure 6.9 on the following page also shows a significantly higher worst-case execution time for un-instrumented builds for `runtime` and `static` schedules. Even when performing two or more warm-up runs before calculating average execution times, we saw that whichever build was executed first was also most likely to show a slightly higher worst-case execution time.

Image demosaicing A key digital photography workload is demosaicing which interpolates sensor data from a color filter mosaic. We used code developed for an embedded MPSoC [26]. It uses several two-dimensional stencils which are from three to five elements wide.

Six successive stencil computations were annotated with a `depends_sc` directive each. The actual, asserted and estimated number of dependencies between a single pair of stencil computations are shown in table 6.4 on page 81. When the chunk size is two, the directive is no more precise than a conservative estimate of the number dependencies. For all other chunk sizes, the number of assumed dependencies is reduced by 3 to 257 depending on the chunk size as shown in table 6.3. This corresponds to a relative improvement between 50% and 99%. The average improvement for all tested chunk sizes is 65%.

A 21 mega-pixel image with a resolution of 5616x3744 pixels was used as input. This input results in a memory consumption of approximately 221 MB, which will not fit into the caches.

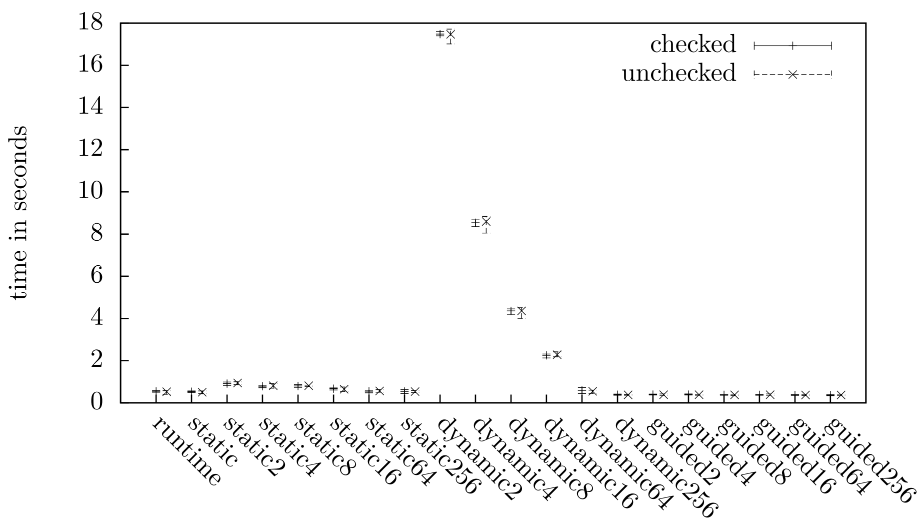


Figure 6.9: Comparison of the maximal, minimal and average running times of *checked* and *unchecked* builds of the heat diffusion simulation with eight threads and various selections of schedule and chunk size. The labels on the x-axis refers to the `schedule` clause.

The results are shown in figure 6.10 on the next page. There were no statistically significant differences between the average running times of the binaries with and without runtime checks inserted.

A slight increase in running time can be observed when increasing values of the chunk size parameter from 16 to 64 to 256 for the `static` schedule. The effect also increases slightly when using fewer threads. We used hardware performance counters to measure cache misses for the different schedule settings. We found that L2 cache misses increases with the chunk size which we believe is explained by a corresponding increase in unsuccessful L2 hardware prefetches. Using chunk size 16 shows around 200K unsuccessful prefetches whereas a chunk size of 256 results in more than 350K unsuccessful prefetches.

Edge and corner detection in images Detecting edges and corners in an image is an important step in machine vision. Two benchmarks, *susan.edges* and *susan.corners*, from the MiBench suite [69] was studied. The large reference input consisted of 384x288 grayscale values which consume 108 KB of memory. Since two buffers are used in susan.edge, its working set is approximately 216

Table 6.4: Actual dependencies, dependencies asserted via directive and conservative estimate of dependencies for the image demosaicing program with various OpenMP schedules.

Schedule	Dependencies			Difference	
	Actual	Asserted	Estimated	Abs.	Rel.
`static`,2 `dynamic`,2	3	4	6	0	0%
`static`,4 `dynamic`,4	3	3	8	3	50%
`static`,8 `dynamic`,8	3	4	12	6	60%
`static`,16 `dynamic`,16	3	3	20	17	85%
`static`,64 `dynamic`,64	3	4	68	64	94%
`static`,256 `dynamic`,256	3	3	260	257	99%
Average	3.0	3.5	61.3	57.8	65%

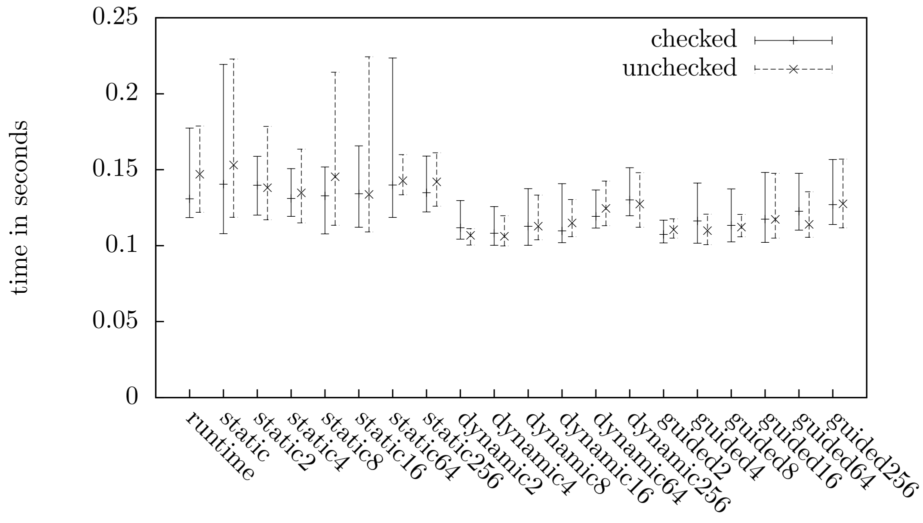


Figure 6.10: Comparison of the maximal, minimal and average running times of *checked* and *unchecked* builds of the demosaicing program with eight threads and various selections of schedule and chunk sizes.

KB. Susan.corners use three buffers and therefore has a working set of around 322 KB, which makes the large reference input fit within the L2 caches.

Both benchmark were annotated with two `depends_sc` directives each. A rough estimate of the time required to insert `depends_sc` directives is about an hour per benchmark or less. Table 6.5 on the following page shows that using the

Table 6.5: Actual dependencies, dependencies asserted via directive and conservative estimate of dependencies for susan.edges with various OpenMP schedules.

Schedule	Dependencies			Difference	
	Actual	Asserted	Estimated	Abs.	Rel.
`static`,2 `dynamic`,2	6	6	10	4	40%
`static`,4 `dynamic`,4	4	7	12	5	42%
`static`,8 `dynamic`,8	3	9	16	7	44%
`static`,16 `dynamic`,16	3	4	24	20	83%
`static`,64 `dynamic`,64	3	4	72	68	94%
`static`,256 `dynamic`,256	2	12	264	252	95%
Average	3.5	7.8	66.3	58.5	63%

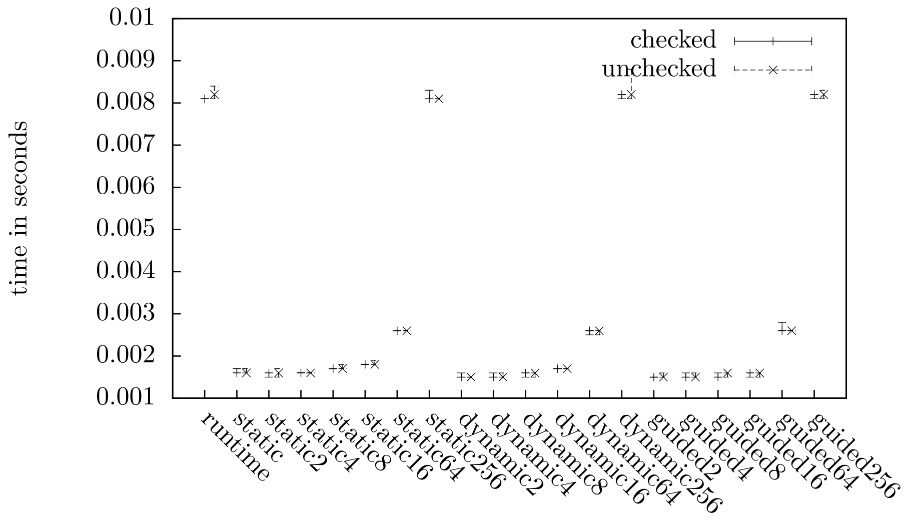


Figure 6.11: Comparison of the maximal, minimal and average running times of *checked* and *unchecked* builds of susan running edge detection using eight threads and various schedules and chunk sizes.

directive rather than a conservative estimate reduces the number of assumed dependencies in susan.edges by 4 to 252 depending on the chunk size. This corresponds to a relative improvement between 40% and 95% with an average improvement of 63%. Similarly, table 6.6 on the next page shows that the directive eliminates 5 to 252 assumed dependencies in susan.corners. The relative improvement is between 42% and 95% with an average of 73% for susan.corners.

Table 6.6: Actual dependencies, dependencies asserted via directive and conservative estimate of dependencies for susan.corners with various OpenMP schedules.

Schedule	Dependencies			Difference	
	Actual	Asserted	Estimated	Abs.	Rel.
`static`,2 `dynamic`,2	7	7	12	5	42%
`static`,4 `dynamic`,4	5	5	14	9	64%
`static`,8 `dynamic`,8	3	4	18	14	78%
`static`,16 `dynamic`,16	3	8	26	18	69%
`static`,64 `dynamic`,64	3	5	74	69	93%
`static`,256 `dynamic`,256	2	14	266	252	95%
Average	3.8	7.2	68.3	61.1	73%

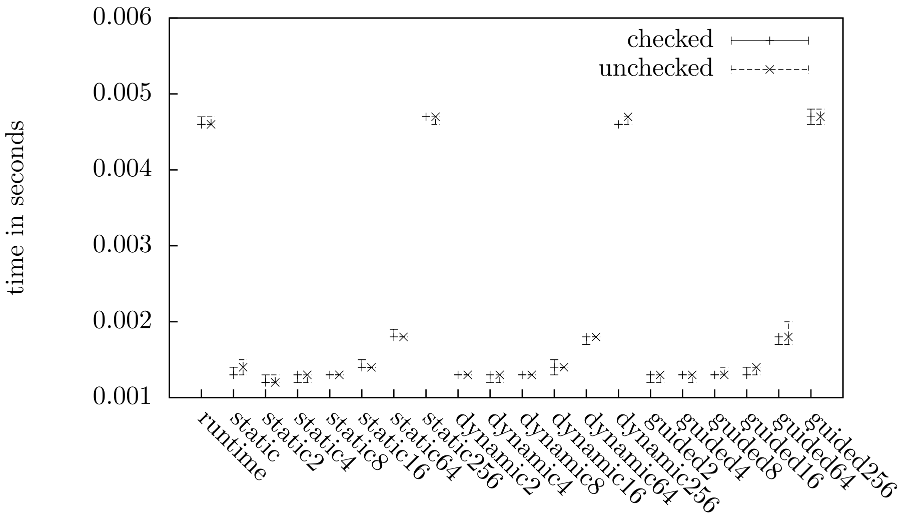


Figure 6.12: Comparison of the maximal, minimal and average running times of *checked* and *unchecked* builds of susan running corner detection using eight threads and various schedules and chunk sizes.

The results are shown in figure 6.11 and figure 6.12. Running times increase sharply for chunk sizes 64 and 256 because the chunk size is close to the number of iterations in the parallelized loops – which causes load imbalance. The imbalance disappears for larger inputs. There were no statistically significant differences in the average execution time with or without runtime checks for correct use of the directive for any of the inputs tested.

6.3 Related Work

Vallerio et al. proposed an automated task graph extraction approach from unmodified C code [159]. Parallelism is not considered and task dependencies are computed using very conservative assumptions about the use of pointers and arrays.

Adve et al. address task graph extraction in the context of high-performance computing [2]. A distributed memory programming model, MPI, is assumed. In contrast, this work targets shared memory programming. They also argue in favor of generating task graphs from OpenMP programs.

Sinnen introduces an OpenMP tasking directive to describe dependencies between certain types of OpenMP regions [146]. The dependence information is used for dynamic scheduling rather than task graph extraction, assign computational weights to tasks and is not runtime checked. Task names are used to express dependence relations between tasks whereas this work use the combination of pointer names and task names to express dependencies. The `depends_t` directive captures which pointers are shared among tasks and which are not. This allows runtime checks to ignore task private pointers.

Thies et al. and Vandierendonck et al. proposed annotations to target programs containing pipeline parallelism [155, 161]. With the former approach, the programmer uses directives to delineate pipeline stages and profiling is used to estimate the dependencies among those. The latter approach uses profiling to detect pipeline parallelism and static analysis to suggest how annotations must be used to eliminate potential dependencies for the parallelization to be legal. In contrast to our work, incorrect use of annotations may go undetected.

Ha has proposed the HoPES programming environment for development and mapping of embedded software to MPSoC's [70]. OpenMP is used to express data-parallelism in programs and task-parallelism is expressed in a synchronous data-flow model. It is unclear if and how the challenges in determining data-dependencies discussed here are handled in the HoPES environment.

Liu and Dick present a tool which can generate communication graphs by tracing loads and stores during execution rather than compile time analysis [107]. The approach is independent on programming language and threading library. Communication graphs accurately capture program behavior for given program input, number of processing elements and operating system scheduler. Unlike task graphs, communication graphs cannot be parameterized or composed hierarchically to analyze application behavior across different program executions and execution platforms.

The hArtes project is an end-to-end framework for real-time embedded systems [22]. The approach to task graph extraction is based on automatic parallelization of sequential C code. However, few types of code can be parallelized automatically. Secondly, being able to derive a parallel program from a sequential program does not imply that an accurate task graph can be derived from the program.

Schmitz et al. [143] determine inter-task dependencies by hand. It was argued in section 4.4 that this approach is labour intensive and prone to errors.

6.4 Summary

This chapter presented two directives to express data-dependencies in programs parallelized with OpenMP. The directives provide dependence information which is hard or impossible to obtain via program analysis. It reduces the number of dependencies in task graphs. This i) allow DSE tools to find more feasible designs and prevents over-provisioning of resources; and ii) lowers the running time of task scheduling algorithms whose asymptotic complexity increase with the number of dependencies.

The impact of the `depends_sc` directive on task graph precision was evaluated on four benchmark kernels. Between one and six directives were added to the benchmarks. This decreased the average number of assumed dependencies between 63% and 74%.

The performance impact of runtime checking for correct use of the `depends_t` directive was evaluated using a parallel sorting benchmark and the `depends_sc` directive was evaluated using four codes containing stencils. All benchmarks were compiled with optimization and measured on a dual socket Intel Xeon based server with eight processing cores and an Intel Atom based machine having two in-order cores. None of the benchmarked programs showed a statistically significant increase in the average running times when adding runtime checks.

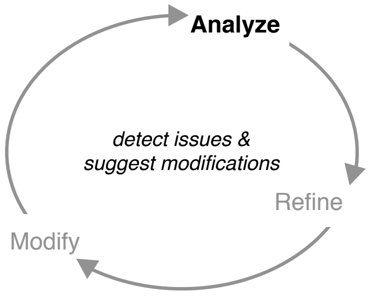
The estimated time required to insert `depends_sc` directives was about an hour per benchmark. Inserting the `depends_t` directives took less than half a working day. To lower the annotation effort, the directives should be combined with static analysis suggesting which variables and loop nests should be annotated. Static analysis could also catch simple errors such as inconsistent use of labels and references to unresolved symbols.

Chapter 7

Generating Compiler Feedback

Parallel programming is one of the biggest challenges faced by the computing industry today. Yet, applications are sometimes written in ways that prevent automatic parallelization or vectorization by compilers [86]. Opportunities for optimization are therefore overlooked. For instance, a recent study of the production compilers from Intel and IBM found that 51 out of 134 loops were vectorized by one compiler but not the other [65].

Several optimizing compilers can generate reports designed to give a general idea of the issues encountered during compilation. The task to determine the particular source code construct that prevents optimization is left to the programmer.



In this study, a production compiler was extended to produce an interactive compilation system. Feedback is generated during compilation. This corresponds to the second step in the analysis feedback loop. The feedback helps the programmer refactor the source code. This leads to code

that is more amenable to automatic
parallelization and vectorization.

The research presented in this chapter was published in the proceedings of the 4th Workshop on Programmability Issues for Heterogeneous Multicores [97]. An expanded version was published as a technical report [98]. This chapter adds an additional case study in which a JPEG decoder was modified to enable automatic vectorization of a frequently executed function.

The following section introduces two benchmarks which exemplify the problems faced by production compilers. The problems observed via the benchmarks are then related to the steps in the auto-parallelization process in Section 7.2. Section 7.3 describes the interactive compilation system to guide code refactoring and facilitate auto-parallelization. The refactoring which enabled auto-parallelization of the two benchmarks is discussed in Sections 7.4 and 7.5 respectively. Refactoring to allow auto-vectorization of a JPEG decoder is discussed in section 7.6. Experimental results are presented in sections 7.7 and section 7.8 surveys related work.

7.1 Exposing Compiler Problems

Programs can be written in many ways. Some ways obstruct automatic parallelization and vectorization. Throughout this chapter, three kernels will exemplify the problems production compilers face. The paragraphs below briefly outlines the kernels.

The first kernel is a realistic image processing benchmark. It was kindly provided by STMicroelectronics Ottawa and Polytechnique Montreal [26]. The kernel consists of 910 lines of C code[1]. it interpolates sensor data from a color filter mosaic [104] in a process called *demosaicing*. Its execution time is concentrated in twelve loop nests whose iterations are independent. In addition to the sequential code, STMicroelectronics wrote a hand-parallelized and optimized `pthreads` version.

The edge detection kernel from the UTDSP benchmark suite [102] was also studied. The kernel consists of 187 lines of C code. It detects edges in a grayscale image and contains three loop nests that may be parallelized. A single loop nest accounts for the majority of the execution time.

[1]All numbers were measured using David A. Wheelers SLOCCount.

The JPEG decoding benchmark from the EEMBC benchmark suite [36] was used to study issues preventing vectorization. It consists of 8592 lines of C code – excluding headers used to embed image data.

The kernels are difficult to parallelize automatically. To test this, the kernels were compiled with `gcc` [63] and `opencc` [35] – both open-source compilers. Three commercial compilers – `icc` from Intel [80], `pgcc` [154] from Portland Group and `suncc` [121] from Oracle – were also tested. The highest available optimization levels and inter-procedural optimization were selected to produce the best results. Level `-O2` was tried in addition to `-O3` for `gcc` since it has been observed to produce better results.

None of the loop nests in the demosaicing code were parallelized by any of the compilers. The results for the edge detection code were mixed as shown in Table 7.1. Only `icc` managed to parallelize the second loop nest where parameter-aliasing is an issue. Interestingly, `icc` succeeds because it makes an inlining decision which rules out aliasing among function parameters. If the function is not inlined, aliasing also prevents `icc` from parallelizing the loop.

