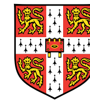
Technical Report

UCAM-CL-TR-796
ISSN 1476-2986

Number 796



UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Facilitating program parallelisation: a profiling-based approach

Jonathan Mak

March 2011

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 2011 Jonathan Mak

This technical report is based on a dissertation submitted November 2010 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St. John's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Summary

The advance of multi-core architectures signals the end of universal speed-up of software over time. To continue exploiting hardware developments, effort must be invested in producing software that can be split up to run on multiple cores or processors. Many solutions have been proposed to address this issue, ranging from explicit to implicit parallelism, but consensus has yet to be reached on the best way to tackle such a problem.

In this thesis we propose a *profiling*-based *interactive* approach to program parallelisation. Profilers gather dependence information on a program, which is then used to automatically parallelise the program at source-level. The programmer can then examine the resulting parallel program, and using critical path information from the profiler, identify and refactor parallelism bottlenecks to enable further parallelism. We argue that this is an efficient and effective method of parallelising general sequential programs.

Our first contribution is a comprehensive analysis of limits of parallelism in several benchmark programs, performed by constructing Dynamic Dependence Graphs (DDGs) from execution traces. We show that average available parallelism is often high, but realising it would require various changes in compilation, language or computation models. As an example, we show how using a spaghetti stack structure can lead to a doubling of potential parallelism.

The rest of our thesis demonstrates how some of this potential parallelism can be realised under the popular fork-join parallelism model used by Cilk, TBB, OpenMP and others. We present a tool-chain with two main components: Embla 2, which uses DDGs from profiled dependences to estimate the amount of *task*-level parallelism in programs; and Woolifier, a source-to-source transformer that uses Embla 2's output to parallelise the programs. Using several case studies, we demonstrate how this tool-chain greatly facilitates program parallelisation by performing an automatic best-effort parallelisation and presenting critical paths in a concise graphical form so that the programmer can quickly locate parallelism bottlenecks, which when refactored can lead to even greater potential parallelism and significant actual speed-ups (up to around 25 on a 32-effective-core machine).

Acknowledgements

First of all I would like to thank my supervisor, Alan Mycroft, for his invaluable advice and support in all matters. Without his guidance and encouragement I would never have been able to complete this thesis. I would also like to thank Matthew Parkinson, my second supervisor, for his support and guidance. In addition, I am profoundly indebted to Karl-Filip Faxén and Sverker Janson from SICS for their help and advice. Much of my thesis builds on Karl-Filip's excellent work.

I would also like to thank numerous people at the Computer Laboratory and elsewhere who have helped me in one way or another during my time here, especially in my early days as a PhD student. They include Christian Fensch, David Greaves, Daniel Greenfield, Ian Leung, Anton Lokhmotov, Simon Moore, Robert Mullins and David Ung.

I am also most grateful to St John's College for its generous financial support.

My family has been a wonderful blessing for me, and I thank my parents and brothers for their love and kindness. Thank you also to Esther, whose love, patience and timely encouragements have kept me going.

I am indebted to the brothers and sisters at the Cambridge Chinese Christian Church, for their wonderful friendship and prayers.

Last but by no means least, I thank Almighty God for his wonderful provision and grace, as everything that I have comes graciously from him. *Soli Deo Gloria!*

Work done in collaboration

Chapter 4 is largely based on an unpublished paper written with Karl-Filip Faxén of the Swedish Institute of Computer Science, a shorter version of which is published as [55].

In terms of the software described in this thesis, the original versions of Embla and Wool are largely due to Karl-Filip Faxén. Embla 2, which extends Embla, results from a joint effort between Karl-Filip and myself. The extensions to Wool described in Chapter 5 are solely my own work, as is Woolifier.

Contents

1 Introduction **11**

2 Survey of existing work **15**

2.1 Automatic parallelisation—static and safe 15

2.2 Dependence profiling—seeing what actually happens 17

2.3 Thread-level speculation—best of both worlds? 18

2.4 Parallel programming languages 19

2.5 Interactive parallelisation—semi-automatic parallelisation with human input 21

3 How much parallelism is out there?—finding limits of parallelism **23**

3.1 Introduction 23

3.2 Deriving limits of parallelism 24

3.2.1 Types of dependences 24

3.2.2 Dynamic Dependence Graphs 26

3.3 Implementation 28

3.4 Results 30

3.4.1 Effects of control dependences 30

3.4.2 Effects of name dependences 32

3.4.3 True dependences only 32

3.4.4 Removing compiler-induced dependences 32

3.5 Related Work 34

3.6 Conclusions 35

4 Separating the wheat from the chaff—estimating task-level parallelism 37

4.1 Introduction 37
4.2 Task Models 39
4.3 Implementation 45
4.3.1 Computing data dependences 47
4.3.2 Discounting spurious dependences due to memory reuse 47
4.3.3 Constructing the Dynamic Dependence Graph 48
4.3.4 Loop recognition 49
4.3.5 Reduction variable recognition 49
4.3.6 Granularity analysis 50
4.4 Results and Evaluation 53
4.4.1 Finding optimal task synchronisation points 54
4.4.2 Amount of parallelism discovered 55
4.4.3 Effects of optimisations 55
4.4.4 Parallelism in other benchmarks 57
4.4.5 Granularity analysis 59
4.5 Related work 62
4.6 Conclusions 63

5 Completing the production line—automating task-level parallelisation 65

5.1 Wool 65
5.2 Cilk and Cilk++ 71
5.3 Reduction operations 73
5.4 Granularity filtering 77
5.4.1 Getting the right threshold 79
5.4.2 Loop granularity 81
5.5 Evaluation 81
5.6 Future Work 84
5.6.1 Spawn hoisting 84
5.6.2 Thread-level speculation 84
5.7 Related Work 85
5.8 Conclusions 85

6 Pushing the boundary—demonstrating the tool-chain for interactive parallelisation 87

6.1 Case Studies . . . 88

6.1.1 sha . . . 89

6.1.2 susan.smoothing . . . 90

6.1.3 181.mcf . . . 90

6.1.4 179.art . . . 99

6.1.5 Other examples . . . 104

6.2 Related work . . . 105

6.3 Conclusions . . . 105

7 Conclusions 107

Chapter 1

Introduction

The prevalence of multi-core processors poses a huge challenge to the programming research community. On one hand, processor clock speeds have stopped rising as processor manufacturers find it increasingly difficult to dissipate heat from faster processors. While still increasing transistor counts in accordance with Moore's Law, manufacturers have kept processor clock speeds almost constant in the 2-3.5 GHz interval. Instead, they have begun to put multiple processing units, or *cores*, on the same chip. Instead of exponential growth in processor clock speeds, we are seeing a (rather slower) exponential growth in the number of cores on a processor.

On the other hand, most programs today are still sequential and single-threaded. This means that while historically these programs have been able to benefit in performance more or less directly without refactoring or even recompilation when run on a newer and faster processor, in the multi-core world this free performance gain no longer applies. The 'free lunch' is over [90].

In order to meet this challenge, programs must now be parallel, or multi-threaded, to stand any chance of benefiting from multi-core processors. Sequential applications must be *parallelised*, or written anew in a parallel language. There are generally two approaches to program parallelisation: with *explicit* parallelism, programmers are responsible for specifying where and how parallelism in the program is to be exploited. The problem, however, is that parallel programming is generally difficult—despite the availability of parallel languages since the early days of computing, most programmers are not familiar with parallel programming paradigms. As a result, programmers may specify parallelism incorrectly, causing concurrency bugs while missing valid parallelism opportunities elsewhere. This is compounded by the non-determinacy of parallel programs—the interleaving of threads is now heavily dependent on the state of the execution environment, and can be different for different runs. This makes concurrency-debugging much more difficult, as concurrency bugs may not be reproducible.

Figure 1.1: Software Development Process

The other approach—*implicit* parallelism—places the burden on the compiler. Static analysis is performed over the program, and safe opportunities for parallelism are identified. The programmer can write programs sequentially as before, and the compiler automatically generates parallel code. The problem with this is that, while complex and sophisticated static analyses have been developed to identify parallelism in programs, especially in loops, these techniques work well only in certain classes of programs, mainly array-based ones. In most pointer-based programs with irregular control flow, static analysis has so far been unable to discover much parallelism.

Faced with this problem, many turn instead to *dynamic* analysis, i.e. profiling programs and gathering information about their execution. This provides information about what *actually* happens rather than what *may* happen, as is the case in static analysis. The major downside to this is that dynamic analysis normally cannot cover all possible execution scenarios, which means that optimisations based only on dynamic analysis may not be safe when the optimised program is executed using new inputs.

Furthermore, a major challenge in extracting parallelism from sequential programs is the removal of unnecessary sequentialising dependences. The software development process can be thought of as a series of conversions, as illustrated in Figure 1.1. One usually begins with a *specification*, a written document detailing desired outcomes. From the specification a human designs an *algorithm*, a general strategy for achieving what the specification requires. The algorithm is then implemented in a *program*, using a programming language. This program is then compiled into an *executable* binary that can be directly executed on a machine. Each conversion, from specification to algorithm, from algorithm to program, or from program to executable, involves design choices that have to be made, some of which have great consequences for the resulting dependences and resulting parallelism. For instance, as we shall see in Chapter 3, artificial dependences on the stack pointer are introduced by the compiler in the conversion from program to executable. By modifying the compiler these dependences can be removed resulting in greater parallelism. However, it is much more difficult for the compiler to identify unnecessary dependences introduced during the conversion from algorithm to program.

In this thesis therefore, we explore a semi-automatic or *interactive* approach to program parallelisation. Dynamic and static analyses gather information that is used to automatically parallelise a program, using language extensions to specify parallel tasks. The programmer can then check that the resulting parallel program is safe and correct. Fur-

thermore, the programmer can use the information gathered by the profiler to identify bottlenecks—dependences in the program that inhibit further parallelism. They can then refactor the bottleneck code in the *sequential* program to enable further parallelism to be discovered and exploited, and then rerun the analyses and parallelisations to get a more parallel program. Such refactoring represents a change of design choices in the conversion process either from algorithm to program or from specification to algorithm. While the former case might simply involve moving a single statement, the latter typically requires much more thought and consideration—parallel versions of sequential algorithms often merit doctoral theses of their own!

The main advantage is that by making the parallelisation visible and comprehensible to the programmer, the programmer and compiler can work together to parallelise a program better than either of them can do on their own in a short amount of time. The machine makes a best effort parallelisation, focusing the programmer on the harder areas that require refactoring. Not only so, but the programmer is also able to parallelise the program without coding in a parallel language. Manual code changes only need to be applied to the *sequential* version of the program.

This parallelisation, based as it is on *dynamic* analysis, is not guaranteed to be safe when run on inputs other than the profiled one. This is because a dependence may arise for some inputs but not others. However, in practice, this is usually not too great a problem. Past research has shown that if we can find inputs to cover all of the code, then we can find most of the dependences as well. Nonetheless, to guarantee safety, systems in the past have been created that insert run-time checks at appropriate points so that the program either crashes or recovers from unexpected dependences (e.g. thread-level speculation schemes) instead of silently producing wrong answers. We note this as a potential feature which has not been implemented in our system, which currently relies only on the programmer to guarantee correctness.

We begin this thesis with a survey of existing work in parallel programming (Chapter 2). We then examine limits of parallelism at all levels in certain sequential benchmarks, and see how they vary from one program to another, and under various models (Chapter 3). Chapter 4 aims to focus only on parallelism that is realisable on multi-cores, especially ignoring fine-grained instruction-level parallelism that is already exploited in VLIW and superscalar processors. We show how our profiling tool, Embla 2, can be used by programmers to understand and exploit the potential for parallelism in their programs. Next (Chapter 5) we show how we can automate the process of using profiler results to parallelise a sequential program, as we have implemented in Woolifier. Embla 2 and Woolifier together form a useful tool-chain that can significantly facilitate program parallelisation for the programmer. We demonstrate this in Chapter 6, focusing on how the critical path information directs the programmer on to the parallelism bottlenecks, which when parallelised would give the greatest performance improvement. We give a final evaluation

and conclusion in Chapter 7.

Below are the main contributions of this thesis:

- We use a unified framework to analyse limits of parallelism in entire programs, under dependence models differing in optimisation techniques required to achieve them. (Chapter 3).
- We extend a Valgrind-based dependence profiler, Embla, to construct a tool, Embla 2, that provides an estimate of potential speed-up for parallelisation using frameworks such as Cilk and OpenMP. Embla 2 can also perform reduction variable recognition and a novel Markov-chain-based granularity analysis.
- Embla 2 also produces a concise graphical representation of critical paths that allow programmers to easily locate parallelism bottlenecks.
- We present Woolifier, a source-to-source compiler that transforms sequential programs into parallel ones that use nested fork-join parallelism, allowing us to realise some of the speed-up as estimated by Embla 2.
- We demonstrate how Embla 2 and Woolifier can be used as a tool-chain for interactive parallelisation. Using several case studies, we show how, with the aid of our tool-chain, good speed-ups can be achieved in many programs with little programmer effort.

Chapter 2

Survey of existing work

Due to the understandable enormous research interest into the problem of program parallelisation, there has been a huge body of literature dedicated to it. As a result, we concentrate our survey of existing work on research closest to our approach, while giving only a sampling of references in other related areas.

2.1 Automatic parallelisation—static and safe

We begin by examining attempts to analyse sequentially written programs to extract parallelism automatically, without assistance from the programmer. The greatest challenge in program parallelisation is dependence analysis—correctly identifying which instructions are *dependent* on each other, that is, the semantics of the program will be different if their order of execution were reversed. If two statements contain instructions that are dependent on each other, then they cannot be executed concurrently, as otherwise nondeterministic behaviour will result based on the interleaving of the instructions—this is known as a *race condition*.

Dependence analysis is an established area of program analysis, not only for program parallelisation but for many compiler optimisations. There are two types of dependences: *control* dependence, which occurs when whether one instruction is executed depends on the result of another, typically a conditional branch or indirect jump; and *data* dependence, which occurs when two instructions access a common resource (register or memory location), and at least one of them writes to it.

Control dependences in a program can generally (except for indirect branches) be worked out precisely (in the syntactic not semantic sense), as in the Program Dependence Graph [27]. Deriving precise data dependences (and control dependences for indirect branches) is much harder, with many static analyses overestimating by a great margin. The main problem is potential aliasing, when different pointers point to the same location. Consider

```
1   void f(int *a, int *b) {
2     *a = 23;
3     *b = *b + 45;
4   }
```

Figure 2.1: Aliasing illustrated—if `a` and `b` alias then line 3 depends on line 2

```
for (i=0; i<n; i++) {
  tmp = i*i+34;
  a[i] = tmp * tmp;
}
```

(a) Example of a privatisable variable. There is a write-after-read dependence between iterations of the loop, which can be removed if `tmp` is privatised.

```
for (i=0; i<n; i++) {
  acc += i*3 - 4;
}
```

(b) Example of a reduction operation. There is a read-after-write dependence between loop iterations, but this is unnecessary as the final value of `acc` is the same regardless of the order of execution.

Figure 2.2: Examples of privatisation and reduction operations.

the example program in Figure 2.1. Line 3 depends on line 2 only if `a==b`, which depending on the context may always, never or sometimes be the case. While interprocedural analysis can deal with simple cases with scalar variables, when arrays and pointer arithmetic are introduced the problem becomes more difficult, especially for non-scientific applications. Kennedy and Allen's book [42] details analyses in this area.

Some dependences are in fact easily removable. For instance, many write-after-read and write-after-write dependences between loop iterations can be removed by privatisation—using a separate location to store the value of a variable in each iteration. Reduction operations are another source of unnecessary dependences. These are accumulation operations on variables where the order in which instances of the associative reduction operation are executed makes no difference to the final value of the accumulator (and intermediate values of the variable are not used). Examples of privatisation and reduction operations are shown in Figure 2.2.

There are a number of parallelising compilers that try to remove these dependences. The most notable are Polaris [10] from the University of Illinois and SUIF [35] from Stanford University. They employ advanced interprocedural scalar and array dependence analysis, as well as privatisation and reduction-operation recognition, to parallelise loops, i.e. allowing different iterations to run in parallel.

Parallelism from parallel execution of loop iterations is known as *DOALL* parallelism. Further parallelism can be extracted if we look beyond DOALL loops, and allow the

compiler to schedule instructions in a function on to different threads. Girkar and Polychronopoulos [31] lay down a detailed theoretical basis for this by specifying exactly when an instruction can be executed before another. Ottoni et al. present compilers that exploit this with Decoupled Software Pipelining [68] and GREMIO [67], both of which involve instruction rescheduling based on program dependences.

Recently there has been interest in using separation logic, an extension of Hoare logic that allows us to easily reason about sections of memory in isolation, to determine more precise dependences [77] for pointer-based programs, at which traditional analyses have not been very successful. Research to automatically parallelise real-world programs using separation logic is ongoing.

2.2 Dependence profiling—seeing what actually happens

Static analysis can generally give us only an over-approximation of what may happen at run-time. Experience shows that such over-approximation is often quite large. This means that valid opportunities for optimisation are often missed. Because of this, many turn to using *dynamic* analysis, in particular profiling, to find out what *actually* happens during execution, rather than what *may* happen. The flip side is of course that the results of dynamic analysis are dependent on inputs, so unless the program takes no inputs or we can exhaustively try all possible inputs, program transformations based only on the results of profiling may not be safe. In other words, profiling generally gives us an underestimate of what may happen. Nevertheless, if we can have inputs we deem representative, then the results of dynamic analysis can still give us information about the program that we may never get with only static analysis.

Agrawal and Horgan [1] present a *Dynamic Dependence Graph*, a variant of the Program Dependence Graph where there is a node for each occurrence of a statement in the program's execution history, in the context of program slicing and debugging. This graph gathers only the execution history, while still using dependences from the static Program Dependence Graph. With more general programs with arrays, however, we need to profile also the memory locations accessed by each statement, in order to find out exactly which statement is dependent on which.

Dependence profiling has been used to study limits of parallelism in programs, performed by finding the *critical path* in a Dynamic Dependence Graph. One such profiling tool is Nethercote and Mycroft's Redux [62]. Because in every path in the DDG each operation must be executed after those preceding it and before those following it, and that the critical path is the longest path in the DDG, the length of the critical path represents the minimum

time required to execute a program in parallel while respecting all dependences. Several studies [99, 47, 7, 72, 86, 95] work at the instruction level, finding the limits of parallelism under various models of execution, some more realistic than others. Some studies look also at how the limits change given a limited number of threads, using measures such as smoothability [91] and slack [76].

In work by Petersen and Padua [70], dependences obtained using various static analyses are compared with those observed at run-time. When DDGs are constructed using these dependences, parallelism figures based on profiling are sometimes found to exceed those based on static analyses by two orders of magnitude, illustrating the benefits of profiling over static analysis.

The limits of parallelism in the works above encompass all levels from instruction-level upwards. However, in practice parallelism that is too fine-grained cannot be exploited by using multiple cores. An independent thread that is only a few instructions long, if scheduled in parallel on a separate core, will cause performance to degrade rather than improve, due to the overheads of task creation, communication and cache misses. Furthermore, fine-grained instruction-level parallelism (ILP) is already being well exploited on single superscalar or Very Long Instruction Word (VLIW) processors, Consequently, several other studies have attempted to exclude ILP from their parallelism figures, focusing on coarser-grained parallelism. Some [48, 103, 46] look at loop-level parallelism—the gain that may result by executing loop iterations fully or partly in parallel with each other. Others [64, 66, 45] focus on function-level parallelism—the gain that may result by executing function calls in parallel with the code before, the code after, or other function calls.

There are many other uses of dependence profiling other than estimating parallelism. Other approaches measure the distances spanned by dependences [104], probabilities of dependences occurring [18], and dependence density [97].

2.3 Thread-level speculation—best of both worlds?

A popular way to use profiling output while ensuring safety is to speculatively execute code above a safety net. Code is executed assuming certain dependences do not arise. This assumption is checked at run-time and if it turns out to be false the offending code is rolled back and rerun, this time respecting the new dependences. This is of course not without cost, and is beneficial only if dependence violations are rare. Much research has studied ways to pick good candidates for speculation.

Thread-level speculation (TLS) schemes range from being software-based to hardware-based. Software-based schemes [102, 19, 65, 23, 9, 80] require no change in hardware, but

are generally prohibitively slow; hardware-based systems [41, 17, 40, 54, 87, 96] have all been implemented so far in simulators only—no processors with TLS support have yet been manufactured.

TLS schemes also differ in how they choose candidate threads. Most focus on loops [19, 65, 17, 87, 80], while others look at function calls [102, 9], and some use both [54]. Still others split instructions into threads in a more unrestricted manner [41, 23, 40, 96]. Another dimension in which they differ is how speculative state is stored in memory. Garzarán et al. [30] provides a taxonomy of approaches to memory management in TLS schemes.

On a related note, 'safe' speculative task execution that does not require a rollback has been explored in the lazy functional language context. It has been found that speculative evaluation of a thunk—a task from which the result may or may not be required—i.e. a type of control speculation, improves performance even on a single processor due to reduced memory traffic [24]. On multiple processors the performance gain is even greater [36].

2.4 Parallel programming languages

We turn our attention now to *explicit* parallelism, where the programmer specifies sources of parallelism in the language. We examine below a selection of the most popular parallel programming languages and task libraries, which generally span the two paradigms, *message passing* and *shared-memory*.

The *message passing* paradigm stems from a class of programming models that have emerged since the late 1970s centring around the concept of concurrent processes communicating through channels. As such interaction by one process with another is made explicit. Hoare's Communicating Sequential Processes [39] and Milner's Calculus of Communicating Systems [59] are the seminal works in this area, influencing later works such as π-calculus [60] and join-calculus [28]. Language implementations of these concepts include Occam [79], Erlang [6] and MPI [85].

In the message-passing model, we generally have a number of concurrent *processes*. Each process has its own local memory, and data can only be shared between processes using explicit message passing. Programs can be written in the "Single Program, Multiple Data" [22] pattern, where the same program is executed by multiple processes, each identified by its unique process identifier. This identifier can be used by one process to send messages to other processes, and for deciding which part of the data to operate on, as well as which part of the program to actually execute.

The advantage of message passing is that as communication between processes is explicit, it is easier to reason about possible interactions between processes and a program written

in the message-passing paradigm is less likely to contain unintentional races due to accidental sharing. However, the concept of message passing is generally alien to programmers who are used to writing sequential software.

The second paradigm, known as the *shared-memory* model, is generally more familiar to programmers. In a typical shared-memory model concurrent *threads* access a common memory address space, communicating via shared variables used as locks, semaphores, etc., or abstractions such as transactional memory [38]. To ensure a gentle learning curve for programmers common implementations are generally in the form of libraries for popular languages or modest language extensions. (Some [12, 14] argue that the latter is a safer approach, as it allows the compiler to reason about concurrency simply by syntax analysis.)

OpenMP [21] is an extension to C and FORTRAN, where all parallelism is expressed through directives and library calls. It is implemented in all widely used C/FORTRAN compilers. The main parallel constructs provided are parallel for-loops, allowing for parallel reduction operations (e.g. for sum, maximum and minimum operations), and synchronisation barriers, where threads wait for each other before proceeding. Thus it is easy to specify loop-level parallelism in OpenMP.

Java (versions 5+) provides concurrency under the Master/Worker model. One can define classes which implement the `Callable` interface, instances of which represent a unit of work to be executed by a thread. Collections of these `Callable` objects can then be passed to a master or `ExecutorService`, which is responsible for creating worker threads and allocating units of work to each thread, either statically or dynamically.

Cilk [89] is a language extension to C that allows asynchronous procedure calls. One of its strengths is its simple interface, which allows programmers to easily fork (`spawn`) a new thread to execute a procedure call, and then join (`sync`) it back together when required. Cilk++ [51] adds support for C++ and parallel for-loops.

One feature of shared-memory models is that they give the programmer no control over where the data is located in memory. While this abstraction might make programming simpler, it may not be so desirable with Non-Uniform Memory Architectures, where the address space is global but some areas in memory are closer to a core than others. The partitioned global address space (PGAS) model is an attempt to bridge the shared-memory and message-passing models by using a global but non-uniform memory model. A prominent example of a language using this model is X10 [16], in which code executing at a certain *place*—abstraction for a computing node, e.g. a processor—can access variables in local memory directly but must explicitly communicate[1] with remote places to access variables there.

[1]This is only a syntactic requirement—the compiler can of course optimise this.

The examples above are merely a representative sample of the tools and libraries available for explicit parallelism, but serve to illustrate the diversity of opinions that exist regarding the best model for parallel computation, from shared memory to message passing, from functional languages to imperative ones, and from libraries to syntactic constructs.

2.5 Interactive parallelisation—semi-automatic parallelisation with human input

Given the problems with automatic parallelisation above, and the overheads of thread-level speculation, another approach has been to involve the programmer in the parallelisation process. Many tools have been created over the years to assist the programmer in this respect. The obvious advantage is that the programmer can provide domain knowledge that the compiler cannot deduce. At the same time, care must be taken so that the programmer is not burdened with work that can be automated.

Earlier tools focus mainly on loops, and typically work by showing statically analysed dependences to the programmer. The ParaScope Editor [43] and PAT [84] are editors that display dependences in FORTRAN programs and assist the programmer in parallelising DOALL loops, flagging specific dependences as ignorable and refactoring loops to increase parallelism. The Parafrase-2-based Graphic Parallelizing Environment [15] displays dependences in a hierarchical manner, making it easier to parallelise a program based on its control structures.

More recently interactive parallelisation tools also employ profiling to gather information. The SUIF Explorer [52] performs both static and dynamic analyses on a program, automatically parallelising coarse-grained loops that are safely parallelisable. From the remaining loops, profiling identifies the significant ones, on which program slicing is applied to identify statements that stop a loop from being parallelisable. Tournavitis et al. [93] present a framework which mark parallelisable loops based on profiled dependences, displaying them to the user for approval. Machine learning is then used to decide on the most profitable loop scheduling policy for each loop, which is architecture-dependent.

A few tools ask the user to identify where parallelism may be found. In work by Thies et al. [92], the programmer specifies potential pipeline boundaries for software pipeline parallelism, which are then checked by a profiler. Several thread-level speculation systems [73, 58, 98] also require user input to identify good candidates for speculation, possibly aided by profiling.

Some forms of user input are more unconventional. Most automatic parallelisation tools aim to preserve *sequential equivalence*—the parallelised code must, given the same input, produce exactly the same output as the original sequential program. Bridges et al.

[13], however, argue that in some cases multiple outputs are acceptable to the user, who should be allowed to specify sources of legal non-determinism. In particular, they propose two annotations of sequential code, which effectively allow the compiler to extract more parallelism by ignoring certain dependences:

1. the *Y-branch*, a branch that can be taken with a specified probability irrespective of whether the corresponding conditional evaluates to true. The authors give an example of a dictionary for a compressor program that needs to be refreshed every so often.

2. the *commutative* annotation on functions, which inform the compiler that calls to these functions can be executed in any order even though dependences may exist between them. A common example is the random number generator function.

Chapter 3

How much parallelism is out there?—finding limits of parallelism

3.1 Introduction

As we have seen, many attempts have been made to extract parallelism out of a sequential program [10, 42, 68, 67]. When we are trying to parallelise a sequential program, one question that would be useful is, how parallel can this program possibly be? If we have made some gains in performance with our existing techniques, how much scope is there for further improvement? Thus we begin by performing a limit study on existing programs.

There have been several studies on the limits of parallelism in a sequential program, typically performed in one of two ways. The first is to re-schedule each dynamic instruction at the earliest cycle possible in a simulator, using the execution trace of a previous run, and deriving the average number of instructions per cycle for the schedule [99, 47, 72]. The other is to construct *Dynamic Dependence Graphs* (DDGs) from the execution trace, from which the average "width" is calculated [7, 86]. The results from either method give a theoretical upper bound on the amount of average parallelism available in general programs, so that we have an idea of how much room there is for further improvement of parallelising compilers. Although both methods should give the same results, we have chosen the latter as it allows more flexibility to transform the graph in ways not possible with the former.

