Compiling Lenient Languages for Parallel Asynchronous Execution

Klaus Erik Schauser

Department of Electrical Engineering and Computer Science
Division of Computer Science
University of California, Berkeley

This work was supported in part by an IBM Graduate Fellowship, by the National Science Foundation Presidential Faculty Fellowship CCR-925370, and LLNL Grant UCB-ERL-92/172. Computational resources were provided, in part, under NSF Infrastructure Grant CDA-8722788.

Compiling Lenient Languages for Parallel Asynchronous Execution

Klaus Erik Schauser
University of California, Berkeley

Abstract

High-level programming languages and exotic architectures have often been developed together, because the languages seem to require complex mechanisms realized by specialized hardware. The implicitly parallel language Id and dataflow architectures are a prime example. The language allows an elegant formulation of a broad class of problems while exposing substantial parallelism, however, its non-strict semantics require fine-grain dynamic scheduling and synchronization making an efficient implementation on conventional parallel machines challenging.

This thesis presents novel compilation techniques for implicitly parallel languages, focusing on techniques addressing dynamic scheduling. It shows that it is possible to implement the lenient functional language Id efficiently by partitioning the program into regions that can be scheduled statically as sequential threads, and realizing the remaining dynamic scheduling through compiler-controlled multithreading. Partitioning the program into sequential threads requires substantial compiler analysis because the evaluation order is not specified by the programmer.

The main contribution of the thesis is the development and implementation of powerful thread partitioning algorithms. In addition, a new theoretical framework is formulated for proving the correctness of the partitioning algorithms. The thesis presents a new basic block partitioning algorithm, separation constraint partitioning, which groups nodes in situations where previous algorithms failed. It presents a new solution for interprocedural partitioning which works in the presence of recursion. All of the partitioning algorithms have been implemented and form the basis of a compiler of Id for workstations and several parallel machines. This execution vehicle is used to quantify the effectiveness of the different partitioning schemes on whole applications. The experimental data, collected from our implementation of Id for workstations and the CM-5 multiprocessor, shows the effectiveness of the partitioning algorithm and compiler-controlled multithreading.

Acknowledgments

There is nothing more exciting than working in a small group of motivated people who want to revolutionize the world. That's what I wrote in the acknowledgments of my Masters, and it is still true. The work described in this thesis is the result of fruitful collaboration within the Berkeley TAM group. I would like to thank all its members, Professor David Culler, Thorsten von Eicken, Seth C. Goldstein, Andrea Dusseau, Lok Tin Liu, Steve Lumetta, Steve Luna, and Rich Martin.

I especially thank my advisor, Professor David Culler, for his support. David, you are an extraordinary person, full of ideas and motivation. Working with you is fun. I really hope we keep in close contact.

I am grateful to Professor David Patterson, my second reader, for teaching me a quantitative approach to computer architecture. You carefully read my thesis and gave many helpful comments. I have the highest respect for your research and your advice.

It is amazing how sad I feel moving on to a new place and leaving behind a group of friends that I could rely on. I didn't have to say good bye to you Thorsten, my friend and my office mate for all those years, as you already left before me. But I am leaving behind Seth. It was a lot of fun working with you as close as we did the last year, let's continue to be friends. Andrea and Steve, you became my new friends after Thorsten left. Thank you all so much for helping me proofread my thesis.

I would also like to thank all members of Professor Arvind's Computation Structures Group of MIT for providing us with their Id compiler, answering all our questions about it, and for making changes to it at our request. Ken Traub, you were the greatest inspiration to me.

And then there are all of our friends with and without kids we are leaving behind.

We will never forget the wonderful time together, after work and at weekends. Susanne, Andreas, Patrick, Oliver, Meiling, Andy, Nina, Jamie, Annie, Keith, Casey, Sidney, Martha, and Luigi you all mean so much to us. Exploring the bay area windsurfing with you Luigi and Andy was exciting.

My deepest appreciation goes to Martina, my wife, for her patience, love, and support, and to my two little daughters, Natalie and Nicole, for their love and laughter. Finally, I want to thank my parents, Esther and Geert Henning for their support.

This work was supported in part by an IBM Graduate Fellowship, by the National Science Foundation Presidential Faculty Fellowship CCR-925370, and LLNL Grant UCB-ERL-92/172. Computational resources were provided, in part, under NSF Infrastructure Grant CDA-8722788.

Für Esther und Geert Henning Schauser

Contents

List of Figures xi

List of Tables xv

1 Introduction 1

1.1 Presentation of Thesis . . . 1
1.2 Implicitly Parallel Languages . . . 2
1.3 Architectural Approaches . . . 4
1.4 Compilation Approach . . . 5
1.5 Contributions . . . 6
1.6 Roadmap . . . 7

2 Lenient Languages and Parallel Machines 9

2.1 Lenient Functional Languages . . . 10
2.1.1 Purely Functional Core . . . 11
2.1.2 I-structures and M-structures . . . 12
2.2 Evaluation Strategies . . . 13
2.2.1 Expressiveness . . . 14
2.2.2 Speculative Computation . . . 18
2.2.3 Parallelism . . . 19
2.2.4 Evaluation Overhead . . . 20
2.3 Implementation Requirements for Lenient Evaluation . . . 21
2.3.1 Synchronization and Dynamic Scheduling . . . 21
2.3.2 Forms of Dynamic Scheduling . . . 25
2.3.3 Memory Management . . . 26

2.3.4 Summary 27
2.4 Massively Parallel Machines 29
2.4.1 LogP Model 29
2.4.2 Compilation Requirements 31
2.5 Compilation Challenge 34
2.5.1 Partitioning 35
2.5.2 Compiler Controlled Multithreading 36
2.5.3 Summary 37

3 Compiler-Controlled Multithreading **39**
3.1 The Threaded Abstract Machine 39
3.1.1 Program Structure 40
3.1.2 Storage Hierarchy 40
3.1.3 Execution Model 41
3.1.4 Efficient Communication 43
3.2 Compiling to TAM 44
3.2.1 Representation of Parallelism 44
3.2.2 Frame Storage Management 45
3.2.3 Communication 46
3.2.4 Partitioning 47
3.3 Thread Generation 47
3.3.1 Instruction Scheduling 49
3.3.2 Lifetime Analysis 49
3.3.3 Frame Slot and Register Assignment 50
3.3.4 Synchronization 50
3.3.5 Thread Ordering 51
3.3.6 Summary 52