Finally, automatic vectorization of two loops in a performance critical function in the JPEG decoder was tested. None of the five compilers could vectorize the two loops.

The obstacles which prevents optimization of the benchmark kernels include:

- Potential aliasing among pointers and arrays.
- Function calls in loop bodies. These may have side effects that create dependencies between loop iterations.
- Control-flow in loop bodies.
- Loop counters that may overflow or lead to out-of-bound array accesses.
- Loop bounds that cannot be analyzed by the compiler.
- Array access patterns that are too complex for the compiler to analyze.
- Loops that may contain insufficient work for parallelization to be profitable.
- operations that cannot be vectorized.

To understand why a compiler may refrain from auto-parallelizing the benchmarks, a single production compiler was studied. This work was based on the

Table 7.1: Loop nests in the edge detection kernel that were auto-parallelized by five different compilers. The compilers are unlikely to parallelize the second or third loop nest.

Origin	Compiler	Loop nests		
		loop1	*loop2*	*loop3*
FOSS	`gcc`	✓		✓
Intel	`icc`	✓	(✓)	✓
FOSS	`opencc`	✓		✓
PGI	`pgcc`	✓		
Oracle	`suncc`	✓	✓	

widely used, open-source `gcc` compiler which is being rigorously tested and enhanced by a vibrant and pragmatic developer community.

7.2 Automatic Parallelization with `gcc`

Automatic parallelization [164] involves numerous analysis steps. Every optimizing compiler must perform a sequence of roughly similar steps. The concrete implementations may vary and this leads to different strengths and weaknesses among compilers. This section explains where and why the analysis steps in `gcc` release 4.5.1 had problems parallelizing the benchmarks.

7.2.1 Alias Analysis

As explained in section 3.1.1 on page 25, alias analysis determines which storage locations may be accessible in more than one way [74]. Aliasing of pointers and function parameters may create dependencies among loop iterations so this analysis is instrumental to auto-parallelization.

The alias analysis implemented in `gcc` is a fast variant. It does not account for the context in which function calls are made nor does it take the control-flow in functions into account. Hence, function parameters of array and pointer types are conservatively assumed to alias. Also, if a statement aliases two pointers at some point in a function, the pointers are assumed to alias not just in the following statements but at all statements in the function. Both types of alias-analysis inaccuracies prevented auto-parallelization of the benchmark kernels studied. The interactive compilation system developed for this study can point to array accesses which are assumed to alias and suggest how annotation can remove this assumption.

7.2.2 Loop Bounds and Strides

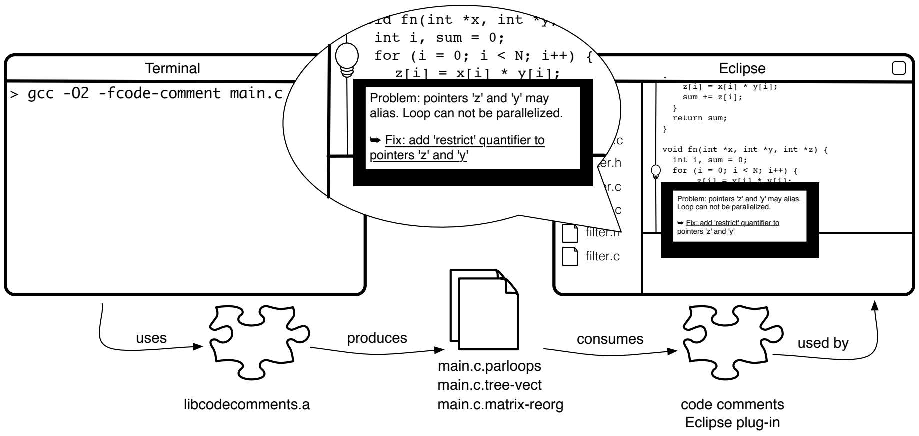
The number of loop iterations must be countable for auto-parallelization to succeed. The compiler must therefore analyze the upper and lower bounds and loop strides. If these values are not constant, it must discover which variables will hold the values at runtime.

The demosaicing kernel contained several loops which have non-unit strides. This prevented `gcc` from computing the number of iterations and was reported by the interactive compilation system. The loop increment can be *normalized* – or changed to one – by multiplying all uses of the loop counter with the original increment. However, `gcc` did not normalize the loops in the demosaicing benchmark so manual refactoring was required. Loop normalization is supported by other compilers such as `opencc` so this may be added `gcc` in a future release.

7.2.3 Induction Variable Analysis

Induction variables are scalars whose values are functions of the loop counters (section 3.1.1 on page 22). Induction and reduction operations lead to inter-iteration dependencies that require special handling during automatic parallelization. Dependencies caused by induction variables are eliminated by substituting these with functions of the enclosing loop counters [131].

In the demosaicing benchmark, `gcc` was either unable to determine which variables serve as induction variables or analyze how the values of induction variables evolve as the loop executes. This prevented auto-parallelization. The problems were mostly due to the use of a one dimensional array to represent two dimensional image data. This is known as linearization and is done since dynamically-sized multi-dimensional arrays are not well supported in C. Indexing into a linearized array involves contributions from two or more loop counters and the resulting expressions are harder to analyze. De-linearization replaces the linearized array with a multi-dimensional one. This breaks each complex indexing expression into several simpler ones which are easier to analyze [109].



```
> gcc -O2 -fcode-comment main.c
```

```
    z[i] = x[i] * y[i];
    sum += z[i];
  }
  return sum;
}

void fn(int *x, int *y, int *z) {
  int i, sum = 0;
  for (i = 0; i < N; i++) {
    z[i] = x[i] * y[i];
```

Problem: pointers 'z' and 'y' may alias. Loop can not be parallelized.

➥ Fix: add 'restrict' quantifier to pointers 'z' and 'y'

Figure 7.1: Illustration of the compilation feedback system. A library extends `gcc` to generate code comments in its diagnostic dump files. A plug-in for the Eclipse CDT environment provides Eclipse with the functionality to i) read the code comments containing feedback ii) display them at appropriate places in the source code and iii) provide refactoring support for the changes suggested by the compiler feedback.

7.2.4 Data Dependence Analysis

Currently, `gcc` contains two different frameworks to analyze data dependencies. The *lambda* framework is the oldest and most mature of the two. It represents data dependencies as distance vectors and implements a multi-dimensional version of the Banerjee test [105, 16]. Much functionality, including the lambda framework is shared between the loop parallelization and vectorization optimizations. Hence, code transformations which enable `gcc` to parallelize a loop using the lambda framework can also help making it vectorizable.

`Gcc` is transitioning to GRAPHITE which is a newer and more capable data dependence framework [129]. The transition is advancing at slow but steady pace and much work remains. In the 4.5 and 4.6 releases of `gcc`, auto-parallelization with GRAPHITE only handles innermost loops. The lambda framework was therefore used in this study.

The goal of data dependence analysis is to determine if a loop has inter-iteration dependencies. In such cases auto-parallelization is usually prohibited. Data is often read and written from arrays or pointers and this may lead to dependencies between loop iterations. Recall from section 3.1.1 on page 22 that each iteration can be identified by a vector in the loop iteration space. Subtracting the iteration vectors of two dependent iterations yields a distance vector. The lambda dependence analysis framework requires that all possibly dependent loop iterations have the same distance vector. If is not the case, auto-parallelization fails. The failure happens because the data access pattern is too complex to be modeled, even if loop iterations are in fact independent. Some loop nests in the demosaicing benchmark were not auto-parallelized for this reason.

7.3 Interactive Compilation Feedback System

The benchmark kernels exposed several coding patterns which obstructs auto-parallelization in production compilers. The previous section then used `gcc` to exemplify why a compiler may be prevented from auto-parallelizing a loop nest. We can now explain how we provide feedback and suggestions on how the programmer can make code amenable to auto-parallelization.

The interactive compilation feedback system is illustrated in Fig. 7.1. It has two parts. The first part is a library, `libcodecomments`, and a set of patches to `gcc`'s auto-parallelization subsystems. This extension of `gcc` generates code comments containing compiler feedback. In contrast to stand-alone tools, the code comments approach leverages the production-quality program analysis and optimization functionality already present in the compiler.

The code comments are generated when one of the steps in the auto-parallelization optimization encounters an issue which prevents further analysis. The functionality in `libcodecomments` is then used to produce a human understandable problem description. This is important because program analysis often fail while processing compiler generated, temporary variables that are meaningless to the programmer. Most importantly, `libcodecomments` is used to reconstruct source level expressions (after preprocessing) and their file locations from compiler generated temporaries.

The generation of diagnostic dump files are controlled via existing compiler flags – the code comments are simply added to these dump files.

The second part of the system is a plug-in for the Eclipse C Development Tools, CDT [153]. The code comments plug-in enables CDT to parse the compiler feedback from dump files. The dump files are read by a custom step in the Eclipse build process and requires no programmer intervention besides adding the appropriate compiler flags. The raw code comments are subsequently converted into *markers*, which are shown as icons in the left margin of the code in the Eclipse source editor. The markers automatically track the source code

Table 7.2: Types of issues reported by compilation feedback system. Second and third columns indicate whether the comment includes guidance to help resolve the issue and whether a refactoring is offered to fix the issue. The fourth column estimates if a non-expert on compilation is likely to comprehend the issue.

type of issue	guides	fix	non-expert	example
pointer aliasing	✓	✓	✓	fig. 7.2
side-effects of fun. call	✓	✓	✓	
unsupported vector op.	✓		✓	fig. 7.4
questionable profitability	✓		✓	
side-effects of volatile var.			✓	
side-effects of inline asm.			✓	
control-flow			✓	fig. 7.3
data reference issue			✓	fig. 7.6
scalar carried dependence				
loop nesting issue				
data alignment issue				
induction variable issue				
reduction variable issue				
unknown num. iterations				

construct, say a loop or variable associated with the code comment. The comment may include a *quick fix* – i.e. a refactoring that automates the suggested transformation. For example, lines may be added or deleted around the construct. The comment in the marker is shown in the *Problems* view in Eclipse, and pops up when the cursor hovers over the marked code as shown in the callout in Fig. 7.1. Similar to compiler warnings and errors, the code comments are automatically updated after each full or incremental build.

Not all the code comments which can be generated by our modified compiler contain concrete advice on how to resolve a given issue. Furthermore, some issues currently require some familiarity with the functioning of auto-parallelization and auto-vectorization and thus must be considered unavailable to non-experts. Table 7.2 gives an overview of issues that can be reported by the system and marks the ones that may be comprehended programmers without a background in compilers. The following three sections describe how the interactive compilation system was used to resolve issues preventing optimization of the benchmark kernels.

7.4 Case Study: Demosaicing

Recall that gcc and the other compilers tested failed to parallelize any of the 12 original loop nests in the demosaicing kernel. The compilation feedback system, however, succeeded in removing the issues preventing parallelization. It was accomplished by iteratively modifying and compiling the code until all relevant loop nests were auto-parallelized. The following sections describe how the code was refactored to accomplish this.

7.4.1 Loop Iteration Counts

Most of the loops in the demosaicing code have a stride of two. This caused gcc's iteration count analysis to fail according to the compiler feedback. As a workaround, the loops were normalized to use unit strides and array indexing expressions were updated accordingly. For instance

```
int x, y, idx;
for(x=2+offset_red;x<H-2;x+=2) {
 for(y=2+offset_blue;y<W-2;y+=2) {
  idx=x*W+y; ... }}
```

was rewritten as:

```
unsigned int x, y, idx;
 for(x=1;x<(H-2)/2;x++) {
  for(y=1;y<(W-2)/2;y++) {
   idx=(2*x+offset_red)*W+2*y+offset_blue;
   ... }}
```

Additionally, the type of the loop counters, x and y were changed from signed to unsigned integers. Finally, we observed that writing the loop upper bound as H/2-1 rather than (H-2)/2 also caused number of iterations analysis to fail. A more powerful analysis can surely digest both variants properly.

7.4.2 Aliasing

As mentioned in section 7.2.1, gcc employs a scalable but imprecise alias analysis. Most importantly, the analysis does not analyze how function arguments are passed from callers to callees, which means that if a function contains several arguments having pointer or array types, gcc must assume they may alias.

This assumption is made even for parameters of incompatible types, due to the weak type discipline employed in C. Its possible to change the assumption that pointers to objects of different types alias with the `-fstrict-aliasing` flag. In our experiments it eliminated potential aliasing in two out of the twelve loop nests.

When potential aliasing prevents parallelization, a code comment contains feedback on the data references that are potential aliases. The code comment as it appears in the IDE is shown in figure 7.2. The comment suggests that the problem can be resolved by annotating the relevant pointers with the `restrict` keyword. This type qualifier was added in the latest revision of the language standard [84]. It also includes an option to automatically transform the code such that the `restrict` type qualifier is added to the relevant pointers declarations. Semantically, if memory addressed by a `restrict` qualified pointer is modified, no other pointer provides access to that memory. It is left to the programmer to determine if the `restrict` qualifier can be added. Based on the suggestions provided by the code comments, we added `restrict` qualifiers to 6 pointer typed formal parameters in two function signatures.

A more precise, inter-procedural alias analysis is also available in `gcc`. It is enabled by the `-fipa-pta` flag. Contrary to our expectations, the inter-procedural alias analysis did not diminish the need to `restrict`-qualify function parameters.

7.4.3 Induction Variables

Normalizing loop strides to one and simplifying the expressions governing loop bounds as described in section 7.4.1 in effect complicated the expressions for the induction variables. For instance, an induction variable that was previously computed as `idx=x*W+y` became `idx = (2*x+offset_red)*W+ 2*y+offset_blue`. However, the compilation feedback helped us understand how to refactor the loops so that induction variable analysis did not prevent auto-parallelization. The original demosaicing kernel uses linearized arrays to represent variable size, two-dimensional image data. The arrays are passed into the three kernels as function parameters. Changing the types of these function parameters allowed us to cast the linearized arrays as two-dimensional arrays. This in turn allowed a simplification of the indexing expressions. By changing a parameter `int *restrict red_array` to `int (*restrict red_array)[W]` where `W` is scalar holding the image width, we changed the indexing expressions from

```
idx = (2*x+offset_red)*W+2*y+offset_blue;
red_array[idx]= RBK_3x3_1(
 red_array[idx-W-1], red_array[idx-W+1],...);
```

to

```
red_array[2*x+offset_red][2*y+offset_blue] =
 RBK_3x3_1(
  red_array[2*x+offset_red-1][2*y+offset_blue-1],
  red_array[2*x+offset_red-1][2*y+offset_blue+1],
 ...);
```

De-linearizing the array accesses arguably increased the readability of the code.

It was also necessary to move the loop-invariant variables `offset_red` and `offset_blue` out of the loop. This was accomplished by introducing a temporary, `restrict`-qualified pointer defined as
`tmp_red_array= red_array+W*offset_red+offset_blue`

Finally, due to an analysis limitation when computing the scalar evolution of expressions containing integers of different sizes we had to suffix the integer literals with `L`'s since we used a 64-bit build environment. Eight loop nests had to be refactored in this manner. Continuing the code example, we arrived at:

```
int (*restrict tmp_red_array)[W]=
 red_array + W*offset_red + offset_blue;
for(x=1;x<(H-2)/2;x++) {
 for(y=1;y<(W-2)/2;y++) {
  tmp_red_array[2L*x][2L*y] =
   RBK_3x3_1(
    tmp_red_array[2L*x-1L][2L*y-1L],
    tmp_red_array[2L*x-1L][2L*y+1L], ...
```

7.4.4 Data dependencies

After transforming the code to allow all preceding analysis steps to succeed, `gcc` was able to perform data dependence analysis on the loop nests. Although iterations of all loop nests in the benchmark are independent, eight of these loop nests update elements in place to reduce memory requirements. The in-place updates are possible when a loop nest writes only "odd" elements and reads only "even" elements or vice versa. From the code comments, however, we could determine that the data dependence analysis failed to discover this. For instance, a possible data dependence was reported between the two following memory *reads*:

```
tmp_blue_array[x*2+1][y*2-1]
```

and

```
tmp_blue_array[x*2-1][y*2+1]
```

Data dependence analysis fails for this pair of references, because the lambda framework cannot compute a distance vector which represents their dependence relation. A possible dependence between these two references must therefore be assumed in lieu of a more precise data dependence analysis.

To avoid reading and writing to the same array – the memory addressed by `tmp_blue_array` in the example – a new temporary array was allocated to hold the writes. This effectively sidesteps a compilers inability to analyze non-overlapping accesses to the same array. However, it also means that updates are no longer done in-place which decreases the locality of the kernel and increases the memory consumption by approximately 8%. Finally, for each of the eight loop nests with in-place updates, a simple "copy" loop was added to write data from the new temporary array back to its original destination. These loops were fairly easy to add and were readily parallelized due to their simplicity.

An alternative solution exists: the programmer could have introduced additional `restrict`-qualified pointers until all potential data dependencies are ruled out. This solution does affects neither the data access pattern nor the memory consumption so performance would be unaffected. This shows that the programmer may need to chose among several alternative ways to refactor – each having a different performance impact. For the experiments, it was pessimistically assumed that the programmer choose the refactoring that is most costly in terms of performance.

The modifications described in this section required 50 source lines to be modified and another 35 to be added. The estimated effort to perform the modifications to allow auto-parallelization is approximately 2-3 working days.

7.4.5 Posix Threads Version

The `pthreads` code received from STMicroelectronics was subsequently optimized to execute all relevant loops in parallel and to minimize synchronization and thread management overhead. The parallelization strategy of the `pthreads` version differs from the auto-parallelized version. Two of the twelve loop nests in the sequential code were fused.

The distribution of iterations among threads also differ. The auto-parallelized version only distributes iterations of the outer loops among threads. The `pthreads` version, however, divides the two-dimensional picture into a number of tiles and assigns each tile to a single thread thereby increasing cache affinity.

Finally the `pthreads` version exploits the task-level parallelism that exists among the eight computationally intensive loop nests. It does so by executing them in pairs of two. The auto-parallelized version executes all loop nests one after another so it only exploits data-level parallelism.

7.4.6 OpenMP Version

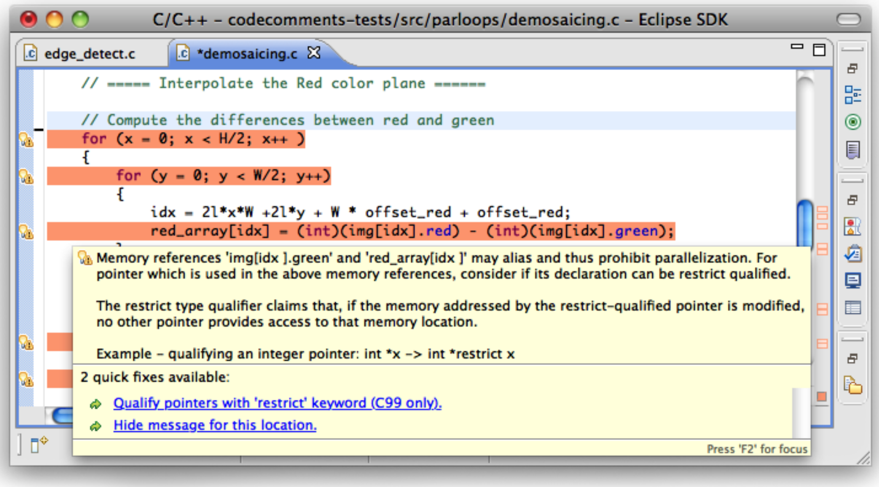
Temporary arrays and extra loop nests were introduced in the auto-parallelized version to work around limitations in `gcc`'s data dependence analysis. Auto-parallelization also uses the combined work-sharing construct `omp parallel for` in OpenMP whereas an experienced OpenMP programmer may enclose several loop nests with `omp for` directives in a single `omp parallel` region to reduce synchronization among threads.