We evaluate the available average parallelism of benchmarks under several models, which differ by the types of dependences considered and graph transformations applied, simulating the effects of possible compiler optimisations. The results allow us to measure the effects different types of dependences have on available parallelism.

We find that when we consider only true dependences—the *essence* of the underlying algorithm—the figures for average parallelism for many benchmarks are over 100. In

```
$4 := add $5, $6
$2 := sub $3, $4
```

(a) True dependence

```
$6 := add $5, $4
$4 := sub $2, $3
```

(b) Anti-dependence

```
$4 := add $5, $6
$4 := sub $2, $3
```

(c) Output dependence

```
   beq $2, $3, L
   $4 := add $5, $6
L: ...
```

(d) Control dependence

Figure 3.1: Examples of the different types of dependences

addition, we argue that certain parallelism-limiting dependences are compiler-induced artifacts, and removing them would increase potential parallelism. For instance, certain name dependences on a linear execution stack can be removed if a *spaghetti stack* is used instead. Thus one of our models simulates the effects of using a spaghetti stack on available parallelism. This removes inter-frame name dependences and breaks long true-dependence chains on the stack pointer register. We find that for some benchmarks this results in a doubling of average parallelism. Taking this idea further, we find that if we consider the extreme (and less realistic) case where all address calculations are disregarded potential parallelism in some programs increases by up to an order of magnitude (details on page 28).

3.2 Deriving limits of parallelism

3.2.1 Types of dependences

In our analysis we consider the parallelisation problem as one of minimising total execution time, counted as the number of instruction cycles, given the constraints imposed by its dependences. Instructions that are independent of each other can be executed in parallel. Parallelisation therefore involves exposing such independence and removing or minimising certain dependences to allow more independent code to be exposed. We consider four types of dependences—as illustrated in Figure 3.1.

- *True* (Read-after-Write) dependences occur when one instruction uses a value produced by a previous instruction, and are generally considered to make up the essence of an algorithm. Most would consider true dependences irremovable without changes on the algorithmic level. In fact, as we will see later, true dependences are sometimes introduced at the assembly level instead by the compiler and may be removed by modifying the compiler rather than the algorithm itself.

- *Anti-* (Write-after-Read) and *Output* (Write-after-write) dependences, collectively known as *name* dependences, occur due to the reuse of registers and memory locations. No value is passed from the source instruction to the target instruction, but if two instructions which have such a dependence between them are executed out of order, use of incorrect values by an instruction may result. One way to remove such dependences is to remap locations. For registers this means mapping the register in the target instruction to a different register if there is no true dependence from an earlier instruction. In fact, register renaming is already performed in many modern processors (e.g. x86) and accordingly we ignore anti- and output dependences on registers in all of our analyses. The renaming of memory locations, however, is in general more difficult, as it may not be possible to work out the memory location accessed by an instruction until the instruction is executed. Privatisation and Single Static Assignment transformations are special cases of memory renaming, typically done at compile-time, but generally only work on the execution stack.

- *Control* dependences occur when an instruction's execution is conditional on the result of a previous instruction, for example after a branch instruction. Control dependences cannot be removed easily, because like true dependences they capture some aspect of the essence of a program. Consider for instance the role of the control dependence in `y = x?23:42;`, which can be mapped to `y = (x&23)|((~x)&42);` (assuming `false` is represented by `0` and `true` by `-1`), showing the similarity to true dependences. However, effects of control dependences may be reduced by such techniques as speculative execution, where the result of an instruction is computed but not committed until we know it is safe to do so.

By considering different subsets of these dependences, we find out the effects each type of dependence has on available parallelism. Different models attempt, in varying degrees of vigour, to remove dependences that may be considered compiler artifacts, leaving only those that are essential to the underlying algorithm of the original program. Ultimately the aim is to explore the level of parallelism if all compiler-induced dependences are removed.

We do not view the results of this study as definitive limits—in particular we do not say that these limits are practically achievable. For example, as we aim to evaluate limits of parallelism in an architecture-independent way, we do not consider any overheads related to threading and inter-processor communication which will impact speed-up in practice. Parallelism-enhancing effects of value prediction [53] are also beyond the scope of our study in this chapter.

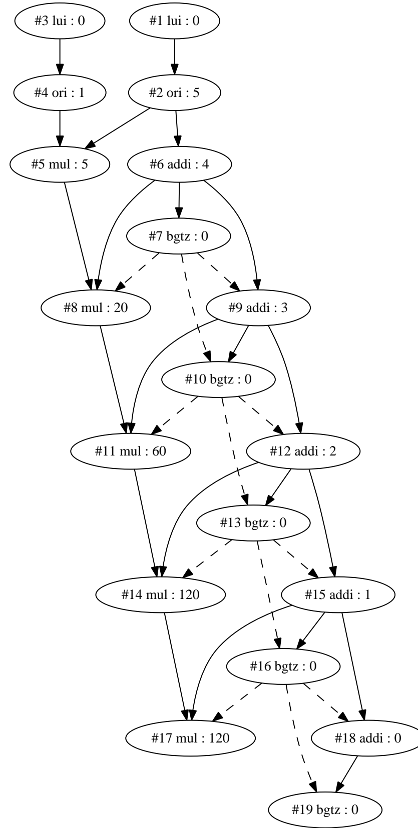


Figure 3.2: An example of a DDG for the calculation of the factorial of 5. Dashed edges denote control dependences—solid edges denote true dependences. Nodes with no outgoing edges (the `mul` and `bgtz` nodes in the bottom) are considered results of the calculation.

3.2.2 Dynamic Dependence Graphs

Dependences described above can be used to create a Dynamic Dependence Graph (DDG) [7], in which a node represents the execution of an instruction, and edges represent the dependences between instructions. Figure 3.2 shows an example of a simple DDG. Consider an idealised machine model where each instruction takes one cycle, but cannot be executed until all the instructions it depends on have been executed. Assume also that there is an unlimited number of processors with zero inter-processor communication overheads. The minimum execution time under this model is the number of instructions in the longest chain of dependences, or in other words the critical path. Average parallelism, measured in instructions-per-cycle, is then calculated as the total number of instructions (graph size) divided by the minimum execution time (length of critical path), which can be viewed informally as the average *width* of the DDG.

We begin with a model that considers all dependences, and then progressively remove certain types of dependences, to evaluate the effects they have on parallelism. In addition to removing control and name dependences, we also consider the effects of removing

```
void main() {
  foo();
  bar();
}
```

(a) Code for two procedure calls inside `main` in C

```
1:      jal     foo           # main: call foo()
2:  •·· addiu   $sp,$sp,-32   # foo: decrement stack pointer (new frame)
3:  •·· addu    $fp,$0,$sp    # copy stack pointer to frame pointer
    |      ... <code for foo()>..
4:  •·· addu    $sp,$0,$fp    # copy frame pointer to stack pointer
5:  •·· addiu   $sp,$sp,32    # increment stack pointer (discard frame)
6:  |   jr      $ra           # return to main()
7:  |   jal     bar           # main: call bar()
8:  •·· addiu   $sp,$sp,-32   # bar: decrement stack pointer (new frame)
9:  •·· addu    $fp,$0,$sp    # copy stack pointer to frame pointer
    |     ... <code for bar()>...
10: •·· addu    $sp,$0,$fp    # copy frame pointer to stack pointer
11: •·· addiu   $sp,$sp,32    # increment stack pointer (discard frame)
12:     jr      $ra           # return to main()
```

(b) Instruction trace of the execution of `main` in MIPS assembly

Figure 3.3: An illustration of the true dependence chain on the stack pointer.

certain classes of true dependences, in particular those involved in address calculations.

Consider the general calling convention of a C program with a downward-growing execution stack. At the heart is a register (`$sp` in MIPS) that stores the stack pointer, the address of the current top of the execution stack. As a function is called, a new frame is pushed on to the stack, and the stack pointer is decremented (assuming a downward-growing stack) to point to the new top of the stack; as the function returns, the frame is popped and the stack pointer is incremented to restore it to the original value. If we have two function calls, as in Figure 3.3, true dependences on the stack pointer would prevent these two calls from executing concurrently on this stack pointer, because the second call requires the value of the stack pointer, which would have been decremented and then incremented back by the first procedure call. In the DDG this would show up as a decrement-increment-decrement-increment sequence of operations on the stack pointer, thus linearising the calls.

It can be argued however, that this dependence is only an artifact of the linear execution stack data structure introduced by the compiler, rather than part of the essence of the original algorithm used by the programmer. Indeed, if a linear stack is replaced by a

spaghetti stack (also known as heap-allocated stack frames [5] or *cactus stack* [32]), such dependences can be eliminated, along with name dependences between different stack frames.

The stack pointer is just one example of address calculations that are only compiler artifacts rather than an essential part of the algorithm. Another example is the heap memory allocator, which typically updates some internal data structure (such as a linked list) every time it is called, giving rise to true dependences between calls, even though commuting the orders of these calls would make no difference to the semantics of the program (assuming sufficient memory is available). Thus taking the idea of removing compiler artifacts one step further, we consider the effects of removing address calculations from the graph altogether. This would show the amount of parallelism available in a world where memory address values do not have to be calculated at run-time, but can rather be pre-determined statically (as in a language with non-recursive static stack allocation such as FORTRAN). It can be argued that such a measure is unrealistic, given that *some* address calculations are in fact an integral part of an algorithm, e.g. radix sort. Nonetheless, it still provides an estimate of the upper bound of parallelism (albeit not the tightest one) if compiler (and certain run-time) artifacts are removed or replaced by data structures (such as the spaghetti stack) which are less parallelism-limiting.

3.3 Implementation

We examined the execution of programs taken mostly from the MiBench [34] suite. We chose this because while they provide a representative sample of real world applications, many of them also have data sets small enough for our analysis—the technology used in this chapter enables us only to examine program traces up to a few tens of millions of instructions long—and we prefer to analyse each program execution in its entirety. In addition we also analysed the synthetic benchmarks Whetstone [20] and Dhrystone [101].

The programs were compiled into MIPS binaries on Linux using gcc with uClibc as the standard library. The binaries were then run on a MIPS simulator (we used the GNU simulator provided by MIPS Technologies [61] and QEMU [8], a uni-processor emulator capable of emulating most Linux system calls.) The simulators have been configured or adjusted to output a trace detailing the opcode of each instruction, as well as the names of registers and addresses of memory locations that it has read from and written to. This trace is then analysed to build the DDGs.

We built DDGs under several models which differed by the types of dependences they considered, as well as the graph transformations applied, as detailed below, and summarised in Table 3.1. Critical paths were then extracted from the graphs, enabling us to calculate the limits of parallelism as we have seen. Note that when we consider control dependences

Model	True deps	Name deps	Control deps	Spaghetti stack	Ignoring addr calcs
TruNamCtl	✓	✓	✓		
TruCtl	✓		✓		
TruNam	✓	✓			
Tru	✓				
TruNamSp	✓	✓		✓	
TruSp	✓			✓	
TruNoAddr	✓				✓

Table 3.1: Table comparing the seven models used in our analysis

in our models, every instruction is made dependent on the most recent branch or indirect jump instruction. As we shall see later, this is an over-approximation of static control dependences, which cannot be deduced simply from the execution trace as control merge points are invisible on execution traces.

TruNamCtl This is the most restrictive (or pessimistic) model. We include in the graph all true dependences for registers and memory locations. We also include name dependences for memory locations (but not registers, as it is relatively straightforward for register renaming to be performed in a modern processor, provided there are enough registers). Finally control dependences are also included, meaning that every instruction is dependent on the most recent branch or indirect jump instruction. This has the effect of limiting parallelism largely to within a dynamic basic block[1].

TruCtl Name dependences for memory locations are ignored, simulating the effects of perfect memory renaming.

TruNam Control dependences are excluded instead (e.g. by perfect branch and jump prediction), allowing parallelism beyond basic blocks to be exhibited.

Tru Only true value dependences are considered here, the "essence" of the program.

TruNamSp Here we model what happens when a spaghetti stack is deployed, in order to reduce compiler-induced dependences on the stack pointer. Two transformations are applied to the graph. The first transformation removes all inter-frame name dependences on the stack, as these different frames which used to occupy the same area on the stack would now occupy different areas in the heap. In the second transformation, every instruction that decrements the stack pointer (i.e. pushes a new

[1] A dynamic basic block is defined as the sequence of instructions between two branch/indirect jump instructions. As opposed to static basic blocks, the target of a branch not taken does not begin a new basic block.

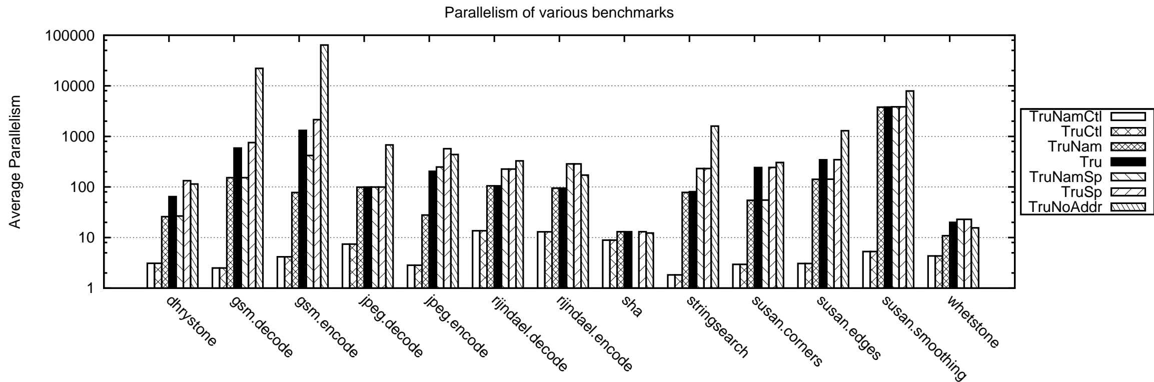


Figure 3.4: Parallelism of various benchmarks under the seven models

activation frame on to the stack) has been replaced with a `malloc_frame` pseudo-instruction. Unlike the original instruction, the new `malloc_frame` instruction has no true dependence on the previous value of the stack pointer, but simply returns the address of a free area in the heap that can be used as a new stack frame, thus breaking true dependence chains on the stack pointer. We ignore the overhead that this instruction might cause in practice, and assume this instruction takes one cycle, just like all other instructions.

TruSp In this model we look at the effect of a spaghetti stack on the DDG when only true dependences are considered.

TruNoAddr In this model, we remove from the graph nodes involved in address computation, in other words, nodes producing values that ultimately contribute to the address component of a memory load or store instruction. Note that although this model subsumes the TruSp model, removing nodes has the effect of reducing the graph size, so if the critical path is not affected significantly by this transformation, the average parallelism could in fact decrease slightly compared to TruSp, even if the program runs faster overall. If address calculations made up a large part of the critical path, however, parallelism would rise.

3.4 Results

Figure 3.4 shows the results of our analysis under the seven models, from which we focus on specific areas to draw several observations.

3.4.1 Effects of control dependences

By comparing the TruNamCtl and TruCtl models with TruNam and Tru respectively in Figure 3.4, we see that parallelism is severely restricted in the presence of control

dependences. This is because in effect they largely prevent code from different basic blocks from being executed in parallel, leaving only intra-(dynamic) basic block parallelism. With control dependences (the TruNamCtl and TruCtl models) most benchmarks only exhibit average parallelism (instructions per cycle) of less than ten, on which the removal of anti- and output dependences have no effect. This generally confirms Wall's findings [99], although due to slight differences in parameters and benchmarks exact comparisons are not possible. In fact a lot of this mostly *intra*-basic block, *instruction-level* parallelism, is already exploited in multiple-issue processors. In order to extract more significant amounts of implicit parallelism, therefore, we need to look beyond basic block boundaries for parallelism at a coarser granularity.

Fortunately there are already techniques which allow us to safely violate some of these control dependences, resulting in potential parallelism higher than what is suggested here. The first of these is the successful use of branch prediction in computer hardware, which results in high rates of correct prediction (at least 80% and often over 95%) with relatively straightforward schemes [83]. Branch prediction allows instructions following a branch instruction to be scheduled before the branch instruction itself, with the results of such speculative instructions discarded if the prediction turns out to be wrong. Nevertheless, even perfect branch prediction would not be able to eliminate all control dependences, as it tends to apply only locally—in other words, the window size[2] is still limited. This means that instructions that are more than a few branches away cannot be scheduled before all these branches even in the absence of other dependences. Thread-level speculation allows us to speculate at a coarser granularity, but at the same time the overheads and misspeculation penalties are higher.

Secondly, it can also be argued that some *inter*-basic block parallelism can be extracted statically while preserving real control dependences. Recall that in our analysis, we make the over-approximation that every instruction is assumed to be control-dependent on the last branch or indirect jump instruction. However, control merge points, which are not visible on the execution trace, may actually eliminate some control dependences. Consider a program of the form `if S1 then { S2 } else { S3 } S4`. While in our analysis `S2`, `S3` and `S4` would all be dependent on `S1`, we note that `S4` would actually be executed regardless of which way the branch went, and therefore can be executed before the result of `S1` is known. Recognising merge points require static analysis such as that used by the Program Dependence Graph [27], and it will be interesting to extend our analysis to examine the increase in parallelism resulting from its use. However, past research suggests the impact of such static analysis is limited in this respect, especially regarding general non-numeric applications [47, 88].

[2] In hardware scheduling, the window size is the maximum number of pending instructions that are considered for scheduling at any one time.

3.4.2 Effects of name dependences

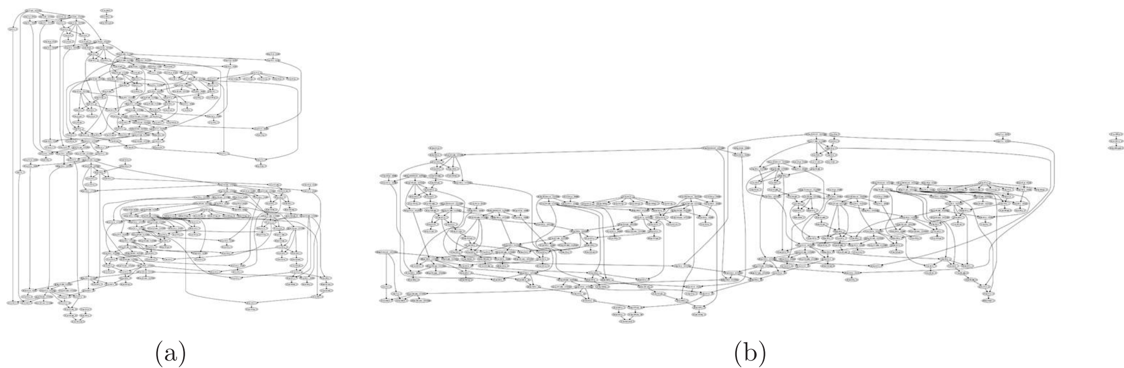
We see the effects of name dependences by comparing the TruNam model with the Tru model. The picture is rather mixed. Some programs exhibit a great increase in parallelism, up to 16 times for gsm.encode, while others (e.g. rijndael, sha and stringsearch) seem largely unaffected. We conclude from this that for certain programs removing name dependences by memory renaming would be important for parallelisation. However, some dependences, e.g. on privatisable variables, are more easily removed than others. A useful extension to this chapter could look at categorising these name dependences based on how they could be removed.

3.4.3 True dependences only

As mentioned earlier, while control dependences can be speculated away, and anti- and output dependences can be reduced with register and memory renaming and other compiler optimisations, true dependences represent the true essence of a program and generally cannot be removed. The Tru model should therefore give us a picture of the limits imposed on parallelism by the algorithm only. Figure 3.4 shows that under this model, many programs exhibit average parallelism of over 100. This is much greater than speed-ups reported for existing implementations of parallelising compilers, which tend mostly to be in single figures [10, 42, 68, 67], showing that there is still a big gap between the amount of speed-up we can achieve with current parallelisation techniques and what is theoretically possible. This gap means that there is still much potential parallelism to be exploited, even if such high parallelism may not be achievable without drastic changes to the compilation method, programming language or even model of computation.

3.4.4 Removing compiler-induced dependences

We argue that even some of the true dependences are only compiler artifacts and do not constitute the essence of the underlying algorithm as specified by the programmer. One example of this is the use of the execution stack and the stack and frame pointers, as described earlier. By comparing models TruNam and Tru with TruNamSp and TruSp respectively, we see how the limits of parallelism change when a *spaghetti stack* is used instead of a linear one. Figure 3.5 shows graphically the effect this replacement has on the DDG of a synthesised program with two identical procedure calls. With just a linear stack, the sub-graph of the first procedure call is vertically above that of the second, while with a spaghetti stack, the sub-graphs of the calls are now side by side, making the critical path roughly half as long.



(a) (b)

Figure 3.5: Shape of Dynamic Dependence Graph of a program with two identical procedure calls (a) without and (b) with the "spaghetti stack" transformation. Note the relative width (average parallelism) and depth (length of critical path) of the two graphs.

Our results in Figure 3.4 again show a mixed picture with this optimisation. For some benchmarks (jpeg.decode, sha, susan), the spaghetti stack makes no difference at all to the available parallelism. For some others (dhrystone, gsm.decode), parallelism only increases with the spaghetti stack when we consider only true dependences—name dependences on the heap and within the same frame would otherwise remain as the bottleneck, preventing the spaghetti stack from affecting the overall parallelism. But for the remaining (rijndael, gsm.encode, jpeg.encode, stringsearch, whetstone) we have a twofold increase in parallelism even if we still consider anti- and output dependences on the heap. This is good news as it shows that even under more realistic assumptions a relatively straightforward change to the compiler can still result in a doubling of available parallelism. It needs to be noted however that this model was used under the assumption that each `malloc_frame` instruction, like all other instructions modelled, takes one cycle to execute, and does not cause other dependences. In practice it is likely that this instruction will take much longer, and depending on its implementation may necessitate new instructions not modelled in our graphs.

Finally, by comparing the Tru and TruNoAddr models in Figure 3.4 we see the effects of ignoring address calculations by removing from the graph all instructions the results of which are subsequently used anywhere as addresses for memory accesses. In some cases (most notably gsm and stringsearch) this has resulted in an increase of an order of magnitude over the Tru model. In other cases there is in fact a slight drop in parallelism, showing that the drop in the critical path length was not great compared to the drop in overall graph size. Generally, however, we see high average parallelism, rising above 1000 in some cases.

3.5 Related Work

Wall [99] produced a comprehensive study on the limits of instruction-level parallelism in SPEC benchmarks. Using an oracle based on the execution trace of a benchmark program, he used a greedy algorithm to schedule each instruction at the earliest possible cycle. He examined the effects on available parallelism of popular optimisations, such as memory disambiguation, register renaming and branch prediction, as well as practical limits on window size (the maximum number of instructions considered for scheduling at any one time) and cycle width (the maximum number of instructions that can be scheduled to run in the same cycle). He found that branch and jump prediction resulted in the highest gain in parallelism in programs, followed by register renaming and memory disambiguation. He also discovered that parallelism was significantly limited when window size was restricted, meaning that much of the parallelism related to instructions that were very far apart. While Wall's concluding view on available parallelism was ambivalent, we believe that by shifting the focus from instruction-level parallelism in superscalar processors to coarser-grained parallelism in multi-core processors, new possibilities can be explored which had been thought difficult to achieve at the time. This is our aim in this chapter, and we explore this further in the rest of the thesis. Our work extends the scope of Wall's study, for instance by also renaming memory locations and considering compiler artifacts.

Lam and Wilson [47] examined in more detail the effects of control dependences on parallelism. Using a similar method to Wall's they looked at parallelism when using a mixture of control dependence analysis, multiple flows of control and speculative execution. Anti- and output dependences were ignored, as were true dependences on the stack pointer and those on call/return instructions. Their conclusions underlined the importance of speculating on control dependences to realising sufficient parallelism.

Austin and Sohi [7] were one of the first to use Dynamic Dependence Graphs (DDGs) of assembly instructions to study limits of parallelism. Their study largely assumed control dependences were perfectly resolved, and focused on the effects of anti- and output dependences on available parallelism. They found that register renaming alone resulted in good levels of parallelism, but more is exposed when memory locations are also renamed. They also obtained a profile of the parallelism of the program over its run-time, and found that parallelism is often unevenly distributed and bursty. Our Dynamic Dependence Graphs are similar to the ones used there, but we have extended their analysis by also examining control dependences and address-related true dependences.

Postiff et al. [72] ran a similar analysis using Wall's method, ignoring control dependences. They reached similar conclusions to Austin and Sohi regarding register and memory renaming. However, they also looked at true dependences on the stack pointer, and found significant gains in parallelism in some cases when they are eliminated. Our methodology of using DDGs is different, but our results generally agree with theirs. Our work also

extends this one by considering control dependences and examining the effects of ignoring all address-related true dependences (not just on the stack pointer).

Stefanović and Martonosi [86] also created DDGs to look at the effects of eliminating address calculations. This was done by removing loads and stores from the graph, replacing them with an edge that goes directly from the producer instruction of the value to the consumer instruction. Instructions from which values are used only as addresses to loads and stores are also removed. They reported vast improvements in parallelism in some cases, but slight deterioration in others where address calculations do not form long dependence chains. Our TruNoAddr model is similar to this analysis, with the exception that only address calculation instructions are removed and not loads and stores, as we reason that loads and stores are still required even if we can now statically determine addresses rather than calculating them at run-time. The results of their study parallel our results for this model.

As can be seen, much work has been done on limits of parallelism. We claim that this chapter is a unified study that explores ideas in the models presented in these previous studies, allowing like-for-like comparison between such models.

In addition to finding limits of parallelism, several studies have also investigated its applicability to limited processors. A new parameter is added to the model that restricts the number of instructions that can be scheduled in each cycle (known as cycle width in [99]). If parallelism is evenly distributed throughout the program, then possible speed-up will remain about the same as cycle width is reduced. If on the other hand parallelism is concentrated in one section then overall speed-up can fall quickly with cycle width. Theobald et al. [91] introduced the concept of *smoothability*, which is the new limit of parallelism given a cycle width equal to the original (infinite cycle-width) limit of parallelism, as a proportion of the original. Rauchwerger et al. [76] used another measure known as *slack*, defined as the average distance between an instruction in a greedy schedule and the same instruction in a lazy schedule.

3.6 Conclusions

This chapter examined the limits of parallelism in benchmark programs, and the effects of various types of dependences on them, by constructing Dynamic Dependence Graphs from execution traces. As with previous studies that we extend, we find that control dependences form the biggest obstacle to realising more than a modest amount of parallelism (above single figures). By disregarding control dependences the available parallelism is much greater, even more so when memory locations are renamed to remove anti- and output dependences. In fact, we find that parallelism for many programs exceeds 100 when only true dependences are considered. We have also looked at true dependences on the

stack pointer, reasoning that these are not an essential part of an algorithm but are only compiler-induced, i.e. there is more parallelism in source code than in assembly code. By using a *spaghetti stack* to remove some of these dependences, we double available parallelism in some programs. It would be interesting to scale up our experimental framework, in order to extend our analysis from the embedded benchmarks of MiBench into server benchmarks. Another direction, which we will take in the next chapter, is to explore how much of the parallelism is coarse-grained enough for profitable exploitation in multi-core processors. We view our findings with qualified optimism, in that while the limits of parallelism are generally quite high, achieving it in practice is not straightforward, and will require significant changes to the current method of compilation, programming language or even model of computation. Nevertheless, we find that in most of these benchmark programs much parallelism is there—the challenge is to find better ways to exploit it.

Chapter 4

Separating the wheat from the chaff—estimating task-level parallelism

4.1 Introduction

In the last chapter we have observed that there is much potential parallelism in most programs. Our goal for the rest of our thesis is to realise some of this potential parallelism in multi-core processors. Using ideas from the last chapter, we have developed a tool-chain for interactive parallelisation. In this chapter, we present the first component of this tool-chain: Embla 2, a tool that estimates the amount of potential *task-level* parallelism in a program and presents valuable information for actual parallelisation.