4 Intraprocedural Partitioning **53**
4.1 Problem Statement 54
4.2 Program Representation 55
4.2.1 Basic Blocks 55
4.2.2 Interfaces 59
4.2.3 Simple Example 61

4.2.4 Summary . . . 63

4.3 Basic Block Partitioning . . . 63

4.3.1 Partitioning a Small Example . . . 65

4.3.2 Dependence Set and Demand Set Partitioning . . . 69

4.3.3 Subpartitioning . . . 70

4.3.4 Iterated Partitioning . . . 71

4.3.5 Examples of Iterated Partitioning . . . 72

4.3.6 Limits of Iterated Partitioning . . . 74

4.4 Separation Constraint Partitioning . . . 77

4.4.1 Indirect Dependencies and Separation Constraints . . . 78

4.4.2 Separation Constraint Partitioning . . . 80

4.4.3 Example of Separation Constraint Partitioning . . . 81

4.4.4 Refined Separation Constraint Partitioning . . . 83

4.4.5 Correctness . . . 86

4.4.6 Merge Order Heuristics . . . 88

4.4.7 Summary . . . 92

5 Interprocedural Partitioning **95**

5.1 Example of Interprocedural Partitioning . . . 96

5.2 Global Partitioning . . . 100

5.3 Global Analysis . . . 102

5.4 Multiple Callers and Conditionals . . . 106

5.4.1 Propagation Rules . . . 107

5.5 Conditional Example . . . 108

5.6 Limitations of Interprocedural Partitioning . . . 111

5.7 Handling Recursion . . . 116

5.7.1 Squiggly Edges . . . 116

5.7.2 Annotations . . . 121

5.8 IO Sets Refinements . . . 122

5.9 Summary . . . 125

6 Experimental Results **127**

6.1 Experimental Framework . . . 128

6.1.1 Id Compiler and TAM backend . . . 128

6.1.2 Machine Characteristics 130
6.1.3 Benchmark Suite 131
6.2 Effectiveness of Partitioning 132
6.2.1 Thread Characteristics 135
6.2.2 TL0 Instruction Distributions 138
6.2.3 Grouping of boundary nodes 140
6.3 Compiler-controlled Multithreading 144
6.3.1 Scheduling Hierarchy 144
6.3.2 Scheduling Hierarchy Under Parallel Execution 149
6.3.3 Handling Remote Accesses 150
6.4 Overall Efficiency 152
6.4.1 Timing Measurements 152
6.4.2 Cycle Distributions 153
6.4.3 Sequential Efficiency 156
6.4.4 Parallel Efficiency 158
6.4.5 Parallel Cycle Distributions 162
6.5 Summary 163

7 Conclusions and Future Work **165**
7.1 Related Work 165
7.2 Future Work 169
7.3 Conclusions 171

Bibliography **175**

A Partitioning: The Theory **183**
A.1 Basic Definitions 184
A.2 Partitioning 188
A.3 Annotation Based Repartitioning 192
A.4 Basic Block Partitioning Algorithm 195
A.5 Congruence Syndromes 200
A.6 Valid Inlet/Outlet Annotations 203
A.7 The Partitioning Algorithm 212

List of Figures

1.1 *Functional view of the hydrodynamics and heat conduction code Simple.* . . 3

2.1 *The three layers of the lenient language Id.* 11

2.2 *The three evaluation strategies and their expressiveness.* 15

2.3 *Different versions of the Fibonacci function, showing the difference in expressive power of strict, lenient, and lazy evaluation.* 16

2.4 *Examples illustrating the need for dynamic scheduling under non-strictness.* 22

2.5 *The dependencies for function f and two possible scenarios, $k = n$ and $l = m$.* 24

2.6 *The LogP model.* 30

2.7 *Possible compilation model under LogP.* 34

3.1 *TAM activation tree and embedded scheduling queue.* 41

3.2 *Thread generation steps.* 48

3.3 *Redundant control arc elimination rule.* 51

4.1 *Dataflow nodes used for the compilation of Id.* 56

4.2 *Partitioning of a small dataflow graph.* 61

4.3 *The dataflow graph for the small example* `x = a + b; y = a * b;` *and the threads derived by various partitioning algorithms.* 66

4.4 *Example of Iterated Partitioning.* 73

4.5 *Example with six nodes requiring five steps of dependence and demand set partitioning to obtain the final solution.* 75

4.6 *Example where iterated partitioning fails to merge two threads.* 76

4.7 *Example showing that merging two nodes under separation constraint partitioning may introduce new separation constraints, and thereby affect the final result.* 82

4.8 *Updating indirect dependencies after merging two nodes with separation constraint partitioning.* . . . 86

4.9 *After merging two nodes u and v, it is not necessary to follow two indirect dependencies.* . . . 87

4.10 *Example showing that separation constraint partitioning may create non-convex threads during intermediate steps.* . . . 88

4.11 *Merging of multiple send and receive nodes.* . . . 90

4.12 *Merging of switch and merge nodes.* . . . 91

4.13 *Order in which the heuristic for separation constraint partitioning tries to merge interior nodes.* . . . 92

5.1 *Example of interprocedural partitioning with annotation propagation across a single-def-single-call interface.* . . . 98

5.2 *Nodes comprising a single-def-single-call interface.* . . . 103

5.3 *Example of interprocedural partitioning across a conditional.* . . . 109

5.4 *Example showing that partitioning of a basic block can yield illegal threads if it uses annotations from global analysis which contain information about the structure of the basic block.* . . . 113

5.5 *Example for determining dependencies from arguments to results.* . . . 118

5.6 *Example of interprocedural partitioning propagating IO sets with new annotations.* . . . 124

6.1 *Overview of Id to TAM compiler. Shown in gray are other compilation approaches.* . . . 129