To measure the resulting performance if the above mentioned deficiencies were removed, the demosaicing code was hand-parallelized with OpenMP pragmas. Like the auto-parallelized version, the OpenMP version only exploits data-parallelism but performs updates of the arrays in-place instead of using temporary arrays.

Furthermore, the entries and exits to and from parallel regions was minimized. It was done using separate `omp parallel` and `omp for` directives in place of the `omp parallel for` directive. This reduced the number of times a parallel section was entered from twelve to five. Using the `nowait` clause on the `omp for` directives finally allowed three implicit barriers to be removed.

7.5 Case Study: Edge Detection

The program consists of a `main` function which calls the function `convolve2d` repeatedly with 3x3 Gaussian and Sobel kernels to do edge detection. The `main` method contains two loop nests but the bulk of the computation takes place in `convolve2d`'s second loop nest. During compilation, `gcc` can parallelize the loop nests in the `main` method but not the work intensive loop in `convolve2d`. The problem is aliasing between three arrays which are passed as parameters


C/C++ - codecomments-tests/src/parloops/demosaicing.c - Eclipse SDK
edge_detect.c
*demosaicing.c

```
    // ===== Interpolate the Red color plane ======

    // Compute the differences between red and green
    for (x = 0; x < H/2; x++ )
    {
        for (y = 0; y < W/2; y++)
        {
            idx = 2l*x*W +2l*y + W * offset_red + offset_red;
            red_array[idx] = (int)(img[idx].red) - (int)(img[idx].green);
```

Memory references 'img[idx].green' and 'red_array[idx]' may alias and thus prohibit parallelization. For pointer which is used in the above memory references, consider if its declaration can be restrict qualified.

The restrict type qualifier claims that, if the memory addressed by the restrict-qualified pointer is modified, no other pointer provides access to that memory location.

Example - qualifying an integer pointer: int *x -> int *restrict x

2 quick fixes available:

- Qualify pointers with 'restrict' keyword (C99 only).
- Hide message for this location.

Press 'F2' for focus

Figure 7.2: A code comment generated by our compilation feedback system and shown in the Eclipse editor. Lines with comments are highlighted with an orange background color and with small lightbulbs in the gutter area. Placing the cursor on a source line with a comment will show an overlay with an explanatory message and a list of available refactoring suggestions (if any). The problem view in the bottom shows code comments in addition to regular warnings and errors.

to the `convolve2d` function. Feedback from the compilation system reported an aliasing problem between pairs of data references and is illustrated in Figure 7.2.

A programmer who understands the roles of the arrays in the edge detection code knows that these will never refer to the same memory. As with the demosaicing code, the lack of aliasing between the function parameters must be communicated using the `restrict` keyword and again `gcc`'s inter-procedural alias analysis did not help. The fact that only pointers can be qualified with `restrict` complicates the situation. Before the `restrict`-qualifier can be used, the parameters to the `convolve2d` function, must be changed as shown below:

```
void convolve2d(
  int input_image[N][N],
  int kernel[K][K],
  int output_image[N][N])
```

to

```
void convolve2d(
  int (*restrict input_image)[N],
  int (*restrict kernel)[K],
  int (*restrict output_image)[N])
```

The edge detection benchmark was subsequently parallelized by `gcc` without further problems. The estimated effort to enable parallelization of the edge detection benchmark is 1-2 hours.

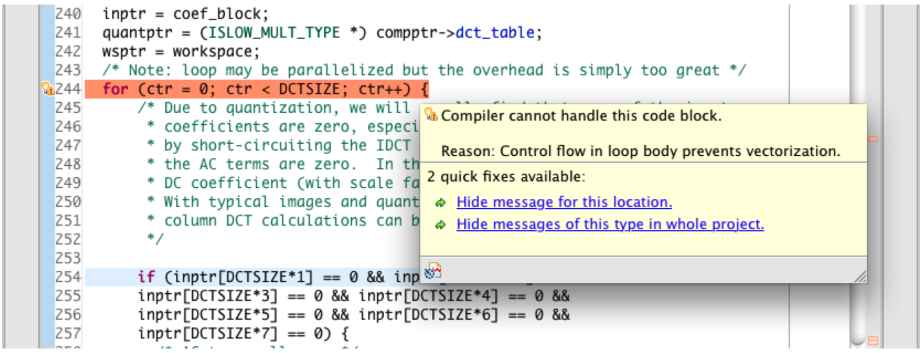
7.5.1 OpenMP Version

To compare the auto-parallelized edge-detection code with a hand-parallelized and optimized version, we inserted OpenMP directives in the sequential code. Similar to demosaicing, separate `omp parallel` and `omp for` directives were used to increase the performance.

7.6 Case Study: JPEG Decoding

The previous two case studies focused on automatic parallelization. This section studies how the compilation feedback system can enable additional loops to be vectorized in a JPEG decoder. Profiling was used to locate the function that accounts for the largest fraction of the execution time. The JPEG decoder was compiled with the highest level of optimization, O3, using `gcc` 4.5.1 and profiled on an Intel Core 2 Duo laptop. The function implementing *inverse discrete cosine transform*, IDCT step accounts for 22% of the total run time. This function was neither parallelized nor vectorized. Since the IDCT function process 8x8 blocks of pixels and is called via a function pointer, parallelization seems unprofitable. Hence, the compilation feedback tool was used to refactor the code until it was successfully vectorized.

Control-Flow in Loops The first issue reported is the existence of control-flow in the two loops comprising the IDCT function. As mentioned in section section 3.1.2 on page 27, if-conversion and loop unswitching can sometimes remove control-flow from loop bodies. However, the `if`-statements in the IDCT function are too complicated to be handled by the former transformation and the latter did not apply. Figure 7.3 on the next page shows the reporting of this issue. The compilation feedback tool cannot determine how to resolve the issue. However, source code comments are associated with both `if`-statements



```
inptr = coef_block;
quantptr = (ISLOW_MULT_TYPE *) compptr->dct_table;
wsptr = workspace;
/* Note: loop may be parallelized but the overhead is simply too great */
for (ctr = 0; ctr < DCTSIZE; ctr++) {
    /* Due to quantization, we will
     * coefficients are zero, especi
     * by short-circuiting the IDCT
     * the AC terms are zero.  In th
     * DC coefficient (with scale fa
     * With typical images and quant
     * column DCT calculations can b
     */

    if (inptr[DCTSIZE*1] == 0 && inp
    inptr[DCTSIZE*3] == 0 && inptr[DCTSIZE*4] == 0 &&
    inptr[DCTSIZE*5] == 0 && inptr[DCTSIZE*6] == 0 &&
    inptr[DCTSIZE*7] == 0) {
```

Compiler cannot handle this code block.

Reason: Control flow in loop body prevents vectorization.

2 quick fixes available:

- Hide message for this location.
- Hide messages of this type in whole project.

Figure 7.3: Compilation feedback reporting that control-flow prevents vectorization.

in the IDCT function. These explain in unambiguous terms that the conditional statements "short-circuit" the following computations. Both if-statements were removed from the code. This is a prime example of how programmer understanding can help modify code to facilitate analysis.

Pointer Aliasing Unlike the automatic parallelization pass in gcc, the vectorizer support loop versioning and runtime checking of aliasing. With versioning, the compiler creates two versions of a loop; one vectorized and one sequential. If runtime checks determine that aliasing prevent vectorization, the sequential version is executed. A runtime check is required among each unique pair of potentially aliased memory accesses. The total number of runtime checks for n accesses is $\frac{n(n-1)}{2}$ – corresponding to the number of edges in a complete graph. A high number of checks degrades program performance so gcc will not emit more than 10 runtime checks by default. Since this limit is exceeded in the IDCT function, the compilation feedback system was used to annotate 5 pointers with the restrict keyword.

Pointer Arithmetic Some compilers, such as IBM's xlc compiler [83], can handle limited forms of pointer arithmetic while several others, including icc and gcc, cannot [65, 21]. Hence, all accesses to three pointers were changed as follows:

```
for(ctr = 0; ... ; ctr++) {
  z2 = DEQUANTIZE(inptr[DCTSIZE*2],
                  quantptr[DCTSIZE*2]);
  ...
  inptr++;
  quantptr++;
}
```

to

```
for(ctr = 0; ... ; ctr++) {
  z2 = DEQUANTIZE(inptr[DCTSIZE*2+ctr],
                  quantptr[DCTSIZE*2+ctr]);
  ...
}
```

A total of 32 source statements were modified to remove pointer arithmetic.

Non-Vectorizable Operations Vectorization uses special instructions to express parallelism. Not all operations on scalars are available in vector form. The compilation feedback reported one such issue. Without the compilation feedback system, `gcc` emits the following message in a diagnostic file:

```
djpeg/jidctint.c:243: note: relevant stmt not supported:
   D.5604_159 = D.5603_158 >> 11;
```

There are two problems with this message. First, It references two variables `D.5604_159` and `D.5603_158` which are compiler generated and therefore has no meaning to the programmer. Second, it makes it seem as if arithmetic right shift operations cannot be vectorized with the Intel MMX and SSE2 vector instructions. This is not the case [43, p. "4-237"]. Rather, the arithmetic right shift cannot be supported for vectors of two 64-bit signed integers. This is reported by the compilation feedback system as shown in figure 7.4 on page 105. It shows the statement containing the unsupported operation The code comment also shows other vector data types for which the right shift *is* in fact supported.

In the source code, the right shift operation is not visible due to the use of a macro. However, the Eclipse IDE shows how macros expand when placing the cursor over a macro. Figure 7.5 on page 105 shows that the expansion of the `DESCALE32` macro includes a right shift operation. The macro operates on data types named `e_s32`, which, to the programmer, signifies a 32-bit signed integer. Yet, the code comment reported that right shift is not available for vectors of two `long` int's which are 64-bit long on the test machine.

Further investigation showed that `e_s32` is a type alias for the `signed long` data type in C. This type is a 32-bit integer on windows systems but a 64-bit integer on 64-bit UNIX systems. Since the EEMBC benchmark suite contains traces of being developed on the Windows operating system, this is most likely a portability problem. The header file defining the `e_s32` type was therefore changed from:

```
typedef unsigned long e_u32;
typedef signed   long e_s32;
```

to

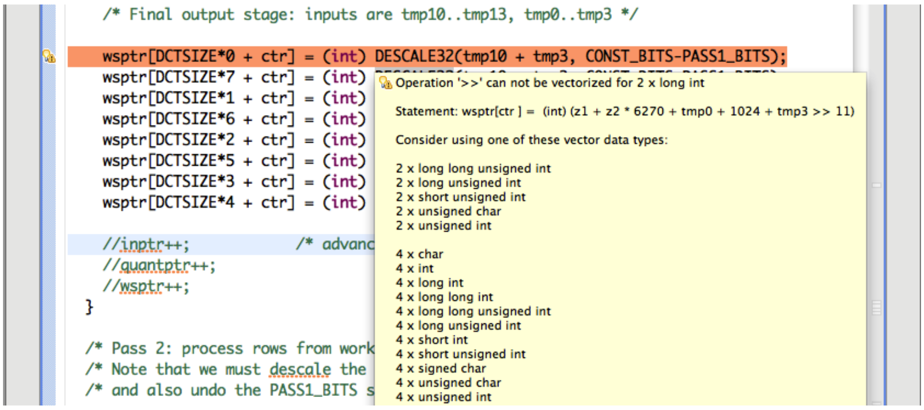
```
/* C99 header to allow more portable code */
#include <stdint.h>
...
typedef uint32_t e_u32;
typedef  int32_t e_s32;
```

Data Reference Analysis One final issue prevented vectorization of the second loop in the IDCT function. The issue was related to data reference analysis and was reported as shown in figure 7.6 on the facing page. The compilation feedback system reports that the statement
`outptr[7] = range_limit[wsptr[ctr*2+7*4]&1023]` could not be analyzed. The reported expression does not exactly match the source code because it has been reconstructed from the compiler intermediate representation after preprocessing and loop unrolling was performed. The programmer should have no problem relating it to the source code as the relevant source statement is highlighted as shown in the figure.

The dependence analyzer requires that affine expressions are used to index into arrays. This is not the case for accesses to the array `range_limit`. There is no obvious way to modify the statement to resolve this. However, the statement can be moved out of the loop nest as shown in listing 7.1 and listing 7.2 on page 106:

```
for (ctr = 0; ctr < DCTSIZE; ctr++) {
  /* compute intensive stuff */
  ...
  /* write back results - prevents vectorization */
  for(int i = 0; i < DCTSIZE; i++)
    outptr[i] =
      range_limit[wsptr[DCTSIZE*ctr+i]&RANGE_MASK];
}
```

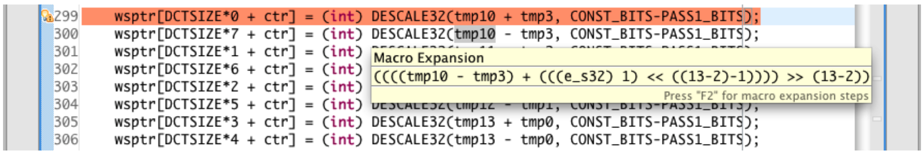
Listing 7.1: Structure of second loop in IDCT function *before* loop fission.



```
    /* Final output stage: inputs are tmp10..tmp13, tmp0..tmp3 */

    wsptr[DCTSIZE*0 + ctr] = (int) DESCALE32(tmp10 + tmp3, CONST_BITS-PASS1_BITS);
    wsptr[DCTSIZE*7 + ctr] = (int)
    wsptr[DCTSIZE*1 + ctr] = (int)
    wsptr[DCTSIZE*6 + ctr] = (int)
    wsptr[DCTSIZE*2 + ctr] = (int)
    wsptr[DCTSIZE*5 + ctr] = (int)
    wsptr[DCTSIZE*3 + ctr] = (int)
    wsptr[DCTSIZE*4 + ctr] = (int)

    //inptr++;            /* advanc
    //quantptr++;
    //wsptr++;
  }

  /* Pass 2: process rows from work
  /* Note that we must descale the
  /* and also undo the PASS1_BITS s
```

```
Operation '>>' can not be vectorized for 2 x long int

Statement: wsptr[ctr ] = (int) (z1 + z2 * 6270 + tmp0 + 1024 + tmp3 >> 11)

Consider using one of these vector data types:

2 x long long unsigned int
2 x long unsigned int
2 x short unsigned int
2 x unsigned char
2 x unsigned int

4 x char
4 x int
4 x long int
4 x long long int
4 x long long unsigned int
4 x long unsigned int
4 x short int
4 x short unsigned int
4 x signed char
4 x unsigned char
4 x unsigned int
```

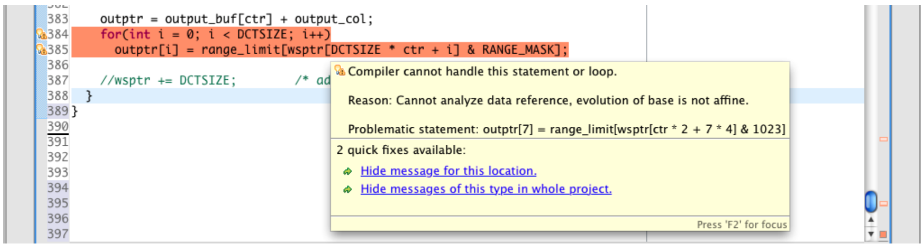
Figure 7.4: Compilation feedback reporting unvectorizable operation.



```
    wsptr[DCTSIZE*0 + ctr] = (int) DESCALE32(tmp10 + tmp3, CONST_BITS-PASS1_BITS);
    wsptr[DCTSIZE*7 + ctr] = (int) DESCALE32(tmp10 - tmp3, CONST_BITS-PASS1_BITS);
    wsptr[DCTSIZE*1 + ctr] = (int)
    wsptr[DCTSIZE*6 + ctr] = (int)
    wsptr[DCTSIZE*2 + ctr] = (int)
    wsptr[DCTSIZE*5 + ctr] = (int)
    wsptr[DCTSIZE*3 + ctr] = (int) DESCALE32(tmp13 + tmp0, CONST_BITS-PASS1_BITS);
    wsptr[DCTSIZE*4 + ctr] = (int) DESCALE32(tmp13 - tmp0, CONST_BITS-PASS1_BITS);
```

```
Macro Expansion
((((tmp10 - tmp3) + (((e_s32) 1) << ((13-2)-1)))) >> (13-2))
                              Press "F2" for macro expansion steps
```

Figure 7.5: Macro expansion in Eclipse IDE helped locate right shift operation in code.



```
    outptr = output_buf[ctr] + output_col;
    for(int i = 0; i < DCTSIZE; i++)
      outptr[i] = range_limit[wsptr[DCTSIZE * ctr + i] & RANGE_MASK];

    //wsptr += DCTSIZE;        /* ad
  }
}
```

```
Compiler cannot handle this statement or loop.

Reason: Cannot analyze data reference, evolution of base is not affine.

Problematic statement: outptr[7] = range_limit[wsptr[ctr * 2 + 7 * 4] & 1023]
2 quick fixes available:
  Hide message for this location.
  Hide messages of this type in whole project.
                                                  Press 'F2' for focus
```

Figure 7.6: Compilation feedback reporting statement which cannot be handled by dependence analysis.

was changed to

```
for (ctr = 0; ctr < DCTSIZE; ctr++) {
  /* compute intensive stuff - now vectorized */
}
/* write back step - still not vectorized */
for (ctr = 0; ctr < DCTSIZE; ctr++) {
  outptr = output_buf[ctr] + output_col;
  for(int i = 0; i < DCTSIZE; i++)
    outptr[i] =
      range_limit[wsptr[DCTSIZE*ctr+i]&RANGE_MASK];
}
```

Listing 7.2: Structure of second loop in IDCT function *after* loop fission.

Loop fission allowed the majority of the second loop in the IDCT function to be vectorized too.

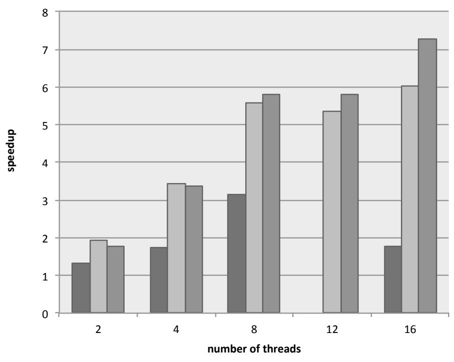
In total, the refactoring described in this section added 4 lines, removed 38 and modified another 39 lines of the IDCT function – which was originally 217 lines of code. To resolve the portability issue, one line was added and two were modified in the header file that define the `e_s32` datatype. The estimated time to perform the modifications was one to two working days. If this study had been limited to a 32-bit environment, the effort would have been significantly lower as the portability problem would not have manifested itself.