We begin this chapter by noting that not all of the parallelism discovered in our last chapter is exploitable by multi-core processors. In particular, parallelism at the instruction level, where only short chains of several instructions are independent from each other at a time, cannot be applied to multi-core processors, as communication between cores would take at least tens of cycles, meaning performance gains from parallelism would be vastly eclipsed by the extra overheads involved in splitting up the work. Instead, instruction-level parallelism is already well exploited with superscalar and VLIW technologies, and it is neither necessary nor desirable to have such parallelism exploited with multiple cores.

For multi-core processors we need parallelism at a *coarser* granularity—*threads* hundreds if not thousands of instructions long that make the overheads of parallelisation worthwhile. In this chapter, we attempt to separate the wheat (coarse-grained or thread-level parallelism) from the chaff (fine-grained or instruction-level parallelism) by restricting ourselves to a certain class of parallelism, namely *Nested Function-level Fork-join Parallelism*, in which function calls/returns are used to delineate tasks. While this is by no

means the only class of coarse-grained parallelism available, it is chosen for the following reasons:

- As a programming abstraction, functions generally represent a reasonable amount of work.
- As programming language constructs, functions require minimal syntactic restructuring to exploit—all the code in the thread is already gathered in one place.
- Functions have a single entry point and return address, facilitating task creation.
- Partly due to reasons above, many popular parallel programming environments, such as Cilk [11], Java [49], TBB [78] and TPL [50], use this as one of their main task models.

Parallel loops, another useful source of coarse-grained parallelism, are also incorporated into our models as an option. Naturally other legitimate sources of coarse-grained parallelism will be left out, such as pipeline parallelism, the extent of which we estimate with another variant model.

Our primary objective in this chapter is to construct a tool which, when given a sequential program and an input, can estimate the amount of parallelism that would be available if the program were to be parallelised using language constructs such as those in OpenMP [21] or Cilk [11]. It can be considered as the degree of speed-up we might expect to get without major structural refactoring. We validate our results against timing measurements of explicitly parallel Cilk programs.

As in the previous chapter, the tool works by profiling data and control dependences. However, here the dependences are mapped back to source code using debugging information. Such functionality is provided by Embla [26], a Valgrind-based [63] profiler which captures and outputs source-level data dependence information. Our tool extends Embla and is accordingly named Embla 2. Our extensions can be summarised as: capturing control dependences, discounting spurious dependences due to heap reuse (Section 4.3.2), construction of Dynamic Dependence Graphs and output of critical paths (Section 4.3.3), loop recognition (Section 4.3.4), reduction variable recognition (Section 4.3.5) and granularity analysis (Section 4.3.6). Embla 2 contains about 6000 lines of C code (compared to 4000 lines for the original Embla).

The focus of traditional studies of parallelism limits [99, 100] is typically on hardware support for some model of parallel execution. As such parameters such as the amount of hardware resources (buffer space, functional units, etc.) are varied. Our focus is rather on explicit parallelisation of the program source code, and we therefore use a different set of parameters to vary and different constraints for parallelisation. The aim is to address

the question, "Will techniques such as thread-level speculation or parallel for-loops make much difference to the possible speed-up when parallelising this program?" Section 4.2 discusses our parallelisation models in depth.

We have used Embla 2 to investigate the potential for parallelisation of three collections of programs: the SPEC CPU 2000 integer programs, MiBench and the example programs in the Cilk 5.4.6 distribution. While the first two collections contain sequential programs, the last one is made up of explicitly parallel programs in the Cilk 5 language. These have the property that eliding the parallel constructs leaves correct sequential code, allowing us to run Embla 2 on that code as validation of our approach. We can thus contrast programs from these different sources as well as show the behaviour of our different models. Section 4.4 reports the results of these experiments.

The contributions can be summarised as follows:

- Dependences output by Embla 2 assist the programmer in parallelisation by identifying appropriate source-level synchronisation points. We show how synchronisation points identified in serial elisions of example Cilk programs by Embla 2 match those in the original hand-parallelised programs (Section 4.4.1). In Chapter 5 we show how this parallelisation can be automated.

- By mapping dynamic dependences back to program source, Embla 2 gives a realistic estimate of potential speed-up for parallelisation using frameworks such as Cilk and OpenMP. Previous studies that we know of [45, 100, 66] have only considered *speculative* task-level parallelism—we give potential speed-ups for programs both with and without the use of thread-level speculation. Using Embla 2 we present estimates of potential parallelism in various example programs and benchmarks (Sections 4.4.2 and 4.4.4).

- Critical paths output by Embla 2 can be used to identify at source-level bottlenecks that restrict the level of potential parallelism. In Chapter 6 we illustrate this using examples from various benchmark suites, and suggest refactorings or algorithmic changes to increase potential parallelism.

4.2 Task Models

Embla 2 uses task models designed to reflect existing and potential capabilities of popular parallel programming environments. We begin by describing our baseline task model, before detailing how other variant models deviate from it. In our model each function call is spawned as a task at its call site. The calling thread can then continue to execute statements that follow the function call without waiting for the call to return, until control

```
p();            spawn p();
q();            spawn q();
r();            sync;
                spawn r();
```

Figure 4.1: An example of function-call spawning and synchronisation, using Cilk-like syntax

reaches a statement that has a dependence on the call. At this point the calling thread must *synchronise* on the task, i.e. wait for the task to complete before going any further.

As an example of this, consider the program fragment in Figure 4.1 (left): Suppose that the calls to `p()` and `q()` are independent, but that the call to `r()` depends on the earlier calls. Then the call to `p()` can be executed in parallel with the call to `q()`, as shown on the right. Here we assume the availability of the constructs `spawn` to start the call in parallel and `sync` to synchronise on all previously `spawn`'ed tasks (cf. Cilk).

As in Cilk, there is an additional requirement that each function call must synchronise on all tasks it has spawned before returning, meaning that tasks must properly nest. Such a design choice may result in some potential parallelism being lost—a task spawned by function `f()` may only have a dependence on it long after `f()` has returned, but under our model this task must be synchronised on before `f()` returns. However, it preserves program modularity by ensuring that each task is responsible for synchronising on the tasks it has spawned.

We assume the underlying machine has an unlimited number of processors with zero overheads for spawning and synchronisation. All instructions, including memory access, are assumed to take one cycle. This means that the estimate derived in this model is an architecture-independent upper bound.

So how do we calculate the potential parallelism for a program under this model? As in Chapter 3, we do this by constructing a *Dynamic Dependence Graph* for each function call, an example of which is shown in Figure 4.2. But here the DDG is a directed acyclic graph $G = (V, E)$ where each node $v \in V$ corresponds to an *instantiation*, or execution, of a *line*[1] of the function, and an edge $(u, v) \in E$ means that u must be completed before v can begin. Edges $u \rightarrow v$ are inserted for:

1. Data dependences, which as before can be read-after-write, write-after-read or write-after-write, between an instruction in the dynamic call tree of u to one in that of v (as observed by the original Embla infrastructure described above). Write-after-read and write-after-write dependences on the stack are ignored (these tend to be on

[1] We would like to have a node corresponding to each execution of a *statement* instead, but the gcc debug tables that Embla uses are per-line not per-statement.

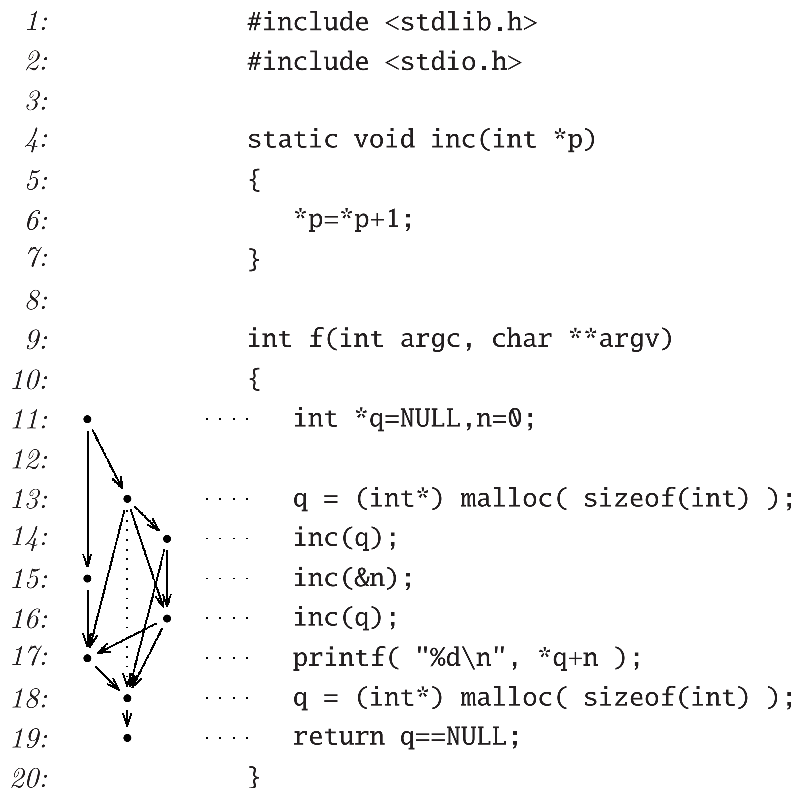


Figure 4.2: Example dependence graph from Embla

easily privatisable variables), as well as all dependences between known commutative library functions, e.g. `malloc`.

2. Control dependences between u and v.

3. Dependences related to the parallelisation model, that is, the restructuring of the code that we allow. E.g. to enforce the requirement that only function calls can be spawned as tasks, there is an incoming edge to each node from the last executed node that does not contain a function call. This effectively linearises the graph except at function call spawns, the only points where forks are allowed.

The *cost* of a node v, $cost(v)$, is the minimum time required to execute the instantiation under our model. This is essentially the number of instructions executed on that line, with the additional requirement that if one of the instructions is a function call, then the length of the *critical path* of that call is included in the *cost* of the node. The *critical path* CP is the path in G with the largest total *cost*. The amount of potential parallelism for a function call is then the cost of serial execution divided by the length of the critical path. The potential parallelism of a program is simply that of its `main` function.

In addition to the baseline model, we alter certain parameters to create variant models, in

```
void g1(int *a) {
  ...
  *a = ...; // Writes to *a
}

void g2(int *b) {
  ... = *b; // Reads from *b
  ...
}

int f(int *a, int *b) {
  int x;
  g1(a);
  g2(b);
}

int main(int argc, char *argv[]) {
  int x=0, y=1;
  f(&x, &y);
  f(&x, &x);
}
```

Figure 4.3: Program illustrating the difference between aggregated and exact data dependences

order to explore how potential parallelism changes with respect to different design choices, compiler optimisations or run-time techniques. These are described below:

Exact data dependences By default, data dependences are *aggregated* per source line over a program's execution and then applied to all instantiations of the line. In other words, in the Dynamic Dependence Graphs every instantiation of a line is dependent on instantiations of the same set of lines. This models the parallelism that is possible by inserting synchronisation points in source code, necessitating the same synchronisation point for all instantiations. A variant is to use *exact* dependences by allowing each instantiation of the same line to have its own set of dependences as observed during execution[2], modelling the effects of dynamic techniques such as thread-level speculation (TLS), in the ideal case where the continuation of each task instantiation runs in parallel with the task right up until the point where a conflict would have been detected.

[2]Cf. the difference between context-sensitive and context-insensitive static analysis.

The difference between the two can be illustrated in the program in Figure 4.3, where there is a dependence between the calls to `g1` and `g2` in `f` only when `a` and `b` alias. A model using aggregated data dependences would ascribe that dependence to all calls to `f`, resulting in no parallelism. This reflects the fact that we cannot statically parallelise the calls to `g1` and `g2` as that would lead to a race condition in the second call of `f`. With exact dependences however, the dependence would only apply in the second call to `f`, resulting in some parallelism in the first call. This reflects that with TLS, we can attempt to run `g1` and `g2` in parallel, which results in potential parallelism in the first call to `f`, at the expense of a possible performance penalty of a rollback in the second call.

Control dependences When it is not known whether a line will be executed until the execution of another (typically a branch), the former line is said to be control-dependent on the latter, and must not begin execution until the latter has completed. In our tool, control dependences are calculated by constructing Control Dependence Graphs [27]. However, our model also allows control dependences to be excluded. This in effect gives us the parallelism achievable with perfect (100% accuracy) control speculation. Previous studies on branch prediction and control speculation [83, 47] suggest that simple prediction algorithms can result in high accuracies for many programs, meaning that a model with control speculation is reasonable.

Loop iterations as tasks In addition to function calls, loops are generally considered to be another source of task-level parallelism, and are the primary parallel programming construct in OpenMP [21]. As mentioned, there is an option to parallelise loops in Embla 2, where an established algorithm [2] is used to identify natural loops. Each instance of a natural loop is spawned as a task, which in turn spawns individual iterations of the loop as tasks just like function calls. Updates to the loop index, which are outside the task boundaries, are still serialised.

Reduction operations Reductions are accumulation operations on variables such as `for (i=0; i<N; i++) acc += f(i);`, where the order in which instances of the associative reduction operation (+) are executed makes no difference to the final value of the accumulator[3]. Embla 2 includes a helper program that statically identifies and annotates reduction operations in loops, dependences between which can be safely ignored to enable greater parallelism.

Spawn hoisting Another way of enabling further parallelism is by spawning tasks earlier (a form of code motion optimisation). In this variant, function calls are spawned

[3]Floating point arithmetic are not *strictly* associative, but are considered so in Embla 2 as in most cases different orders of application lead to insignificant differences in the results.

```
// Unrelated work
...
/*spawn*/ f();
```

(a) Original program extract

```
/*spawn*/ f();
// Unrelated work
...
```

(b) With spawn hoisting

Figure 4.4: An example of spawn hoisting

as early as dependences allow, rather than only when control in the calling thread reaches the call site (the default). It can be thought of as moving the function call higher. This is illustrated in Figure 4.4, where in the original program, the call to `f()` follows some unspawned statements unrelated to `f()`. Under the baseline model, `f()` can only be spawned when control reaches its call site, i.e. after the unrelated work has been executed, resulting in no parallelism. With spawn hoisting, however, `f()` can be spawned before the unrelated work is executed, so that they can be executed together. Spawn hoisting is modelled by relaxing constraint 3 on page 41 so that the incoming edge is only inserted to each node that does not contain a function call.

Line-level parallelism By only considering function calls and potentially loop iterations as tasks, other forms of task-level parallelism are naturally excluded, namely those that are not delineated by function calls/returns or loop boundaries. Examples include pipeline and DOACROSS parallelism, as well as parallelism between arbitrary portions of code that could only be exploited with code refactoring or transformations. To see how much more parallelism there is, we introduce a variant model where all lines can be spawned, regardless of whether they contain function calls. This is modelled by not applying constraint 3 at all. Note however that while this allows us to see more parallelism, some of the extra parallelism discovered will be too fine-grained to be exploitable. Nonetheless, the remainder could well be coarse-grained enough to be profitably exploited by refactoring certain lines into a new task.

Granularity filtering On the other hand, there is no guarantee every function call is coarse-grained enough that the program is sped up when it is parallelised. Inter-processor communication and the overheads involved in spawning, scheduling and synchronising on a task can lead to inferior performance if task granularity is too small. Thus, in reality, some function calls are better off *inlined* (run sequentially) than spawned. This leads to the question of how much parallelism is left if we do not spawn tasks that are too fine-grained. We answer this question by only allowing a function call to be spawned if its profiled mean size is above a specified granularity threshold. Again this is done by altering constraint 3 such that there is an incoming edge to each node from the last

executed node that does not contain a function call *with a mean size above the threshold*. We detail how we obtain mean task sizes in Section 4.3.6.

However, this extension was not applied to spawned loop iterations, because of the way many parallel task libraries execute parallel for-loops. Instead of spawning each iteration as a separate task, they usually use divide-and-conquer to recursively partition the iteration space in half, spawning each half as a task. This continues until the iteration space is small enough, at which point the partition is executed sequentially.

This means that parallel for-loops with small bodies can still be beneficially executed in parallel, as iterations can be lumped together into one big task (often known as a *chunk*). (In fact, many recursive divide-and-conquer functions can be parallelised this way also.) Embla 2 could be modified to model iteration-lumping as a potential extension, but we feel that the difference in results would be small.

4.3 Implementation

This section provides details on how various aspects of Embla 2 have been implemented. Embla 2 is based on the data dependence profiler Embla [26], which computes static data dependence information. Both Embla and, because of its heritage, Embla 2 are based on the Valgrind binary instrumentation framework [63]. Neither tool uses the source code of the program under investigation, which makes them relatively language independent, with one caveat: if used in a managed code environment such as .NET or a Java virtual machine, they profile the virtual machine rather than the program running on top of it.

While the tools work entirely on machine code, they report results in terms of the source code of the program, using debugging information to bridge the gap. However, since they do not know about the code generator, it is necessary to separate dependences arising from the program from dependences arising from artifacts in the code generator, such as register spills and reloads, that should not affect the dependences reported or the critical path measurements.

We use the following strategy to make clear the connection between the instruction level (which the tools see) and the source level (which the user cares about) without being confused by artifacts of the code generator: run the tools on code compiled with register allocation turned off and record only dependences through memory, not through registers. This works well for languages (or implementations) that adhere to the rule that the abstract program state should be in memory at sequence points in the code. This is the case for instance for C.

While this strategy only gives safe information about subprograms compiled without optimisation, it still allows parts of the program untouched by the parallelising transfor-

Input/Output	Parameter name	Description	Required for
Input	`--dep-file`	List of control and data dependences	Aggregated dependences
Input	`--hidden-func-file`	List of commutative functions (e.g. `malloc`)	Ignoring dependences between commutative functions (this file is supplied for all our models)
Input	`--loop-file`	Line numbers of loops	Spawning loop iterations
Input	`--task-size-file`	Mean sizes of tasks	Granularity filtering
Output	`--trace-file`	Profiled data dependences	Subsequent run with aggregated data dependences
Output	`--edge-file`	Profiled control flow	Loop identification and control dependence identification
Output	`--critpath-file`	Critical paths	Programmer perusal (see Chapter 6)
Output	`--lengths-file`	Profiled lengths of lines	Subsequent granularity analysis
Output	`--task-size-out-file`	Profiled mean sizes of tasks and their continuations	Granularity filtering

Table 4.1: Input and output file parameters for Embla 2

mations to be optimised, so for instance pre-compiled libraries can be handled. Memory-based dependences between these and the rest of the code are correctly tracked and any register-based dependences (return values and possibly arguments) will be reflected in memory in the unoptimised code.

Our strategy suffers from two drawbacks. Firstly, by using debugging information we are only able to attribute dependences to different source lines rather than statements or function calls. Furthermore, by profiling *unoptimised* binaries our calculations become less accurate as performance predictions for the eventual *optimised* programs. An alternative approach is to insert instrumentation at a higher level representation of the program, such as on the abstract syntax tree. However, this approach would require customising a compiler. Also, pre-compiled libraries can no longer be profiled.

Table 4.1 lists the different input and output file parameters supplied to Embla 2. Different subsets of parameters may be passed to Embla 2 depending on the functionality required by the various models. Embla 2 may need to be run several times in order to obtain a parallelism estimate under certain models. For instance, for aggregated dependences Embla 2 must be run twice, first to gather dependences over the whole program's execution (producing `--trace-file` and `--edge-file` as outputs), then again to construct Dynamic Dependence Graphs and perform critical path analysis (using `--dep-file` as input). The only file required for all of our models is `--hidden-func-file`, which contains a list of commutative functions (mostly ones used for memory management, e.g. `malloc` and `free`, but the list can also be extended by the programmer).

As a dependence profiler that instruments every memory access instruction, Embla 2 does incur a significant overhead. When run on an x86 desktop, using Embla 2 to collect dependences causes an average of 1500x slowdown of the profiled program, while critical path analysis causes around 700x. Valgrind itself, on which Embla 2 is based, causes an

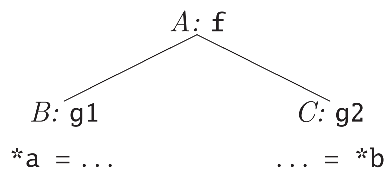


Figure 4.5: Part of the execution tree of the program in Figure 4.3

average slowdown of 8x. It is hoped that further optimisations can bring overheads down.

4.3.1 Computing data dependences

We now describe in brief how Embla maps instruction-level data dependences back to source code. Further details can be found in [26]. Under Valgrind's binary instrumentation framework, Embla tracks data dependences between executed instructions, which are then mapped back to pairs of source lines within the same function. These instructions might not be directly part of the function, but could instead be part of other functions transitively called by the function. For instance, in Figure 4.3 the instructions causing the dependence are part of the *bodies* of `g1` and `g2`, respectively. However, Embla instead maps the dependence to the *calls* to `g1` and `g2` in the definition of `f`.

To do this, Embla uses two main data structures: an *execution tree*, which is a dynamic call tree with individual instructions as leaf nodes (part of the execution tree for the program in Figure 4.3 is shown in Figure 4.5), and a *memory table*, which maps memory addresses to nodes in the execution tree that performed the most recent write (for RAW and WAW dependences) and all reads since that write (for WAR dependences). For each memory access instruction executed, Embla uses the memory address to look up the dependence source nodes in the memory table. For each source node, the *Nearest Common Ancestor* (NCA) node (node A in the example) is then located by traversing upwards through the execution tree. The dependence is then attributed to the two children of the NCA that are respectively ancestors of the source and target instructions (nodes B and C in the example). Line numbers of these two nodes are provided by debugging information.

4.3.2 Discounting spurious dependences due to memory reuse

In order to only capture relevant dependences we must also identify and discount spurious dependences due to memory reuse. These occur both in the execution stack, as we have seen in Chapter 3 with the effects of spaghetti stacks, as well as on the heap, due to freed memory that is reallocated.

For the former, Embla keeps track of the value of the stack pointer, which points to the top of the execution stack, over execution. For each dependence observed, Embla checks

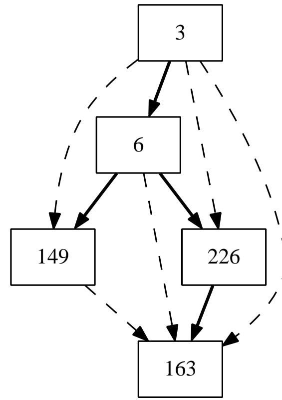


Figure 4.6: Figure showing our optimisation for DDG edges. The solid edges represent the critical path edges for each node, which are stored. The dashed edges represent edges not in the critical path for any node, and so are discarded.

if the stack pointer has crossed the address concerned in the time between the source instruction and the target instruction. If so, the dependence is discounted.

As for the heap, calls to memory allocation functions (`malloc`, `calloc` and `memalign`) are instrumented to record the size of memory requested and the address of the allocated memory returned. If subsequently this address is passed to `free`, then the memory table entries for the corresponding addresses are cleared. The `realloc` function is treated as memory allocation, deallocation or both, depending on its actual behaviour.

4.3.3 Constructing the Dynamic Dependence Graph

Under Embla 2, whenever a new instruction with a different line number from the last is executed, a new node is created in the Dynamic Dependence Graph corresponding to the new line instantiation. For each function call, a new DDG is initialised; similarly the DDG is finalised at the corresponding function return.

For *exact* data dependences, edges are inserted into the DDG between nodes that correspond to the source and target nodes in the execution tree, as dependences arise. For *aggregated* data dependences, a prior run of Embla 2 collects all data dependences as pairs of line numbers. Then in the second run, for each node n corresponding to line l, edges are inserted to n from all instantiations of lines on which l is dependent in the current DDG.

In order to save memory space, we note that for critical path analysis it is not necessary to store all the edges in a DDG. It suffices for each node to have a pointer to one other node, corresponding to the predecessor on the longest path so far terminating at the node (the *critical path* for that node). Figure 4.6 illustrates this with an example. For each new dependence, if the new source node has a longer critical path than the existing one, then

```
for (i=0; i<N; i++) {
  task1();
  task2();
}
```

Figure 4.7: An example loop with DOACROSS parallelism

the pointer is changed to point to this new node; otherwise no changes need to be made. This optimisation is correct as edges are always only added to the latest node, meaning that critical paths of existing nodes remain fixed once we move on to a new node. When the function call returns, the longest critical path of all the nodes becomes the critical path of the DDG.

4.3.4 Loop recognition

Embla can output a list of all line-level control flow edges, which can be used to construct Control Flow Graphs (CFGs) for each function. Using these CFGs, single-entry/single-exit loops are identified using the *natural* loop identification algorithm described in [2] among other places. The decision to use Embla output to construct CFGs was mostly for convenience—static analysis could have been used to construct CFGs, but this would make little difference to our final results.

However, it is in fact not always profitable to spawn loop iterations, as it precludes DOACROSS or pipeline parallelism. Consider the loop in Figure 4.7, where there are dependences between calls to `task1` and between calls to `task2` as well as from `task1` to `task2` but crucially not from `task2` to `task1`. Hence there are loop-carried dependences. If loop iterations were spawned as tasks, each iteration would have to have completed execution serially before the next iteration could begin. However, if the iterations were not spawned, then each call to `task2` could in fact be spawned and not synchronised on until after the call to `task1` in the next iteration, allowing us to overlap these two calls and get parallelism. Thus, we have introduced a further model where only iterations of parallel loops, i.e. those with no loop-carried dependences as observed by Embla, are spawned.

4.3.5 Reduction variable recognition

In order to recognise reduction operations, a static analysis is performed on program source and annotations are added to these operations. We do this by using the ROSE source-to-source compiler framework [82], with which we search for variables x in loops that satisfy the following requirements:

1. x is of scalar type.

2. x is not aliased (address-taken).

3. x is declared outside the loop (not privatised).

4. All occurrences of x in the loop are of the form $x = x \oplus expr$, where $\oplus$ can be addition, subtraction, multiplication or a primitive bitwise or logical operator (all associative), but must remain the same throughout the loop. x must also not occur inside $expr$. Alternatively if all occurrences of x in the loop are of the form $x = expr$ (where x does not occur inside $expr$) this is an *assignment* reduction operation and is also permitted[4].

Once these reduction variables have been identified, macros are wrapped around all reduction operations, which when expanded place *client requests* before and after the operation. Client requests are series of instructions that do nothing except notify the Valgrind run-time system. This way, Embla 2 knows when an instruction belongs to a reduction operation, and can tag its entry in the memory table. When a dependence is encountered of which both the source and target entries are tagged, this dependence can be disregarded.

4.3.6 Granularity analysis

In order to filter out tasks that are too fine-grained, we need to find out the sizes of function calls. We did that by extending Embla to record mean serial lengths of function calls, measured in terms of the total number of instructions, just as for the critical path. Due to the instrumentation infrastructure of Embla, we record lengths on the source-line level, but record separate figures for a line depending on whether it calls a function at run-time. This deals with the case `if (...)  f();` where every call to `f()` is large—if many instantiations of this line do not in fact call `f()` then the overall mean length may still be small, even though if and when we call `f()` the task will be sufficiently coarse-grained. We define *mean task length* as the mean serial length of a line when counting only instantiations that actually call a function at run-time.

We then extended Embla to take into account the granularity of a function call, such that only function calls with a granularity above a certain threshold are spawned. Analogous to the "Aggregated data dependences" model, either all or none of the instantiations of each line are spawned, based on the average granularity. This reflects what happens when we specify at source level whether to spawn or inline a function call. We do not consider run-time techniques to dynamically decide whether to spawn a function.

[4]This can be thought of as another associative operator $\oplus$ defined as $x \oplus y \equiv y$.

A naïve approach would be to define granularity simply as *mean task length*. The problem with this, however, is that it would still not be beneficial for a very long task to be spawned if it is to be synchronised soon after. Another way to see this is to consider a spawn as a fork operation that creates two tasks—the function call and the *continuation*—and a synchronisation as a joining of these two tasks. To gain useful parallelism it is necessary for both the function call and the continuation to be sufficiently long. We must therefore introduce another measure, the *mean continuation length*, or the expected amount of work between the task and the first synchronisation point encountered.