6.2 *Dynamic thread and inlet distributions for the benchmark programs under various partitioning schemes.* . . . 136

6.3 *Dynamic TL0 instruction distribution for the benchmark programs under various partitioning schemes.* . . . 139

6.4 *Average number of argument and result send operations per function call.* 142

6.5 *Average number of switch instructions per conditional.* . . . 143

6.6 *Dynamic run-time on a SparcStation 10 for the benchmark programs under various partitioning schemes.* . . . 153

6.7 *Dynamic cycle distribution of TL0 instructions for a Sparc processor for the benchmark programs under various partitioning schemes.* . . . 154

6.8 *Speedup for Gamteb and Simple on the CM-5 from 1 to 64 processors.* . . . 160

6.9 *Distribution of processor time for the benchmark programs.* . . . 163

A.1 *Example showing the use of the call tree grammar.* . . . 187

A.2 *This graph shows the situation for the proof of Proposition 10.* 211

A.3 *Structure of correctness proof of interprocedural algorithm.* 213

List of Tables

2.1 *Comparison of the three evaluation strategies.* 15

2.2 *Forms of dynamic scheduling and their memory requirements.* 26

4.1 *Order in which heuristic for separation constraint partitioning tries to merge nodes.* 89

6.1 *Benchmark programs and their inputs.* 131

6.2 *Dynamic TL0 thread and inlet statistics.* 135

6.3 *Dynamic function calls and conditional statistics.* 141

6.4 *Dynamic TAM scheduling statistics.* 146

6.5 *Dynamic scheduling characteristics under TAM for the programs on a 64 processor CM-5.* 150

6.6 *Percentages of split-phase operations into instruction types and locality.* . . 151

6.7 *Dynamic run-time in seconds on a SparcStation 10 for the benchmark programs under various partitioning schemes.* 152

6.8 *Dynamic run-time for several of the Id benchmark programs and equivalent C or FORTRAN programs.* 156

6.9 *Dynamic run-time in seconds on the CM-5 for Gamteb.* 159

6.10 *Dynamic run-time in seconds on the CM-5 for Simple.* 159

6.11 *Dynamic run-time in seconds of Gamteb (9-cell), running 40,000 particles on a variety of machines (the same TL0 code). Also shown are the run-times for a C version of naive Matrix Multiply for size 400 x 400.* 161

6.12 *Dynamic run-time on Monsoon for Gamteb and Simple.* 161

Chapter 1

Introduction

1.1 Presentation of Thesis

This thesis studies compilation techniques for implicitly parallel languages, focusing in particular on the extended functional language Id. These languages allow an elegant formulation of a broad class of problems while exposing substantial parallelism. However, their semantics require fine-grain dynamic scheduling and synchronization, making an efficient implementation on conventional parallel machines challenging. This thesis presents sophisticated compilation techniques which improve the run-time behavior of implicitly parallel programs on conventional distributed memory machines, thus minimizing the need for specialized architectures. Specifically, it focuses on compilation techniques that minimize dynamic scheduling and synchronization. This dissertation shows that it is possible to obtain an efficient implementation by partitioning the program into regions that can be scheduled statically as sequential threads and implementing the remaining dynamic scheduling efficiently through compiler-controlled multithreading.

Partitioning the program into sequential threads requires substantial compiler analysis because, unlike in imperative languages, the evaluation order is not specified by the programmer. Care has to be taken to generate threads which obey all dynamic data dependencies and do not lead to deadlock. An interprocedural partitioning algorithm is presented which uses global analysis to form larger threads. Even with this sophisticated partitioning algorithm some dynamic scheduling remains, largely due to asynchronous inter-processor communication. This scheduling is implemented efficiently through compiler-controlled multithreading which focuses on the storage hierarchy. Exposing the scheduling, synchronization, and communication to the compiler enables a multitude of optimizations which are not possible on machines where the scheduling and communication are directly implemented in hardware.

A compiler for Id has been developed along with backends for the CM-5 multiprocessor, J-Machine, and workstations. This experimental platform shows the effectiveness of the partitioning algorithms and compiler-controlled multithreading. In addition, it allows us to study the behavior of implicitly parallel programs on existing distributed memory machines, to identify remaining bottlenecks, and to quantify the value of various architectural features for the execution of these programs.

1.2 Implicitly Parallel Languages

The implicitly parallel languages studied in this thesis, non-strict functional languages, form an attractive basis for parallel computing since, unlike sequential languages, they expose all the parallelism present in the program. This is supported by their clean semantics, which can be expressed using rewrite rules. In side-effect free functional languages the arguments to a function call can be evaluated in parallel. Furthermore, non-strict execution can substantially enhance the parallelism, since functions can start executing before all of the arguments have been provided.

Usually non-strictness is combined with lazy evaluation. Under lazy evaluation an expression is only evaluated if it contributes to the final result. Lazy evaluation decreases the parallelism because the evaluation of an expression is only started after it is known to contribute to the result. Non-strictness can also be combined with eager evaluation. This combination, termed lenient evaluation by Ken Traub [Tra88], exhibits more parallelism than lazy evaluation while retaining much of its expressive power.

Let us take a look at an example. Figure 1.1 shows the top-level function calls and dependencies for the hydrodynamics and heat conduction code called Simple [CHR78, AE88]. This problem can be expressed very naturally in a functional language. The