7.7 Experimental Results

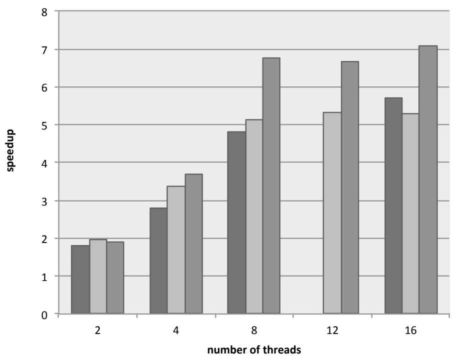
The differences in sequential performance between the original and modified versions of the edge detection and demosaicing kernels were measured and found to be negligible in both cases. The sequential performance of the JPEG decoder will be discussed separately.

The reference input for the edge detect benchmark is an image with 128x128 pixels which was scaled to 4096x4096 to increase running times well above the timing resolution. A large and a small color image was used as input to the demosaicing kernel. The large image had 5616x3744, 24-bit pixels while the small image consisted of 768x512, 24-bit pixels. The inputs are summarized in Table 7.3.

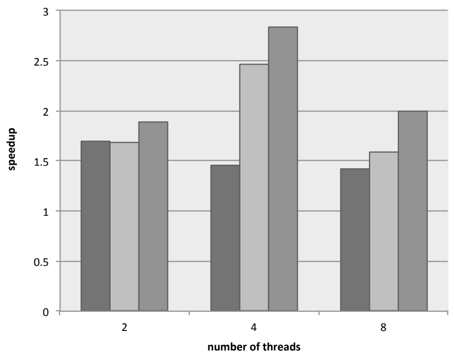
Two different systems were used to evaluate the impact of our modifications to the benchmarks. The Intel system was a dual-socket server equipped with



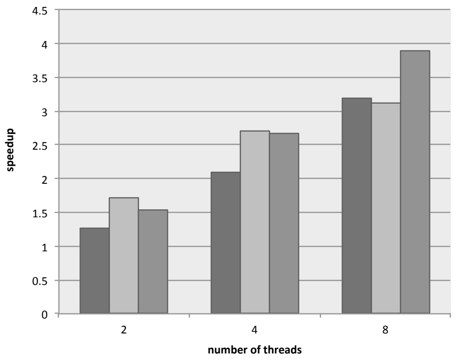
(a) Demosaicing of low resolution image on x86-64



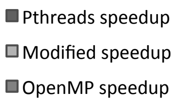
(b) Demosaicing of high resolution image on x86-64



(c) Demosaicing of low resolution image on POWER



(d) Demosaicing of high resolution image on POWER



(e) Legend

Figure 7.7: Demosaicing speedups on x86 and POWER platforms. Three parallelized versions are compared to the original, sequential program version: a version modified and auto-parallelized by `gcc`, a hand written version using `pthreads`, and a hand written OpenMP version. The `pthreads` version does not take advantage of all parallelism inherent in the benchmark. Also, it does not support 12 threads, so this data point is unavailable. Figures 7.7a and 7.7c speedups results for a image with a resolution of 768x512 and 7.7b and 7.7d show speedups for an image with a resolution of 5616x3744 pixels.

Table 7.3: Characteristics of the benchmark inputs.

Benchmark	Large input	Small input
Demosaicing	5616x3744, 24-bit color	768x512, 24-bit color
Edge Detection	4096x4096, 8-bit grayscale	-

Table 7.4: Characteristics of the processing elements and memory hierarchies in the systems used for benchmarking.

System	#Processors	#Cores	#Threads	Frequency
Intel	2	8	16	2.93 GHz
IBM	2	4	8	4.0 GHz
System	L1D	L2/Core	L3/Core	DRAM
Intel	32 KB	256 KB	2 MB	12 GB DDR3
IBM	64 KB	4 MB		8 GB DDR2

two quad-core 2.93 GHz Xeon 5570 CPUs and a total of 12 GB DDR3 RAM. It contained eight cores each of which supports two hardware threads. It had 32 KB L1 instruction cache, 32 KB L1 data cache, 256 KB L2 cache per core and 8 MB shared L3 cache per CPU. The operating system was Linux using the 2.6.36 kernel. The IBM system was a JS22 (7998-61X) blade with two dual-core 4.0 GHz POWER6 SCM processors. Like the Intel Xeon system, each core supports two hardware threads. The system had 8 GB DDR2 RAM, 64 KB L1 instruction cache, 64 KB L1 data cache and 4 MB L2 cache per core. The operating system kernel was Linux 2.6.27. The characteristics of the processing units are summarized in Table 7.4

Version 4.5.1 of `gcc` with our modifications to generate compiler feedback was used for all experiments. The `-O2` compilation flag was used for optimization since the auto-parallelization does not always succeed at `-O3`. Measurements were made for 2-16 threads on the x86 platform and 2-8 on the POWER platform. Numbers were calculated as averages over three consecutive program executions on an unloaded system. The time spent on IO was excluded from the measurements.

7.7.1 Demosaicing Speedups for Intel Xeon

The speedups of parallelizing the modified demosaicing code was compared to the performance of the original, sequential code. The speedups of the hand written pthreads and OpenMP versions were measured similarly. Finally, the sequential performance of the modified demosaicing code was measured.

Two images were used as input for the demosaicing benchmark: a high resolution 21 mega-pixel image and a small image with 768x512 pixels. The speedups on the x86-64 platform are summarized in Fig. 7.7a and Fig. 7.7b.

When auto-parallelizing the unmodified code no loops are parallelized so no speedups are shown. When auto-parallelizing the modified code, the observed speedups range from 1.9 to 6 for 2-16 threads for the low resolution image. The highest speedup of 6.0 was obtained using 16 threads. A speedup of 5.6 is already obtained using 8 threads and the using 12 threads did not produce more than a 5.3 speedup over the sequential code. The meager increases after 8 threads are most likely explained by the fact that no cache capacity is added past 8 threads. Seemingly, the system uses symmetric multi-threading aware scheduling. This means that it assigns one thread to each of the eight cores before any of the cores receive an additional thread.

The high resolution image shows speedups ranging from 1.9 to 5.3 for 2-16 threads. The speedup using 8 threads, which is 5.1, is again close to the maximal speedup of 5.3 on 16 threads. The weaker scaling when using a high resolution image may be explained by the fact that the effect of the temporary arrays become more pronounced once the working set sizes exceed the capacity of the last level caches.

Performance relative to the OpenMP version Speedups for the OpenMP version range from 1.8 to 7.2 for 2-16 threads for the low resolution image and from 1.9 to 7.0 for the high resolution image. Performance increases steadily for 2,4,8 and 16 threads whereas there is little difference between 8 and 12 threads. Interestingly, the auto-parallelized version outperforms the OpenMP version by 8% on 2 threads and performs similarly with 4 threads. It is outperformed by 4% and 17% on 8 and 16 threads respectively.

Performance relative to the `pthreads` version The speedups obtained by the pthreads version were: 1.8-5.7 for the high resolution image and 1.3-3.2 for the low resolution image, executed by 2-16 threads. On the big image, the auto-parallelized code performed within 93-108% of the pthreads code – outperforming it on all but 16 threads. With the small image, the auto-parallelized version outperformed the pthreads code by 44%-239%.

Sequential performance of modified demosaicing code As described in section 7.4.4 on page 97, eight "copy" loops were added to work around an issue preventing data dependence analysis. This decreases the temporal locality of memory references and increases memory consumption. Without automatic parallelization, the modified demosaicing code only ran 1% slower than the original demosaicing code. Furthermore, retaining all modifications except the eight "copy" loops increased the sequential performance by 9%.

7.7.2 Demosaicing Speedups for IBM POWER

An anomaly was encountered when compiling the demosaicing code on POWER. Four of the 20 loop nests in the modified benchmark were not parallelized by `gcc`. All loops are successfully parallelized with Linux or Mac OS X on Intel platforms. It was also ensured that `gcc` were configured and built identically on the two platforms. This leads us to believe that the differences are caused by target dependent optimization decisions.

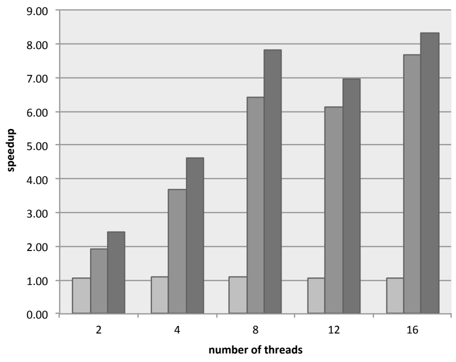
As a work-around, `parallel for` pragmas was manually inserted where the auto parallelization step in `gcc` would have done the same. It was verified that the workaround where 16 loops are auto-parallelized and 4 hand parallelized performs identical to the version where all loops were auto-parallelized on the Intel platform.

The experimental runs of the demosaicing benchmark were repeated on the POWER platform using the same high and low resolution images and the same compiler version. The speedups on this platform are summarized in Fig. 7.7c and Fig. 7.7d. The demosaicing code generally scaled significantly worse on the POWER platform which suggest that the benchmark needs additional tuning – e.g. improving the use of the memory hierarchy though loop tiling – to make the best use of this platform.

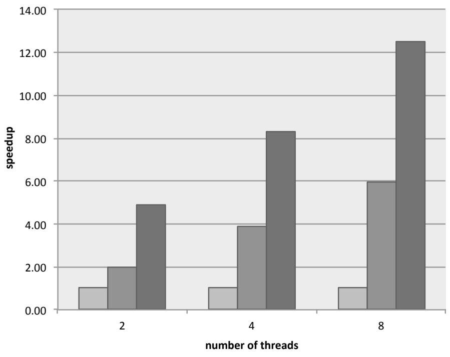
For the version modified to allow auto-parallelization, the speedups on the low resolution image range from 1.7 to 2.5 for 2-8 threads with the best performance observed for 4 threads. The speedups for the high resolution image were 1.7-3.1 and here the best result used all 8 threads.

The OpenMP code showed slightly better scaling with speedups ranging from 1.9-2.8 on 2-8 threads for the low resolution image and produced the best speedup using 4 threads. For the high resolution image, speedups were from 1.5 to 3.8 and the best result was obtained using 8 threads similar to the auto-parallelized version. Comparing the performance of the OpenMP and auto-parallelized versions shows that the latter delivers 79-89% of the performance of the former with the low resolution image and 80-111% for the high resolution image. Again the auto-parallelized version compares most favorably to the OpenMP version with 2-4 threads.

The `pthreads` version showed more modest speedups on the POWER platform. With the low resolution image, speedups were 1.7 on 2 threads but only 1.5 and 1.4 on 4 and 8 threads respectively. With the high resolution image, speedups were: 1.3 on 2 threads, 2.1 on 4 and 3.2 on 8 threads. With the small image, the auto-parallelized code delivered the same performance on 2 threads and 111-167% on 4 and 8 threads. With the big image, auto-parallelization delivers 130-135% on 2 and 4 threads and the same performance on 8 threads.



(a) Speedups on Intel Platform.



(b) Speedups on IBM Platform.

▨ Unmodified speedup

▨ OpenMP speedup

▨ Modified Speedup

(c) Legend

Figure 7.8: Edge detection speedups on x86 and POWER platforms. Speedups for the modified and auto parallelized code is shown with respect to the sequential performance and with respect to the unmodified, auto-parallelized code. In the version modified on the basis of comments, all three loop nests in the program are parallelized by `gcc`, in the unmodified version, only the two loop nests in the `main` function are parallelized.

7.7.3 Edge Detection Speedups for Intel Xeon

We measured the speedups when gcc parallelized the original edge detection code and the modified code relative to sequential execution and relative to the performance of the unmodified, auto-parallelized code. Finally, the results of auto-parallelization are compared with hand-parallelized OpenMP code. The speedups on the Intel Xeon system are summarized in Fig. 7.8a.

When auto-parallelizing the unmodified edge detection code with gcc, speedups are within 5%-10% since the most work intensive loop is not parallelized.

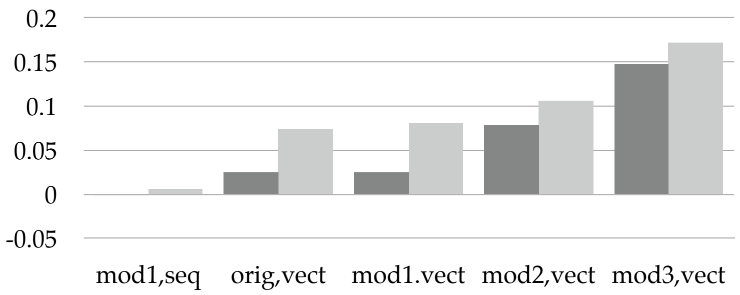
When auto-parallelizing the modified code, all three loop nests are transformed. The highest speedup of 8.32 used 16 threads, but a speedup of 7.8 is already obtained at 8 threads and 12 threads only resulted in a speedup of 7.0. Speedups on 2 and 4 threads are 2.44 and 4.62 which is super-linear.

The speedups of the OpenMP version ranged from 1.92 on 2 threads to 7.67 on 16 threads. Super-linear scaling was not observed. In effect, the auto-parallelized code outperformed the OpenMP code by 9-27% and the difference was greatest on 2 threads. The reason, we discovered, was that gcc was able to unroll the most frequently executed inner loop in the auto-parallelized version. The use of manually inserted OpenMP pragmas on the other hand seems to prevent such unrolling.

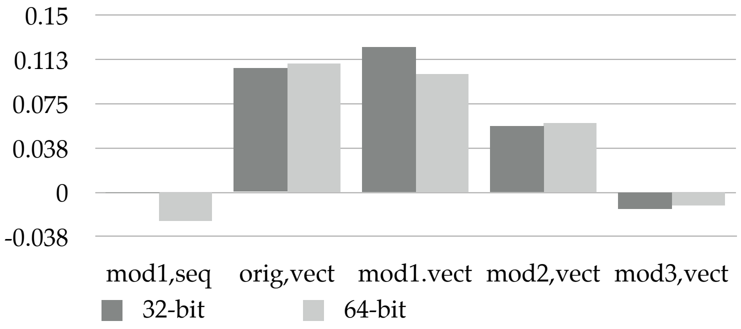
7.7.4 Edge Detection Speedups for IBM POWER

The speedups that were observed on the POWER6 machine are summarized in Fig. 7.8b. When running the unmodified, auto-parallelized edge detection code, we observed a performance improvement of 2% on 2-8 threads.

When auto-parallelizing the code with modification based on the code comments, however, we observe speedups ranging from 4.9 on two threads and up to 12.5 on 8 threads. In contrast, the OpenMP version saw speedups of 2.0 on 2 threads, 3.9 on 4 and 6.0 on 8 threads. Hence, the performance of the auto-parallelized version was 210-242% relative to the OpenMP version. Again, we attribute the difference to gcc's unrolling of the inner loop in the auto-parallelized version.



(a) JPEG Decoding Speedups on Intel Platform



(b) Speedups on IBM Platform

Figure 7.9: JPEG Decoding Speedups on Intel and IBM architectures. Modification 1 is the change from `signed long int` to `int32_t`. Modification 2 is the changes to allow vectorization of the first loop of the IDCT and modification 3 allowed vectorization of both loops.

7.7.5 JPEG Decoding Speedups for Intel Xeon and IBM POWER

The JPEG decoder was compiled in both 32 and 64-bit modes to determine how changing the word size affects the performance.

The modifications were grouped in three sets as follows: **mod1** is the redefinition of the typedef `e_s32` from a 64-bit integer (on 64 bit systems) to a 32-bit integer on all systems; **mod2** extends **mod1** with the modifications to allow the first loop in the IDCT routine to be vectorized; and **mod3** extends **mod2** with the modifications to allow the second loop in the IDCT routine to be vectorized. Figure 7.9a and figure 7.9b shows the performance before and after the three sets of modifications for the Intel and IBM systems. Unlike the previous measurements, the x-axis shows number of images decoded per second.

Intel Xeon platform Vectorizing the unmodified code increases performance by 2 and 7% for 32 and 64-bit builds respectively. With and without vectorization, the 64-bit build shows just a slight performance increase from **mod1**. Modifications to vectorize the first loop, **mod2**, increase 32-bit performance by an additional 5% and 64-bit performance by 3%. The modifications to fully vectorize the IDCT function, **mod3**, improve 32-bit performance by 12% and 64-bit performance by 10% relative to the original code with vectorization. Note that the Xeon platform has 128-bits vector units whereas newer Intel platforms – codenamed "Sandy Bridge" widens the vector units to 256-bit [79]. This would allow a higher vectorization factor and, possibly, a greater impact of the modifications.

IBM POWER Platform The first modification, **mod1** resulted in a 2% improvement over the original 32-bit code when vectorizing. Further modification to allow vectorization degraded the performance. It seems that the extra-work introduced by removing the short-circuit if-statements was not outweighed by the speedup from vectorizing the loops on this architecture.

7.8 Related Work

Early work which pioneered user interaction in an auto-parallelization process include the ParaScope Editor, SUIF Explorer and PAT [87, 106, 10]. They parallelize sequential FORTRAN codes based on stand-alone analysis and user-interaction.

This work targets the C language whose properties complicate analysis compared to FORTRAN. These properties include pointers, dynamic memory allocation and others [12]. It leverages the extensive analysis capabilities of a production compiler. This means that compiler feedback will adjust in response to improvements in the compiler analysis and in response to the use of different compiler flags. The integration with a production compiler is also important since the analysis of loop nests benefits from scalar optimizations such as if-conversion and function inlining and from optimizing at link time.

The mechanisms used to rule out potential data dependencies also differ. ParaScope and PAT store information on potential data dependencies which the programmer has suppressed outside the source code so this information can be obsoleted by changes to the source code. The compilation feedback system, on the other hand, suggest that the `restrict` keyword is used to eliminate sets of dependencies. This is a standardized mechanism understood by most compilers. It also works when the code is changed.

ReLooper is an Eclipse plug-in that can help the programmer parallelize regular loop nests in Java code [151]. Parallelization is done using the `ParallelArray` framework and not OpenMP. The former is limited to handle fewer kinds of parallel loops. Like our tool, ReLooper also relies on static data-dependence analysis to detect parallelism – but unlike in `gcc` the data-dependence analysis is inter-procedural. Using ReLooper, the programmer picks a target array and is then told if the loops which access the array can be parallelized safely. She can then choose to parallelize unsafe loops as is or make changes before re-running the analysis. This mode of interaction differs from our tool since we may also give refactoring suggestions that increase amenability to parallelization. Also, our approach is not data or array driven, but rather focuses on loops.

Sean Rul et al. proposed the Paralax infrastructure which also exploits programmer knowledge for optimization [161]. Paralax is comprised of three parts i) a compiler for automatic parallelization of outer loops containing coarse-grain pipeline-style parallelism, ii) a set of annotations which annotate data-dependencies which cannot be eliminated via static analysis and which are verified dynamically and iii) a tool which suggests how the programmer may add annotations to the program.