In order to derive mean continuation lengths we first need to identify synchronisation points for each function call, which would require dependence information from a previous run of Embla 2. Having identified synchronisation points, we then need to work out the mean number of instructions between each function call and the first synchronisation point encountered. We have thus further extended Embla to profile such information in a new prior run.

As an alternative to an extra run of Embla 2 we can also estimate this figure probabilistically based on information already gathered by Embla—the mean lengths of each line (both when it calls a function at run-time and when it does not), the Control Flow Graph (extended to give the number of times each edge is taken during profiling), and the dependence information. This estimate must take into account the different possible control paths in the continuation, and the relative frequencies of these paths.

Let L be the set of source lines in the program. For each line $l \in L$, take n_l to be the total number of instantiations of l and c_l to be the *cost*, or mean run-time length over all instantiations of l measured in terms of number of instructions.

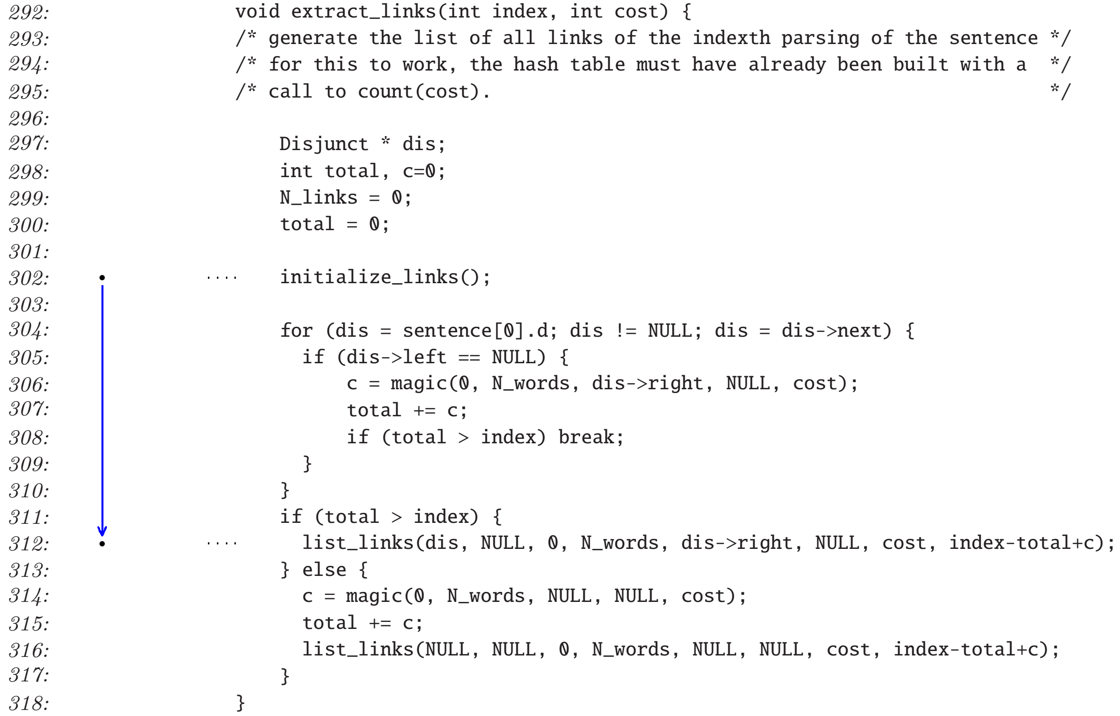
Let $E \subseteq L \times L$ be the set of control flow edges in the program. For each edge $(i, j) \in E$, we define n_{ij} to be the number of times the edge is taken as profiled. Then the *probability* that an edge (i, j) is taken (given control reaches line i) is derived by:

$$p_{ij} = \frac{n_{ij}}{n_i}$$

Let s be a line with a function call. To estimate the mean continuation length of the task at s we first define $L^{s\rightarrow} \subseteq L$ as:

$$L^{s\rightarrow} = \{l \in L \mid \exists \text{ path } P \text{ from } s \text{ to } l \text{ and } \forall m \in P \setminus \{l\}.\ m \text{ is not dependent on } s\}$$

Under the simplifying assumption that the probability of a control flow edge being taken is independent of prior control flow—the *Markov property*—we have a *Markov chain* where $L^{s\rightarrow}$ is the set of states, and $\mathbf{P} \equiv (\mathbf{P}_{ij})$ is the transition probability matrix. Line s is the starting state, and at each step, the chain moves from state i to state j with probability $\mathbf{P}_{ij}$.



```
292:    void extract_links(int index, int cost) {
293:    /* generate the list of all links of the indexth parsing of the sentence */
294:    /* for this to work, the hash table must have already been built with a  */
295:    /* call to count(cost).                                                   */
296:
297:        Disjunct * dis;
298:        int total, c=0;
299:        N_links = 0;
300:        total = 0;
301:
302:        initialize_links();
303:
304:        for (dis = sentence[0].d; dis != NULL; dis = dis->next) {
305:          if (dis->left == NULL) {
306:              c = magic(0, N_words, dis->right, NULL, cost);
307:              total += c;
308:              if (total > index) break;
309:          }
310:        }
311:        if (total > index) {
312:          list_links(dis, NULL, 0, N_words, dis->right, NULL, cost, index-total+c);
313:        } else {
314:          c = magic(0, N_words, NULL, NULL, cost);
315:          total += c;
316:          list_links(NULL, NULL, 0, N_words, NULL, NULL, cost, index-total+c);
317:        }
318:    }
```

Figure 4.8: Example function from 197.parser and its dependences

For each line i that does not depend on s the transition probabilities are:

$$\forall j \in L^{s\rightarrow}.\ \mathbf{P}_{ij} = p_{ij}$$

Function exits and lines that depend on s are *absorbing* states—for each absorbing state i the transition probabilities are:

$$\forall j \in L^{s\rightarrow}.\ \mathbf{P}_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

Let $\mathbf{Q}$ be the submatrix of $\mathbf{P}$ formed by taking only the rows and columns for non-absorbing (a.k.a. *transient*) states. Let $\mathbf{I}$ be the identity matrix with the same dimensions as $\mathbf{Q}$. Further let $\mathbf{c}$ be the vector of mean run-time lengths of non-absorbing states, i.e.:

$$\forall i \in \text{non-absorbing states of } L^{s\rightarrow}.\ \mathbf{c}_i = c_i$$

It can be shown [33] that the vector $\mathbf{t}$ of expected total costs incurred before an absorbing state is reached, starting from non-absorbing states, can be calculated as the solution to $(\mathbf{I} - \mathbf{Q})\mathbf{t} = \mathbf{c}$, and therefore the expected run-time length from the spawn at line s until the first synchronisation point (excluding the length of the function call itself), in other words the *mean continuation length*, is $\mathbf{t}_s - \mathbf{c}_s$.

As an example, take the `initialize_links()` function call on line 302 of `extract-links.c` from the SPECint benchmark `197.parser`. The conditional on line 311 always evaluates to true during profiling, and only line 312 is dependent on this call in this function. Therefore, $L^{302\rightarrow} = \{302, 304, 305, 306, 307, 308, 311, 312\}$. Only line 312 is an absorbing state, so matrix $\mathbf{Q}$ would have 7 rows and columns, and $\mathbf{c}$ would have 7 rows.

$$\mathbf{Q} = \begin{array}{c} \\ 302 \\ 304 \\ 305 \\ 306 \\ 307 \\ 308 \\ 311 \end{array} \begin{array}{c} \begin{array}{ccccccc} 302 & 304 & 305 & 306 & 307 & 308 & 311 \end{array} \\ \begin{pmatrix} & 1 & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & & & & 1 & & \\ & & & & & 1 & \\ & \frac{1}{2} & & & & & \frac{1}{2} \\ & & & & & & \end{pmatrix} \end{array} \qquad \mathbf{c} = \begin{array}{c} 302 \\ 304 \\ 305 \\ 306 \\ 307 \\ 308 \\ 311 \end{array} \begin{pmatrix} 53 \\ 5 \\ 4 \\ 197 \\ 2 \\ 3 \\ 3 \end{pmatrix}$$

(Row and column labels denote line numbers.) Solving for $(\mathbf{I} - \mathbf{Q})\mathbf{t} = \mathbf{c}$, we have:

$$\mathbf{t} = \begin{array}{c} 302 \\ 304 \\ 305 \\ 306 \\ 307 \\ 308 \\ 311 \end{array} \begin{pmatrix} 478 \\ 425 \\ 420 \\ 416 \\ 219 \\ 217 \\ 3 \end{pmatrix}$$

Here $\mathbf{t}_{302}$ is 478, and so the expected continuation length for this task is $\mathbf{t}_{302} - \mathbf{c}_{302} = 478 - 53 = 425$ instructions.

While actual execution of a program is deterministic and not probabilistic—strictly speaking the Markov property does not hold because in theory control flow history can determine with certainty the next edge to be taken in the Control Flow Graph—we argue that it may still model reality in most cases. We will evaluate the accuracy of our probabilistic model by comparing mean continuation lengths calculated probabilistically with the mean continuation lengths as profiled by Embla 2 (see Section 4.4.5).

4.4 Results and Evaluation

In this section we present some outputs of Embla 2 to demonstrate both how Embla 2 can be used and how much potential parallelism it has found in different programs.

Program	Description	Parameters
cholesky	Matrix decomposition	size=256, nonzeros=1000
cilksort	Merge sort	size=100000
fft	Fourier transform	size=512*512
fib	Naïve Fibonacci calculation	n=30
heat	Differential equation solver	nx=ny=512, nt=1
lu	Matrix decomposition	n=256
magic	Magic squares search	n=4
matmul	Matrix multiplication	n=128
plu	Matrix decomposition	n=128
strassen	Matrix multiplication	n=512

Table 4.2: Description and parameters for Cilk 5.4.6 examples used

```
384:   spawn cilksort(A,tmpA,quarter);
385:   spawn cilksort(B,tmpB,quarter);
386:   spawn cilksort(C,tmpC,quarter);
387:   spawn cilksort(D,tmpD,size-3*quarter);
388:   sync;
389:
390:   spawn cilkmerge(A,A+quarter-1,B,B+quarter-1,tmpA);
391:   spawn cilkmerge(C,C+quarter-1,D,low+size-1,tmpC);
392:   sync;
393:
394:   spawn cilkmerge(tmpA,tmpC-1,tmpC,tmpA+size-1,A);
```

Figure 4.9: An extract from cilksort and its corresponding relevant dependences

We begin our demonstration of Embla 2 with example Cilk programs packaged with the 5.4.6 release of Cilk, as described in Table 4.2[5]—programs known to have lots of task-level parallelism. Furthermore, the nature of Cilk means that these programs can be translated into semantically equivalent programs in ordinary C (known as *serial elisions*) simply by stripping the Cilk keywords[6]. We thus have a set of C programs and their manually-parallelised counterparts to compare with the parallelism discovered by Embla 2.

4.4.1 Finding optimal task synchronisation points

Programmers can use aggregated dependences discovered by Embla 2 to realise the parallelism discovered using a language like Cilk, where function calls are spawned as tasks and later joined (or *synchronised*), by synchronising on previously spawned tasks just before the first line dependent on them. As an example, Figure 4.9 shows an extract from

[5]We have omitted programs that use the `inlet`, `abort` and `SYNCHED` keywords, as their translation into ordinary C is not straightforward.

[6]Namely, `cilk`, `spawn` and `sync`.

the cilksort program, along with aggregated dependences discovered by Embla 2 on the program's serial elision. By spawning function calls and synchronising before the first line that depends on previously spawned tasks, we arrive at the Cilk program with optimal parallelism, which in this case is the same as the original program. An algorithm for automating this process is described in the next chapter. However, the programmer should be aware that as with all dynamic analyses, these dependences are based on running the program on sample inputs only and so the parallelisation may not be safe for all possible inputs. Nevertheless, previous work [26] has shown a correlation between dependence coverage and code coverage, meaning that by using inputs that can exercise all branches in the code one can generally get most if not all possible dependences.

4.4.2 Amount of parallelism discovered

Figure 4.10(a) compares (on a logarithmic scale) the amount of parallelism found by Cilk's timing infrastructure (averaged over 60 runs) to that found by Embla 2 on their serial elisions. For this comparison we use our baseline model, with aggregated data dependences, no loop parallelisation or spawn hoisting—the closest model we have to Cilk's. The graph shows that Embla 2 is able to find all of the task-level parallelism in most of the original Cilk programs. One notable exception is magic, and this is because the Cilk program uses an implicit *inlet*[7] at the statement `count += spawn execute(...);`, which is not a feature of our baseline model. Later we show how, by extending our model with loop-level parallelism and reduction recognition, Embla 2 also discovers the parallelism here.

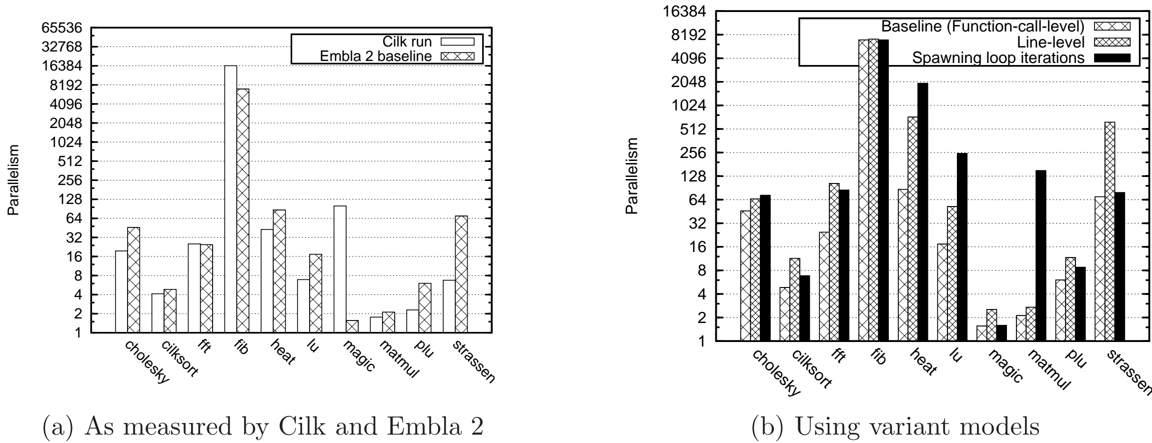
For a few examples Embla 2 can discover more parallelism than explicitly specified in the original Cilk program. We found several functions called synchronously that could have been spawned, as well as C library function calls that can be spawned with the addition of simple wrappers.

4.4.3 Effects of optimisations

We now look more closely at the variant optimisation models described in Section 4.2 to see whether they affect potential parallelism in these programs. As it is impractical to explore all possible combinations of parameters we only present here the results of using models that differ from the baseline for one parameter, while acknowledging that these parameters are by no means orthogonal.

Exact data dependences In most programs in the Cilk test suite the difference between this model and the baseline is negligible. This suggests that for such programs most

[7] In Cilk an *inlet* is a block of code that is executed atomically after a task completes.



(a) As measured by Cilk and Embla 2

(b) Using variant models

Figure 4.10: Parallelism of Cilk programs (note logarithmic scale)

of the potential parallelism is statically achievable—run-time techniques such as thread-level speculation (TLS) would give few performance benefits. One exception is lu, which gives an almost-fourfold speed-up over the baseline. This is because in one nested loop the dependence for a certain task on itself exists only between iterations of the outer loop but not of the inner loop. With aggregated dependences this results in all instantiations of the task being serialised, while with exact dependences instantiations from the same iteration of the outer loop can still be parallelised. While TLS would address this issue, a cheaper method is to parallelise the inner loop—part of the speed-up seen in the third column for lu in Figure 4.10(b) can be attributed to this task.

Control dependences In all the programs in the Cilk test suite parallelism is virtually unchanged, compared to the baseline figures in Figure 4.10(a), when control dependences are ignored, meaning that for these programs control speculation does not have any effect on available parallelism. This is in contrast to the limits of parallelism from Chapter 3, where the removal of control dependences resulted in a gain of parallelism of an order of magnitude. One of the reasons for this difference is that even when control dependences are ignored, our model's restriction that a task is only spawned when control reaches its call site, as well as the enforcement for *nestedness*, are preventing us from seeing the gains reported in the last chapter. Furthermore, these programs mainly work on arrays and have little irregular control flow, meaning that control dependences are not so much of a hindrance here.

Loops Looking at the first and third columns of Figure 4.10(b), which compares the potential parallelism of Cilk programs without and with spawning loop iterations, we can see that the use of parallel for-loops benefits most of the programs considered here, especially matmul, which sees around a 64-fold increase in parallelism. This confirms the

view that the use of parallel for-loops is an excellent way to express task-level parallelism, in addition to spawning function calls. In fact, some parallel programming environments (e.g. Cilk++ [51]) optimise parallel for-loops by deriving loop induction variables by divide-and-conquer instead of linearly, meaning that actual parallelism may be even higher than these figures (as Embla 2 preserves linear increments of induction variables). Embla 2 not only finds the amount of loop-level parallelism in a program, but can be used to easily identify candidate loops for parallelisation. This can simply be done by searching through the dependences output by Embla 2 for dependences between iterations of the loop concerned. If there are no such dependences, then the loop can be parallelised.

Reduction operations When the dependences between reduction operations are discounted, the parallelism of magic vastly increases from under 2 to 376. This means that a parallel reduction mechanism, such as Cilk's implicit inlets, *hyperobjects* in Cilk++ and similar operations in OpenMP, is essential for the program's parallelism potential to be realised.

Spawn hoisting With these Cilk programs, we find that spawn hoisting has a negligible effect on potential parallelism, as parallelism measurements with spawn hoisting is almost the same as those for the baseline in Figure 4.10(a). This suggests that, perhaps unsurprisingly, most function calls in these Cilk example programs are already spawned at the earliest possible point, and little further hoisting is possible.

Line-level parallelism The amount of line-level parallelism in these programs is shown in the second columns of Figure 4.10(b). We can see that for most of these programs the amount of line-level parallelism is around twice or more that of function-call-level parallelism. This is mostly due to simple statements performing arithmetic operations inside a function call or loop that can run in parallel. Each of these operations takes a small number of cycles, which means that it is not viable for each of these to be spawned. As mentioned, some of this fine-grained (mostly instruction-level) parallelism is realised already in existing superscalar processors. Nevertheless, operations may be grouped and extracted into tasks that are sufficiently large to see performance gains.

4.4.4 Parallelism in other benchmarks

We also ran the same analysis using Embla 2 on some of the benchmark programs in the SPEC CPU 2000 [37] (with the MinneSPEC reduced data input set [44]) and MiBench [34] suites, the results of which are displayed in Table 4.3. As before, the baseline model uses aggregated data dependences, and considers only function-call-level parallelism without

	Baseline	Exact data deps	Ignoring control deps	Spawning all loops	Spawning parallel loops	Spawn hoisting	Spawn hoisting and ignoring control deps	Line-level	Everything
SPECint 2000									
164.gzip	2%	2%	2%	34%	28%	2%	3%	61%	113%
175.vpr_place	4%	4%	4%	10%	5%	5%	12%	227%	373%
175.vpr_route	35%	35%	35%	27%	43%	43%	50%	162%	234%
176.gcc	23%	35%	31%	29%	33%	29%	54%	165%	391%
181.mcf	116%	118%	150%	41%	120%	116%	163%	517%	1519%
186.crafty	9%	9%	14%	18%	20%	10%	29%	222%	496%
197.parser	8%	9%	14%	9%	8%	11%	245%	31%	310%
252.eon_cook	122%	136%	179%	80%	133%	153%	235%	197%	343%
252.eon_kajiya	110%	125%	191%	82%	122%	141%	260%	178%	364%
252.eon_rushmeier	128%	155%	197%	84%	143%	162%	259%	204%	404%
253.perlbmk	1%	1%	1%	1%	1%	4%	4%	63%	74%
254.gap	6%	7%	7%	10%	10%	10%	15%	59%	106%
255.vortex	20%	23%	20%	19%	19%	31%	33%	104%	115%
300.twolf	10%	16%	11%	30%	16%	12%	35%	131%	237%
SPECfp 2000									
171.swim	15%	15%	15%	1882%	1486%	15%	15%	1722%	5964%
172.mgrid	1%	1%	1%	6176%	5951%	1%	1%	3187%	7176%
173.applu	4%	4%	4%	572%	549%	7%	27%	237%	2205%
178.galgel	11%	11%	11%	328%	550%	14%	14%	174%	372%
179.art	0%	0%	0%	47%	36%	5%	5%	220%	772%
187.facerec	6%	6%	6%	243%	268%	6%	6%	91%	94%
188.ammp	45%	46%	63%	13%	42%	47%	67%	167%	328%
189.lucas	4%	4%	4%	50%	80%	4%	4%	324%	326%
191.fma3d	2%	2%	2%	3%	3%	3%	3%	5%	5%
200.sixtrack	0%	0%	0%	1%	1%	1%	1%	79%	79%
MiBench									
basicmath	15%	15%	15%	13%	15%	15%	15%	15%	15%
bitcount	5%	128%	5%	22%	5%	5%	5%	43%	258%
blowfish.decode	143%	143%	143%	3%	143%	143%	143%	170%	170%
blowfish.encode	145%	145%	145%	3%	145%	145%	145%	168%	168%
dijkstra	0%	0%	0%	0%	0%	1%	2%	4%	5%
FFT	4%	6%	4%	17%	9%	4%	4%	13%	19%
FFT.inverse	7%	9%	7%	34%	16%	7%	7%	24%	38%
gsm.decode	10%	10%	10%	10%	15%	10%	10%	38%	422%
gsm.encode	4%	4%	4%	16%	13%	4%	4%	437%	727%
jpeg.decode	63%	63%	63%	6%	64%	64%	64%	479%	698%
jpeg.encode	34%	36%	34%	85%	86%	35%	36%	499%	793%
mad	39%	40%	39%	52%	49%	44%	49%	390%	462%
patricia	16%	17%	16%	13%	16%	20%	26%	21%	26%
qsort	0%	0%	0%	0%	0%	1%	1%	2%	2%
sha	16%	16%	16%	16%	18%	16%	16%	36%	107%
stringsearch	85%	85%	86%	29%	102%	85%	87%	102%	102%
susan.corners	0%	0%	0%	1246%	1045%	0%	0%	1967%	2167%
susan.edges	0%	0%	0%	1148%	904%	0%	0%	2271%	3108%
susan.smoothing	0%	0%	0%	19%	0%	0%	0%	127%	39530%
tiff2bw	4%	4%	4%	11%	4%	4%	4%	49%	49%
tiff2rgba	0%	7%	0%	0%	0%	1%	1%	1%	7%

Table 4.3: Parallelism of benchmark programs, expressed in terms of percentage speed-up

loop parallelisation or spawn hoisting. Most models are self-explanatory. The Everything model in the final column considers all possible optimisations—line-level parallelism, exact data dependences and ignoring control dependences[8]. From Table 4.3 we make the following observations:

- In general, we see that few benchmarks exhibit the level of parallelism seen in the Cilk examples. In fact, none of these benchmarks exhibit parallelism of over 3 (i.e. over 200%) under the baseline model, suggesting that spawning existing function calls alone is insufficient to effectively parallelise them.

[8]Spawn hoisting and spawning loop iterations are subsumed by the line-level model.

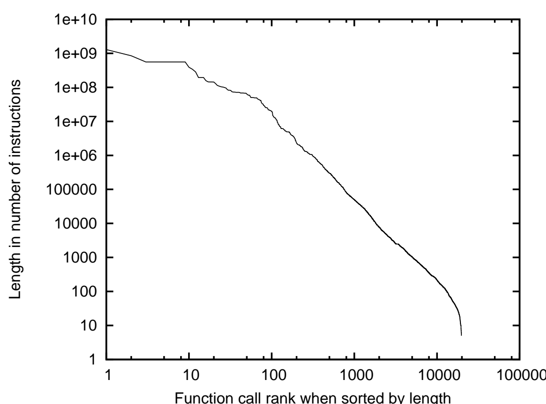


Figure 4.11: Mean run-time lengths of function calls

- There are, however, some benchmarks with a significant amount of loop-level parallelism, such as those from the SPEC floating point benchmark suite. The susan benchmarks from MiBench also exhibit lots of loop-level parallelism. These programs perform image smoothing, corner detection and edge detection on an image, and are data-parallel—the same computation is performed on each pixel in the image and the results for each pixel are independent of each other. This is reflected in our results, which show that using the small inputs, both susan.corners and susan.edges have potential parallelism of over 12 when loop iterations are spawned. This is not the case for susan.smoothing, however, as we will explain in Chapter 6.

- For some programs, e.g. 252.eon, spawning loop iterations actually results in a lower level of parallelism than the baseline. As explained earlier, this is because the baseline allows for partial overlap between loop iterations (a.k.a. software pipelining) whereas spawning loop iterations is an all or nothing proposition; the iterations are either completely independent or completely serial.

- Most programs do exhibit a significant amount of line-level parallelism, showing that function-level and loop-level parallelism are not the only forms of parallelism in these programs. Whether any of this extra parallelism that is excluded by our other models is actually exploitable on multi-core processors requires further investigation.

4.4.5 Granularity analysis

We now show the results of our granularity analysis, beginning with the mean lengths of all function calls in our benchmark programs. Figure 4.11 shows the sorted mean lengths of function calls. The graph appears to show three straight line regions, starting off with a gentle slope, getting steeper around 100, and then dropping abruptly after 10000. One straight line would suggest a power law of $ln^k = constant$, with n being the rank of the

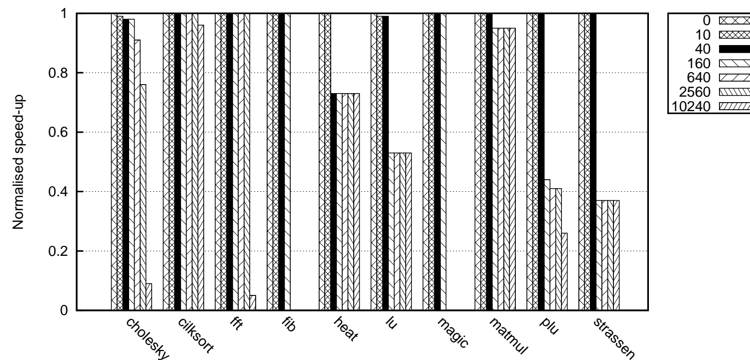


Figure 4.12: Sensitivity to granularity of speed-up for Cilk benchmarks

function call when sorted by length, l its length averaged over program execution, and k the scaling constant. The slightly convex shape suggests that there are more big functions and fewer small functions than we would expect from a power law. But this can in fact be explained by arguing that programmers do not tend to write functions unless they do a significant amount of work—single-line functions should be rare as programmers would tend to inline them[9]. Nevertheless, a trend line on the graph shows k, the scaling constant, to be around 2.5.

We turn to the two methods of calculating mean continuation length. Over all the benchmark programs we find that in around 10% of cases the difference between the expected continuation length calculated probabilistically and one profiled by Embla 2 is above 100 instructions; the difference is over 1000 instructions in just over 5% of function calls. This suggests that our probabilistic model is accurate in most cases, although in a small number of cases the difference can be large.

To see the effects of limiting spawnable tasks to those with mean granularity—defined as the smaller of mean task length and mean continuation length—above a certain threshold, we have extended Embla to read in the results of granularity analysis. The threshold is configured with a run-time option, and the speed-ups, normalised to the speed-up for a threshold of 0 (i.e. no tasks are filtered), are seen in Figures 4.12 to 4.15. These figures use the profiled continuation lengths, but the figures for probabilistically derived continuation lengths are very similar.

From these graphs we see varied results. While for a small number of programs parallelism is not affected at all, most programs are sensitive to granularity filtering. For some programs parallelism collapses quickly when the threshold is raised, while for others the drop is much more gradual. Overall, from these graphs we do see that many tasks are fine-grained, meaning that low task-spawning overheads are essential. It is also

[9]Accessor methods in object-oriented languages are a notable exception, but most of our benchmarks are non-OO imperative programs.