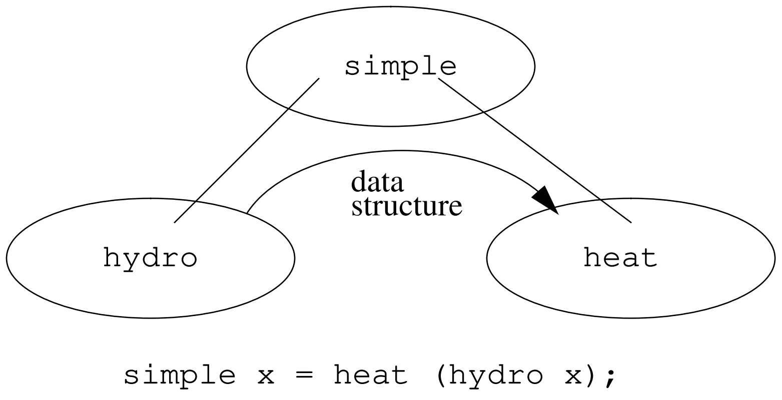


Figure 1.1: *Functional view of the hydrodynamics and heat conduction code Simple.*

algorithm consists of two parts — hydrodynamics and heat conduction.[1] To solve the whole problem `(simple x)`, we first solve the hydrodynamic component `(hydro x)` and then apply the heat conduction function `heat`. The hydrodynamics computation creates a large data structure which is consumed by the heat conduction computation. On a parallel machine we would like to pipeline the execution of both parts so that they can execute in parallel. For this to be possible we need non-strict evaluation, which lets the heat conduction function start executing before its arguments have been fully evaluated. Care has to be taken that the heat conduction function (the consumer of the data structure) does not read parts of the data structure which have not yet been produced. This can be ensured by data structures that automatically synchronize between producer and consumers. If a consumer accesses a data element before it has been produced, the computation dependent on it is deferred and re-scheduled once the value is available. Read accesses to synchronizing data structures may be deferred when an element is not present. Split-phase accesses are required to implement this. In split-phase accesses the request and the response are separate operations. With split-phase accesses the processor can continue working after issuing the request, making it possible to hide the communication latency with computation not dependent on the requested data.

The language Id provides both lenient evaluation and synchronizing data structures, so called I-structures. As we saw, dynamic scheduling may be required for two reasons. First, the non-strict semantics of the language can make it impossible to statically determine the order in which operations execute. Expressions can only be evaluated after all of the argu-

[1]The actual program consists of more parts for which conceptually the same discussion applies.

ments they depend on are available. Second, long latency inter-processor communication makes is necessary to dynamically schedule the computation dependent on the messages, including accesses to synchronizing data structures.

1.3 Architectural Approaches

Unfortunately, dynamic scheduling comes at a high cost on commodity microprocessors, which only support a single thread of control efficiently and incur a high cost for context switching. Therefore, the emerging implicitly parallel languages were accompanied early on by the development of specialized computer architectures, *e.g.*, dataflow machine such as the TTDA [ACI+83], Manchester dataflow machine [GKW85], Epsilon-2 [GH90], ADAM [MF93], EM-4 [SYH+89], and Monsoon [PC90]. These architecture not only support dynamic scheduling and synchronization in hardware but also support fine-grained communication.

At first glance, compilation of non-strict languages for architectures supporting dynamic scheduling and communication directly in hardware seems relatively straightforward. Unfortunately, relying completely on hardware for the scheduling introduces new problems. First, the large amount of parallelism present in the implicitly parallel programs causes excessive resources usage and resource management becomes a critical issue [Cul90, Rug87]. Second, most dataflow machines exhibit low single thread performance, which severely limits the performance for those parts of the programs with little parallelism. Multithreaded machines, such as the HEP [TS88] and Hybrid [Ian88], were developed to combine fast single thread performance with efficient scheduling. Subsequent developments, such as P-RISC [NA89] and TAM [CSS+91], go a step further by formulating abstract machines which can be implemented efficiently on stock hardware, but rely extensively on the compiler to eliminate as much dynamic scheduling as possible by partitioning the program into sequential threads.

This evolution is similar in spirit to other high level languages requiring complex mechanisms, where initial language and hardware developments where closely related. Historical examples include the Burroughs B5000 [Bar61], a stack architecture with displays to support lexical level addressing of ALGOL like languages, Lisp machines with tag hardware to support dynamic operand types [Moo85], and Prolog machines with hardware support for unification [Dob87]. Improvements in compilation techniques made it possible to compile the complex mechanisms either completely away, to decompose them into smaller pieces that could be implemented more efficiently using fast hardware primitives, or to special case them depending on the context. These advances in compilation techniques made the special hardware support less valuable. In most cases, commodity RISC architectures provide an efficient execution vehicle due to the RISC methodology of providing primitives not solutions, relying on the compiler to map complex constructs to the appropriate primitives.

1.4 Compilation Approach

An important observation is that although non-strict languages require dynamic scheduling, they do not require it throughout. Therefore it makes sense to study execution models where groups of instructions are statically scheduled into sequential threads and dynamic scheduling occurs only between threads. These sequential threads can be executed efficiently on commodity microprocessors. To derive these threads it is necessary to expose both forms of dynamic scheduling to the compiler, those due to the non-strictness of the language and those due to inter-processor communication. The optimization goal is to minimize the number of thread switches.

The task of identifying portions of the program that can be scheduled statically and ordered into threads is called partitioning. Partitioning is done using dependence analysis. Two operations can only be placed into the same thread if the compiler can statically determine the order in which these execute and prove that there does not exist a long-latency dependence among them. Communication and the non-strictness of the language impose limits on the size of the threads that can be produced.

The cost of the remaining dynamic scheduling can be reduced by switching to related threads first (*e.g.*, in the same function). This allows high-speed processor state, such as registers or the call frame, to be carried over from one thread to the next. The full price of a context swap is only paid when a swap to an unrelated thread in a different function activation becomes necessary.

Non-strict languages also require fine-grain communication for accessing global data structures and sending arguments and results between function activations residing on different processors. Communication occurs very often, and therefore has to be implemented efficiently, by having the compiler produce all of the code for sending and handling of messages. Other issues which need to be addressed are data layout and work distribution.

1.5 Contributions

The main contribution of this thesis is the development of powerful thread partitioning algorithms, which go substantially beyond what was previously available [Tra88, HDGS93, Sch91]. The algorithms presented in [TCS92] served as a starting point. This thesis extends that work in several ways:

- It presents a new basic block partitioning algorithm, separation constraint partitioning, which is more powerful than iterated partitioning, the previously best known basic block partitioning.

- It shows how separation constraint partitioning can be used successfully as part of the interprocedural partitioning algorithm for the partitioning of call and def site nodes.

- It extends the interprocedural analysis to deal with recursion and mutually dependent call sites which previous analysis could not handle.