Paralax is complimentary to our work. It parallelizes irregular, pointer-intensive codes whereas we focus on codes amenable to automatic parallelization after some modification. The suggestions generated by the Paralax tool rely on both static analysis and profiling information whereas our suggestions, so far, do not require program profiling.

Suggestions for locality optimizations, SLO, provides refactoring suggestions at the source level aiming to reduce reuse distances and thus the number of cache misses [23]. The suggestions are based on cache profiling runs and are complimentary to the types of refactoring suggested by our tool. For instance, SLO does not help the programmer expose parallelism in the source code.

The latest releases of IBM XL C/C++ and Intel `icc` can generate compilation reports and may suggest changes in response to code which cannot be analyzed by the compiler. [54, 82]. This work is complimentary to ours and relies on vendor specific pragmas. We used `icc`'s *Guided Auto-Parallelism* feature on the demosaicing kernel by inserting `icc`-specific pragmas as suggested by the compiler. This allowed `icc` to parallelize a minority of the loops. The resulting performance varied from a marginal speedup to a sizable slowdown.

7.9 Summary

Many source codes are written in ways that prevent auto-parallelization of loops. A compilation feedback system was developed to address this. This infrastructure was evaluated on via two image processing kernels and a JPEG decoding program that pose problems for current production compilers. By refactoring application source code, more loops can be automatically parallelized and vectorized.

A demosaicing kernel delivered speedups of up to 6.0 on the Intel Xeon system (and up to 3.1 on the IBM POWER6 system). A edge-detection kernel was sped up by factors of up to 8.3 the Intel Xeon system and up to 12.5 on the IBM POWER6 system. The performance of a JPEG decoder was improved by 10 and 12% on the Intel Xeon system and by 2% on the IBM POWER6 system.

Automatic parallelization was compared with with manual-parallelization across different program inputs, systems and benchmark programs. Auto-parallelization delivered the best result in 12 cases, while hand-parallelization was better in 11 remaining situations. At low and medium thread counts auto-parallelization generally performed similar to or better than hand-parallelized and optimized codes.

For all three benchmarks, the performance improvement varied considerably between the two platforms. This shows that auto-parallelization and vectorization must be combined with platform-specific tuning. The codes studied also showed that programmer to chose among refactoring alternatives which affect performance differently. Finally, the study showed that prioritization of compiler feedback remains important: information from the compilation feedback system as well as other compilers are likely to consist of hundreds of individual comments, even when the code contains only a handful of missed opportunities for optimization.

CHAPTER 8

Refining Compiler Feedback

The compilation feedback mechanism discussed in the previous chapter emits a code comment whenever an optimization succeeds or must be aborted. An optimization may need to be aborted for several reasons. The optimization may simply not be legal and should indeed be aborted. Alternatively, the programmer may inadvertently have written code in a way that prevents automatic optimization. Often, the optimization may need to be aborted due to conservative correctness checks during code analysis. Finally, it may be the case that the analysis capabilities of the compiler lack the sophistication required to determine whether the optimization is legal or not. The compiler does not know which of these reasons explain the failure to optimize. It therefore outputs a code comment in all cases.

This means that far from all code comments are indicators of missed opportunities for optimization. The combination of large code bases and superfluous comments make it difficult for the programmer to identify comments that lead to improved program performance.

This study is based on compilation feedback from the EEMBC benchmark suite [36], which consists of 214 thousand source lines of ANSI C code[1]. Four production compilers report an average of 873 issues preventing parallelization

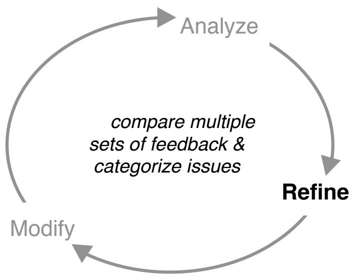
[1]When counted with SLOCCount by David A. Wheeler.

or vectorization when compiling EEMBC. The exact number of code comments depends on the compiler and compiler options. Expecting the programmer to examine each individual issue does not seem realistic.

The previous chapter was only concerned with code comments that report issues preventing optimization. This chapter is concerned with both *back offs*, which report issues; as well as *optimization* comments, which report successes. Further, a comment is said to be *resolvable* if it is likely to represent a missed optimization. Whether or not the optimization itself is profitable is not implied by this. Similarly, *un*resolvable comments are those which are unlikely to represent a missed optimization.

As with all kinds of optimization, most effort should be spent on back offs that improve program performance if resolved. Program profiling can identify code comments which relate to code that is executed often. Knowing the execution frequencies of program statements, however, can only rank a comment relative to *other* comments. Execution frequencies do not indicate if an individual code comment is likely to represent a missed optimization opportunity or not.

This chapter closes the analysis feedback loop by examining the refinement step which leads back to the modification step. Two techniques are examined which, unlike profiling, do not require programs to be executed. The techniques estimate whether each individual comment is likely to signify a missed optimization or not. These techniques compliment, rather than replace, profiling as a way to prioritize compilation feedback.



The following section explains the approach used to refine compiler feedback. Section 8.2 discusses implementation aspects and 8.3 presents the experimental results. Section 8.4 surveys related work and section 8.5 summarizes.

8.1 Approach

This study refines compilation feedback by comparing multiple sets of feedback generated by compiling the same source code. Different feedback is generated from each build by changing the compilation options or changing the compiler itself. The process consists of four steps:

Assistance The first step is to generate compilation feedback from multiple compilers. Compilers differ in their capabilities and their optimization strategies. Hence, one compiler may successfully optimize where others cannot. Compilation options that may increase performance or benefit automatic parallelization and vectorization are enabled. This includes inter-procedural analysis and link-time optimization.

Speculation The second step generates an additional set of compilation feedback for each compiler. Correctness checks that may otherwise block automatic parallelization and vectorization are relaxed via compilation options in these builds. The feedback generated in this step is speculative since it is unknown whether the relaxed assumptions preserves the correctness and performance of the compiled programs.

Pre-processing This step transforms all sets of compilation feedback into a common format so feedback from different compilers can be compared.

Comparison The intuition behind this step is simple: if one set of feedback contains a back off comment on a given code location and another set of feedback contains an optimization comment on the same location, the back off is potentially resolvable. If the optimization comment was generated in the assistance step, then the back off may be caused by a compiler limitation. If the optimization comment was generated in the speculation step, then the back off may be caused by conservative correctness checks. If a non-speculative build generated a back off and a speculative build using the same compiler generated no code comment at the same location, the compiler refrained from optimizing or providing further feedback. Such back offs are most likely unresolvable.

After the refinement process, each back off falls in one of three categories: potentially resolvable, potentially unresolvable or unknown. Ideally, the last category is as small as possible as it provides no guidance to the programmer. The list of potentially resolvable back offs can be further prioritized via program profiling. It is then up to the programmer to determine which of the potentially resolvable back offs are actually resolvable.

Table 8.1: Names and versions of the compilers used in this study.

Compiler	Product name	Version
`icc` [80]	Intel C++ Composer XE 2011 for Linux	12.0.2
`xlc` [83]	IBM XL C/C++ for Linux	11.1
`pgcc` [154]	PGI C/C++ Workstation for Linux	11.2
`suncc` [121]	Oracle Solaris Studio	12.2

8.1.1 Speculative Feedback

Certain compilation options allow a compiler to perform optimizations that it otherwise cannot. The options to relax correctness checks may also cause incorrect behavior in the compiled programs or degrade their performance. The compiled programs must therefore be discarded. However, the compilation feedback generated from speculation is potentially useful.

By assuming that no pointers are aliased in the code being compiled, for instance, the compiler may report optimizations in places where back off comments would otherwise appear. When an optimization comment appear in place of back off, the latter is likely to represent a missed opportunity for optimization.

In addition to the existence of various forms of aliasing, several other assumptions can be made. Speculation can also alter assumptions about side-effects from function calls, profitability of optimizations, conformance to language standards and the possibility of overflows in the intermediate computation of induction variables in loops [44]. By performing multiple speculative builds, which relax different assumptions, the programmer can be told more about the nature of the issue preventing optimization.

Last, the relaxed assumptions may allow the compiler to proceed past correctness checks that would otherwise have generated a back off only to abort the optimization at a later stage. Then, the compiler may or may not generate another back off. In the latter case, the programmer has no way to understand the issue, so the initial back off is considered unresolvable.

8.1.2 Assistive Feedback

Speculation can identify locations where conservative assumptions prevent optimization. However, when a single compiler is used to generate speculative and non-speculative builds, the feedback cannot detect limitations of that particular compiler. Using multiple compilers to generate compilation feedback from same source code identifies locations where one compiler determines to optimize and another generates a back off comment. The names and versions of the compilers used in this study are shown in table 8.1 on the facing page. The `gcc` compiler used in the previous study was not included since it does not support speculative compilation. The `opencc` compiler [35] was not included since it does not generate code comments. Finally, the `pathcc` compiler from PathScale [124] was not included as source code locations are not reported accurately during link-time optimization.

The idea is that all locations where one or more compilers optimize are safe to optimize similarly by all other compilers – assuming no compiler makes an error when determining to optimize. In addition, feedback from other compilers can identify locations where a compiler neither optimizes nor generates a back off comment. This too may be the result of a compiler limitation.

8.1.3 Pre-processing of Feedback

The syntax and semantics of code comments differ among compilers. Some compilers write feedback to the console just like errors and warnings. Others write feedback to files. Some compilers write all comments to a single file – others write a file for each translation unit. Most compilers use a custom syntax while `xlc` outputs XML. Finally, some files may be compiled more than once during the same build, this creates duplicate comments, which must be ignored.

During a single invocation, a compiler may generate both back off comments and optimization comments for the same source code location. For instance, the `icc` compiler generates an optimization comment in the same location where it emits one or more back off comments. These optimization comments do not report to actual optimizations and must also be ignored.

For these reasons, compiler feedback from different compilers and even from different compilation passes need to be pre-processed before feedback from multiple compilers and multiple builds can be analyzed in a unified fashion. For each comment, pre-processing marks whether the comment is an optimization or back off, whether it is speculative or not and the type of optimization that generated the comment, etc.

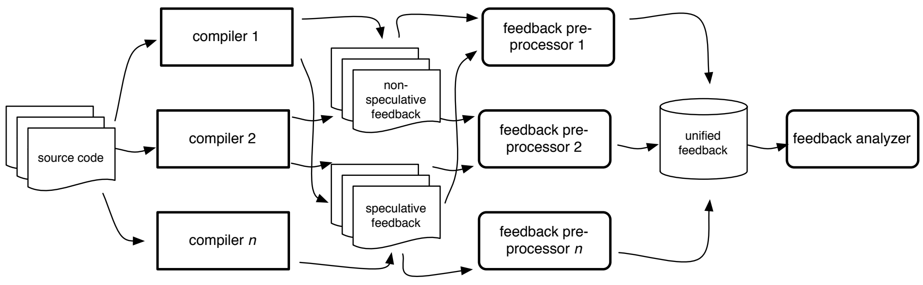


Figure 8.1: Refinement of compilation feedback. Feedback from multiple compilers and multiple builds is pre-processed and analyzed to estimate which back offs are potentially resolvable and which are not.

8.2 Implementation

The current implementation of the techniques outlined in the previous section is shown in figure 8.1. It consists of a set of Python scripts. The compilation feedback from each build is then pre-processed via a set of *feedback pre-processors*. One specialized to parse the feedback from a specific compiler as show by the column of three rounded rectangles. This allows the subsequent analysis to be compiler independent and makes it straightforward to add support for additional compilers. A separate set of scripts analyze the unified feedback. These correspond to the rightmost rectangle in the figure.

8.2.1 Correlating Code Comments

Source code locations consisting of a file path and a line number are used to correlate code comments. Code comments, which do not contain both of these, are therefore discarded.

Even when locations are properly reported, care must be taken when correlating code comments from different compilers. Consider the two code snippets shown in figure 8.2 on the facing page showing two common coding styles. Compilers may output code comments on the line which begins the loop or on the line which contains the opening bracket. For the coding style on the left, this does not matter. With the coding style on the right, however, different compilers may place comments relating to the same loop on different lines.

For the benchmark code and the compilers used in this study, this was not a problem. The current implementation can detect but not automatically handle situations where related code comments refer to adjacent lines.

```
for(i = 0; i < LEN; i++) {

```

(a) Opening brackets on *first* line of loops.

```
for(i = 0; i < LEN; i++)
{
```

(b) Opening brackets on *second* line of loops.

Figure 8.2: Differences in coding styles must be accounted for when correlating code comments.

Correlating comments via loop hierarchies This study uses source code locations to correlate code comments. Additional code comments could be compared if the loop hierarchy is taken into account. Assume that compiler a reports optimization at line l_1 and that another compiler b reports an optimization at line l_2 where $l_1 \neq l_2$. Without considering the nesting of the loops, it may seem that the compilers compliment each other. Each finds an optimization opportunity that the other one does not. Then assume that line l_1 begins a loop nested inside the loop starting on line l_2 and that both compilers perform loop parallelization. In that case, the compilers do not compliment each other, rather, compiler b does better than compiler a since parallelization of the outer loop is likely to be more profitable. Using the loop hierarchy to correlate additional code comments is left for future work.

8.2.2 Choosing Compiler Options

Modern compilers implement a large number of optimizations that interact in intricate ways [38]. A pair of compiler optimizations may interfere in both positive and negative ways depending on the program being compiled [31]. Though empirical evaluation, *iterative compilation* can find optimization strategies that work well for a particular combination of compiler, program and dataset, etc.

It has been shown that iterative compilation significantly improves compilation results (with respect to a single or multiple objectives) relative to compilers that apply optimizations in a fixed order [39, 3, 30, 77]. The main drawback of iterative compilation is the time required to repeatedly recompile and execute the same program.

The number and type of code comments can be viewed as objectives to be optimized through iterative compilation. Hence, iterative compilation could be used to find the empirical best set of compilation options for both assistive and speculative builds.

The current implementation does not support automatic iterative compilation to evaluate the effect of compilation options. Manual experimentation was used to evaluate several compiler options. It was found that high optimization levels, aggressive inline expansion and link-time optimization allows more loops to be auto-parallelized and vectorized.

8.3 Experimental Results

This section evaluates four aspects related to the refinement of compiler feedback. First, it evaluates how the locations of code comments overlap among different builds. Second, the amount of back offs that can be classified as potentially resolvable or not via speculation. Third, the number of back offs that can be classified by comparing compiler feedback generated from the four compilers listed in table 8.1 on page 120. Finally, the synergies between the speculation and assistance techniques are evaluated.

A non-speculative build and a speculative build of the EEMBC benchmark for each compilers form the basis of the experimental evaluation. Table 8.2 on the facing page shows the compiler options used to control optimization, feedback and speculation for the eight builds. Assistive builds use options in the optimization and feedback categories. Speculative builds adds the options listed in the speculation category. The compiler options were chosen experimentally by observing the amount of feedback generated as mentioned in the previous section. Inter-procedural analysis and link-time optimization were enabled in all instances.

8.3.1 Overlap of Compiler Feedback

Table 8.3 on page 126 shows how the locations of 3490 non-speculative back offs overlap with the locations of other code comments. The majority of the code comments (94%) are co-located with other comments. This is important since this study only compares code comments that reference the same source code location.

Approximately 20% of the back offs are only co-located with code comments generated by the same compiler. This means that only speculation can be

Table 8.2: Compiler options used to generate feedback from speculative and non-speculative builds.

Purpose	Compiler Options
	Intel C++ Composer XE 2011 for Linux
Optimization	`-fast -parallel -vec`
Feedback	`-guide -opt-report 3 -vec-report5 -par-report5`
Speculation	`-ansi-alias -no-ansi-alias-check -fargument-noalias -fargument-noalias-global -opt-subscript-in-range -par-threshold0 -vec-threshold0`
	IBM XL C/C++ for Linux
Optimization	`-O5 -qsimd -qsmp=auto -qthreaded -qhot`
Feedback	`-qlistfmt=xml=all -qreport`
Speculation	`-qalias=ansi:global:allptrs -qrestrict -qlibansi -qinline=auto:level=10 -qassert=refalign -qdebug=NSIMDCOST`
	PGI C/C++ Workstation for Linux
Optimization	`-fastsse -mp -Mvect -Mconcur -Mipa=fast -Minline`
Feedback	`-Minfo -Mneginfo`
Speculation	`-Mconcur=cncall,noaltcode -vect=noaltcode -Msafeptr=all`
	Oracle Solaris Studio
Optimization	`-fast -xautopar -xparallel -xvector -xipo -xtarget=native`
Feedback	`-xloopinfo`
Speculation	`-xrestrict=%all -xalias_level=strong`

used to address these back offs. About 29% of the back offs are co-located with optimizations from the same or another compiler. This means that either speculation or assistance can potentially address these back offs. Finally, 45% of the back offs are co-located with back offs from other compilers. In absence of evidence to the contrary, these back offs are categorized as unaddressable.

The percentages are shown as pie-charts in the left half of figure 8.3. The right half shows the types overlap among locations with code comments rather than percentage of overlapping back offs. Up to 92% of all locations with code comments can be addressed by the techniques presented here. The percentage differs from the percentage of back offs (94%) because several back offs can share a single source code location.

Table 8.3: Co-location of back offs and other code comments.

Correlation	back offs	back offs (%)
Single compiler, single build (isolated comments)	201	6%
Single compiler, multiple builds	704	20%
Multiple compilers, optimizations and back offs	998	29%
Multiple compilers, back offs only	1587	45%

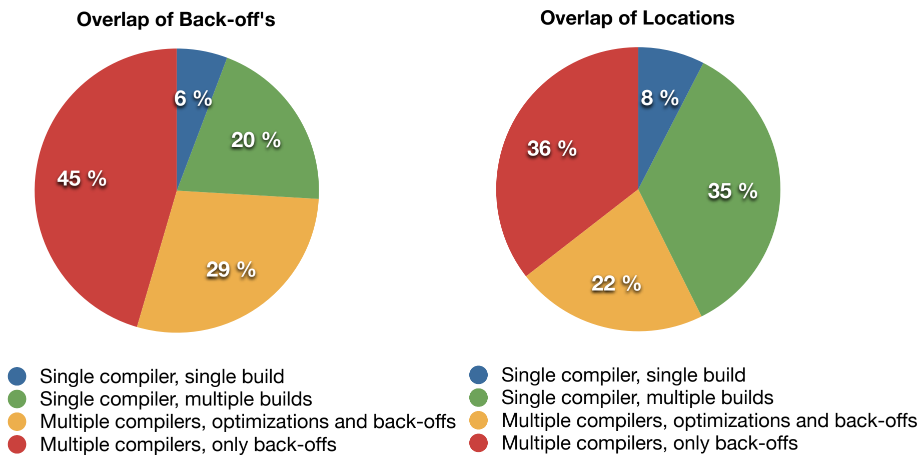


Figure 8.3: Types of overlap among back offs and among all locations with code comments.

8.3.2 Refinement via Speculation

The type and degree of speculation depend on the compiler options available. The compiler options added to enable speculation were specified in "Speculation" category in table 8.2 on the preceding page. With `icc`, speculative builds assume that no aliasing is possible (parameters containing `alias`), that computation of induction variables do not cause overflows (`-opt-subscript-in-range`) and that vectorization and parallelization and vectorization is always profitable (`-par-threshold0` and `-vect-threshold0`).