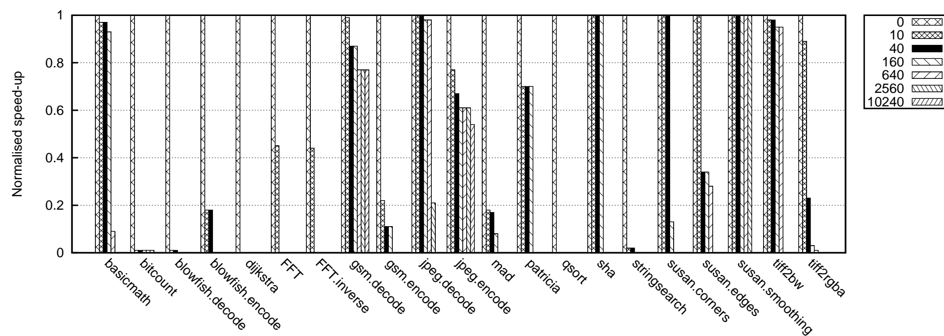


Figure 4.13: Sensitivity to granularity of speed-up for MiBench programs

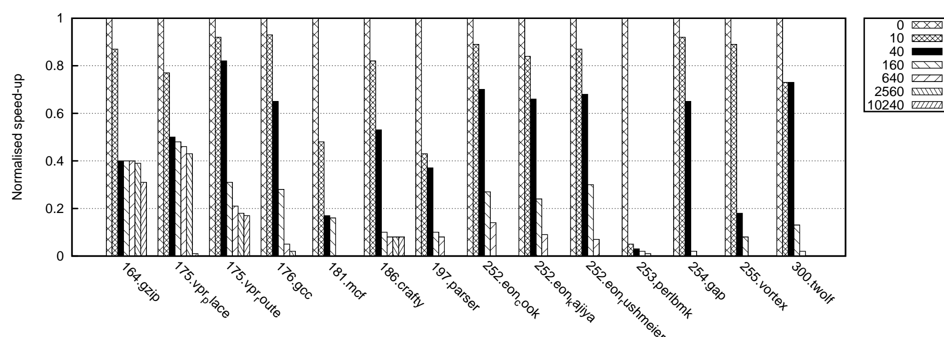


Figure 4.14: Sensitivity to granularity of speed-up for SPEC 2000 integer benchmarks

interesting to see the differences between different types of programs. Parallelism for programs considered more regular or array-based—Cilk example programs, the SPEC floating point benchmarks and some of the MiBench benchmarks—drops off at a higher granularity threshold than more irregular or pointer-based programs—those in the SPEC integer benchmarks and the rest of the MiBench suite. This suggests that the main tasks of regular programs are generally coarser-grained than those of irregular programs.

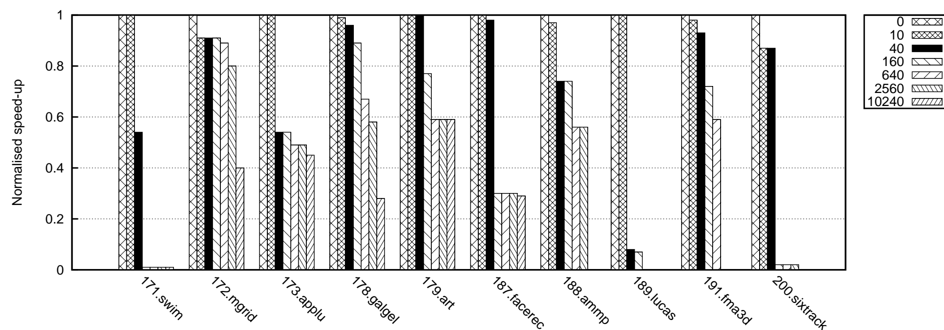


Figure 4.15: Sensitivity to granularity of speed-up for SPEC 2000 floating point benchmarks

4.5 Related work

Various languages and libraries have been created to allow easy expression of task-level parallelism. Cilk(++)'s [11, 51] model of fork/join parallelism matches our function-spawning model most closely, although a Cilk `sync` always joins with all tasks spawned in the current procedure activation rather than being able to pick a specific task to join as in our model. While OpenMP [21] originally focused almost exclusively on loops, version 3.0 adds support for function-level parallelism. Other notable examples offering similar functionality include Java's concurrency library [49], Intel's TBB [78] and Microsoft's TPL [50]. SMPSs [69] uses a related model where the programmer specifies dependences rather than synchronisation points.

Several studies have been done on limits of instruction-level parallelism (e.g. [99]), but fewer have tried to separate out task-level parallelism from instruction-level parallelism. Kreaseck et al. [45] explored limits of speculative task-level parallelism by executing function calls early, similar to the way we hoist spawns. They have however imposed the restriction that spawned function calls must be joined at their original call sites, which is a restriction we have felt to be unnecessary, and thus have not imposed in our analysis[10]. In our model function calls can be joined as late as dependences allow (but always before the parent call returns). Other studies [100, 66] have shown that data-value prediction, especially in regard to return values, is effective at increasing task-level parallelism. This is something we wish to consider further in the future.

Embla's source-level profiling algorithm [26] has been used elsewhere to discover dependences and assist programmers with parallelisation. Most systems [103, 93, 48] are concerned with the parallelisation of loops only. The Alchemist tool [104] uses it to select good task candidates for thread-level speculation. Nguyen et al. [64] use it to detect function-level parallelism, but uses a more restrictive parallelisation model that is based on Scheme. We believe Embla 2 is the first that can assist in parallelising function calls as well as loops, while providing a realistic estimate of potential speed-up.

There are parallels between Embla 2 and on-the-fly data race detection [57, 81]. Both use instrumentation to infer properties of the program through dynamic analysis. The main difference is that while race detection seeks to find unsafe parallelism (i.e. bugs) in an explicitly multi-threaded program, Embla 2 seeks to find potential safe parallelism in a sequentially written program[11].

There has been much research into automatic parallelisation [42, 10], most of which uses

[10]However, it would be straightforward to model this by extending our "Spawn hoisting" model. We simply insert in the DDG an incoming edge to each node that *does not* contain a function call from the last executed node that *does* contain a function call.

[11]With the usual caveat that dynamic analysis can only give approximations as program execution is input-dependent. To guarantee safety a safety mechanism (e.g. thread-level speculation) is required.

only static analysis. However, the lack of precision when statically analysing dependences remains a barrier, something which dynamic analysis tools such as Embla 2 can address. The downside is that the resulting parallelisation may not be valid for runs not covered by the training set of input data. A hybrid approach would therefore be best.

Work by Aleen and Clark [3] finds commutative functions in user code by symbolically executing each function f twice using two randomly generated inputs, I_1 and I_2. Function f is said to be commutative if $f(I_1); f(I_2)$ and $f(I_2); f(I_1)$ result in memory layouts indistinguishable using any subsequent I/O. Consecutive calls to commutative functions can then be executed in parallel, but mutual exclusion locks may be required. By identifying commutative functions in user code, this approach helps to remove more unnecessary dependences, which can lead to greater parallelism being discovered. However, as with other static analyses this approach suffers from a lack of precision. Also, the costs of locks required to execute commutative functions in parallel may wipe out any potential performance gains from parallelisation.

Some of the speculative task-level parallelism uncovered by Embla 2 can be exploited with thread-level speculation (TLS) (e.g. [87]), although we have observed this is not needed for the Cilk test suite programs. One important factor affecting the performance of TLS is selecting good candidate threads for speculation, as frequent rollbacks would offset any gains in parallelism [54]. We believe that Embla 2 can be used to identify good candidates, as it allows us to look at the frequencies at which potential dependences materialise.

As to the use of Markov chains to model control flow, Ramalingam [75] used Markov chains constructed from Control Flow Graphs with edge probabilities to perform probabilistic data flow analysis. Mehofer and Scholz [56] extended this to make probabilities for each edge conditional on the previous edge taken. This same extension may be applied to our probabilistic model for estimating task granularity to improve its accuracy.

4.6 Conclusions

This chapter has presented Embla 2, a tool designed to aid program parallelisation by estimating and locating potential parallelism in programs as well as pinpointing bottlenecks. It works by profiling dependences and mapping them back to program source. The underlying model of parallelism treats each function call as a spawnable task, which is synchronised on as late as dependences allow. Variants of this model allow us to estimate the potential benefits of parallel for-loops and optimisations such as spawn hoisting, parallel reduction operations and thread-level data dependence and control speculation.

We have shown that Embla 2 is able to discover all of the plentiful declared parallelism in most example Cilk programs, and even to find parallelism in places not explicitly parallelised. Having run the same analyses over benchmark programs from the SPEC CPU

2000 and MiBench suites, we observed that most of them do not exhibit the same amount of task-level parallelism as the Cilk examples. This suggests that most general-purpose programs tend to have little potential parallelism which is exploitable by spawning function calls and loops. However, critical paths output by Embla 2 can help the programmer locate parallelism bottlenecks, which when refactored may greatly improve the parallelism of a program. This will be explored in Chapter 6.

Our model of nested function-level parallelism may not be the only practical task model. For instance, it is possible to remove the constraint that tasks must be properly nested, by allowing tasks to be synchronised anywhere within the program rather than only in the parent task. Loss of program modularity and clarity may result, which is why not many languages allow this. It would nevertheless be of interest to estimate how much parallelism is gained by removing this constraint.

One future enhancement to Embla 2 is the addition of thread-spawning overheads to our cost model. This would give us an even more realistic estimate of potential speed-up. Related to this is the ability to aggregate small threads into a bigger thread in order to save overheads.

Embla 2 forms the first major component of our interactive parallelisation framework. In the following chapters we describe the rest of the framework and demonstrate how they combine to help programmers parallelising legacy sequential applications using parallel programming environments like Cilk.

Chapter 5

Completing the production line—automating task-level parallelisation

Following on from the dependence profiling that Embla 2 provides, our next step is to derive a parallelisation which, in an ideal world, would have the same speed-up as the parallelism estimate Embla 2 gives. By construction the Nested Function-level Fork-join Parallelism model that Embla 2 uses is very similar to the task models of programming languages like Cilk(++)[11, 51], TBB[78] and others. This chapter looks at how sequential programs can be transformed into parallel programs in these languages using profiled dependences. We introduce the second component of our interactive parallelisation tool-chain, named Woolifier. Woolifier implements algorithms to convert sequential C programs into Wool (Section 5.1) and Cilk/Cilk++ (Section 5.2), along with useful extensions (Sections 5.3 and 5.4). We evaluate our Woolifer implementation in Section 5.5.

5.1 Wool

Wool[25] is a macro-based task library for C designed by Karl-Filip Faxén at the Swedish Institute of Computer Science. Tasks are essentially spawnable functions, and are defined and used with the following macros.

- `TASK_`n`(`*return_type*`,` f`,` *arg₁_type*`,` *arg₁_name*`, ...,` *argₙ_type*`,` *argₙ_name*`)` creates the signature of a task named f. The body of the task definition is the same as for a normal function.

- `SPAWN(`f`,` arg_1`, ...,` arg_n`)` spawns a task instance of f. Code following the `SPAWN` can be run while the task is still being executed.

```
int fib(int n) {
  if (n<2) {
    return n;
  } else {
    int x,y;
    x = fib(n-1);
    y = fib(n-2);
    return x+y;
  }
}
```

(a) Original function

```
TASK_1(int, fib, int, n) {
  if (n<2) {
    return n;
  } else {
    int x,y;
    SPAWN(fib, n-1);
    y = CALL(fib, n-2);
    x = SYNC(fib);
    return x+y;
  }
}
```

(b) In Wool

Figure 5.1: A sequential Fibonacci function and its parallelisation in Wool.

- `SYNC(`f`)` joins on the most recently spawned task, which must be an instance of f (Parameter f is redundant in naïve semantics but enables more optimisation). Code following the `SYNC` cannot be run until execution of the task has been completed. `SYNC` returns the (possibly void) value returned by the task.

- `CALL(`f`,` arg_1`,` ...`,` arg_n`)` calls task f synchronously just as in sequential code. It returns only when execution of the task has been completed. It is semantically equivalent to a `SPAWN` followed immediately by a `SYNC` but is slightly faster in the implementation. The use of `CALL` is sometimes known as *task inlining*, as the practice of executing a task in the same thread to avoid spawning overheads is analogous to the practice of inlining a function to avoid function-call overheads.

- `LOOP_BODY_`n`(`*loop_name*`,` *body_size*`,` *idx_type*`,` *idx_name*`,` *arg₁_type*`,` *arg₁_name*`,` ...`,` *argₙ_type*`,` *argₙ_name*`)` defines the loop body of a parallel for-loop (much like how `TASK_`n defines a task), indexed by the variable *idx_name*. Loop bodies must be defined like stand-alone functions in Wool. *body_size* is provided to Wool as an indicator of the likely granularity of each loop iteration, and is used to derive the value of the *chunk_size* parameter in Algorithm 1.

- `FOR(`*loop_name*`,` idx_{begin}`,` idx_{end}`,` arg_1`,` ...`,` arg_n`)` executes a parallel for-loop of the given `LOOP_BODY_`n (just as `SPAWN` spawns a task instance), with one iteration for each integral value of *idx* in the range $[idx_{begin}, idx_{end})$.

Figure 5.1 shows a simple example of a Fibonacci program that uses Wool macros.

We can see that Wool's task model closely matches that of Embla 2. Similar to Embla 2's baseline model, only function calls can be spawned. As in Embla 2, Wool enforces nested[1] parallelism—all tasks spawned by each function must be synchronised on before the function may return, or else errors may result. Wool also allows the spawning of loop bodies for parallel for-loops, mirroring Embla 2's "Spawning loop iterations" model. In addition, both Embla 2 and Wool assume no inter-task communication, except at task spawning and joining[2].

Nevertheless, there are minor differences which must be taken into account when *Woolifying* a program with Embla 2's results.

The first is the restrictive nature of `SYNC`, which can only synchronise on the last spawned task. This means that `SYNC`s must be ordered so that tasks are synchronised on in a last-in-first-out order (in addition to the above requirement for all tasks spawned by each function to be synchronised on before the function may return), and that at run-time there must be as many `SYNC`s as there are `SPAWN`s. While such a design lends itself to efficient implementation, it could be difficult to program, as the number of spawns at run-time may be variable, and care must be taken to ensure that the same number of `SYNC`s are called to avoid undefined behaviour. Embla 2, on the other hand, allows spawned tasks to be synchronised on in any order.

Thus to accommodate Embla 2's more flexible model, we extended Wool to allow arbitrary synchronisation orders. This is done by introducing the concept of *handles*, which are simply linked lists of spawned tasks. A handle is usually defined at the beginning of a function, and every `SPAWN` must now be associated with a handle. `SYNC`s now also require a handle, and simply traverses the handle's linked list of spawned tasks, synchronising on each of them in turn. As a result, each `SYNC` may synchronise on zero, one or multiple tasks. A necessary consequence of this is that `SYNC`s no longer return the result of the task. Instead, destinations must be specified when spawning non-void tasks. Figure 5.2 shows some example usage of handles.

Increased flexibility might come at a cost to performance. The original Wool program uses a simple LIFO task queue—`SPAWN`s push tasks on to the queue and `SYNC`s pop from it. In our extension, synchronisation is no longer guaranteed to be on the task at the top of the queue and so can no longer pop from it. Instead, space on the task queue can only be reclaimed when a task returns—the programmer is required to ensure that all tasks spawned by a function have been synchronised on before returning. However, empirical evidence shows that the performance cost is negligible—for Woolifications of serial elisions

[1]With respect to the dynamic call tree, the execution of a task must be *nested* within the execution of its parent.

[2]One can always implement inter-task communication in Wool with concurrent data structures, but this is not expected to play a big part in most Wool programs, where tasks are generally expected to be isolated.

```
TASK_1 (long, primal_net_simplex, network_t *, net) {
  HANDLE (handle_1);
  HANDLE (handle_0);
  ...
  while (!(opt != 0L)) {
      SYNC (update_tree, handle_1);
      if (bea = CALL (primal_bea_mpp, m, arcs, stop_arcs, &red_cost_of_bea)) {
          ...
          iminus = CALL (primal_iminus, &delta, &xchange, iplus, jplus, &w);
          if (!iminus) {
              ...
              if (delta != 0L)
                VOID_SPAWN (primal_update_flow, handle_0, iplus, jplus, w);
          } else {
              ...
              VOID_SPAWN (update_tree, handle_1, !xchange,
                          new_orientation, delta, new_flow, iplus, jplus,
                          iminus, jminus, w, bea, red_cost_of_bea,
                          ((flow_t) (net->feas_tol)));
              ...
              if (!((*iterations - 1) % 20)) {
                  SYNC (update_tree, handle_1);
                  *checksum += CALL (refresh_potential, net);
                  ...
              }
          }
      } else {
          opt = (1);
      }
    }
  SYNC (update_tree, handle_1);
  *checksum += CALL (refresh_potential, net);
  CALL (primal_feasible, net);
  CALL (dual_feasible, net);
  SYNC (primal_update_flow, handle_0);
  return 0;
}
```

Figure 5.2: Extract from an example *Woolified* function from 181.mcf from SPECint 2000, illustrating the use of handles (highlighted in bold)

of Cilk's example programs, the difference in performance between our extended version of Wool and the original is insignificant.

Another minor difference is with the restrictions Wool places on parallel for-loops, in that it must be a natural loop with an incrementable loop index (integer or pointer) and a range that is known at the start of the loop. This is required due to its divide-and-conquer implementation (see Algorithm 1), which is more efficient generally than sequentially spawning each iteration.

Algorithm 1 Wool's Divide-and-conquer Algorithm for executing parallel for-loops

procedure PARALLEL_FORALL($loopbody, start, finish, chunk_size$)
  if $finish - start \leq chunk_size$ **then**
    for $i = start$ to $finish - 1$ **do**
      $loopbody(i)$
    end for
  else
    $mid \leftarrow start + \lfloor \frac{finish-start}{2} \rfloor$
    spawn PARALLEL_FORALL($loopbody, mid, finish, chunk_size$)
    call PARALLEL_FORALL($loopbody, start, mid, chunk_size$)
    sync
  end if
end procedure

This restriction, however, necessarily precludes parallel loops where the range is unknown at the start of the loop. A classic example is a loop that updates every element in a linked list. Embla 2's model allows this kind of DOALL parallelism, but such a loop cannot be implemented with Wool's parallel for-all loops. Our initial solution is to revert to spawning the loop body for every iteration. The drawback of this approach is firstly that spawns and joins are sequentialised, leading to $O(n)$ overhead time requirement rather than $O(log(n))$ (for unlimited processors). Secondly, it is more difficult to coalesce consecutive loop iterations for fine-grained loop bodies, meaning that if the loop body is small parallelising it could degrade rather than improve performance. Later (Section 5.4) we describe how granularity analysis can be used to filter out such loops.

Finally neither Wool nor Cilk allows library functions to be spawned directly. However, this can be easily solved by providing a task library of spawnable wrappers. Library functions that hold internal data structures though may need to be modified to ensure thread safety (e.g. with a global lock), if they are not already thread-safe.

Once these differences have been resolved, how can a C program be transformed into a Wool program, or *Woolified*? The algorithms for transforming function calls and loops are described in Algorithms 2 and 3 respectively. The first algorithm utilises a subroutine

Algorithm 2 Algorithm for Woolifying function calls

```
procedure INSERT_SYNCS(line, spawn)
    for all unvisited successors of line, s do
        if s is either dependent on spawn or is an exit then
            add SYNC(spawn) before s if one is not already there
        else
            INSERT_SYNCS(s, spawn)
        end if
    end for
end procedure
procedure SPAWN_TASKS(program)
    for all function calls in program, c do
        if all successors of c are either dependent on c or exits then
            replace c with synchronous call, CALL(c)
        else
            replace c with asynchronous call, SPAWN(c)
            INSERT_SYNCS(c, c)
        end if
    end for
end procedure
```

Algorithm 3 Algorithm for Woolifying loops

```
procedure PARALLELISE_LOOPS(program)
    for all loops identified in program, loop do
        guard ←guard of loop
        body ←body of loop
        if guard is not dependent on body and body is not dependent on itself then
            if loop is canonical then
                outline body into LOOP_BODY_n construct
                replace site of entire loop with FOR construct
            else
                outline body into a TASK, f
                replace site of body with an asynchronous call, SPAWN(f)
                add SYNC(f) after the loop
            end if
        end if
    end for
end procedure
```

for placing joins at a *synchronisation frontier*, the set of latest points in the program where the joins must occur in order for dependences to be respected. Formally, consider the Control Flow Graph for a function, $CFG = (V, E_c)$, where each node $v \in V$ is a source line in the function, and $E_c \subseteq V \times V$ represents the possible flows of control in the function. The successor function $succ : V \to \mathcal{P}(V)$ is defined as $v \in succ(u)$ iff $(u, v) \in E_c$. Consider also a dependence graph $DG = (V, E_d)$, where $(u, v) \in E_d$ iff v depends on u. Then a synchronisation frontier of a node u is defined as the set $SF(u) \subseteq V$ such that $v \in SF(u)$ iff (i) either $(u, v) \in E_d$ or v is an exit (i.e. $succ(v) = \emptyset$), and (ii) there exists a path P from u to v in CFG such that for all nodes w in P except u and v, $(u, w) \notin E_d$.

In practice some details need to be considered when inserting `SYNC`s at the correct place, as we are working on the line-by-line level. For instance when the line before which we need to insert a `SYNC` is a loop header, we need to consider whether the `SPAWN` is before or inside the loop. If it is before the loop, then the `SYNC` should be placed before the entire loop; if it is inside the loop, then the `SYNC` should be placed inside the loop body.

Algorithm 3 describes how to Woolify a loop. The main requirement for a parallel loop is that there is no inter-iteration (or *loop-carried*) dependence. The loop header (the *guard*) can be dependent on itself, and the loop body can be dependent on the guard, but the guard cannot be dependent on the loop body. However, as mentioned, we need to check if the loop is *canonical*, i.e. of the form:

```
for (i=...  ; i < ⟨loop-invariant expression⟩; i++) {...}
```

This ensures that the loop satisfies the requirement for Wool to run it as a DOALL loop. Otherwise we must spawn each iteration individually.

Note that while our implementation uses profiled dependences, the algorithms do not depend on how the dependences are derived. Thus, the dependences could equally have been statically determined. As we have said earlier, while static analysis generally gives a safe estimate of all potential dependences, it tends to be too conservative and deduce dependences that rarely or never arise in practice. Embla 2 on the other hand finds dependences dynamically, but dependences found may not include all dependences that could arise in any execution. In such a case the programmer should check that the parallelisations are indeed safe, or else a safety net such as a run-time dependence check and rollback mechanism is required, such as the ones used in thread-level speculation schemes [80, 88].

5.2 Cilk and Cilk++

Cilk [11] is a language extension of C that is very much similar to Wool in its task model. The main differences are mostly syntactical: a task is defined by placing the `cilk`

```
cilk int fib(int n) {
  if (n<2) {
    return n;
  } else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return x+y;
  }
}
```

(a) Cilk

```
int fib(int n) {
  if (n<2) {
    return n;
  } else {
    int x,y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x+y;
  }
}
```

(b) Cilk++

Figure 5.3: Parallelisation of Fibonacci function in Cilk and Cilk++.

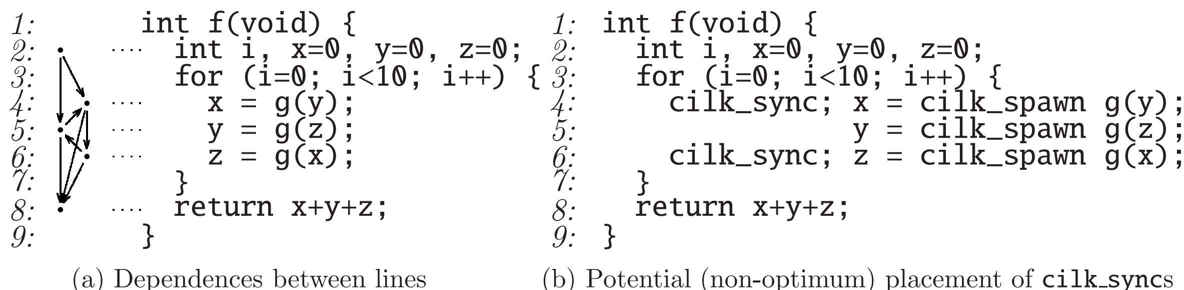
keyword in front of a function definition, and `SPAWN(`f`, `arg_1`, …, `arg_n`)` becomes `spawn` f`(`arg_1`, …, `arg_n`)`. All tasks must be `spawn`ed—even if they must be synchronised on immediately after. Cilk does not have primitives for DOALL loops.

Cilk++ [51] is a commercialisation of Cilk by Leiserson which is now owned by Intel. Apart from minor syntactic changes, and the fact that tasks do not now have to be spawned, Cilk++ introduces support for DOALL loops, as well as hyperobjects, which we will discuss later. Figure 5.3 shows example programs in Cilk and Cilk++.

One of the more important differences between Wool and Cilk/Cilk++ is in how tasks are synchronised. In Cilk/Cilk++ a `sync` will synchronise on all tasks spawned by the current task. The advantages of such a design decision are its simplicity for programmers and that it lends itself well to an efficient implementation. In terms of expressiveness, however, it does not give the programmer control over synchronising on individual tasks that Wool gives. There is also an implicit `sync` before each function return.

The insertion of synchronisation points must therefore now take into account all tasks spawned in the same function. As a result, a synchronisation point placed at one spawn's synchronisation necessarily but prematurely synchronises on all other tasks that have yet to be synchronised. This on the other hand might make insertion of synchronisation points later on unnecessary. To find the optimal synchronisation points, given spawn points[3], we define recursively the set of open spawn points—lines at which tasks may have been spawned that may not yet be synchronised on:

[3]Not spawning certain tasks might allow other tasks to be synchronised later and lead to better performance. This is however beyond the scope of the thesis.



(a) Dependences between lines (b) Potential (non-optimum) placement of `cilk_sync`s

Figure 5.4: Example of a program where non-greedy Cilkification might not terminate

$$open_spawns(n) = \begin{cases} is_spawn(n) & \text{if } n \text{ is preceded by a } \texttt{cilk_sync} \\ is_spawn(n) \cup \bigcup\limits_{p \in pred(n)} open_spawns(p) & \text{otherwise} \end{cases}$$

$$is_spawn(n) = \begin{cases} \{n\} & \text{if } n \text{ contains a } \texttt{cilk_spawn} \\ \emptyset & \text{otherwise} \end{cases}$$

The new algorithm, which uses a forward data-flow analysis to insert synchronisations, is described in Algorithm 4. It is an iterative algorithm, but is greedy in the sense that once a `cilk_sync` is inserted it is not removed later even if rendered redundant by another `cilk_sync`. One could modify the algorithm to permit the removal of `cilk_sync`s if they become redundant. However, doing this naïvely can cause non-termination in the Cilkification algorithm. Consider the program in Figure 5.4(a). Each of the calls to `g` can be spawned, but there is no *optimum* correct placement of `cilk_sync`s, in that wherever the `cilk_sync`s are placed, there will be one before a line n that does not depend on any of the lines in $\bigcup_{p \in pred(n)} open_spawns(p)$ (such as the `cilk_sync` on line 4 in Figure 5.4(b)). A Cilkification algorithm that permits the removal of `cilk_sync`s when they become sub-optimal may not terminate in this case, unless a mechanism is included to detect infinite looping and stop the algorithm in pathological cases.