- It develops a theoretical framework for proving the correctness of our partitioning approach. Developing the correctness proofs revealed problems with prior partitioning approaches.

- It implements the partitioning algorithms, resulting in a running execution vehicle of Id for workstations and several parallel machines.

- It quantifies the effectiveness of the different partitioning schemes on whole applications.

1.6 Roadmap

The next chapter studies non-strict languages and derives the requirements that these impose on implementations for conventional parallel machines. Chapter 3 presents TAM, a compiler-controlled multithreading approach that efficiently handles dynamic scheduling and communication, and briefly discusses how to compile for it. Chapter 4 focuses on the partitioning algorithm for obtaining the sequential threads and shows how to partition individual functions. Chapter 5 extends this to interprocedural partitioning using global analysis techniques. We show that the compiler has to be conservative in determining the threads to avoid deadlock. Chapter 6 presents experimental results which show the effectiveness of the partitioning algorithms and the compiler-controlled multithreading in dealing with the dynamic scheduling. It also highlights remaining inefficiencies and quantifies the benefit of architectural support in overcoming them. Finally, Chapter 7 contains the summary and conclusion. The proof of correctness for the partitioning algorithm appears in Appendix A.

Chapter 2

Lenient Languages and Parallel Machines

This chapter motivates our focus on lenient functional languages and discusses what is involved in compiling these languages for massively parallel machines. Its goal is to distill the salient features of non-strict languages and to discuss their implications on efficient parallel implementations. For this it is also necessary to develop an understanding of the architecture of existing parallel machines and their performance characteristics.

Functional languages are declarative in nature, meaning that the programmer describes what is to be computed without specifying the actual order in which the computation is performed. The order in which expressions are evaluated depends strongly on the evaluation strategy of the implementation. We discuss three different evaluation strategies that have been proposed for functional languages: strict, lenient, and lazy evaluation. As we will see, the three evaluation strategies differ substantially in their expressiveness, parallelism, and efficiency. For our purposes — a parallel implementation of functional languages — lenient evaluation seems to be the best choice: being non-strict it obtains much of the expressive power of lazy evaluation at a much lower overhead, while exhibiting more parallelism than both strict or lazy evaluation. The language Id is the only language we know of which is based on lenient evaluation.[1]

[1] An ongoing development, parallel Haskell (pH), which integrates many concepts of Id into Haskell, may also be based on lenient evaluation.

Non-strictness may make it necessary for a caller to continue after invoking a function and may result in multiple callees running concurrently. Therefore, lenient evaluation requires fine grain dynamic scheduling and synchronization, and places a strong demand on memory management because the call structure at any point in time forms a tree rather than a stack. In addition, synchronizing data structures provided in Id require tagged heap storage. These issues make an efficient implementation on conventional parallel machines challenging.

Parallel machines impose additional requirements on the implementation of a parallel language. In recent years technological factors have led to a convergence of parallel architectures towards physically distributed memory machines, consisting of many workstation-like nodes connected by a fast communication network. This kind of general purpose MPP architecture is present in our main experimental platform, the Thinking Machines CM-5. Communication and synchronization is expensive on these machines. To produce code which achieves good speedups it is necessary to distribute the work evenly among the processing nodes, to place the data as to minimize the communication, to reduce the communication overhead, and to avoid idling processors during long-latency communication.

The structure of this chapter is as follows. Section 2.1 presents the various layers of the lenient functional language Id. Section 2.2 studies how the evaluation strategy affects expressiveness, parallelism, and efficiency. Section 2.3 summarizes the requirements that lenient languages impose on an implementation, while Section 2.4 studies the additional requirements imposed by parallel machines. Section 2.5 ends the chapter with a summary of how the compiler can deal with these requirements. The compiler can obtain sequential efficiency by partitioning the program into sequential threads and implementing the remaining dynamic scheduling efficiently through compiler controlled multithreading.

2.1 Lenient Functional Languages

Id is a functional language augmented with synchronizing data structures. As shown in Figure 2.1 the language consists of three different layers. The functional core has all of the properties of purely functional languages, including referential transparency and determinacy. Pure functional languages do not handle data structures efficiently and cannot express non-deterministic computation. Id solves these problems by adding two non-functional extensions, I-structure and M-structures.[2]. I-structures are write-once data structures which provide automatic synchronization between the producer and consumers. Being write-once, I-structures preserve determinacy but sacrifice referential transparency. Non-deterministic computation can be expressed with M-structures, mutable data structures that provide support for atomic updates.

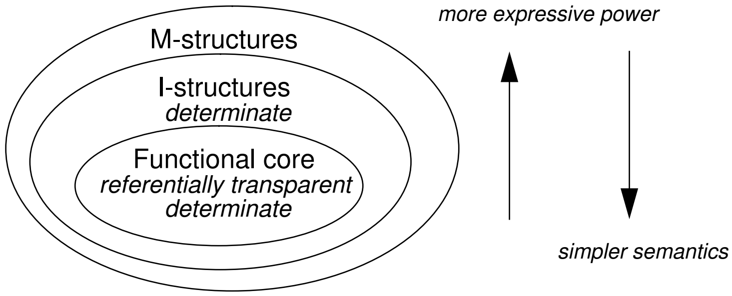


Figure 2.1: *The three layers of the lenient language Id (from [Nikhil93]).*

We now discuss the various layers of the language in more detail and show how they are affected by the evaluation strategy.

2.1.1 Purely Functional Core

The core of Id is a purely functional language with simple semantics and is similar in spirit to other functional languages, such as LML, Haskell, and Miranda. It is a strongly typed language, with overloading and polymorphic functions. The language provides the standard set of basic operators, recursive letrec bindings, user defined type constructors, pattern matching, and higher order functions. The language is side-effect free and therefore referentially transparent, implying that any two occurrences of an expression denote the same value, and optimizations such as common subexpression elimination can always be applied.