With `xlc`, speculative builds also assume lack of aliasing (`-qalias=...`), that pointers can be treated as restrict qualified (`-qrestrict`), that functions with the name of ANSI C library functions in fact from the ANSI C library (`-qlibansi`). It is also assumed that pointers point to naturally-aligned data (`-qrefalign`) and that vectorization is always profitable (`-qdebug=NSIMDCOST`). Finally, the

Table 8.4: Effect of speculation.

	Optimizations			back offs		
Compiler	Baseline	Speculative	Delta	Baseline	Speculative	Delta
`icc`	126	577	451	224	68	156
`xlc`	58	82	24	671	724	-53
`pgcc`	75	103	28	1186	983	203
`suncc`	71	73	2	1409	1389	20

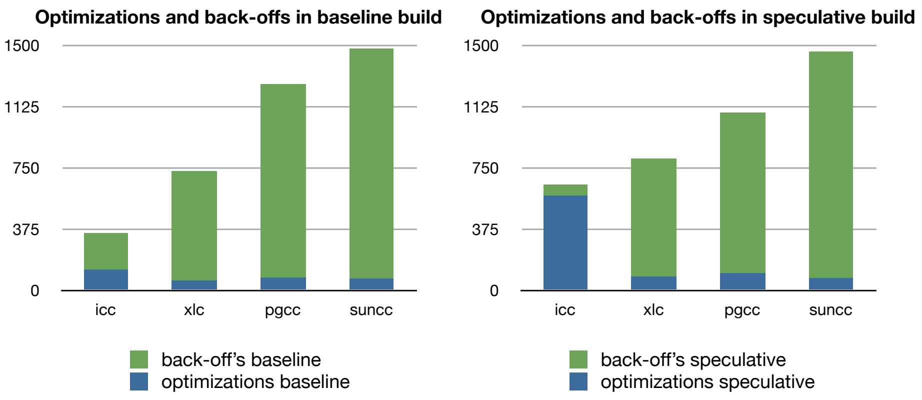


Figure 8.4: Optimization and back off comments in baseline and speculative builds.

compiler is asked to perform inline expansion whenever possible (`-qinline=auto:level=10`).

The last two compilers, `pgcc` and `suncc`, offer fewer compiler options to control speculation. Speculative builds using `pgcc` only assume absence of aliasing (`-Msafeptr=all`) among pointers, that parallelization is profitable and that calls in loops are safe to parallelize (`-Mconcur=cncall`). Also, when parallelizing and vectorizing speculatively, the compiler should not generate alternate serial code (`-Mconcur=noaltcode -Mvect=noaltcode`). Finally, speculative builds with `suncc` only assume that pointers can be treated as being restrict qualified (`-xrestrict=%all`) and that type-based alias-analysis can be used `-xalias_level=strong`) to rule out aliasing between variables that have incompatible types.

The effect of speculation is shown in table 8.4 and graphically in figure 8.4. As expected, speculation enables more optimizations to be performed. When speculation converts a back off comment into an optimization, the back off is likely to represent an opportunity for optimization. The `icc` compiler reports 451 additional optimizations whereas `suncc` performs only two additional optimizations. The reasons are likely twofold: `icc` offers many ways to relax assumptions whereas `suncc` can only relax assumptions about aliasing. Moreover, it seems that loops parallelized by `suncc` include runtime checks for aliasing. This means that with `suncc` aliasing is unlikely to be a major obstacle to parallelization or vectorization in the first place. Except for `xlc`, all compilers generate fewer back offs when compiling speculatively.

The effect of speculation for each of the compilers is detailed in figure 8.5 on the next page. These are determined by counting how optimizations and back offs correlated between baseline and speculative builds. Eight combinations are possible:

1. baseline optimization → speculative optimization
2. baseline back off → speculative optimization
3. no comment in baseline build → speculative optimization
4. baseline optimization → speculative back off
5. baseline back off → speculative back off
6. no comment in baseline build → speculative back off
7. baseline optimization → no comment in speculative build
8. baseline back off → no comment in speculative build

Relations 2 and 8 are of particular interest since they indicate that a back off can or cannot be addressed cf. section 8.1.1 on page 120. Relation 3 is also an indicator of missed opportunities for optimization. However, it does not help the programmer prioritize existing back offs, which is the goal of this study.

Out of a total 224 back offs generated by `icc`, 88 (39%) are likely to be resolvable (relation 2) and an additional 30 (13%) back offs are unlikely to be resolvable (relation 8). Overall 118 (53%) of the back offs produced by `icc` can therefore be addressed by speculation.

Speculation with `xlc` identifies 14 (2%) of back offs as resolvable and 252 (38%) as unresolvable – in combination this addresses 40% of all back off comments generated by `xlc`.

For `pgcc`, speculation identifies 26 (2%) of back offs as resolvable and 195 (16%) as unresolvable. This addresses 19% of the total back off comments generated

	Speculative Optimizations	Speculative Back offs	Speculative Nothing
Baseline optimizations	114	0	6
Baseline back offs	88	106	30
Baseline nothing	254	6	
	Shared	Unique	Overlap
Locations	251	200	55.7%

(a) Effect of speculation with `icc`.

	Speculative Optimizations	Speculative Back offs	Speculative Nothing
Baseline optimizations	14	10	26
Baseline back offs	14	369	252
Baseline nothing	73	294	
	Shared	Unique	Overlap
Locations	229	307	42.7%

(b) Effect of speculation with `xlc`

	Speculative Optimizations	Speculative Back offs	Speculative Nothing
Baseline optimizations	73	0	2
Baseline back offs	26	895	195
Baseline nothing	4	21	
	Shared	Unique	Overlap
Locations	924	221	80.7%

(c) Effect of speculation with `pgcc`

	Speculative Optimizations	Speculative Back offs	Speculative Nothing
Baseline optimizations	56	1	0
Baseline back offs	8	1348	9
Baseline nothing	0	2	
	Shared	Unique	Overlap
Locations	1123	1083	50.9%

(d) Effect of speculation with `suncc`

Figure 8.5: Effect of speculation for the individual compilers. Back offs identified as likely resolvable are highlighted with green and back offs that are unlikely to be resolvable with red. Additional back offs generated by speculation are highlighted with yellow.

by `pgcc`. Finally, speculation with `suncc` merely identifies 8 of back offs as resolvable and 9 as unresolvable – which constitutes little more than a percent of the total back offs reported by `suncc`.

Speculation with `icc` is particularly suited to identify back offs that are likely to be resolvable. This may be explained by `icc`'s many options to relax assumptions and its conservative cost model. With `xlc` and `pgcc` speculation is most effective at finding back offs which are unlikely to be resolvable. Speculation is mostly ineffective at addressing the back offs reported by `suncc`. Unsurprisingly, the compilers allowing several types of speculation did better than `suncc`, which offers few options to relax assumptions. The next section evaluates the assistance technique which compares code comments generated by different compilers.

8.3.3 Refinement via Assistance

The idea behind refinement via assistance is simple: if one compiler emits a back off comment where another compiler emits an optimization comment, then the back off comment is potentially resolvable. This section examines how each of the four compilers can be assisted by the three others.

Refining `icc` feedback via assistance Tables 8.5-8.7 on page 131 compares the number and types of code comments generated by `icc` with those generated by the three other compilers used in this study. Table 8.5 shows that `icc` generated 25 back off comments in locations where `xlc` optimized. This means that `xlc` can be used to mark 11% of the comments generated by `icc` as likely to be resolvable. Similarly, `pgcc` and `suncc` can mark 51 (23%) and 37 (17%) of `icc` generated back offs as resolvable. When combined, the three compilers allows 74 (33%) of the `icc` back offs to be marked as likely to be resolvable.

Refining `xlc` feedback via assistance Contrary to `icc`, assistance is mostly ineffective at addressing back off comments generated `xlc`. Tables 8.8-8.10 on page 132 compares the number and types of code comments generated by `xlc` with those generated by `icc`, `pgcc` and `suncc`. Table 8.8 shows that `xlc` generated 14 back off comments in locations where `icc` optimized. The remaining compilers did not optimize where `xlc` output a back off comment, which means that they were not of assistance. In total, only 2% of the comments generated by `xlc` can be marked as potentially resolvable by comparing with feedback generated by the three other compilers.

Table 8.5: Number of code comments generated by `icc` and `xlc` correlated by location and categorized by the eight combinations of comment pairs. Back offs generated by `icc` which may represent missed opportunities for optimization are highlighted in green.

	Opt.'s `xlc`	back offs `xlc`	Nothing `xlc`
Opt.'s `icc`	7	6	112
back offs `icc`	25	49	150
Nothing `icc`	25	567	
	Shared	Unique	Overlap
Locations	59	526	10.1%

Table 8.6: Number of code comments generated by `icc` and `pgcc` correlated and categorized as table 8.5.

	Opt.'s `pgcc`	back offs `pgcc`	Nothing `pgcc`
Opt.'s `icc`	24	73	29
back offs `icc`	51	107	66
Nothing `icc`	2	953	
	Shared	Unique	Overlap
Locations	206	994	17.2%

Table 8.7: Number of code comments generated by `icc` and `suncc` correlated and categorized as table 8.5.

	Opt.'s `suncc`	back offs `suncc`	Nothing `suncc`
Opt.'s `icc`	8	107	11
back offs `icc`	37	159	28
Nothing `icc`	21	1092	
	Shared	Unique	Overlap
Locations	247	921	21.1%

Table 8.8: Number of code comments generated by `xlc` and `icc` correlated by location and categorized by the eight combinations of code comment pairs. Back offs generated by `xlc` which may represent missed opportunities for optimization are highlighted in green.

	Opt.'s `icc`	back offs `icc`	Nothing `icc`
Opt.'s `xlc`	7	24	25
back offs `xlc`	14	54	567
Nothing `xlc`	112	150	
	Shared	Unique	Overlap
Locations	59	526	10.1%

Table 8.9: Number of code comments generated by `xlc` and `pgcc` correlated and categorized as table 8.8.

	Opt.'s `pgcc`	back offs `pgcc`	Nothing `pgcc`
Opt.'s `xlc`	16	27	13
back offs `xlc`	0	225	410
Nothing `xlc`	59	941	
	Shared	Unique	Overlap
Locations	176	1126	13.5%

Table 8.10: Number of code comments generated by `xlc` and `suncc` correlated and categorized as table 8.8.

	Opt.'s `suncc`	back offs `suncc`	Nothing `suncc`
Opt.'s `xlc`	4	51	0
back offs `xlc`	0	341	294
Nothing `xlc`	63	1069	
	Shared	Unique	Overlap
Locations	240	1007	19.2%

Refining pgcc feedback via assistance Tables 8.11-8.13 on page 134 compares the number and types of code comments generated by pgcc with those generated by the three other compilers. Table 8.11 shows that pgcc generated 90 back off comments in locations where icc optimized. Consequently, icc can mark 8% of the comments generated by pgcc as likely to be resolvable. On the other hand, xlc and suncc can only mark 23 and 27 (2% each) of pgcc generated back offs as resolvable. Assistance from the three other compilers allows 133 (11%) of the pgcc back offs to be marked as likely to be resolvable.

Refining suncc feedback via assistance Tables 8.14-8.16 on page 135 compares feedback generated by pgcc with feedback generated by the other compilers. Table 8.14 shows that suncc generated 141 back off comments in locations where icc optimized. Hence, icc can be used to mark 10% of the comments generated by suncc as likely to be resolvable. However, xlc and pgcc can only mark 44 and 47 (3% each) of the back offs as resolvable. The combined assistance of three other compilers lets 194 (14%) of the suncc back offs be classified potentially resolvable.

The compiler that benefitted most from assistance was icc. One third of all back offs reported by icc reference locations optimized by one of the other compilers. Each compiler could address at least 11% of the back offs reported by icc and pgcc alone could address more than twice as much. Recall that speculative compilation with icc lead to a dramatic increase in the number of reported optimizations. Together, these two observations suggest that icc uses a conservative cost model relative to the other compilers. Two other compilers, pgcc and suncc, also benefit from from assistance – primarily from icc. Finally, xlc could address 11% of the back offs generated by icc but other compilers were mostly unable to address the back offs generated by xlc.

Stand-alone optimizations The first row, third column in tables 8.5-8.16 shows that, in several instances, one compiler reports optimizations where another compiler does not output a back off comment. For instance, icc optimizes at 112 times at locations where xlc does not emit any comment according to table 8.8 on the preceding page. Similarly, pgcc reports 59 optimizations in places where xlc does not report failure to optimize.

This may happen for one of the following reasons:

1. Compilers emitted code comments in different locations. Consider optimization of a loop in a function which was inlined. Comments from one compiler may refer to the location of the call site. Another compiler may

Table 8.11: Number of code comments generated by `pgcc` and `icc` correlated by location and categorized by the eight combinations of code comment pairs. Back offs generated by `pgcc` which may represent missed opportunities for optimization are highlighted in green.

	Opt.'s `icc`	back offs `icc`	Nothing `icc`
Opt.'s `pgcc`	24	49	2
back offs `pgcc`	90	73	953
Nothing `pgcc`	29	66	
	Shared	Unique	Overlap
Locations	206	994	17.2%

Table 8.12: Number of code comments generated by `pgcc` and `xlc` correlated and categorized as table 8.11.

	Opt.'s `xlc`	back offs `xlc`	Nothing `xlc`
Opt.'s `pgcc`	16	0	59
back offs `pgcc`	23	152	941
Nothing `pgcc`	13	410	
	Shared	Unique	Overlap
Locations	176	1126	13.5%

Table 8.13: Number of code comments generated by `pgcc` and `suncc` correlated and categorized as table 8.11.

	Opt.'s `suncc`	back offs `suncc`	Nothing `suncc`
Opt.'s `pgcc`	28	46	1
back offs `pgcc`	27	719	370
Nothing `pgcc`	2	446	
	Shared	Unique	Overlap
Locations	775	699	52.6%

Table 8.14: Number of code comments generated by `suncc` and `icc` correlated by location and categorized by the eight combinations of code comment pairs.

	Opt.'s `icc`	back offs `icc`	Nothing `icc`
Opt.'s `suncc`	8	41	21
back offs `suncc`	141	132	1092
Nothing `suncc`	11	28	
	Shared	Unique	Overlap
Locations	107	985	21.1%

Table 8.15: Number of code comments generated by `pgcc` and `xlc` correlated by location.

	Opt.'s `xlc`	back offs `xlc`	Nothing `xlc`
Opt.'s `suncc`	4	0	63
back offs `suncc`	44	252	1069
Nothing `suncc`	0	294	
	Shared	Unique	Overlap
Locations	225	867	19.2%

Table 8.16: Number of code comments generated by `suncc` and `pgcc` correlated by location.

	Opt.'s `pgcc`	back offs `pgcc`	Nothing `pgcc`
Opt.'s `suncc`	28	41	2
back offs `suncc`	47	872	446
Nothing `suncc`	1	370	
	Shared	Unique	Overlap
Locations	775	699	52.6%

emit comments at the location of the loop as it appears in the source code file.

2. Two compilers choose to optimize at different levels of a loop nest. One compiler may optimize at the outermost level and one at the innermost level – and neither generated back off comments for the levels not optimized.

3. Compiler did not attempt to optimize because it did not discover an opportunity to do so.

4. Compiler attempted optimization but refrained and did not report why it did so.

Cases 1 and 2 may be addressed by determining the nesting of loops as discussed in section 8.2 on page 122 as well as the locations of loops, functions and function calls in the source code.

Cases 3 and 4 help compiler writers identify areas where a particular compiler underperforms. In case 3, stand-alone optimizations indicate places where a compiler did not recognize an opportunity for optimization. This may be caused by an unfortunate selection of compiler options or due to lack of sophistication in the compiler itself. In the last case, the compilation feedback mechanism can be improved. Clearly, if the compiler deliberately chooses not to optimize it should at least report that optimizing seems unprofitable.

8.3.4 Combining Speculation and Assistance

Some back off comments may be categorized by either speculation or assistance. Yet others can only be categorized by the combination of speculation and assistance. The categories are as follows:

Assistance or speculation Back offs in this category can be addressed by either of the two techniques evaluated in the preceding sections.

Assistance and speculation These back offs can only be addressed when the two techniques are combined.

Assistance only Assistance is required to address such back offs.

Speculation only Only speculation can be used to categorize back offs as resolvable.

Suppressible Speculation can be used to categorize back offs as unresolvable.

Pseudo back off Back offs that appear in conjunction with optimization comments from the same build are not considered to be as important as regular back offs. Either the compiler overcame the issue initially preventing optimization – or it reported an additional opportunity for optimization.

Table 8.17 shows the breakdown of back offs generated by each of the four compilers. The data is also shown graphically in figure 8.6 on page 139.

The ability of the techniques to classify back offs vary among the compilers. With `icc`, assistance and speculation both work well and in most cases either of the techniques expose resolvable back offs. Combining speculation and assistance does not help much however. In total, 60% of the back offs `icc` can be addressed by assistance and speculation.

With `xlc`, suppressing back offs via speculation is by far the most effective technique. Only a modest number of back offs, 10%, can be categorized as potentially resolvable. In total, 43% of `xlc` back offs are addressable.

For `pgcc` suppression of back offs via speculation is also most effective. Assistance and the combination of assistance and speculation is also fairly effective and categorized 22% of the back offs as potentially resolvable. The share of addressable back offs is 37%.

Finally, `suncc` also benefits most from assistance combined with speculation and assistance alone. On the other hand, self speculation is mostly ineffective with `suncc`. This means that back offs are difficult to address if other compilers cannot be used for assistance. Overall, 30% of all back `suncc` offs can be addressed

Table 8.17: Back offs by category. Pseudo back offs are not considered to be addressable.

	`icc`	`xlc`	`pgcc`	`suncc`
assistance or speculation	62	0	3	0
assistance and speculation	6	37	109	215
assistance only	12	14	130	194
speculation only	26	14	23	8
suppressible	28	222	179	9
pseudo back off	0	36	70	44
unaddressable	90	348	672	939
total back offs	224	671	1186	1409
total addressable (%)	60%	43%	37%	30%

Table 8.18: Feedback locations by category.

	`icc`	`xlc`	`pgcc`	`suncc`
assistance or speculation	58	0	3	0
assistance and speculation	2	18	100	148
assistance only	7	6	102	145
speculation only	22	8	23	5
suppressible	27	96	178	6
pseudo back off	0	25	69	20
unaddressable	50	190	639	768
total locations	116	343	1114	1092
total locs. addressable (%)	70%	37%	36%	28%

– the majority are categorized as potentially resolvable. This makes `suncc` the compiler that responds least favorably to the techniques evaluated in this chapter. However, `suncc` is also the compiler that generates the most feedback by far. It generated 1409 back offs compared to only 166 back offs generated by `icc`. This suggests that the fewer back offs a compiler generates, the more back offs are addressable by the techniques evaluated here. In total, these techniques can address 43% of the 3490 non-speculative back offs generated by the four compilers.