5.3 Reduction operations

We now look at how we can extend our Woolification algorithm to reflect optimisation models described in the last chapter, beginning with reduction operations. Recall that a reduction variable is one that acts as an accumulator, i.e. its value can only be used as an argument to an associative operator, the result of which is written to the accumulator

Algorithm 4 Algorithm for Cilkifying function calls

```
procedure INSERT_SYNCS(spawns)
    for all lines in function, line do
        open_spawns[line] ← ∅
    end for
    syncs ← ∅
    worklist ← spawns
    while worklist is non-empty do
        line ← remove top of worklist
        if line ∉ syncs then
            if line is dependent on any line in ⋃_{p∈pred(line)} open_spawns[p] then
                insert a cilk_sync before line
                syncs ← syncs ∪ {line}
                open_spawns[line] ← is_spawn(line)
            else
                open_spawns[line] ← is_spawn(line) ∪ ⋃_{p∈pred(line)} open_spawns[p]
            end if
            if open_spawns[line] has changed then
                worklist ← worklist ∪ succ(line)
            end if
        end if
    end while
    for all lines in syncs, line do
        add cilk_sync before line
    end for
end procedure
procedure SPAWN_TASKS(program)
    spawns ← ∅
    for all function calls in program, c do
        if not all successors of c are either dependent on c or exits then
            replace c with asynchronous call, cilk_spawn c
            spawns ← spawns ∪ {c}
        end if
    end for
    INSERT_SYNCS(spawns)
end procedure
```

itself. More formally, for a variable x to qualify as a reduction variable there must exist a monoid $(T, \oplus, e)$, where T is a set, $\oplus$ is an associative binary operation $\oplus : T \times T \rightarrow T$, and e is an identity element with respect to $\oplus$. $x \in T$ and is only used in this form: $x \leftarrow x \oplus E$, where E is an expression not containing x. In our implementation T is typically the set of possible values for numerical types, $\oplus$ is addition, subtraction, multiplication or a primitive bitwise or logical operator, or the *assignment* reduction operation defined as $x \oplus y \equiv y$.

In fact, the assignment reduction operation on its own, however, does not have an identity element e such that $\forall x. x \oplus e = x$. To address this we must alter the monoid to $(T \cup \{\bot\}, \oplus, \bot)$, where $\oplus$ is now defined as:

$$x \oplus y = \begin{cases} x & \text{if } y = \bot \\ y & \text{otherwise} \end{cases}$$

If the only dependences between iterations of a loop are on reduction variables, then the loop can still be parallelised given a special mechanism to safely accumulate these reduction variables. This is actually quite straightforward given the divide-and-conquer approach used by Cilk++ and Wool to execute DOALL loops. When we split the range into half, we pass the current value of the reduction variable into the left child and pass the identity value into the right child. Each child then updates its own copy and returns the final value. When we join the two children, we simply apply the reduction operation to the two children's final values and return the result.

Cilk++ provides this functionality in the form of reducer *hyperobjects* [29]—C++ classes with overloaded operators that allow them to be used as normal variables but at the same time guarantee correct parallel execution. These reducers work across task spawns and synchronisations, not just parallel for-loops. For the sake of simplicity, however, in Wool we introduce reduction variables for use with for-loops only.

Here we describe how we extended Wool to support the parallelisation of reduction operations in a way similar to Cilk++'s hyperobjects. As Wool is based on C macros rather than C++, we did not have overloading or template functionality available to us, and thus the syntax we introduced was more clumsy, but the underlying mechanism is similar. Reducer definitions are introduced with a `REDTYPEDEF_`*n* macro, which defines a new `struct` type containing all the reduction variables a loop contains, as well as identity and reduction functions for this new type. Algorithm 1 is now extended to deal with reducers in Algorithm 5. The programmer must also copy initial values into the reducer before the loop and copy the final values back out afterwards, using the `COPYIN` and `COPYOUT` macros respectively.

After this was implemented in Wool, we extended our Woolification algorithm to allow parallelisation of for-loops if the only loop-carried dependences are on reduction variables.

```
for( arc = net->arcs; arc != (arc_t *)stop; arc++ ) {
    if( arc->flow ) {
        if( !(arc->tail->number < 0 && arc->head->number > 0) ) {
            if( !arc->tail->number ) {
                operational_cost += (arc->cost - net->bigM);
                fleet++;
            } else
                operational_cost += arc->cost;
        }
    }
}
```

Figure 5.5: Loop with reduction variables from 181.mcf

```
REDTYPEDEF_2(redtype_14,
  long, fleet, PLUS,
  cost_t, operational_cost, PLUS);

REDLOOP_BODY_1(OUT_14, SMALL_BODY, redtype_14, arc_t*, arc, network_t*, net) {
  if (arc->flow) {
      if (!(arc->tail->number < 0 && arc->head->number > 0)) {
          if (!arc->tail->number) {
              DEREF(operational_cost) += (arc->cost - net->bigM);
              DEREF(fleet) += 1;
            } else {
              DEREF(operational_cost) += arc->cost;
            }
        }
    }
}
...
redtype_14 OUT_14_reducer;
COPYIN(redtype_14, &OUT_14_reducer, fleet, operational_cost);
FOR(OUT_14, &OUT_14_reducer, net->arcs, (arc_t *) stop, net);
COPYOUT(redtype_14, &OUT_14_reducer, &fleet, &operational_cost);
```

Figure 5.6: Parallelisation of Figure 5.5 in Wool

Algorithm 5 Extending Wool's parallel for-loops to deal with reductions (reducer is passed by reference)

procedure PAR_FORALL_WITH_REDUCE($loopbody, reducer, start, finish, chunk_size$)
  if $finish - start \leq chunk_size$ **then**
    for $i = start$ to $finish - 1$ **do**
      $loopbody(i, reducer)$
    end for
  else
    $reducer' \leftarrow$ identity element
    $mid \leftarrow start + \lfloor \frac{finish-start}{2} \rfloor$
    spawn PAR_FORALL_WITH_REDUCE($loopbody, reducer', mid, finish, chunk_size$)
    call PAR_FORALL_WITH_REDUCE($loopbody, reducer, start, mid, chunk_size$)
    sync
    $reducer \leftarrow$ REDUCE($reducer, reducer'$)
  end if
end procedure

As mentioned in Chapter 4, Embla 2 can mark out such dependences. Figures 5.6 and 5.7 show the parallelisation in Cilk++ and Wool with reducers (with manual renaming and formatting to improve readability) of an example loop from 181.mcf in the SPEC 2000 integer benchmarks, shown in Figure 5.5.

5.4 Granularity filtering

All implementations of task-parallel libraries or languages have overheads when spawning and synchronising on a task, and as a result tasks that are too fine-grained will cause performance to degrade rather than improve as speed-up due to parallel execution is eclipsed by overheads. To ensure optimal performance we therefore need to identify and inline fine-grained tasks, even though dependences might allow them to be spawned in our original Woolification/Cilkification algorithms. In the previous chapter we have seen how Embla 2's parallelism estimate changes with increasing spawn threshold. We now apply the same threshold to our Woolification algorithm—i.e. only statically spawn a task if the smaller of its mean length and mean continuation length (which we shall call *mean task size*) is above a certain threshold.

There are drawbacks to simply only using a granularity threshold. In particular we envisage two common scenarios:

1. Task sizes vary over the course of the program's execution. As it is not necessarily true that performance gain/loss is proportional to task size, the mean task size

```
inline void OUT_14(network_t *net,arc_t *arc,
  cilk::reducer_opadd<long>&fleet,
  cilk::reducer_opadd<cost_t> &operational_cost) {
  if (arc->flow) {
    if (!(arc->tail->number < 0 && arc->head->number > 0)) {
      if (!arc->tail->number) {
        operational_cost += (arc->cost - net->bigM);
        fleet += 1;
      } else {
        operational_cost += arc->cost;
      }
    }
  }
}
...
cilk::reducer_opadd<long> fleet_red;
cilk::reducer_opadd<cost_t> operational_cost_red;
cilk_for (arc_t *arc = (net->arcs); arc != ((arc_t *)stop); arc++) {
  OUT_14(net,arc,fleet_red,operational_cost_red);
}
operational_cost += operational_cost_red.get_value();
fleet += fleet_red.get_value();
```

Figure 5.7: Parallelisation of Figure 5.5 in Cilk++

may not be the best representative of whether the overall effect of spawning a task would be beneficial. A common case of wide task size variance is with recursive functions, e.g. implementations of divide-and-conquer algorithms. To truly decide the profitability of spawning a task it may be necessary to run the programs with and without inlining and compare run-times. A better alternative is to predict at run-time whether a certain instance of a task should be spawned or inlined. This argues for an extension of Wool in the form of `SPAWNIF(`f`, `*condition*`, `arg_1`, `..., arg_n`)` that only spawns if *condition* evaluates to `true`, and inlines otherwise. For quicksort, *condition* could be that the number of elements is greater than 100. The condition is provided by the programmer, but it would be straightforward to extend a profiler to provide assistance.

2. Task sizes may also vary for different inputs. If the training data set supplied to Embla 2 is too small, it might lead to certain tasks being deemed too fine-grained by our Woolification algorithm, when normal data sets would ensure that the tasks

```
#include "cilk.h"
#include <stdlib.h>

void f (int n)
{
  int i;
  int y;
  for (i = 0; i < n; i++) {
    y |= i;
  }
}

int cilk_main(int argc, char* argv[])
{
  int x = atoi(argv[1]);
  long long int r = 0x100000000 / x;
  int i;
  for (i=0; i<r; i++) {
    cilk_spawn f(x);
    f(x);
    cilk_sync;
  }
  return 0;
}
```

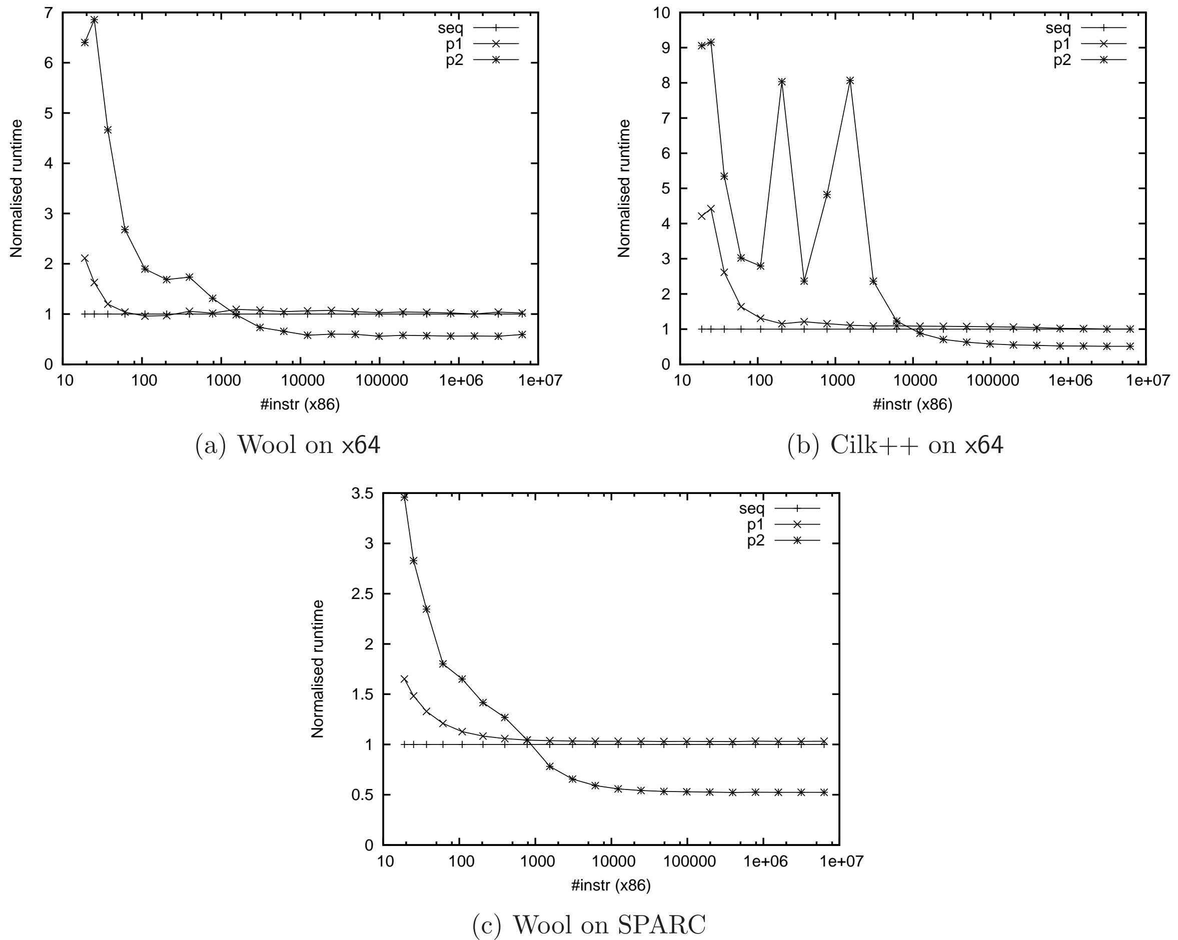
Figure 5.8: Benchmark program to determine suitable spawn threshold in Cilk++ syntax. The programs for Cilk and Wool are similar.

are sufficiently coarse-grained. Again the `SPAWNIF` construct could be useful in this instance.

Nevertheless, we believe that for most cases mean task size is a good indicator of potential performance gain/loss.

5.4.1 Getting the right threshold

To begin with, we needed to determine how big the threshold should be. If it is too small, then overheads of smaller tasks will hinder performance; if it is too large, then opportunities for parallelism may be unnecessarily lost. We developed a benchmark program, shown in Figure 5.8, that can help us with this. In this program, the function `f(n)` is a



(a) Wool on x64

(b) Cilk++ on x64

(c) Wool on SPARC

Figure 5.9: Normalised running times of benchmark program as task size changes

computation-intensive task, the length of which is set by a command line argument. This program repeatedly forks and joins two identical instances of `f(n)`. By contrasting the running times of this program when run on one and two processors we can find out the point at which it is more beneficial to run tasks in parallel.

Figure 5.9 shows the running times of our benchmark program on three different language-architecture combinations, all normalised to `seq`, the running times when the program is run sequentially without spawning any tasks. The `p1` line shows the running times when the program does spawn and synchronise on tasks but is run on a single processor; the `p2` line shows the running times when the program spawns and synchronises on tasks and is run on two processors. The `x64` (64-bit x86) machine contains an Intel Core 2 Quad processor clocked at 2.40GHz, running Linux kernel version 2.6.23. The `SPARC` machine is a Sun SPARC Enterprise T5140 Server with 128 hardware threads, running Solaris 10 8/07.

From these results we can draw a number of observations.

- For very small task sizes `p1` is significantly slower than `seq`, as the overheads of spawning a task dominate over the cost of executing the task body itself. For tasks larger than 100 x86 instructions the difference between the two becomes insignificant.
- For small task sizes `p2` is much slower than `p1`, showing that the overheads of stealing and later synchronising on a task are even greater than those of spawning.
- There are two humps on the `p2` curve for Cilk++ on `x64`, which is strange as it seems that increasing granularity may in fact degrade performance. We have not been able to examine the Cilk++ internals in detail for us to attribute this anomaly.
- The most important result to take from this is where the `p2` line crosses the `seq/p1` lines, as that is the point at which spawning a task becomes beneficial. For Wool on `x64`, that happens at around 1500 x86 instructions; for Wool on SPARC, it is around 800 x86 instructions. This suggests a lower communication-to-computation cost ratio for the SPARC. For Cilk++ on `x64`, however, the crossing point is much higher, suggesting much higher spawning and stealing overheads for Cilk++.

Our observations show that the thresholds are highly dependent on language implementation as well as architecture. It also demonstrates the importance of not spawning tasks that are too small, as they can seriously degrade performance.

5.4.2 Loop granularity

Recall that DOALL loops are implemented in Wool and Cilk++ with divide-and-conquer, as seen in Algorithm 1. This means that the *chunk_size* parameter can still be set appropriately to combine short loop iterations into more coarse-grained tasks. We have seen just now how to find a task threshold—the minimum mean number of instructions in a task for its parallelisation to be profitable. We can use this threshold to derive granularity simply by:

$$chunk_size = \frac{threshold}{loop_body_length}$$

This will ensure that the mean length of each task spawned for this loop is greater than the threshold.

5.5 Evaluation

We have implemented our Woolification and Cilkification algorithms in Woolifier, a source-to-source transformer based on the ROSE framework [82]. We now present the results of

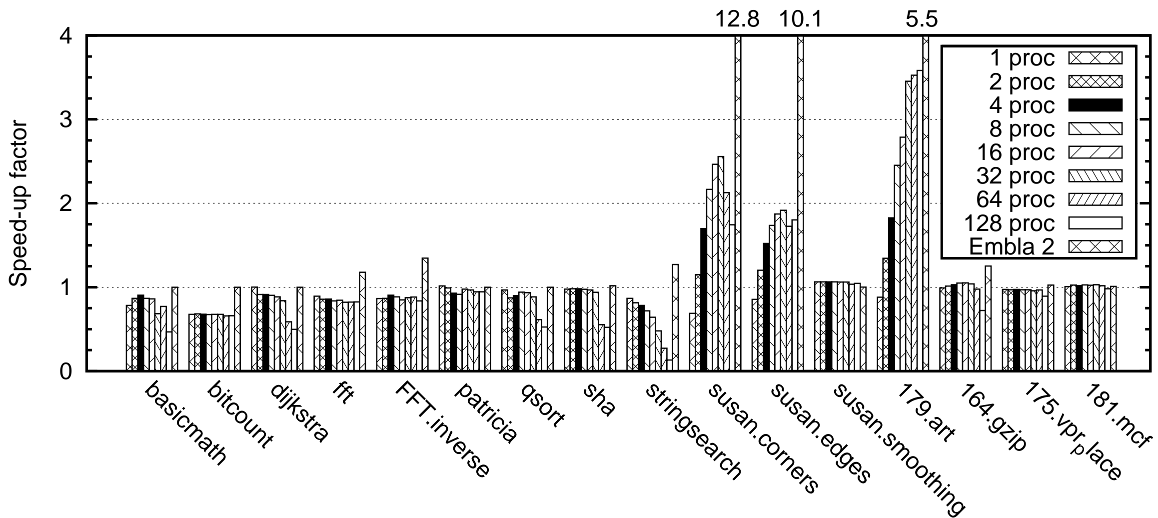


Figure 5.10: Actual speed-ups of Woolified programs from SPEC and MiBench compared with Embla 2's estimates

Woolifying some benchmark programs, which we compare with the parallelism estimates given by Embla 2. Each program was profiled with a *training* input, and the results were used by our Woolification algorithm to convert the C program into Wool, using Wool's support for parallel tasks and DOALL loops with reduction operations. The granularity filter was set at 1500 x86 instructions. Actual speed-ups were obtained by timing the sequential and parallel programs on a Sun SPARC Enterprise T5140 Server with 128 hardware threads, running Solaris 10 8/07. This machine consists of two UltraSPARC T2 processors, each with eight cores running on a clock speed of 1.2 GHz. Each core has two instruction pipelines, but supports eight threads, which time-share the two pipelines. This means that effectively there are around 32 cores[4]. To the operating system, however, the 128 threads appear as individual processors. There is a 24 KB level-1 cache for each core, and a 4 MB level-2 cache shared by all cores on a processor. Each speed-up figure is calculated using the mean execution time over three runs. The execution times have a standard deviation of around 5% on average.

From Figures 5.10 and 5.11, which show the actual speed-ups of various programs compared with Embla 2's estimates, we make the following observations:

- As expected, there is a correlation between Embla 2's estimates and actual speed-ups. The maximum speed-up factor was around 25 (achieved by `heat`), which is reasonable given the effective number of cores in the system is around 32.

- Not only so, but we notice also that speed-up factors fall after reaching a peak as the number of workers is increased. On closer inspection, we observe that the

[4]Nevertheless, if instructions from the different threads on the same core can be perfectly scheduled in between each other's memory stalls, then a speed-up of above 32 is possible.

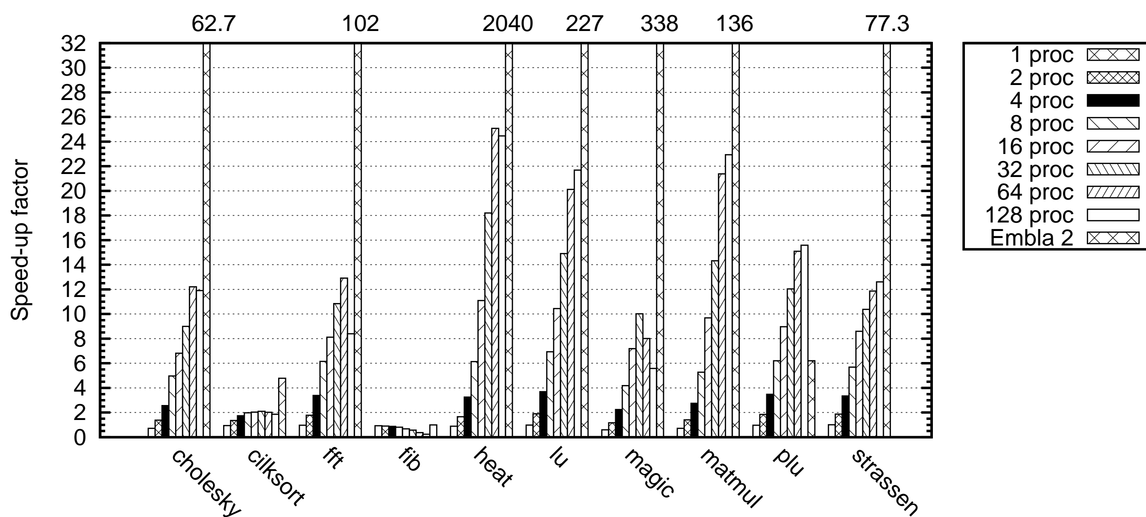


Figure 5.11: Actual speed-ups of Woolified programs from Cilk's example distribution compared with Embla 2's estimates

less parallelism there is in a program, the lower the number of workers at which performance peaks. We believe it is partly because of competition for execution time between threads on the same core. When there is little parallelism, only a few threads are doing useful work, while most others are busy looking for non-existent work to steal. These non-working threads take up valuable execution time on the core that could otherwise have been used by the working threads.

- Sometimes actual speed-up factors and Embla 2's parallelism estimates do not correlate as well as one would expect. This is because smaller inputs are used for Embla 2 than for timing Woolified programs. For programs such as plu, where the amount of parallelism increases with input size, figures for actual speed-up can exceed Embla 2's estimates. Conversely, for the susan benchmarks, the amount of parallelism actually falls as input size increases, which explains why the actual speed-up figures are not as high as Embla 2's estimates would suggest. A potential enhancement to our work here would run Embla 2 on a program using inputs of various sizes, in order to obtain a better prediction of actual speed-up for the program given an input of a certain size.

- For short-running programs where execution times were short (under one second), such as stringsearch and fib, performance vastly degrades as the number of processors is increased, as worker initialisation costs, which increase with the number of processors, dominate the execution times.

5.6 Future Work

5.6.1 Spawn hoisting

Another optimisation that can be applied to our Woolification algorithm is spawn hoisting. (As there was little change in parallelism found by Embla 2 for spawn hoisting, this has not been implemented.) Recall spawn hoisting is a type of code motion where we move a task spawn statement to as early a point as possible in order to increase potential parallelism. It can be seen as the reverse of synchronisation-point insertion, described earlier, which tries to insert a synchronisation point at as late a point as possible. So here instead of searching forward for lines that depend on the spawned task, we now search backwards for lines on which the spawned task depend.

There is one significant difference, however. In our synchronisation-point insertion algorithm there needs not be a one-to-one run-time correspondence between synchronisation and spawning. We are free to synchronise on the same task at multiple program points because as long as the task is synchronised on at least once the semantics is preserved—synchronising on a task that either has not been spawned or has already been synchronised on is semantically equivalent to a no-op. Similarly, multiple tasks can be synchronised on at one synchronisation point.

With spawn hoisting, however, a one-to-one run-time correspondence between the hoisted spawn and the original call site is essential[5]. In other words, the hoisted task should only be spawned if and only if it would necessarily have been called at the original site, and the number of times the hoisted task would be spawned is the same as the number of times it would have been called at the original site. This condition is sometimes known as *execution equivalence* [54].

5.6.2 Thread-level speculation

Another potential but unimplemented extension is to add support for thread-level speculation to Wool. This would be useful for two reasons. Firstly parallelisations based on profiling are input-dependent and therefore are not guaranteed to be safe for all inputs. Thread-level speculation is one way to guarantee safety. Secondly some dependences may occur so infrequently during execution that it may be beneficial to obtain greater parallelism by speculatively executing a task, only rolling back the task when the dependence actually materialises. As noted in Chapter 2, while software-based TLS is prohibitively slow, hardware-based TLS is still not available in current processors. One compromise is to have a system that tracks memory accesses but simply exits with an error instead of

[5] In the absence of speculation.

rolling back the task whenever a dependence violation is observed. This alerts the programmer to a concurrency bug, which can then be fixed in the program (e.g. by rerunning the program under Embla 2 with the offending data input).

5.7 Related Work

Work on automatic parallelisation has been summarised in Chapter 2. Most however only look at parallelising DOALL loops and not function calls.

Podobas et al. [71] present a comparison of task-creation and synchronisation overheads of task libraries including Wool, Cilk++ and various OpenMP implementations, using a number of microbenchmarks and applications. In this study, Wool generally has the lowest overheads, with an overhead of 19 instructions per task for inlined tasks, and 2200 instructions per task for stolen tasks on two cores. It notes also, however, that task creation and synchronisation overheads increase as the number of cores is increased.

There has also been work to relieve the programmer or profiler from setting the value of *chunk_size* in Algorithm 1. Tzannes et al. [94] have implemented a scheme that decides at run-time whether to further sub-divide and spawn the iteration range based on the current status of the task queue. When run on a processor prototype, their *Lazy Binary Splitting* scheme outperforms the static profile-directed scheme by an average of around 20%.

Another approach to identifying unprofitable spawns is auto-tuning. The PetaBricks system [4] compares the relative performance of using different algorithms and parameters at each level of data granularity, producing a program that switches between alternative algorithms at run-time based on certain parameters to achieve better performance. This approach can be applied here, where we use auto-tuning to set the *chunk_size* parameter as well as the *condition* parameter in the `SPAWNIF` construct suggested in Section 5.4.

5.8 Conclusions

This chapter has described the Woolification process as implemented in our Woolifier, where we use dependence information to transform a sequential program into a Wool program that maximises parallelism. We described the steps needed to reconcile Wool's task model with that of Embla 2's, and the algorithms for C-to-Wool and C-to-Cilk transformations, including those for filtering tasks that are too fine-grained. Our evaluations show that our automatically parallelised programs achieve reasonable speed-ups. In most cases Embla 2's estimates are never attained, but this is expected as Embla 2's model assumes zero overheads in the task library, instant memory access and an unlimited number

of processors. The large gap in parallelism between Cilk's example programs and most SPEC and MiBench benchmark programs in Chapter 4 is mirrored in the performance figures here. We therefore require more than automatic parallelisation in these cases. In the following chapter we will look at how Embla 2 and our Woolifier can be used as part of an interactive parallelisation tool-chain, which performs a best-effort parallelisation, and focuses the programmer on the bottlenecks that require manual changes.

Chapter 6

Pushing the boundary—demonstrating the tool-chain for interactive parallelisation

As we mentioned in the Introduction, a major challenge in program parallelisation is the removal of unnecessary sequentialising artifacts in the conversion process as shown in Figure 6.1. In the previous chapters we have looked at how we can reverse the effects of artifacts introduced by compilation, and in some specific cases, implementation (e.g. reduction operation recognition). Our results in Chapter 4 suggest that while the example programs from Cilk have lots of inherent task-level parallelism (the programs were implemented with task-level parallelism in mind), most general-purpose programs tend to have little and cannot be transformed into highly concurrent programs simply by spawning existing function calls and loops. To address this issue, Embla 2 outputs the critical path of each function call, allowing us to examine the bottlenecks that prevent greater parallelism from being realised. These bottlenecks could be due to artifacts of implementation or design, or (more rarely we believe) the inherently sequential nature of the specification itself. Here we demonstrate the use of critical path information to locate and characterise bottlenecks in some of the simpler examples and suggest possible refactorings or algorithmic changes that would increase parallelism.