Purely functional languages have a strong mathematical foundation in the λ-calculus. The λ-calculus defines a set of conversion rules for evaluating expressions. Applying these rules transforms an expression into its normal form. The λ-calculus is commonly

[2]The name I-structure stands for "Incremental"-structure since the elements can be filled in incrementally. M-structure is derived from "Mutable"-structure because every element can be updated several times

used to define the operational and denotational semantics of a functional language. The operational semantics of a function describe its algorithm, *i.e.*, the sequence of operations required to compute the result. The denotational semantics describe the intended meaning of a function, *i.e.*, its mapping of arguments to results.

Id is a language with *non-strict* (denotational) semantics. A function f of arity n is said to be *strict* in its i-th argument if $f x_1 \ldots x_{i-1} \perp x_{i+1} \ldots x_n = \perp$ for all $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$. The symbol $\perp$ (bottom) represents no information and is used to indicate non-terminating computation. With non-strict semantics it is possible to define functions which return a result even if one of the arguments diverges. This makes it possible to create cyclic data structures, since portions of the result of a function call can be used as arguments to that call. As we will see, non-strictness increases the expressiveness of the language but requires a more flexible evaluation strategy than strict evaluation, which evaluates all arguments before calling a function. Before discussing evaluation strategies we briefly present the two non-functional layers of Id.

2.1.2 I-structures and M-structures

Early functional languages had difficulties expressing scientific computation requiring arrays or other large data structures efficiently. These problems have been alleviated by array comprehensions, similar in spirit to list comprehensions. These are purely functional constructs and define an array by defining each element. Successfully implemented in Id, they have been folded back into modern functional languages such as Haskell. With array comprehensions, elements of the array can be defined in terms of other elements. One limitation is that in order to be purely functional, the contents of the array has to be defined completely within the array comprehension. This ensures that referential transparency is preserved. There are certain problems where this is not sufficient [ANP87], therefore, Id also provides two non-functional data structures, I-structures and M-structures.

I-structures are write-once data structures, which separate the creation of the structure from the definition of its elements. There are two operations defined on I-structures: I-fetch, which takes an I-structure and an index and returns the value of the corresponding element; and I-store, which writes a value into an I-structure location. Like functional arrays, I-structures provide efficient indexed accesses to the individual elements. In addition, I-structures provide synchronization between the producer and consumer on an element by element basis. At the implementation level, each element of an I-structure can be in one of three states: *empty*, *deferred*, or *full*. At the beginning, all elements of an I-structure are empty. When an I-store writes a value into a location, its state is changed to full, and subsequent I-fetches return the value of the element. An I-store to a location which is full results in a run-time error. I-fetches occurring before the I-store are deferred and are serviced once the I-store is executed. Depending on the target architecture, this mechanism can be implemented either directly in hardware through presence bits, or emulated in software. I-structures are not purely functional, since they lose referential transparency. Any function obtaining a pointer to an I-structure may store into it. On the other hand, determinacy is preserved because each location can be stored into at most once.

M-structures can be used to express non-determinacy, by defining data structures which can be updated. Again, the creation of a data structure is separated from the operations on it. The basic operations are *put* and *take*. Each structure element again has three states: *empty*, *deferred*, and *full*. A put on an empty location writes the value into it and marks it full. A take on a full location returns its value and marks it empty. Takes on an empty location defer, and a subsequent put services only one of the deferred takes. This mechanism can be used to efficiently implement data structures that are updated multiple times, such as accumulators. Updates occur atomically and can be used for example to implement complex fetch and ops. We now continue with a discussion of possible evaluation strategies.

2.2 Evaluation Strategies

The two most widely used evaluation strategies for functional languages are strict and lazy evaluation. We compare them to lenient evaluation, the evaluation strategy used for Id. The evaluation strategies influence the order in which the evaluation of expressions proceeds and, therefore, describe the operational semantics of a language.

The three evaluation strategies differ in how they handle the evaluation of type constructors and function calls. To evaluate a function call $f e_1 \ldots e_n$,

Strict evaluation first evaluates all of the argument expressions e_1 to e_n, and then evaluates the function body, passing in the evaluated arguments.

Lenient evaluation starts the evaluation of the function body f in parallel with the evaluation of all of the argument expressions e_1 to e_n, evaluating each only as far as the data dependencies permit.

Lazy evaluation starts the evaluation of the function body passing the arguments in unevaluated form.[3]

The evaluation rules for constants, arithmetic primitives, and conditionals are the same for the three evaluation strategies. The evaluation of a constant yields that constant. Arithmetic operators are strict in their arguments, so they evaluate the arguments before performing the operation. The evaluation of a conditional, *if* e_1 *then* e_2 *else* e_3, first evaluates the predicate e_1. Depending on the result either e_2 or e_3 is then evaluated. Our three evaluation strategies treat the conditional conservatively: they do not start evaluating e_2 and e_3 speculatively before knowing the value of the predicate.

We see that under strict evaluation it is impossible to call a function with some, but not all of the arguments, obtain a result and pass the result or parts of it back into the function. On the other hand, this scenario is possible under both lenient and lazy evaluation; therefore, both implement non-strict semantics. Lazy evaluation ensures that only expressions which contribute to the final answer are evaluated. The evaluation is demand driven. In contrast, strict and lenient evaluation evaluate all expressions (with the exception of those inside the arms of conditionals). We also say that they use eager evaluation, which is data driven because an expression can be evaluated as soon as its data inputs are available.

As Table 2.1 shows, the three evaluation strategies differ substantially in their expressiveness, the amount of parallelism they expose, the implementation overhead, and overall efficiency. This difference is discussed in the next subsections.

2.2.1 Expressiveness

This subsection compares the expressiveness of the three evaluation schemes. The additional expressiveness of non-strictness is important, because programs can be written

[3]Strict evaluation is often also termed call-by-value, while lazy evaluation is called call-by-need. For the λ-calculus the terms applicative-order and normal-order evaluation are used.