Compilers often generate multiple back off comments on the same location. These typically describe different issues preventing the same optimization. The effects of assistance and speculation can therefore also be calculated by categorizing locations rather than individual back offs. Table 8.18 use the same categories as table 8.17 on the previous page but it shows the number of locations rather than the number of individual back offs in each category. These numbers supports the conclusions drawn from table 8.17. Up to 70% of locations containing back offs from `icc` can be addressed while this is only true for 28% of the locations where `suncc` reports a back off. The average percentage of addressable locations is 43% – similar to the average percentage of addressable back offs.

8.4 Related Work

Callahan et al. [28] describe 100 loops written in Fortran with the purpose of testing the effectiveness of automatic vectorizing compilers. Their results show which loops are fully or partially vectorized by 19 different compilers of varying degrees of maturity. Like this study, it was found that the capabilities among

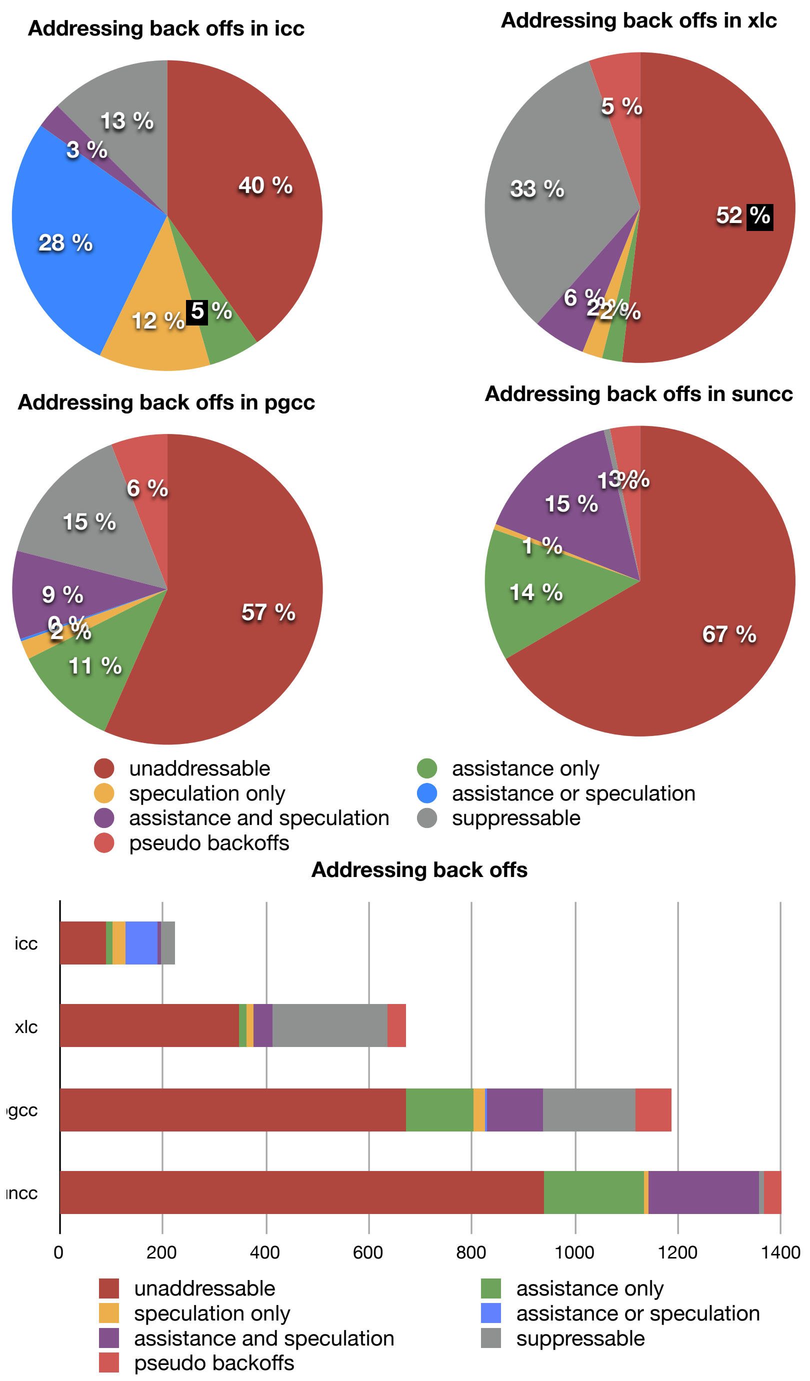


Figure 8.6: Addressing back offs in each of the four compilers by classifying them according to seven different categories based on their overlap with other code comments.

compilers varies significantly. On average 55% of the loops were vectorized whereas the best compiler could vectorize as much as 80%. Only 4% of the loops could not be vectorized by any of the compilers tested. The test suite was later converted to C by Smith [147] and tested with four additional compilers. Like the original study, this revealed significant differences in the vectorization capabilities of the compilers. Whereas this study seeks to quantify the synergies between different compilers, the works by Callahan et al. and Smith focus on the quality of the compilers tested and on the quality of the test suite itself.

More recently, Garzarán et al. [65] studied the vectorization capabilities of two of the compilers also studied here: `xlc` from IBM and `icc` from Intel. The study used C versions of the loops described by Callahan et al. [28] as well as additional loops contributed by the authors. It was found that out of 134 vectorizable loops, only 49 loops were vectorized by both compilers whereas 51 loops were vectorized by one compiler but not the other. Similar to this work, the study demonstrates the existence of synergies between different compilers.

The techniques used in this study are related to *interative compilation* [39, 3, 30, 77, 64]. Recompiling the same code while varying the context – e.g. optimization flags – is central to both approaches. As mentioned in section 8.2 on page 122, this work could use iterative compilation to find the empirical best compilation options – e.g. those which allow the greatest number of loops to be optimized in both speculative and non-speculative builds. Conversely, the research on iterative compilation could also benefit from this effort to expose additional optimization opportunities. Annotation and refactoring change the relative impact of compiler optimizations. Hence, interactive compilation may discover an improved optimization strategy after additional parallelism has been exposed.

Fursin and Temam's [64] work on collective optimization is perhaps closest to this study. They compare optimization results between multiple builds on various architectures, programs and datasets and show that the resulting performance is highly dependent on the compilation options. Similarly, this study compared results between multiple builds obtained from different compilers and show that the type and volume of code comments varies significantly with the choice of compiler and compilation options.

8.5 Summary

Compilers generate back offs for several reasons. These include restrictive assumptions about aliasing, profitability, etc. Compilers may also generate back offs due to lack of sophistication and when a given optimization is simply not legal. This results in a high volume of comments and creates a “needle in a haystack” problem for programmers.

This study explored compile time techniques that help programmers refine compilation feedback. These techniques do not require program profiling to work. Profiling, however, can be used to further prioritize compilation feedback. The techniques used in this study can also help compiler writers understand strengths and weaknesses of one compiler relative to others.

Four compilers, `icc`, `xlc`, `pgcc` and `suncc`, were used to evaluate how speculation and assistance help refine compilation feedback. The impact of the techniques varied considerably between the compilers. For one compiler, assistance was most effective. For another, speculation performed better. For yet another, the combination of assistance and speculation was most effective. In conclusion, both techniques are equally important and synergies exist between them.

Both speculation and assistance worked well with `icc` and the majority of the back offs generated were addressable by both techniques. On the other hand, assistance had modest impact on back offs from `xlc`. However, speculation was effective in finding `xlc` back offs that are not likely to be resolvable. With `pgcc` assistance was quite helpful on its own and when combined with speculation. Speculation on was also helpful in finding unresolvable back offs. Finally, `suncc` generated the greatest amount of feedback and did not benefit from self speculation. Assistance on its own and when combined with speculation by other compilers was able to address almost a third of the `suncc` back offs though.

Overall, 43% out of 3490 back offs generated collectively by the four compilers can be categorized as potentially profitable or unprofitable. The study also showed that speculation and assistance was most effective for the compiler that generated fewest back offs and offered the most options for speculative builds. Finally, it was observed that one compiler may report optimizations in places where another compiler generates no code comments. This indicates that the latter compiler can be improved – either it should generate a back off comment explaining the lack of optimization or it should be enhanced to discover the missed opportunities for optimization.

Chapter 9

Conclusions

Limitations in our ability to analyze program source codes make it challenging to discover parallelism in sequential programs *and* make it challenging to compute the task graph representing a parallel program.

This thesis argues that software development relies on feedback loops; that the interaction between program analyzers and the programmer forms a feedback loop; and, finally, that addressing the challenges at each step in this feedback loop increases our ability to analyze, model and optimize parallel programs.

9.1 Annotations for Task Graph Extraction

The first study in the thesis is motivated by the need to extract task graphs from programs expressed in imperative programming languages. The data dependencies among statements in such programs are implicit. On the other hand, data dependencies in task graphs are explicit to capture precedence constraints among tasks. Manually extracting task graphs is tedious and error prone whereas automated extraction via program analysis will conservatively over-approximate dependencies among tasks. This suggested a hybrid approach.

The study aimed to demonstrate that programmer inserted annotations compliment information gathered from program analysis and affords more precise task graphs. By design, the correct use of such annotations cannot be verified via program analysis. Hence, the study also intended to show that incorrect use of the annotations can be detected at runtime without impairing performance.

Two directives, `depends_t` and `depends_sc`, were introduced. Annotating programs with the directives reduces the number of assumed data-dependencies among tasks in OpenMP programs. Unlike results from a dynamic analysis, information from the directives is valid across all program inputs assuming correct use of the directive. Support to check that the runtime observable dependencies match the claims of the directives was also developed.

Three aspects of the directives were evaluated: i) the programmer effort required to insert the proposed directives; ii) the performance impact of the runtime checks; and iii) the ability of the second directive to exclude data dependencies that would otherwise have been assumed to exist between tasks in two different stencil computations.

Thirteen `depends_t` directives were added to a sorting benchmark. Between one and six `depends_sc` directives, less than four on average, were inserted in the four remaining benchmarks. Annotating the integer sort required less than half a working day. The estimated effort to insert `depends_sc` directives was about an hour per benchmark.

The number of estimated dependencies between two stencil computations with annotations was compared to a conservative estimate. Four benchmarks saw decreases between 50-99%. The average decrease across the benchmarks was 69%.

Two-sided, unpaired t-test was used to determine if the cost of runtime checking had a statistically significant impact on the average execution times of instrumented programs. None of the benchmarks showed a statistically significant decrease in execution time.

Valuable observations were made during the experimental work. Annotating programs require programmer effort, which is a valuable resource. The `depends_t` directive requires reasoning about data dependencies – not just locally inside a function, but across functions, which is harder. Likewise, the `depends_sc` directive requires reasoning about how two stencil computations share data and reasoning about the scheduling of iterations in parallel loops.

This raised the following question: how can the annotation effort be lowered? In the ideal case, the programmer only inserts annotations in places where they have significant impact. This lead to the hypothesis for the second study: when further analysis is prevented, program analysis can generate feedback which helps the programmer determine where and how to annotate the code.

To the best of my knowledge, no suitable task graph extraction tools are available to test this hypothesis. Focus was therefore shifted from dependencies in task graphs to a related problem: automatic parallelization and vectorization.

9.2 Generating Compilation Feedback

Programs are often written in ways that prevent auto-parallelization and vectorization of loops. The second study thus sought to show that program analysis can generate feedback that guides the programmer to annotate a program. Furthermore, resolving the issues preventing analysis allows the compiler to optimize more loops and thereby increase program performance.

An interactive compilation system was implemented to help the programmer annotate and refactor programs. For each issue, which prevents further analysis, and for each optimization performed, the system emits a code comment. The system was used to perform an extensive performance evaluation. Three benchmark kernels were studied on two parallel architectures. The evaluation used three sequential kernel benchmarks that pose problems for current production compilers. By annotating and refactoring application source code, additional loop nests could be optimized automatically. The speedups from auto-parallelization were compared with speedups after manual parallelization across different program inputs, systems and benchmark programs.

The experimental results showed speedups of up to 6.0 for a demosaicing kernel on an Intel Xeon system and up to 3.1 on an IBM POWER6 system after automatic parallelization. An edge-detection kernel was also sped up by factors of up to 8.3 the Intel Xeon system and up to 12.5 on the IBM POWER6 system.

Speedups from parallelization were measured for different numbers of threads and input sets. Auto-parallelization delivered the best result in 12 cases, while manual parallelization was better in 11 remaining situations. At low and medium thread counts auto-parallelization performed similar to or better than hand-parallelized and optimized codes.

An image decompression code was used to demonstrate the speedups after modifications to allow additional vectorization. On the Intel Xeon system, the code ran 10-12% faster. On the IBM POWER6 system, the modifications generally degraded performance except in one case, which showed a 2% speedup.

The speedups demonstrate the opportunities to extract additional parallelism from code containing regular loops. The differences in speedups on the two platforms suggest that auto-parallelization and vectorization should be combined with platform-specific tuning to extract additional performance. The experimental work also revealed that there are sometimes several ways to resolve an issue preventing further analysis. The programmer must choose among the alternatives and these choices may significantly affect the resulting performance.

This experimental work also led to interesting observations. Most importantly, resolving a reported issue does not necessarily enable optimization. Resolving one issue may reveal an additional issue that cannot be resolved. The second observation concerns the volume of the feedback. Reporting a high number of issues creates a “needle-in-the-haystack” problem for the developer. Not all code comments can be examined, so which comments are worthy of the programmers attention? Limiting the volume of feedback, on the other hand, runs the risk of not reporting issues that would lead to additional optimization.

It is most effective to resolve reported issues that relate to hot code areas. Profiling can find these areas and thereby help prioritize comments. However, the execution frequencies of code do not help estimate if an individual comment may lead to additional optimization or not. The question was then: besides profiling - what techniques help determine which code comments should be addressed?

9.3 Refining Compiler Feedback

The third and final study sought to identify code comments as potentially resolvable or unresolvable. The idea is as follows. The likelihood that a back off comment represents a missed optimization can be determined by comparing multiple sets of compilation feedback. Different sets of feedback are generated by building the same program several times while varying the compiler and the compilation options.

The experimental work used two approaches to estimate if back offs can be resolved or not. The first is speculative compilation. It relaxes certain assumptions during compilation and this lets the compiler perform optimizations otherwise prevented under the default assumptions. The second technique, assistance, compares feedback from multiple compilers. Compilers use different heuristics and differ in their optimization capabilities, so this can identify more opportunities for optimization than a single compiler can. The effect of combining speculation and assistance was also studied. The techniques were evaluated using four production compilers and an embedded benchmark suite.

The experimental results showed that speculation and assistance can categorize up to 60% of the issues reported by a single compiler as potentially resolvable (47%) or unresolvable (13%). On average speculation and assistance could categorize 43% out of a total of 3490 issues reported by the four compilers as potentially resolvable (27%) or unresolvable (15%).

There was no single "best" technique to categorize the issues reported. Rather, the efficiency of the approaches, alone and when combined, depends on the individual compiler. Besides the differences in optimization capabilities, this is most likely explained by the differences in how feedback was generated. There was more than a six-fold difference between the number of issues reported by the least and most verbose compiler. Also, the four compilers offered varying degrees of support for speculative compilation in terms of the number of relaxable assumptions. Categorization of issues was most effective with the compiler that reported the fewest issues and had the best support for speculation. This suggests that the other compilers should extend the support for speculation and perhaps limit the reported issues to those where a resolution can be suggested.

The third study used source code locations as a simple way to correlate the code comments. Correlation could be improved by taking code structure into account. This could identify when compilers emit related code comments at different source code locations. Integration with profiling tools might also be interesting. As previously mentioned, profiling can help prioritize code comments relating to frequently executed code. Profiling may also categorize code comments where no compiler is able to optimize even when optimization *is* possible. Such profiling would have to observe memory accesses in loops to find inter-iteration dependencies and would be more intrusive than profiling to find hot functions. However, such profiling would only be necessary for loops where the static techniques cannot estimate if a code comment is resolvable or not.

9.4 Outlook

Regrettably, annotating programs and examining compiler feedback is currently reserved for experts and compiler writers. A widespread adoption can only happen if these techniques become more accessible. This thesis demonstrates several ways to increase the accessibility of feedback and annotations. Annotations may be checked at run-time without compromising performance and compiler feedback can be shown directly in the source code editor via IDE integration.

Fully automatic parallelization may or may not increase application performance – but wastes no programmer effort in either case. With the programmer-in-the-loop approach, a human must spend effort to understand the reported issues and to consider workarounds. This thesis shows two ways to reduce this effort. The first combines the reported issues with suggestions explaining how the programmer can resolve these. The second way reduces the amount of feedback that must be considered by estimating if a reported issue represents a missed optimization or not.

Some may argue that improved compilers and runtimes will allow ample parallelism to be exploited without involving the programmer. If history is any indicator, this seems unlikely. Others may argue that the relatively limited success of automatic parallelization makes it largely irrelevant. In my opinion, both views are too simplistic.

The programmer-in-the-loop approach advocated in this thesis reframes this discussion. Rather than arguing over the merits of fully automatic or completely manual approaches, this thesis explores how to get the best from both worlds. For this to happen, existing tools and languages must evolve to allow better interaction with the programmer.

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] Vikram S. Adve and Rizos Sakellariou. Compiler synthesis of task graphs for parallel program performance prediction. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, LCPC '00, 2001.

[3] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, 2006.

[4] A. Agbaria, Dong-In Kang, and K. Singh. LMPI: MPI for heterogeneous embedded distributed systems. In *Proceedings of the International Conference on Parallel and Distributed Systems*, ICPADS '06, 2006.

[5] I. Ahmad, Yu-Kwong Kwok, and Min-You Wu. Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors. In *Proceedings of the 1996 International Symposium on Parallel Architectures, Algorithms and Networks*, ISPAN '96, pages 207–, Washington, DC, USA, 1996. IEEE Computer Society.

[6] Ishfaq Ahmad, Yu-Kwong Kwok, Min-You Wu, and Wei Shu. Casch: A tool for computer-aided scheduling. *IEEE Concurrency*, 8(4):21–33, 2000.

[7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Boston, MA, USA, 2006.

[8] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

[9] Zahira Ammarguellat and W. L. Harrison, III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. *SIGPLAN Not.*, 25:283–295, June 1990.

[10] Bill Appelbe, Kevin Smith, and Charlie McDowell. Start/Pat: A parallel-programming toolkit. *IEEE Softw.*, 6:29–38, July 1989.

[11] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[12] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26:345–420, December 1994.

[13] 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 ACM/IEEE conference on Supercomputing*, SC '91, 1991.

[14] Utpal Banerjee. An introduction to a formal theory of dependence analysis. *The Journal of Supercomputing*, 2:133–149, 1988.

[15] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[16] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[17] Michael Barr. Real men program in C. `http://embeddedgurus.com/barr-code/2009/08/real-men-program-in-c/`, 2007. Date accessed: June 5, 2011.

[18] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized multiframe tasks. *Real-Time Syst.*, 17:5–22, July 1999.

[19] Sanjoy K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Syst.*, 24:93–128, January 2003.

[20] Sanjoy K. Baruah, Louis E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Syst.*, 2:301–324, October 1990.

[21] Mieke Van Bavel and Michel Tilman. Interactive C-code cleaning tool supports multiprocessor SoC design. http://www.dspdesignline.com/208803005, July 2008. Date accessed: June 1st 2009.