Figure 6.1: Software Development Process

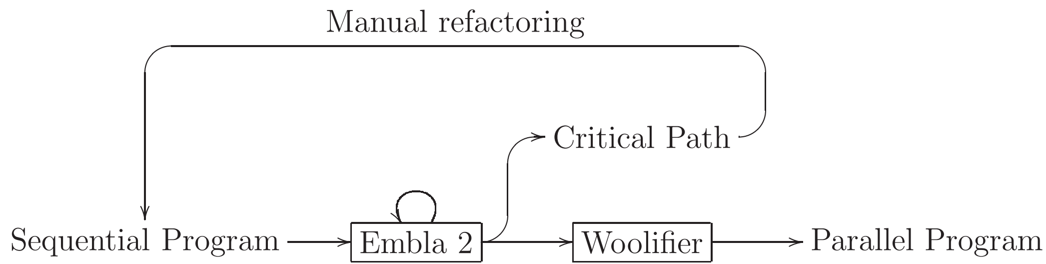


Figure 6.2: Our tool-chain for program parallelisation

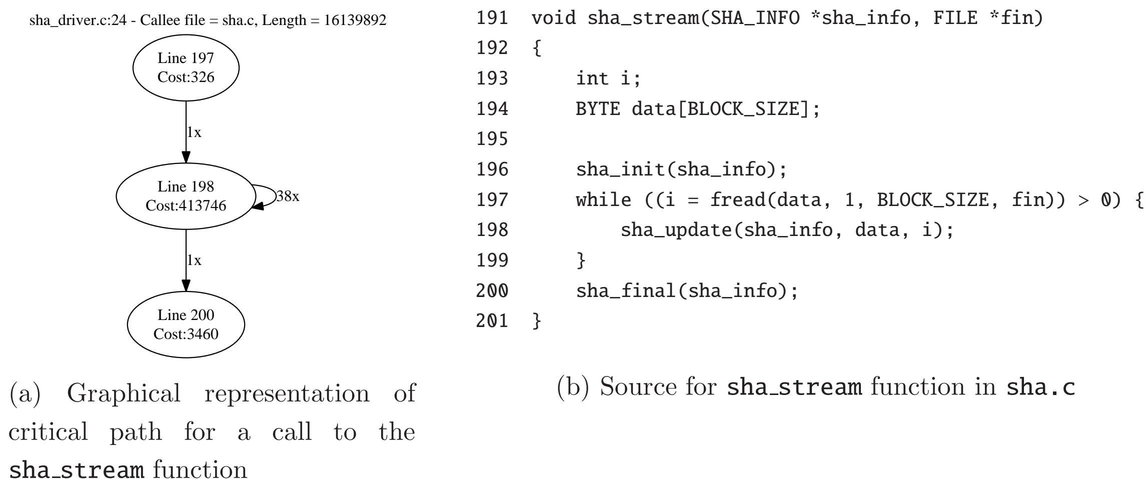
Embla 2's profiler outputs all line instantiations in each critical path, the cost of each instantiation, and the length of the critical path itself. As the size of each Dynamic Dependence Graph, and consequently its critical path, is proportional to the number of line instantiations, the critical path can contain many line instantiations and be very long. We have implemented in Embla 2 a small script that can turn this into a more accessible, graphical representation of the critical path. We aggregate all critical paths for each static function call in the source over program execution in a graph, giving us one graph per function call site. In this graph, each node represents a different source line in the function, marked with line number and its cost averaged over all instantiations of this line. Edges between nodes denote dependences in the critical paths, each labelled with the number of times that dependence occurs in the critical paths. This is more concise as the number of nodes in this graph is bounded by the number of lines in the function definition, and the number of edges by the square of that (but is normally much lower). Figure 6.3(a) gives an example.

The graph can also be viewed as a deterministic finite state machine, with nodes being states and edges being transitions labelled with the line number of the edge's target[1]. This finite state machine could then generate all critical paths for a static function call in the program's execution.

6.1 Case Studies

Figure 6.2 illustrates how the tool-chain we have developed, comprising of Embla 2 (Chapter 4) and Woolifier (Chapter 5), can be used as an interactive and iterative program parallelisation aid. We demonstrate this process with a number of case studies based on benchmark programs, focusing on the manual refactoring performed to achieve greater estimated and actual parallelism. The critical paths in this chapter are all generated

[1]Strictly speaking a deterministic finite state machine must have one transition for every state and symbol. But our graph can be augmented to satisfy this requirement simply by adding a new *rejecting* state and directing all missing transitions to this state.



(a) Graphical representation of critical path for a call to the `sha_stream` function

(b) Source for `sha_stream` function in `sha.c`

Figure 6.3: Critical Path Analysis of `sha`

using an Embla 2 model that considers aggregated data and control dependences, spawns parallel loops with support for parallel reduction operations, and a granularity filter with the threshold set to 1500 x86 instructions, modelling Wool's (augmented) capabilities.

6.1.1 sha

Sha, or Secure Hash Algorithm, is a program that computes a 160-bit hash value from the contents of an input file. Even under our 'Line-level parallelism' model, Embla 2 reports a critical path of length of around 16 million instructions out of the total work of around 22 million, resulting in a parallelism estimate of only 1.36. Figure 6.3 illustrates part of our analysis, using critical paths, into the reason for the lack of inherent parallelism there. Most of the critical path (16,139,892 instructions) can be attributed to a function call on line 24 in the file `sha_driver.c`. By examining the critical path of this line (Figure 6.3(a)), we see that the critical path consists mainly (38 times) of dependences between instantiations of line 198 in `sha.c`, which have an average cost of 413,746 instructions. These correspond to calls to `sha_update` in the program source (Figure 6.3(b)).

Further examination of `sha_update` reveals that the function takes the existing hash value (the *digest*) and derives a new hash value which replaces it. Consequently, each call to this function must depend on the last as it requires the digest computed by the last call. We conclude that the serialisation of these iterations are necessary by design, suggesting that in order to increase the amount of parallelism, the underlying algorithm must be modified, e.g. by dividing the file into blocks and computing independent digests for each block, which are then combined into one final digest. Actual design of an alternative

algorithm for secure hashing is a non-trivial process, and is well beyond the scope of our study.

6.1.2 susan.smoothing

Not all the changes required are as drastic, however. As mentioned in Section 4.4, susan.smoothing is a data-parallel image smoothing application. But unlike susan.corners and susan.edges, little potential parallelism is found here even when loop iterations are spawned. Under the Embla 2 model with aggregated data and control dependences, reduction recognition and loop-iteration spawning, the parallelism figure is 1.002 (47.6M/47.5M). Most of the critical path can be attributed to the function call to `susan_smoothing`, the critical path of which is shown in Figure 6.4. From this we deduce that the main components are the loops, in order of significance, on lines 675-679 (1.6 million iterations in the critical path), 674-681 (108,000 iterations), 667-687 (7220 iterations) and 660-662 (225 iterations). The first three of these are in fact nested within each other.

Examination of these loops reveals that the dependences are due to several variables that are incremented by a fixed amount during each iteration, otherwise known as *induction* variables. By expressing these variables as linear functions of the index variables instead, we break these loop-carried dependences and the critical path drops under the same model dramatically to around 19,000, giving a parallelism estimate of 2490.

This improvement is reflected (though not on the same scale) in actual performance. The sequential version takes over 23 seconds to transform a 3MB image, while the refactored and Woolified program takes 0.93 second on the SPARC machine, corresponding to a speed-up of over 25.

6.1.3 181.mcf

181.mcf is a benchmark in the SPEC 2000 integer benchmark suite that performs single-depot vehicle scheduling. It takes as input start and end times of timetabled trips and the cost of going from the end of one trip to the start of another. The algorithm used is a variant of the Network Simplex algorithm. This algorithm represents the problem as finding an *optimal* spanning tree in a graph. Beginning with a *feasible* spanning tree, it repeatedly replaces an edge in the tree with another not in the tree to get a better solution, until none can be found, at which point the tree is optimal. This algorithm is inherently sequential at the top level, as each iteration simply builds on the result of the last. But at lower levels (within each iteration) we can still make some gains with the help of our parallelisation tool-chain.

Without modifications to the program, Embla 2 finds very little parallelism there, giving an estimate of only 1.01. In the output of feeding the program to our parallelisation

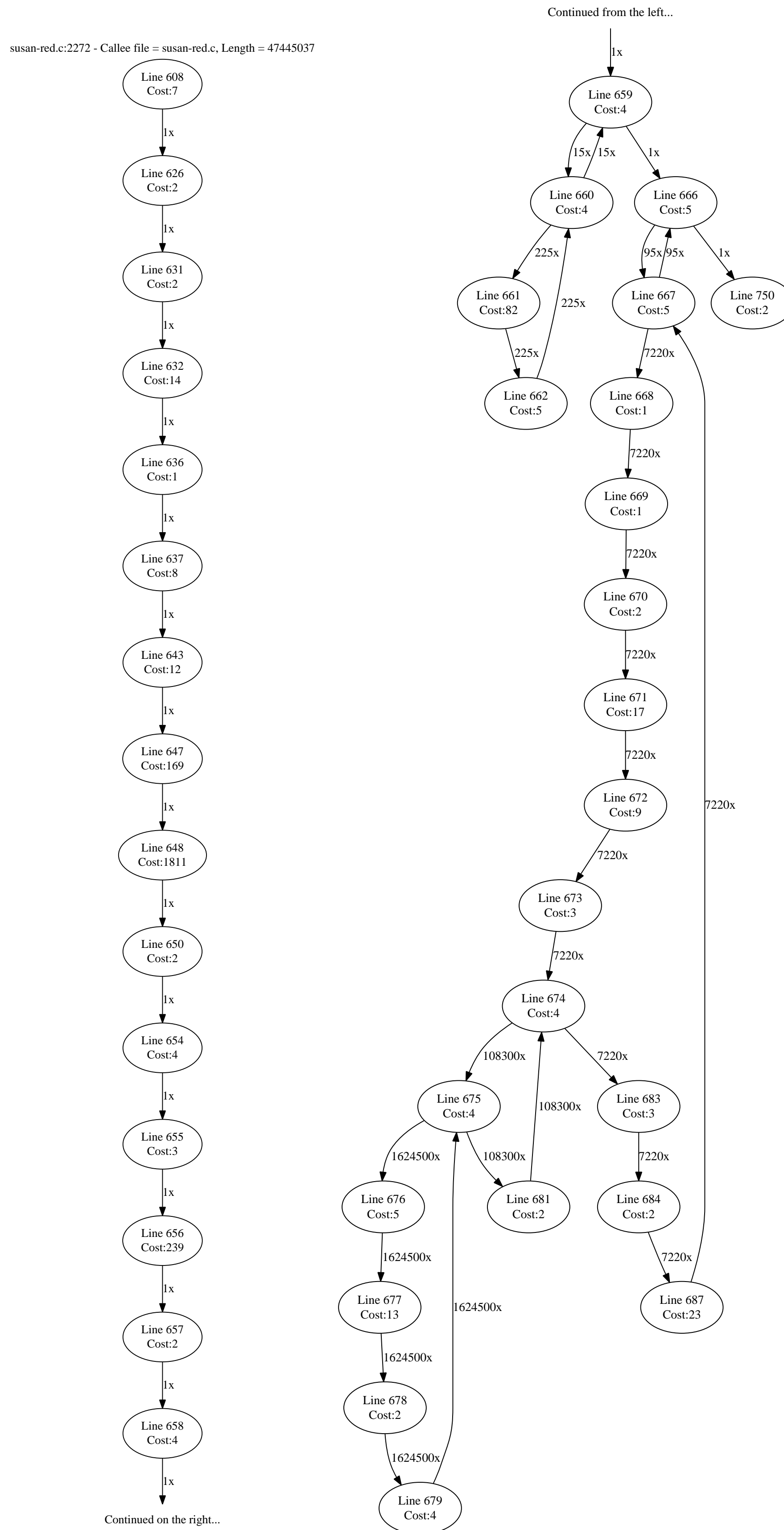


Figure 6.4: Critical path for a call to the `susan_smoothing` function

tool-chain, no function calls became successfully parallelised—either because the tasks or their continuations are too short, or because they were not executed at all in the profile run. Several loops were parallelised, however, but most have small loop bodies. When run on the SPARC machine (with 4 workers) with a larger input, there is a speed-up of around 2% compared to sequential execution.

The first thing we did was to canonicalise as many parallel loops as possible, as only canonical for-loops[2] can be converted into divide-and-conquer style DOALL loops. Most parallel loops could be easily canonicalised, except two loops in `treeup.c`, which climb up a tree from a node, following parent pointers. As the bodies of these two loops are too small, they were not parallelised at all. However, this resulted in little change, with speed-up remaining at around 2%.

We now look at the critical paths to see where they can be shortened. There are two ways to shorten the critical path—one is to break the chain by removing a critical dependence; the other is to reduce the cost of a critical node by recursively shortening the critical path in the function called by that node. Note however that one cannot tell exactly how much the critical path can be shortened as a result without rerunning the program under Embla 2, as a non-critical dependence may become critical if we remove a dependence or reduce the size of a node in a critical path.

We begin at the function call to `global_opt`, which has a critical path length of almost 70M. The critical paths produced by Embla 2 are shown in Figure 6.5. With an average cost of 31.7 million instructions and occurring twice in the critical path, giving a total contribution of 63.5 million instructions to the critical path, line 27 is the costliest node by far, followed by line 32. Little can be done at this level to cut any dependence, as the second iteration needs to use the input of the first, so we focus on the costliest nodes, lines 27 and 32, and look at their critical paths.

We begin with line 27 of `mcf-red.c`. Figure 6.6 shows the critical paths produced by Embla 2 for this line, from which we see that the main component of the critical path is the loop on lines 40-84. This is the bulk of the main loop of the network simplex algorithm. As each iteration seeks to improve on the result of the previous iteration, parallelising this loop would involve major algorithmic changes, perhaps involving a new mathematical theory. Instead, we again focus on the costliest nodes and see whether we can reduce their cost. The costliest nodes are line 41, which has a cost of around 21000 but is in the critical path over 2500 times; and line 88, with a cost of around 37000 and is in the critical path over 130 times.

We first look at line 41, which is a call to the function `primal_bea_mpp`. This is the function that looks at candidate edges to insert into the tree. The critical paths produced

[2]Recall a canonical for-loop is of the form
`for (i=... ; i < ⟨loop-invariant expression⟩; i++) {...}`.

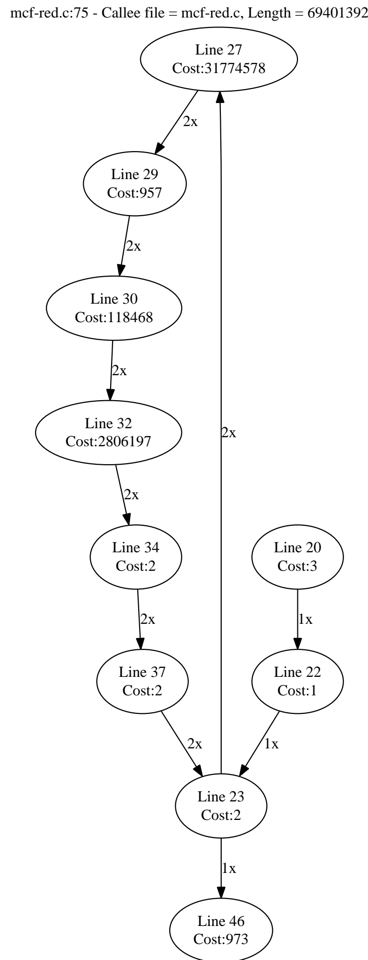


Figure 6.5: Critical Path for `mcf-red.c:75 - global_opt()`

by Embla 2 are shown in Figure 6.7, and it can be seen that the main components are the loops on lines 102-109 and 118-125 (both loops are in the critical path 100,000s of times). To find out why these loops are not parallel in their current form, we turn to the source code.

Figure 6.8 shows the source code for these two loops. The main data structure used here is an array called `perm`, that stores candidate `arc`s (or edges). This array works as a stack, with variables `next` and `basket_size` pointing at the top element at any time. Every new element is inserted to the right of the last inserted element. Both loops are trying to find candidate arcs: the first loop looks at candidate arcs from the last iteration to see if they are still eligible, and if so pushes them on to the stack; the second takes regular samples of arcs, again to append on to the array those that are eligible. It can be seen that the reason the loops are not automatically parallelisable by our tool-chain is the dependences on the variables `next` and `basket_size` that index into `perm`. A number of possible solutions present themselves, each with overheads involved:

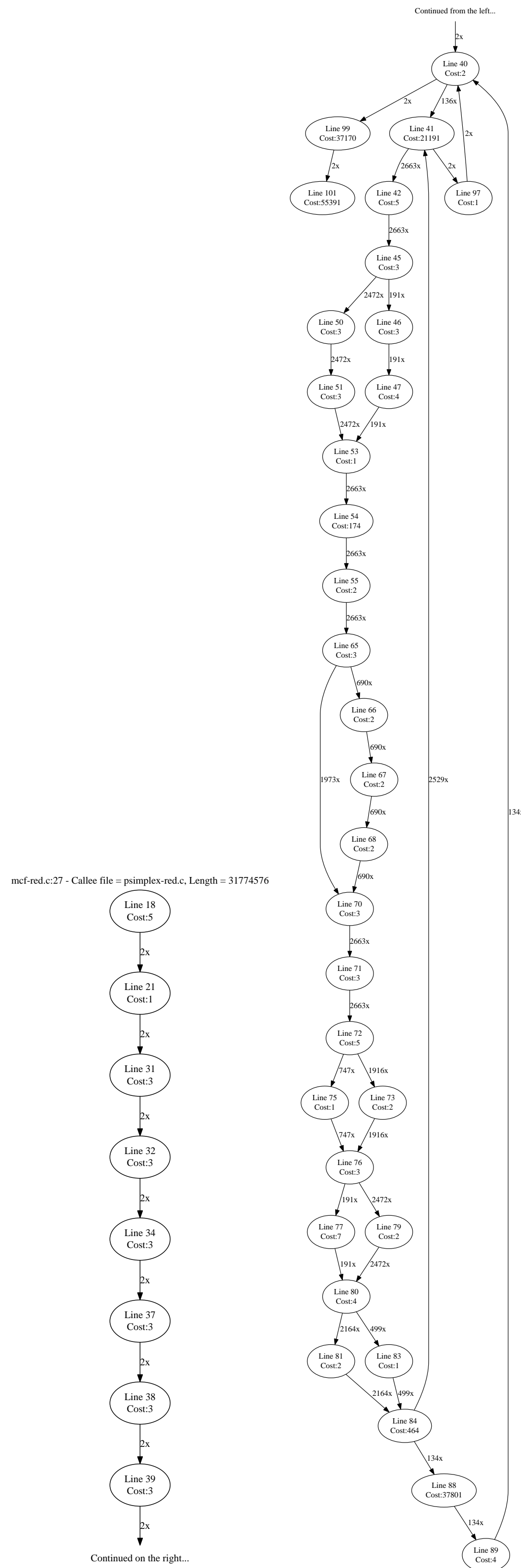


Figure 6.6: Critical Path for `mcf-red.c:27 - primal_net_simplex(&net)`

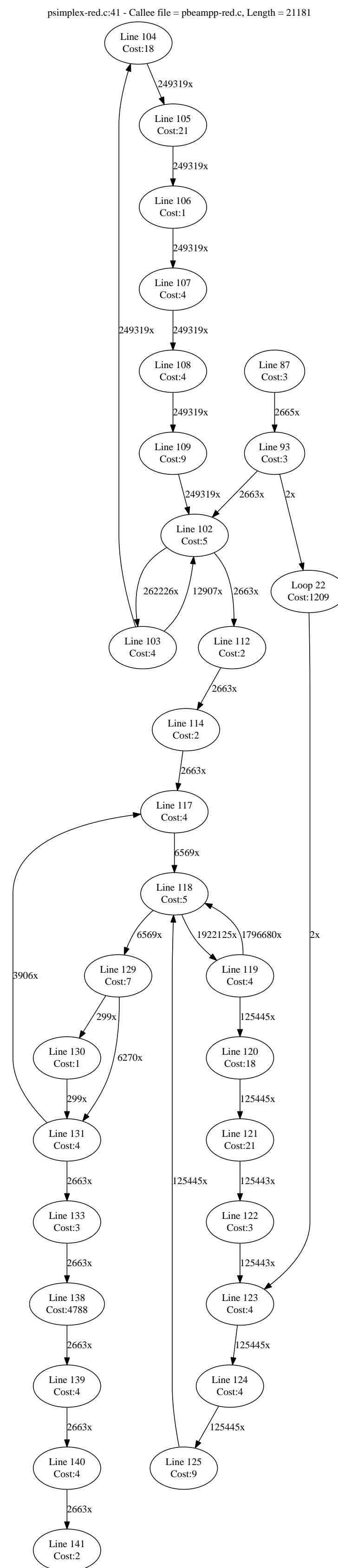


Figure 6.7: Critical Path for `psimplex-red.c:41 - primal_bea_mpp(...)`

```
102     for (((i = (2)) , (next = (0))); (i <= (100)) && (i <= basket_size); i++) {
103       arc = (( *(perm[i])).a);
104       red_cost = bea_compute_red_cost(arc);
105       if (bea_is_dual_infeasible(arc,red_cost) != 0) {
106         next++;
107         ( *(perm[next])).a = arc;
108         ( *(perm[next])).cost = red_cost;
109         ( *(perm[next])).abs_cost = ((red_cost >= (0))?red_cost : -red_cost);
110       }
111     }
...
118   for (; arc < stop_arcs; arc += nr_group) {
119     if ((arc -> ident) > (0)) {
120       red_cost = bea_compute_red_cost(arc);
121       if (bea_is_dual_infeasible(arc,red_cost) != 0) {
122         basket_size++;
123         ( *(perm[basket_size])).a = arc;
124         ( *(perm[basket_size])).cost = red_cost;
125         ( *(perm[basket_size])).abs_cost = ((red_cost >= (0))?red_cost : -red_cost);
126       }
127     }
128   }
```

Figure 6.8: Code for two loops in `pbeampp-red.c` from 181.mcf

1. Change the loop into a DOALL loop, implemented by divide-and-conquer, in which every iteration gets an array cell—i.e. the array index variable is incremented at every iteration regardless of whether an arc is stored there. Effectively this makes `next` and `basket_size` induction variables. The result is a fragmented array, which needs to be compacted afterwards. We introduce a parallel reduction operation that compacts the elements at the joining stage of divide-and-conquer. Thus parallelism is introduced at the expense of extra work in compaction.

2. Change the `perm` array into a linked list structure. The index variable is no longer required, as new elements can simply be appended to the list. As list appending is a simple reduction operation, the loops are now parallelisable. One problem with this approach is that the first loop can no longer be canonical (without adding significant work), as it must now traverse an unindexed linked list. This means that the first loop, though parallelisable, cannot be implemented by divide-and-conquer in Wool as it stands. At the same time, the loop body is too fine-grained for each iteration to be spawned as a task, and therefore the loop must remain sequentially executed. Also, calls to `malloc` and `free` are required for each cell, creating additional overheads.

3. Notice that the array is sorted later on in the function, and therefore the order in which arcs are appended does not in fact matter. However, many arcs do compare equal, meaning the result could be different from the sequential version, although

Method	Embla 2's estimate	Actual speed-up
1	1.56	22%
2	1.18	-17%
3	1.77	13%

Table 6.1: Parallelism results for the three methods of refactoring the loops in Figure 6.8

still valid. By making the index variable increments atomic operations and marking them as *commutative*[3], the loop becomes parallelisable. However, atomic operations are generally expensive, and prone to contention between threads.

To see which method is best, we implemented and evaluated each of the methods by feeding the refactored programs through our tool-chain. The results are shown in Table 6.1. Looking at Embla 2's estimates it would appear that method 3 gives the shortest critical path, but it does not take into account locking and contention costs due to the atomic operation. When parallelised, it is in fact method 1 that gives the best speed-up on the SPARC, with a speed-up of 22% over sequential execution using 8 worker threads.

From line 41 we turn to line 88 in `psimplex-red.c`, the other costly node in the critical path of `mcf-red.c:27`. The critical paths from Embla 2 are shown in Figure 6.9, from which we can see lines 51-72 are the most significant.

The code for these lines are shown in Figure 6.10. The two inner loops work together to perform a depth-first tree traversal, updating the potential of each node using the potential of its parent. While the loops are not parallelisable in their present form, we note that depth-first tree traversal can be implemented instead using divide-and-conquer. However, children of each node are stored in linked lists, giving rise to two problems: the first is that there is a checksum accumulator for the whole tree, and while adding to the checksum is a reduction operation as we have seen it is not easy to implement efficient divide-and-conquer style reduction for linked lists. Secondly the tree can be very flat, giving rise to lots of small tasks that are too small to give any performance gains. Indeed, this second factor means that even if we are only spawning tasks at the top level of the tree, the task granularity is still too small—average task size reported by Embla 2 is around 200-300. (This again shows the usefulness of Embla 2 in helping the programmer make parallelisation decisions.) Thus, these loops must remain unparallelised.

Returning to the critical path for the call to `global_opt` in Figure 6.5, we look at the second costliest node there—line 32. The critical paths from Embla 2 are shown in Figure 6.11, in which the main component is the loop on lines 175-190. This loop iterates through a linked-list-like structure and either does nothing, or appends an arc to an array,

[3]Recall that this makes Embla 2 ignore dependences between instantiations of these operations.

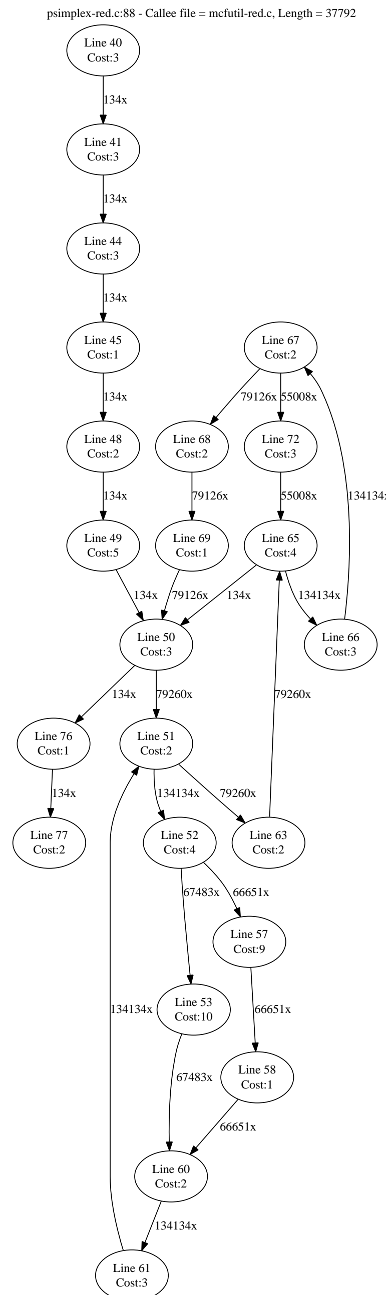


Figure 6.9: Critical Path for `psimplex-red.c:88 - primal_bea_mpp(...)`

or replaces an arc. Examination of the code suggests that the sequentialising dependences are inherent in the algorithm, and that only a change in algorithm can make this loop parallelisable.

We have shown here how we examined the most critical nodes, with the help of Embla 2, and raised both Embla 2's estimated parallelism (from 1.01 to 1.56) and actual speed-up on a multi-core machine (from 2% to 22%). Not all critical nodes were successfully parallelised, but we did succeed in refactoring two significant loops to achieve this speed-

```
50    while(node != root){
51      while(node != (0)){
52        if ((node -> orientation) == (1))
53          node -> potential = ((( *(node -> basic_arc)).cost) + (( *(node -> pred)).potential));
54        else
55  /* == DOWN */
56  {
57          node -> potential = ((( *(node -> pred)).potential) - (( *(node -> basic_arc)).cost));
58          checksum++;
59        }
60        tmp = node;
61        node = (node -> child);
62      }
63      node = tmp;
64  {
65        while((node -> pred) != (0)){
66          tmp = (node -> sibling);
67          if (tmp != (0)) {
68            node = tmp;
69            break;
70          }
71          else
72            node = (node -> pred);
73        }
74      }
75    }
```

Figure 6.10: Code for loops in `mcfutil-red.c` from 181.mcf

up.