Evaluation strategy	Strict	Lenient	Lazy
Expressiveness	low	high	very high
Parallelism	high	very high	low
Overhead	low	high	very high
Speculative computation	some	some	none
Overall efficiency	good	very good	poor

Table 2.1: *Comparison of the three evaluation strategies.*

in a more efficient way. It is easy to see that any program which returns an answer under strict evaluation also does so under lenient and lazy evaluation. Likewise, any program which returns the answer under lenient evaluation also does so under lazy evaluation. As shown in Figure 2.2 the expressiveness of the three evaluation strategies forms a three level hierarchy. Lenient evaluation obtains non-strict semantics and therefore gives a programmer more expressive power because circular data structures can be created. For example, the recursive binding `a = (cons 1 a)` denotes a simple cyclic list, and `b = (cons 2 (hd b))` denotes a tuple containing the same element. Lazy evaluation, in addition, can handle infinite data structures, as long as only a finite part is accessed.

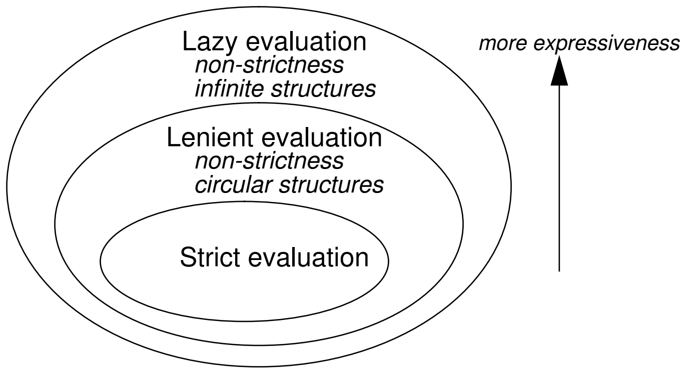


Figure 2.2: *The three evaluation strategies and their expressiveness.*

To illustrate this difference in expressiveness, we use a very simple example, the Fibonacci function. Figure 2.3 shows the code for several versions of the Fibonacci function, each illustrating an important point. For readers unfamiliar with the Id syntax, curly braces enclose mutually recursive variable bindings with the keyword `in` denoting the result. The first definition is the standard doubly recursive definition. It only requires

```
% strict version   O(2^n)
def fib n = if n < 2 then 1
               else fib (n-1) + fib (n-2);

% lenient version   O(n^2)
def fib n = { li = (1 : 1 : (fib_list li 2 n));
             in nth n li };
def fib_list l i n =            % list of Fibonacci numbers from i to n
    if i > n then nil
          else (((nth (i-1) l) + (nth (i-2) l)) : (fib_list l (i+1) n));

% lazy version   O(n^2)
def fib n = { li = (1 : 1 : (fib_list li 2))
             in nth n li };
def fib_list l i =              % infinite list of Fibonacci numbers from i
       (((nth (i-1) l) + (nth (i-2) l)) : (fib_list l (i+1)));

% efficient strict version   O(n)
def fib n = if n < 2 then 1
               else fib_aux 1 1 2 n;
def fib_aux r1 r2 i n = if i == n then r1+r2
                           else fib_aux r2 (r1+r2) (i+1) n;

% non-strict array comprehension   O(n)
def fib n = { A = { array (0,n) of
                     | [0] = 1
                     | [1] = 1
                     | [i] = A[i-1] + A[i-2] || i <- 2 to n};
             in A[n]};
```

Figure 2.3: *Different versions of the Fibonacci function, showing the difference in expressive power of strict, lenient, and lazy evaluation.*

strict evaluation: the arguments can be completely evaluated before starting the recursive calls. The complexity is exponential in n, $\mathcal{O}(2^n)$. The next definition shows how non-strictness can be used to increase efficiency. It uses dynamic programming to keep track of all Fibonacci values produced and thereby ensures that each value is only produced once. This implementation uses a list instead of a table and its complexity is $\mathcal{O}(n^2)$. A list of the first n Fibonacci numbers is created and then the n-th element of that list is returned. This function definition cannot execute under strict evaluation because the function that produces the list of Fibonacci numbers from 2 to n also requires this list as one of its arguments. Lenient evaluation correctly executes this non-strict program.

With lazy evaluation we can further simplify this example as shown in the next definition. Now we just produce the infinite list of all Fibonacci numbers and return the n-th element. Lazy evaluation ensures that the infinite list is only being produced up to the n-th element. In effect, lazy evaluation needs to implicitly maintain the control structure which is explicitly present in the previous definition. With lenient evaluation this version would fail to terminate because it would try to construct the entire infinite list. With fair scheduling, lenient evaluation would eventually produce the answer (but it would still run forever as it would try to produce the rest of the infinite list not required for the answer). Therefore, from a theoretical standpoint it has the same denotational semantics as lazy evaluation but different termination properties. In practice though, even with fair scheduling, non-terminating computation can result in an arbitrarily large slowdown, especially if the computational resources required by the non-terminating computation grows much faster than those for the useful computation. Under lenient evaluation it is therefore the programmer's responsibility to avoid non-terminating computation. One way to achieve this is to insert explicit delay statements into the code [Hen80, Hel89].

In the last two definitions dynamic programming was used only in a very limited form. The recursive function only needs to look up the previous two Fibonacci numbers to compute the next element. Instead of using a list containing all Fibonacci numbers we could also have passed the last two in explicitly, as shown in the next definition. This reduces the complexity to $\mathcal{O}(n)$. The final definition for the Fibonacci function uses an array comprehension (analogous to list comprehension) which provides constant time lookup and therefore alleviates the problem of expensive lookups using lists. Non-strictness is required, as the table of Fibonacci numbers is recursively defined in term of its own elements. This definition also has the same computational complexity of $\mathcal{O}(n)$.

Either lazy or lenient evaluation can be used for non-strict array comprehensions. For I-structures or M-structures on the other hand, lazy evaluation is not sufficient. The reason is that the allocation of a structure and definition of its elements are separated. A read of an empty element cannot identify the producer and therefore cannot start its evaluation. It is necessary to evaluate all functions and expressions which obtain a pointer to this structure, as any of those may produce the value and store it into the location.

As our small examples show, non-strictness provides a powerful tool to the programmer, since it permits defining data structures recursively and creating circular data structures. This can improve the time and space requirements of a program substantially, by avoiding recomputing results or traversing data structures multiple times [Bir84]. It can be used to naturally express the recursive dependencies that arise when implementing parsers [Hut92], attribute grammars [Joh87a], cooperating processes [FH88], or physical simulations, such as successive over-relaxation [ANP87].