[22] Koen Bertels, Georgi Kuzmanov, Elena Moscu Panainte, Georgi Gaydadjiev, Yana Yankova, Vlad Mihai Sima, Kamana Sigdel, Roel Meeuws, and Stamatis Vassiliadis. HARTES toolchain early evaluation: Profiling, compilation and HDL generation. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, FPL '07, 2007.

[23] Kristof Beyls and Erik D'Hollander. Refactoring for data locality. *IEEE Computer*, 42(2):62–71, 2 2009.

[24] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the Intel architecture. *Int. J. Parallel Program.*, 30:65–98, April 2002.

[25] William Blume and Rudolf Eigenmann. The range test: a dependence test for symbolic, non-linear expressions. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, SC '94, 1994.

[26] Y. Bouchebaba, B. Lavigueur, B. Girodias, G. Nicolescu, and P. G. Paulin. MPSoC memory optimization for digital camera applications. In *Proceedings of the 10th Euromicro Conference on System Design Architectures, Methods and Tools*, DSD '07, 2007.

[27] J. Mark Bull and Darragh O'Neill. A microbenchmark suite for OpenMP 2.0. *SIGARCH Comput. Archit. News*, 29(5):41–48, 2001.

[28] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: a test suite and results. In *Proceedings of the ACM/IEEE conference on Supercomputing*, SC '88, 1988.

[29] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6), November 1986.

[30] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, 2007.

[31] Milind M. Chabbi, John M. Mellor-Crummey, and Keith D. Cooper. Efficiently exploring compiler optimization sequences with pairwise pruning. In *Proceedings of the ACM SIGPLAN 1st International Workshop on Adaptive Self-Tuning Computing Systems*, 2011.

[32] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007.

[33] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, Cambridge, MA, USA, 2007.

[34] Robert N. Charette. This car runs on code. http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code/, 2009. Date accessed: April 9, 2011.

[35] Computer Architecture and Parallel Systems Laboratory. Open64. http://www.open64.net/. Date accessed: March 13, 2011.

[36] The Embedded Microprocessor Benchmark Consortium. Eembc 1.1. http://www.eembc.org/. Date accessed: June 8, 2011.

[37] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, 1971.

[38] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. *SIGPLAN Not.*, 34:1–9, May 1999.

[39] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23:7–22, August 2002.

[40] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, New York, NY, USA, 2nd edition, 2001.

[41] IBM Corporation. The restrict type qualifier. http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/topic/com.ibm.xlcpp8a.doc/language/ref/restrict_type_qualifier.htm, 2006. Date accessed: July 14, 2011.

[42] IBM Corporation. XL C/C++ for Linux, V11.1 compiler pragmas reference. http://publib.boulder.ibm.com/infocenter/lnxpcomp/v111v131/index.jsp, 2011. Date accessed: July 14, 2011.

[43] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual combined volumes 2a and 2b: Instruction set reference, a-z. http://www.intel.com/products/processor/manuals/, 2010. Date accessed: June 9, 2011.

[44] Intel Corporation. Intel C++ composer XE 2011 for Linux documentation. http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/lin/index.htm, 2010. Date accessed: May 15, 2011.

[45] Intel Corporation. Intel-specific pragma reference. http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011/compiler_c/cref_cls/common/cppref_bk_pragmas_intel_specific_ref.htm, 2011. Date accessed: July 14, 2011.

[46] Oracle Corporation. Sun Studio 12: C user's guide. http://download.oracle.com/docs/cd/E19205-01/819-5265/bjaby/index.html, 2001. Date accessed: July 14, 2011.

[47] M. Cosnard and M. Loi. Automatic task graph generation techniques. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, HICSS '95, 1995.

[48] Francis Cottet, Joëlle Delacroix, Claude Kaiser, and Zoubir Mammeri. *Scheduling in Real-Time Systems.* John Wiley & Sons, Hoboken, NJ, USA, 2002.

[49] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[50] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[51] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science and Engineering*, 5:46–55, 1998.

[52] B.P. Dave, G. Lakshminarayana, and N.K. Jha. Cosyn: Hardware-software co-synthesis of heterogeneous distributed embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):92 –104, march 1999.

[53] Robert P. Dick, David L. Rhodes, and Wayne Wolf. Tgff: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, CODES/CASHE '98, 1998.

[54] Yong Du et al. Explore optimization opportunities with XML transformation reports in IBM XL C/C++ and Fortran compilers for AIX. http://public.dhe.ibm.com/software/dw/rational/emi/Explore_XL_CCplus_and_Fortran_Compilers_for_AIX_XML_Transformation_Reports_Options.pdf, 2010. Date accessed: January 13, 2011.

[55] Alejandro Duran, Josep M. Pérez, Eduard Ayguadé, Rosa M. Badia, and Jesús Labarta. Extending the OpenMP tasking model to allow dependent tasks. In *Proceedings of International Workshop on OpenMP*, IWOMP '08, 2008.

[56] 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 '04, 2004.

[57] Christine Eisenbeis and Jean-Claude Sogno. A general algorithm for data dependence analysis. In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, 1992.

[58] Petru Eles, Zebo Peng1, Krzysztof Kuchcinski, and Alexa Doboli. Hardware/software partitioning with iterative improvement heuristics. In *Proceedings of the 9th International Symposium on System synthesis*, ISSS '96, 1996.

[59] Stijn Eyerman, Lieven Eeckhout, and Koen De Bosschere. Efficient design space exploration of high performance embedded out-of-order processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '06, 2006.

[60] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22, 1988.

[61] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901 – 1909, dec. 1966.

[62] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[63] Free Software Foundation. GNU Compiler Collection. http://gnu.gcc.org. Date accessed: September 11, 2010.

[64] Grigori Fursin and Olivier Temam. Collective optimization: A practical collaborative approach. *ACM Trans. Archit. Code Optim.*, 7:20:1–20:29, December 2010.

[65] María J. Garzarán, Saeed Maleki, William Gropp, and David Padua. Program optimization through loop vectorization. http://sc10.supercomputing.org/schedule/event_detail.php?evid=tut140, 2010. Tutorial. Date accessed: December 19, 2010.

[66] P.P. Gelsinger. Microprocessors for the new millennium: Challenges, opportunities, and new frontiers. In *IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, ISSCC '01, 2001.

[67] 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 '91, 1991.

[68] Matthias Gries. Methods for evaluating and covering the design space during early design development. *Integr. VLSI J.*, 38(2):131–183, 2004.

[69] M. R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization*, WWC-4, 2001.

[70] Soonhoi Ha. Model-based programming environment of embedded software for MPSoC. In *Proceedings of the Asia and South Pacific Design Automation Conference*, ASP-DAC '07, Washington, DC, USA, 2007. IEEE Computer Society.

[71] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, San Rafael, CA, USA, 2nd edition, 2010.

[72] Mikael T. Heath. Canny detector source code. ftp://figment.csee.usf.edu/pub/Edge_Comparison/source_code/canny.src, 1996. Date accessed: March 16, 2011.

[73] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[74] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, 2001.

[75] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, 2000.

[76] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32:60–65, March 2001.

[77] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '08, 2008.

[78] IEEE std. 1003.1c-1995 thread extensions. Technical report, Institute of Electrical and Electronics Engineers, 1995. Formerly POSIX.4a. now included in 1003.1-2004.

[79] Intel Corporation. Intel Advanced Vector Extensions Programming Reference. http://software.intel.com/file/35247/. Date accessed: June 8, 2011.

[80] Intel Corporation. Intel C++ Composer XE 2011 for Linux. http://software.intel.com/en-us/articles/intel-compilers/. Date accessed: March 13, 2011.

[81] Moore's law: Raising the bar. Technical report, Intel Corporation, 2005. ftp://download.intel.com/museum/Moores_Law/Printed_Materials/Moores_Law_Backgrounder.pdf.

[82] Intel Corporation. Guided auto-parallelism (GAP). http://software.intel.com/en-us/articles/guided-auto-parallel-gap/, 2010. Date accessed: March 16, 2011.

[83] International Business Machines. IBM XL C/C++ for Linux. http://www.open64.net/. Date accessed: March 13, 2011.

[84] International Organization for Standardization. ISO/IEC 9899:1999, December 1999.

[85] H. Jin, M. Frumkin, and H. Yan. NPB-OpenMP 3.0. Technical Report NAS-99-011, NASA Ames Research Center, Moffett Field, CA 94035-1000, 1999.

[86] Arun Kejariwal, Alexander V. Veidenbaum, Alexandru Nicolau, Milind Girkarmark, Xinmin Tian, and Hideki Saito. Challenges in exploitation of loop parallelism in embedded applications. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, 2006.

[87] K. Kennedy, K. S. McKinley, and C. W. Tseng. Interactive parallel programming using the parascope editor. *IEEE Trans. Parallel Distrib. Syst.*, 2:329–341, July 1991.

[88] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[89] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, ASAP '97, 1997.

[90] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *Int. J. Parallel Program.*, 28:347–361, August 2000.

[91] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[92] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. Parallel Distrib. Comput.*, 59:381–422, December 1999.

[93] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28, September 1979.

[94] Per Larsen, Sven Karlsson, and Jan Madsen. Identifying inter-task communication in shared memory programming models. In *Proceedings of the 5th International Workshop on OpenMP*, IWOMP '09, 2009.

[95] Per Larsen, Sven Karlsson, and Jan Madsen. Expressing inter-task dependencies between parallel stencil operations. In *Proceedings of 3rd Workshop on Programmability Issues for Heterogeneous Multicores*, MULTIPROG '10, 2010.

[96] Per Larsen, Sven Karlsson, and Jan Madsen. Expressing coarse-grain dependences among tasks in shared memory programs. *Special Issue of IEEE Transactions on Industrial Informatics*, 2011. Accepted for Publication.

[97] Per Larsen, Razya Ladelsky, Sven Karlsson, and Ayal Zaks. Compiler driven code comments and refactoring. In *Proceedings of 3rd Workshop on Programmability Issues for Heterogeneous Multicores*, MULTIPROG '11, 2011.

[98] Per Larsen, Razya Ladelsky, Jacob Lidman, Sally A. McKee, Sven Karlsson, and Ayal Zaks. Automatic loop parallelization via compiler guided refactoring. Technical Report IMM-Technical Report-2011-12, DTU Informatics, Technical University of Denmark, 2011. http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=6041.

[99] 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 '00, 2000.

[100] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. Techniques for increasing and detecting memory alignment. Technical report, Massachusetts Institute of Technology, 2001. Technical Memo 621, MIT LCS.

[101] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, 2004.

[102] Corinna Lee et al. UTDSP benchmark suite. http://www.eecg.toronto.edu/corinna/DSP/infrastructure/UTDSP.html, 1998. Date accessed: July 4, 2009.

[103] Edward A. Lee. The problem with threads. *Computer*, 39:33–42, May 2006.

[104] Xin Li, Bahadir Gunturk, and Lei Zhang. Image demosaicing: a systematic survey. *Visual Communications and Image Processing 2008*, 6822, 2008.

[105] Zhiyuan Li, Pen-Chung Yew, and Chuag-Qi Zhu. Data dependence analysis on multi-dimensional array references. In *Proceedings of the 3rd international conference on Supercomputing*, ICS '89, 1989.

[106] Shih-Wei Liao et al. Suif explorer: an interactive and interprocedural parallelizer. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '99, 1999.

[107] Ai-Hsin Liu and Robert P. Dick. Automatic run-time extraction of communication graphs from multithreaded applications. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, 2006.

[108] Shankar Mahadevan, Kashif Virk, and Jan Madsen. ARTS: A SystemC-based framework for multiprocessor systems-on-chip modelling. *Des Autom Embed Syst*, 11(4), 2007.

[109] Vadim Maslov. Delinearization: an efficient way to break multiloop dependence equations. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI '92, 1992.

[110] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, Boston, MA, USA, 2004.

[111] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[112] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proceedings of The 20th IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS '98, 1998.

[113] Gordon E. Moore. Readings in computer architecture. In Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, editors, *Readings in computer architecture*, chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[114] K. W. Morton and D. F. Mayers. *Numerical Solution of Partial Differential Equations: An Introduction*. Cambridge University Press, New York, NY, USA, 2005.

[115] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[116] Richard Nass. Annual study uncovers the embedded market. http://www.eetimes.com/design/embedded/4007166/Annual-study-uncovers-the-embedded-market, 2007. Date accessed: June 5, 2011.

[117] Peter Newton and James C. Browne. The code 2.0 graphical parallel programming language. In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, 1992.

[118] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis (2. corr. print)*. Springer, Heidelberg, Germany, 2005.

[119] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, 2006.

[120] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: revisited for short SIMD architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, 2008.

[121] Oracle Corp. Oracle Solaris Studio. http://www.oracle.com/technetwork/server-storage/solarisstudio. Date accessed: March 13, 2011.

[122] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29:1184–1201, December 1986.

[123] Joseph C. H. Park and Mike Schlansker. On predicated execution. Technical Report HPL-91-58, HP Software and Systems Laboratory, 1991. www.hpl.hp.com/techreports/91/HPL-91-58.html.

[124] PathScale Corporation. PathScale EKOPath 4 Compiler Suite. http://www.pathscale.com/ekopath-compiler-suite. Date accessed: June 21, 2011.

[125] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach 4th Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, September 2006.

[126] Paul M. Petersen and David A. Padua. Static and dynamic evaluation of data dependence analysis techniques. *IEEE Trans. Parallel Distrib. Syst.*, 7:1121–1132, November 1996.

[127] Andy D. Pimentel, Louis O. Hertzberger, Paul Lieverse, Pieter van der Wolf, and Ed F. Deprettere. Exploring embedded-systems architectures with artemis. *Computer*, 34:57–63, November 2001.

[128] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with starss. *Int. J. High Perform. Comput. Appl.*, 23:284–299, August 2009.

[129] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. A. Silber, and N. Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proceedings of the 4th GCC Developer's Summit*, 2006.

[130] Sebastian Pop, Albert Cohen, and Georges-Andre Silber Silber. Induction variable analysis with delayed abstractions. In Tom Conte, Nacho Navarro, Wen-mei Hwu, Mateo Valero, and Theo Ungerer, editors, *High Performance Embedded Architectures and Compilers*, volume 3793 of *Lecture Notes in Computer Science*, pages 218–232. Springer Berlin / Heidelberg, 2005.

[131] William Morton Pottenger. Induction variable substitution and reduction recognition in the polaris parallelizing compiler. Master's thesis, University of Illinois at Urbana-Campaign, 1995.

[132] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Concurrency*, 4:63–79, 1996.

[133] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the ACM/IEEE conference on Supercomputing*, SC '91, 1991.

[134] G. Ramalingam. The undecidability of aliasing. *TOPLAS*, 16(5):1467–1471, 1994.

[135] J. Ramanujam, S. Krishnamurthy, J. Hong, and M. Kandemir. Address code and arithmetic optimizations for embedded systems. In *Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, ASP-DAC '02, 2002.

[136] Lawrence Rauchwerger and David Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. *SIGPLAN Not.*, 30:218–232, June 1995.

[137] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[138] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of Jade. *TOPLAS*, 20(3):483–545, 1998.

[139] Andreas Rodman and Mats Brorsson. Programming effort vs. performance with a hybrid programming model for distributed memory parallel architectures. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, Euro-Par '99, 1999.

[140] Peter Rundberg and Per Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction-Level Parallelism*, 3:2002, 2001.

[141] E.M. Saad, M. El Adawy, H.A. Keshk, and S.M. Habashy. Task graph generation. In *Proceedings of the National Radio Science Conference*, NRSC '06, 2006.

[142] M. Saldana and P. Chow. Tmd-mpi: An mpi implementation for multiple processors across multiple fpgas. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, FPL '06, 2006.

[143] Marcus T. Schmitz, Bashir M. Al-Hashimi, and Petru Eles. *System-Level Design Techniques for Energy-Efficient Embedded Systems.* Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[144] N. Shah. Understanding network processors, 2001. Masters Thesis.

[145] Oliver Sinnen. *Task Scheduling for Parallel Systems.* Wiley-Interscience, Hoboken, NJ, USA, May 2007.

[146] Oliver Sinnen, Jsun Pe, and Alexei Kozlov. Support for fine grained dependent tasks in OpenMP. In Barbara Chapman, Weiming Zheng, Guang Gao, Mitsuhisa Sato, Eduard Ayguadé, and Dongsheng Wang, editors, *A Practical Programming Model for the Multi-Core Era*, volume 4935 of *Lecture Notes in Computer Science.* Springer Berlin, 2008.

[147] Lauren L. Smith. Vectorizing C compilers: how good are they? In *Proceedings of the ACM/IEEE conference on Supercomputing*, SC '91, 1991.

[148] G. Snider. Spacewalker: Automated design space exploration for embedded computer systems. Technical Report HPL-2001-220, HP Laboratories Palo Alto, 2001.

[149] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI – The Complete Reference, Vol. 1, The MPI Core, 2nd ed.* The MIT Press, Cambridge, MA, USA, September 1998.

[150] Ian Sommerville. *Software engineering (8th ed.).* Addison Wesley, Redwood City, CA, USA, 2006.

[151] Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph Johnson. Relooper: refactoring for loop parallelism in java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '09, 2009.

[152] OpenMP Architecture Review Board. OpenMP application program interface, version 3.0. Technical report, OpenMP Architecture Review Board, May 2008.

[153] The Eclipse Foundation. Eclipse C Development Tools. http://eclipse.org/cdt/. Date accessed: March 13, 2011.

[154] The Portland Group. PGI C/C++ Workstation for Linux. http://www.pgroup.com/products/pgiworkstation.htm. Date accessed: March 13, 2011.

[155] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, 2007.

[156] Takao Tobita and Hironori Kasahara. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5):379–394, 2002.

[157] J. D. Ullman. NP-complete scheduling problems. *J. Comput. Syst. Sci.*, 10:384–393, June 1975.

[158] *Unknown Author*. List of htc phones. http://en.wikipedia.org/wiki/List_of_HTC_phones, 2010. Date accessed: April 7, 2011.

[159] Keith S. Vallerio and Niraj K. Jha. Task graph extraction for embedded system synthesis. In *Proceedings of the 16th International Conference on VLSI Design*, VLSID'03, 2003.

[160] Ashlee Vance. British chip designer prepares for wider demand. http://www.nytimes.com/2010/09/20/technology/20arm.html, 2010. Date accessed: April 7, 2011.

[161] 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 '10, 2010.

[162] J. A. Williams, I. Syed, J. Wu, and N. W. Bergmann. A reconfigurable cluster-on-chip architecture with mpi communication layer. In *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '06, 2006.

[163] Michael J. Wolfe. Beyond induction variables. *SIGPLAN Not.*, 27:162–174, July 1992.

[164] Michael J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Boston, MA, USA, 1995.

[165] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23:20–24, March 1995.

[166] Tao Yang and Apostolos Gerasoulis. Pyrros: static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, 1992.

[167] Ti-Yen Yen and W. Wolf. Communication synthesis for distributed embedded systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers*, ICCAD-95, 1995.

[168] H. Youness, M. Hassan, and A. Salem. A design space exploration methodology for allocating task precedence graphs to multi-core system architectures. In *Proceedings of the International Conference on Microelectronics*, ICM '10, 2010.