6.1.4 179.art

Art is a benchmark program from the SPEC2000 floating point suite. It implements an image-recognition neural network, which is first trained to recognise certain objects, in this case a helicopter and an aeroplane, and is then tasked with finding such objects in a new image.

For the unmodified version of the program, Embla 2 gives an estimate of parallelism of 5.49. When automatically Woolified, the program speeds up by a factor of 3.7 with 64 workers. The critical path length for our training data set is 1.5 billion instructions, most of which can be attributed to line 1089 in `scanner-red.c`, with a cost of around 5 million instructions and around 250 occurrences in the critical paths. The critical path produced by Embla 2 for this line is depicted in Figure 6.12. The main component of this critical path is the loop on lines 401-460. The guard condition of this loop is dependent on a variable assigned to in the loop body, meaning that the loop is inherently sequential and not easily parallelisable. However, we can focus on two nodes: loop 50, which has an average cost of around 430,000 but occurs around 1300 times in the critical path; and line

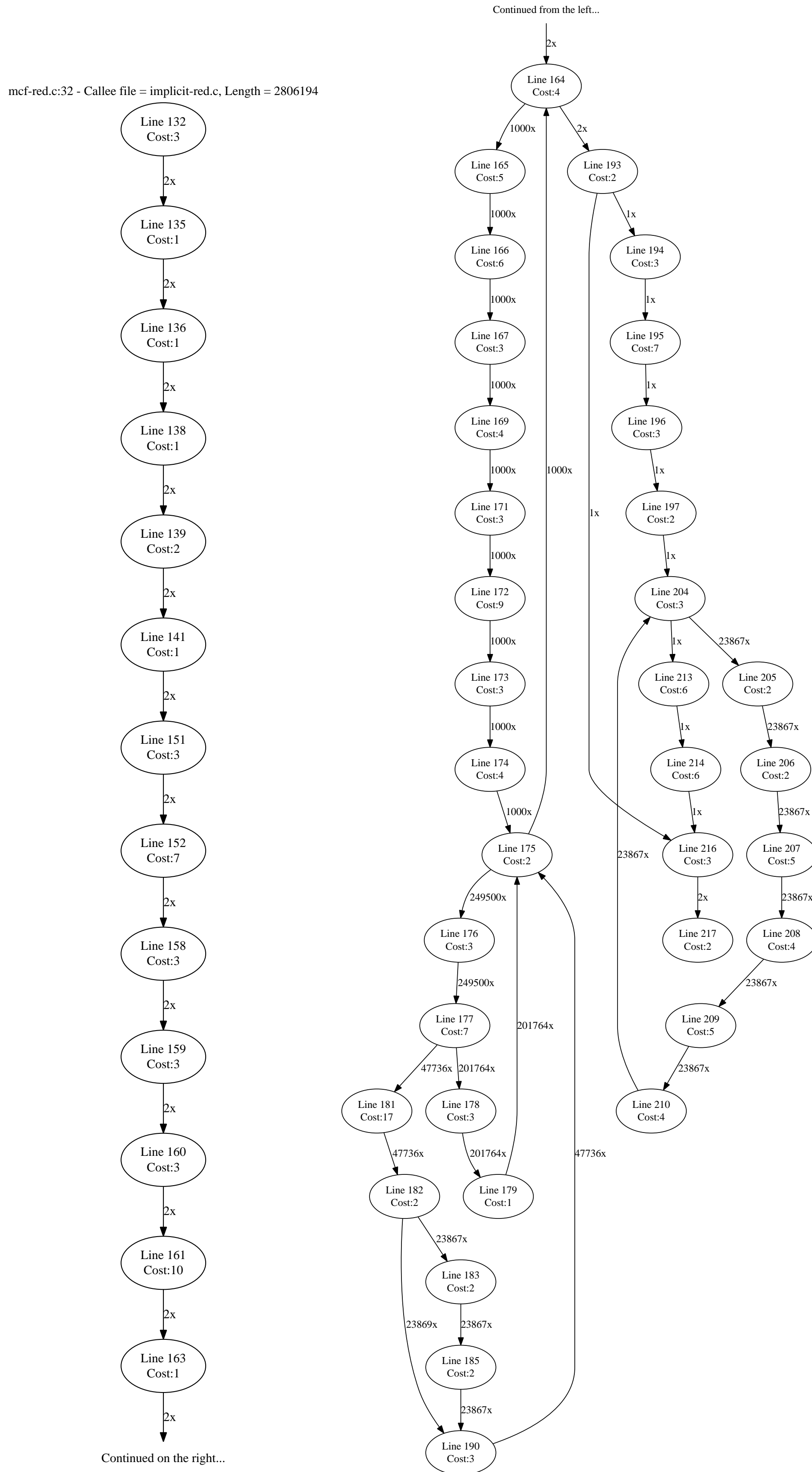


Figure 6.11: Critical Path for `mcf-red.c:32 - price_out_impl(&net)`

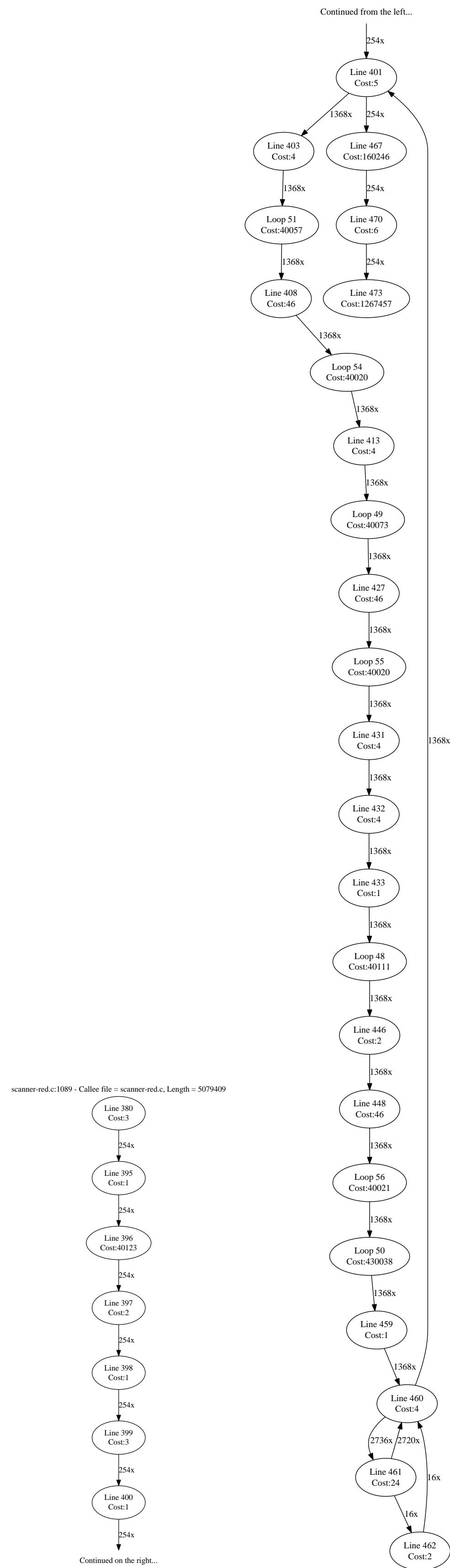


Figure 6.12: Critical path for `scanner-red.c:1089 - train_match(0)`

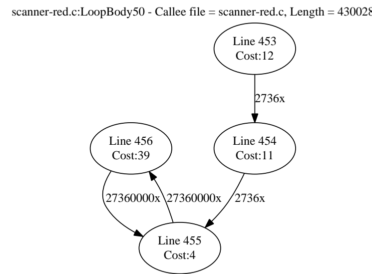


Figure 6.13: Critical path for loop 50 in `scanner-red.c`

```
455           for (ti = 0; ti < numf1s; ti++)
456             Y[tj].y += ((f1_layer[ti].P) * ((bus[ti])[tj]));
```

Figure 6.14: Code for a loop in `scanner-red.c` from 179.art

473, which has an average cost of around 1.2 million and occurs around 250 times in the critical path. We focus on the critical paths of each in turn.

The fact that Embla 2 represents loop 50 by a single node means that this loop is spawned as a separate task. As the model we are using spawns only DOALL loops, this means that this loop is parallelisable as it is. However, even when it is parallel, the cost of the loop (being roughly the critical path length of one iteration of the loop) is still significant. Thus we examine the body of the loop to find out why. Figure 6.13 shows the critical path of the body of loop 50 as produced by Embla 2. The main component here is the inner loop on lines 455-456, shown in Figure 6.14. Examining the loop shows that each time the loop is executed an amount is added to `Y[tj].y`. As the address of `Y[tj].y` remains constant throughout the loop, this field in effect acts as an accumulator for a reduction operation. However, as this is a field rather than a local variable, the operation has not been recognised as a reduction operation[4]. By reducing a local variable inside the loop instead, and copying the final value of this local variable into the `Y[tj].y` immediately after the loop, we make the addition operation a suitable reduction operation, making this inner loop parallelisable. This results in an increase in both Embla 2 estimated parallelism (from 5.49 to 8.51) as well as actual speed-up on the SPARC (from a factor of 3.7 to 4.8 with 64 workers).

We now turn to line 473 (Figure 6.15), the other significant line in Figure 6.12. We focus on the main component of the call on that line, which is loop 58. Again, although loop 58 is parallelisable as it is, its single-iteration cost is still significant, and therefore we look at the critical path of its body, as shown in Figure 6.16, to find the bottleneck. The main component of the critical path there is the inner loop on lines 189-200.

[4]Recall our requirement for a reduction variable to be of *scalar* type.

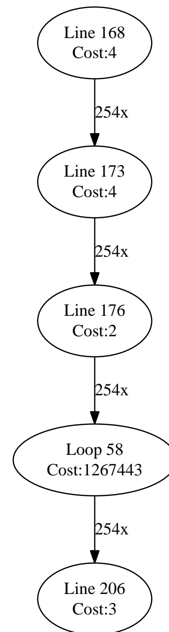


Figure 6.15: Critical path for `scanner-red.c:473`

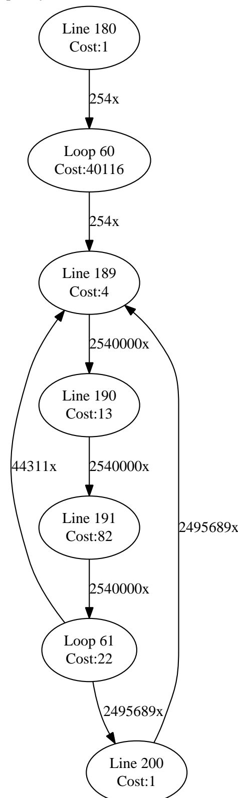


Figure 6.16: Critical path for loop 58 in `scanner-red.c`

```
189     for (j = 0; j < numf1s; j++) {
190       temp = ((bus[j])[i]);
191       (bus[j])[i] += ((g(i) * ((f1_layer[j].P) - ((bus[j])[i]))) * delta_t);
192       if ((fabs((temp - ((bus[j])[i]))) <= er) && (resonant != 0)) {
193 #ifdef DEBUG
194 #endif
195         resonant = 1;
196       }
197       else {
198 #ifdef DEBUG
199 #endif
200         resonant = 0;
201       }
202     }
```

Figure 6.17: Code from `scanner-red.c`

Figure 6.17 shows the code for this inner loop. From it we see that the reason the loop as it stands is not parallelisable is the presence of a read-after-write dependence on a global variable, `resonant`, which can be written to in one iteration and read in the next iteration. The key observation here is that the read is not required—`resonant` is only ever assigned `0` or `1`, and once it is `0`, it will remain so for the remainder of the loop. In other words, the effect of the loop is to set `resonant` to `0` if for any iteration `(fabs((temp - ((bus[j])[i]))) <= er)` is true (and `1` otherwise). We can thus transform the loop body by removing the second condition on line 192 and the assignment on line 195. The only use of `resonant` that remains is the write on line 200, making it an eligible assignment reduction variable. Making this change raises Embla 2's parallelism estimate to around 12.3, and actual speed-up to around 5.8 with 64 workers.

Thus we have again shown how effectively Embla 2 has focused our attention on the parallelism bottlenecks with critical paths, and what sorts of refactoring are required to increase parallelism.

6.1.5 Other examples

Applying the same analysis to other examples, we find that input/output forms a large part of several benchmarks. In dijkstra, for example, a tenth of the program's sequential execution time is taken up by calls to `scanf`. In FFT (MiBench), printing the results takes up around 80% of processing time. As input/output is unparallelisable without significant re-implementation, Amdahl's law would mean that the maximum speed-up, even if we were able to parallelise the rest of the program perfectly, would still be low. This suggests that for some of the benchmarks examined, a parallel implementation of input/output would be very useful.

6.2 Related work

Tools that display programs along with statically and dynamically analysed dependences to the user have been discussed in Chapter 2. Here we focus on work that tries to manually parallelise programs, especially those that overlap with our own case studies.

Prabhu and Olukotun [73, 74] present how they manually parallelised loops in several SPEC 2000 applications using profiled thread-level speculation, in particular achieving 47% speed-up with 181.mcf and 135% speed-up with 179.art. Zhong et al. [105] have automated some of the common transformations applied by Prabhu and Olukotun, achieving 40% speed-up with 181.mcf and around 790% speed-up with 179.art. Bridges et al. [13] use a software pipelining model, coupled with thread-level speculation, to parallelise SPEC 2000 applications. They report a speed-up from simulation of 184% for 181.mcf.

Our case studies differ from all of these in two important ways. Firstly, instead of reporting simulation results, we report actual speed-ups by timing the sequential and parallelised versions of the programs on a real multi-core processor, along with a more architecture-independent 'limit' provided by Embla 2. Secondly, we have not used thread-level speculation in our parallelisation. This allows us to produce parallel programs that can run on existing commodity hardware. Nevertheless, we believe that if thread-level speculation support is added to our framework, speed-ups for these programs could increase further.

In addition, we believe that we are the first to implement and use critical path information to find parallelism bottlenecks. Prior work generally only used relative sequential execution times of loops to determine what parts of the program to examine. However, the results of using relative sequential execution times do not depend on whether those parts are already parallelisable. With critical paths, we can focus immediately on the parts of the program that are not automatically parallelisable.

6.3 Conclusions

This chapter has shown how a parallelisation tool-chain consisting of Embla 2 and Woolifier can aid program parallelisation by making a best-effort automatic parallelisation and producing an easy-to-understand representation of critical paths that focuses the programmer on parallelism bottlenecks. We have also shown the kinds of program refactoring at the bottlenecks that can significantly increase parallelism in the programs. From our case studies these refactorings are mostly alternative implementations of the same algorithm, e.g. moving induction variable increments out of the loop (susan.smoothing), and removing unnecessary reads of variables (179.art). Some involve minor algorithmic changes, e.g. the compaction method for 181.mcf, while for others (e.g. sha) major algorithmic redesigns are required which are beyond the scope of our work.

While some of the refactorings, such as that of the induction variable in susan.smoothing, can be performed by a compiler, in general most artifacts of program implementation can only be removed by the programmers themselves (e.g. 181.mcf). By automating everything that does not require the programmer's input, and by highlighting time-critical sections of the program which need programmer rewriting, we have made interactive parallelisation more efficient.

Chapter 7

Conclusions

We began this thesis by describing the challenge faced by the software community from the advance of multi-core processors. Throughout this thesis, we then described our approach to tackling this problem, namely profiling-based *interactive* parallelisation. We first looked at the amount of parallelism inherent in benchmark programs, before focusing on Nested Function-level Fork-join Parallelism that is the basis of many popular parallel languages and task libraries.

Our approach to interactive parallelisation centres around a tool-chain with two major components: Embla 2, a tool that profiles dependences in a sequential program and derives an estimate of the amount of parallelism that could be achieved with fork-join parallelism; and Woolifier, a source-to-source program paralleliser that aims to realise the parallelism found by Embla 2. Using Embla 2 we showed that the amount of potential parallelism differs widely from program to program. Profiling results of Embla 2 are then fed to Woolifier, which transforms the program using a parallel language or task library, performing optimisations such as parallelising reduction operations and inlining fine-grained tasks. Finally, we showed how critical paths output by Embla 2 can be used by the programmer to quickly focus on parallelism bottlenecks that need to be refactored in order to get greater parallelism. We saw that by refactoring these bottlenecks we achieved significant gains in actual speed-up.

Our aim was to automate the parallelisation process as much as we can, and present critical path information to focus the programmer on bottlenecks in the program for manual refactoring. Sometimes the refactorings can in fact be automated by the compiler, and there is certainly scope for extending our Woolifier to perform more parallelism-enhancing transformations. However, we have also seen in many cases that programmer intervention is required, either in the form of a reworking of the implementation of the same algorithm, or in a new more parallel algorithm altogether.

Our approach is mainly based on profiling, which by nature can be very input-dependent. In particular, the absence of a certain dependence in a particular profile run does not mean

that no such dependence exists for any input. With our tool-chain the responsibility is left with the programmer to verify the safety of the parallelised programs, and in our experience we have had to manually hoist synchronisation points in a small number of cases due to certain dependences not materialising in the profile runs. Regarding this weakness we make the following points:

- When dependences are missed by a profiler, it is normally either because a certain control path was not taken in a profile run, or that the input data was too small. Thus we should ensure that a large enough data set is used for profiling, and can enlist the help of a test coverage tool to ensure all areas of code are covered.
- Dependence information can be extracted by *static* rather than dynamic analysis. We know that static analysis can often over-approximate, giving dependences that may never occur in practice. However, it would be very interesting to see how different statically analysed dependences would be to those produced by Embla 2, given recent advances in alias analysis and separation logic.
- Another technique to mitigate this problem is the use of run-time memory monitoring, as employed by thread-level speculation frameworks. Dependences are tracked at run-time, and when they are observed to be violated the relevant tasks are rolled back and re-executed in the right order. However, overheads associated with such a mechanism are high, potentially wiping out all the potential performance gains from parallelism. We propose a lighter weight version which simply exits with an error when a dependence violation is observed, saving the overheads required to keep past memory states to roll back a task.

We believe the techniques used by our tool-chain are useful to programmers trying to parallelise large legacy systems, especially those with which they may be unfamiliar. Our tool-chain may also be useful for programmers with little knowledge of parallel languages developing new programs, as the development process does not require parallel programming at all—all the refactorings are done on the sequential code.

Our opinion, however, is that it is better for programmers to consider parallelism right from the beginning and program in a parallel language. This is because, as we have noted, each conversion in the development process, from specification to algorithm, from algorithm to program, and from program to executable, can introduce artifacts that unnecessarily restrict parallelism. Our tool-chain certainly does well in removing some of these restrictions, but it is much better for these restrictions not to have been introduced in the first place. We believe a paradigm shift in software design is required—just as programmers in the past adapted to object-oriented principles when faced with increasing size and complexity of software projects, so they must now be familiar with design

patterns in parallelism in order to adapt to the world of multi-core processors and other parallel computer architectures. How to achieve this, though, would be the subject of another thesis.

Bibliography

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256. ACM, 1990.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[3] F. Aleen and N. Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *ASPLOS '09: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 241–252. ACM, 2009.

[4] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–49. ACM, 2009.

[5] A. W. Appel and Z. Shao. An empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6:47–74, 1994.

[6] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[7] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 342–351. ACM and IEEE Computer Society, 1992.

[8] F. Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the USENIX Annual Technical Conference*, pages 41–46. USENIX Association, 2005.

[9] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA '09: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 81–96. ACM, 2009.

[10] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, pages 141–154. Springer-Verlag, 1994.

[11] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, Aug. 1996.

[12] H.-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–268. ACM, 2005.

[13] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–81. IEEE Computer Society, 2007.

[14] P. A. Buhr. Are safe concurrency libraries possible? *Communications of the ACM*, 38(2):117–120, 1995.

[15] C. R. Calidonna, M. Giordano, and M. M. Furnari. A graphic parallelizing environment for user-compiler interaction. In *ICS '99: Proceedings of the 13th International Conference on Supercomputing*, pages 238–245. ACM, 1999.

[16] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 519–538. ACM, 2005.

[17] M. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 434–446. ACM, 2003.

[18] T. Chen, J. Lin, X. Dai, W. Hsu, and P. Yew. Data dependence profiling for speculative optimizations. In *Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 57–72. Springer-Verlag, 2004.

[19] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24. ACM, 2003.

[20] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *Computer Journal*, 19(1):43–49, 1976.

[21] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[22] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, April 1988.

[23] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 223–234. ACM, 2007.

[24] R. Ennals and S. P. Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *ICFP '03: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 287–298. ACM, 2003.

[25] K.-F. Faxén. Wool–a work stealing library. *SIGARCH Computer Architecture News*, 36(5):93–100, December 2008.

[26] K.-F. Faxén, K. Popov, S. Jansson, and L. Albertsson. Embla – data dependence profiling for parallel programming. In *CISIS '08: Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 780–785. IEEE Computer Society, 2008.

[27] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[28] C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. In *Proceedings of the Applied Semantics Summer School (APPSEM)*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer-Verlag, 2000.

[29] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA '09: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, pages 79–90. ACM, 2009.

[30] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 191–202, 2003.

[31] M. Girkar and C. D. Polychronopoulos. Extracting task-level parallelism. *ACM Transactions on Programming Languages and Systems*, 17(4):600–634, 1995.

[32] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, Aug. 1996.

[33] C. M. Grinstead and L. J. Snell. *Introduction to Probability*. American Mathematical Society, 1997.

[34] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the International Workshop on Workload Characterization*, pages 3–14. IEEE Computer Society, 2001.

[35] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, page 49. ACM, 1995.

[36] T. Harris and S. Singh. Feedback directed implicit parallelism. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 251–263. ACM, 2007.

[37] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[38] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. ACM, 1993.

[39] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[40] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 59–70. ACM, 2004.

[41] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 205–214. ACM, 2007.

[42] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[43] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, 1991.

[44] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(1):7, 2002.

[45] B. Kreaseck, D. Tullsen, and B. Calder. Limits of task-based parallelism in irregular applications. In *Proceedings of the International Symposium on High Performance Computing*, volume 1940 of *Lecture Notes in Computer Science*, pages 43–58. Springer-Verlag, 2000.

[46] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Caşcaval. How much parallelism is there in irregular applications? In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–14. ACM, 2009.

[47] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 46–57. ACM and IEEE Computer Society, 1992.

[48] J. R. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):812–826, 1993.

[49] D. Lea. A Java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 Conference on Java Grande*, pages 36–43. ACM, 2000.

[50] D. Leijen and J. Hall. Parallel performance: Optimize managed code for multi-core machines. *MSDN Magazine*, Oct 2007.

[51] C. E. Leiserson. The Cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527. ACM, 2009.

[52] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam. SUIF Explorer: an interactive and interprocedural parallelizer. In *PPoPP '99: Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 37–48. ACM, 1999.

[53] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *ASPLOS VII: Proceedings of the Seventh International Conference*

on Architectural Support for Programming Languages and Operating Systems, pages 138–147. ACM, 1996.

[54] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. In *PPoPP '06: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 158–167. ACM, 2006.

[55] J. Mak, K.-F. Faxén, S. Janson, and A. Mycroft. Estimating and exploiting potential parallelism by source-level dependence profiling. In *Euro-Par 2010: Proceedings of the 16th European Conference on Parallel and Distributed Computing*, volume 6271 of *Lecture Notes in Computer Science*, pages 26–37. Springer, 2010.

[56] E. Mehofer and B. Scholz. Probabilistic data flow system with two-edge profiling. In *DYNAMO '00: Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 65–72. ACM, 2000.

[57] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 24–33. ACM, 1991.

[58] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–14. ACM, 2010.

[59] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[60] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[61] MIPS Technologies, Inc. *MIPS SDE Programmers' Guide*, 2007.

[62] N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. In *Proceedings of the Third Workshop on Runtime Verification*, pages 149–170. Elsevier, 2003.

[63] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100. ACM, 2007.

[64] V. H. Nguyen, K. Taura, and A. Yonezawa. Parallelizing programs using access traces. In *LCR '02: Proceedings of the 6th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 2002.

[65] C. E. Oancea and A. Mycroft. Set-congruence dynamic analysis for thread-level speculation (TLS). In *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*, volume 5335 of *Lecture Notes in Computer Science*, pages 156–171. Springer-Verlag, 2008.

[66] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 303–313. IEEE Computer Society, 1999.

[67] G. Ottoni and D. I. August. Global multi-threaded instruction scheduling. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 56–68. IEEE Computer Society, 2007.

[68] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118. IEEE Computer Society, 2005.

[69] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 142–151. IEEE Computer Society, 2008.

[70] P. Petersen and D. Padua. Static and dynamic evaluation of data dependence analysis techniques. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1121–1132, 1996.

[71] A. Podobas, M. Brorsson, and K. Faxén. A comparison of some recent task-based parallel programming models. In *Proceedings of the Third Workshop on Programmability Issues for Multi-Core Computers*, 2010.

[72] M. A. Postiff, D. A. Greene, G. S. Tyson, and T. N. Mudge. The limits of instruction level parallelism in SPEC95 applications. *Computer Architecture News*, 27(1):31–34, 1999.

[73] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *PPoPP '03: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12. ACM, 2003.

[74] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in SPEC2000. In *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 142–152. ACM, 2005.

[75] G. Ramalingam. Data flow frequency analysis. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI'96)*, pages 267–277. ACM, 1996.

[76] L. Rauchwerger, P. K. Dubey, and R. Nair. Measuring limits of parallelism and characterizing its vulnerability to resource constraints. In *MICRO 26: Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 105–117. IEEE Computer Society, 1993.

[77] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York*, volume 5502 of *Lecture Notes in Computer Science*, pages 348–362. Springer-Verlag, 2009.

[78] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[79] A. W. Roscoe and C. A. R. Hoare. The laws of OCCAM programming. *Theoretical Computer Science*, 60(2):177 – 229, 1988.

[80] P. Rundberg and P. Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction-Level Parallelism*, 3, 2001.

[81] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[82] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer-Verlag, 2003.

[83] J. E. Smith. A study of branch prediction strategies. In *ISCA '81: Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–148. ACM, 1981.

[84] K. Smith, B. Appelbe, and K. Stirewalt. Incremental dependence analysis for interactive parallelization. In *ICS '90: Proceedings of the 4th International Conference on Supercomputing*, pages 330–341. ACM, 1990.

[85] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

[86] D. Stefanović and M. Martonosi. Limits and graph structure of available instruction-level parallelism. In *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 1018–1022. Springer-Verlag, 2000.

[87] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[88] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2000.

[89] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, 2001.

[90] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[91] K. B. Theobald, G. R. Gao, and L. J. Hendren. On the limits of program parallelism and its smoothability. In *MICRO 25: Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 10–19. IEEE Computer Society, 1992.

[92] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–368. IEEE Computer Society, 2007.

[93] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–187. ACM, 2009.

[94] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–189. ACM, 2010.

[95] N. Vachharajani, M. Iyer, C. Ashok, M. Vachharajani, D. I. August, and D. Connors. Chip multi-processor scalability for single-threaded applications. *SIGARCH Computer Architecture News*, 33(4):44–53, 2005.

[96] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT 2007: Proceedings of*

the 16th International Conference on Parallel Architectures and Compilation Techniques, pages 49–59. IEEE Computer Society, 2007.

[97] C. von Praun, R. Bordawekar, and C. Caşcaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 185–195. ACM, 2008.

[98] C. von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–89. ACM, 2007.

[99] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 176–188. ACM, 1991.

[100] F. Warg and P. Stenström. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 221–230. IEEE Computer Society, 2001.

[101] R. P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[102] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 439–453. ACM, 2005.

[103] P. Wu, A. Kejariwal, and C. Caşcaval. Compiler-driven dependence profiling to guide program parallelization. In *LCPC '08: Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 232–248. Springer-Verlag, 2008.

[104] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 47–58. IEEE Computer Society, 2009.

[105] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, pages 290–301. IEEE Computer Society, 2008.