2.2.2 Speculative Computation

Lazy evaluation only evaluates expressions which contribute to the final answer. It therefore only performs computation that is really required. Strict and lenient on the other hand might do unnecessary computation since they are eager. As the following simple example illustrates they perform computation speculatively, *i.e.*, do work which is ultimately wasted.

```
def cond p x y = if p then x+1 else y+2;
def main = cond true (fib 5) (fib 10);
```

The function `cond` takes three arguments: a predicate, an argument used if the predicate is true, and an argument used if the predicate is false. When invoking this function from `main` with strict or lenient evaluation both arguments are evaluated although only one of them is actually used. Lazy evaluation avoids this extra work.

2.2.3 Parallelism

The evaluation strategy can have a strong impact on the amount of parallelism. Consider the first doubly recursive definition of the function `fib` from Figure 2.3. Here the two calls can be executed in parallel because they are completely independent. This parallelism is exploited under strict and lenient evaluation, as both evaluate expressions eagerly. As soon as the evaluation reaches the "else" arm of the conditional, both function calls can execute in parallel. For this example even a lazy evaluator can exploit the parallelism because the addition is strict in both of its operands. Therefore, before evaluating the addition, both recursive calls can be evaluated in parallel. In general, non-strict operators and non-strict functions may be present, in which case lazy evaluation cannot exploit the inherent parallelism in the evaluation of multiple independent arguments because it first needs to show that these are actually required for the final answer. In fact, this is one reason why strictness analysis plays such a crucial role for lazy evaluation.

In addition to this simple form of independent function call parallelism, lenient evaluation can exploit producer-consumer parallelism. Consider the function `flat` which produces a list of the leaves of a binary tree using accumulation lists.

```
def flat tree acc = if (leaf tree) then (cons tree acc)
                       else flat (left tree) (flat (right tree) acc);
```

This example does not require non-strict semantics; therefore it can be executed with any of the three evaluation strategies. However, under strict evaluation this code exhibits little parallelism. The right subtree is completely flattened before starting with the left subtree. And this holds for every level, so the tree is flattened sequentially.

Under lenient execution the flattening of the two subtrees can be pipelined. Flattening of the right subtree can be started simultaneously with the flattening of the left subtree. After the result list for the right subtree has been produced it is fed into the function working on the left subtree. In effect, the entire list is constructed in parallel. This example shows that overlapping the consumer and producer can yield more than just a constant factor increase in parallelism [Nik91]. Simulations for the dataflow architecture TTDA show that with lenient evaluation the critical path on a full binary tree of depth 10 consists of 250 time steps with a maximum parallelism of 1776 and an average parallelism of 266 instructions (assuming the resources are available). If executed strictly, the critical path would grow to 26,650 time steps, with a maximum parallelism of 4 instructions.

Lazy evaluation flattens the subtrees in the opposite order from strict evaluation. Lazy evaluation first flattens the left subtree, and then starts working on the right subtree.[4] Therefore lazy evaluation also does not exhibit any parallelism.

In summary, we have seen that both lenient and lazy evaluation obtain non-strictness, *i.e.*, they may start the evaluation of a function body before evaluating the arguments. When non-strictness is combined with eager evaluation, as in lenient evaluation, the parallelism is increased substantially, because all arguments can be evaluated in parallel with the function body. Of course, this increase in parallelism only occurs if the program does not require non-strictness. For programs which require non-strictness, dependencies from portions of the result back to some of the arguments exist. These dependencies force a sequentialization in the evaluation process, *i.e.*, some arguments can only be evaluated after the part of the result they depend on has been produced. Obviously, if such dependencies exist they may force a complete sequential evaluation even under eager evaluation, and lenient evaluation may not exhibit more parallelism than lazy evaluation.

2.2.4 Evaluation Overhead

As should already be clear, the evaluation strategies differ substantially in their overhead. The biggest differences are in the costs for argument passing and dynamic scheduling.

Strict evaluation has the lowest overhead. All arguments can be evaluated before calling a function, and therefore can be passed by value. Furthermore, it is possible to statically schedule all of the computation inside a function and produce efficient sequential code. The compilation of strict functional languages is therefore similar in spirit to well understood implementations of sequential languages.[5]

Lazy evaluation has a high overhead because it requires arguments to be passed in unevaluated form, unless it can be shown that they contribute to the final result. There are

[4] Of course, lazy evaluation would only produce as much of the flattened list as is actually required.

[5] Additional issues arise due to the use of higher-order functions, partial applications, parallelism, and single assignment limitations. Optimizations of continuations [Kra88, App92] and update analysis [HB85] are important techniques to deal with them.

several ways to implement lazy evaluation: using explicit delay and force [Hen80], graph reduction [PvE93], or abstract machines [Joh87b]. It is even possible to implement lazy evaluation with eager evaluators by making the control flow of demands explicit [PA85]. Although these schemes seem to be quite different at first glance, they produce very similar code after optimizations [Tra88, Pey92].

Delaying the evaluation of an argument essentially requires creating a thunk (closure) for corresponding argument expression. This thunk has to capture the addressing environment and a pointer to the code to be executed. When the argument is needed, the thunk is forced and updated with the result to ensure that it is evaluated only once, even when used multiple times. It should be clear from this description that this mechanism — producing the thunk, forcing and updating it, and finally reclaiming its heap space — is fairly complex and expensive to implement.

The overhead of lenient evaluation falls between that of lazy and strict evaluation. In the next section we present examples showing this overhead and discuss what requirements this places on an implementation.

2.3 Implementation Requirements for Lenient Evaluation

The overhead for lenient evaluation is less than for lazy evaluation, but still high. As we will see this is mainly due to fine grain synchronization, dynamic scheduling, complex memory management, and synchronizing data structures.

2.3.1 Synchronization and Dynamic Scheduling

Under lenient evaluation it is not necessary to create thunks for argument expressions, but it still may not be possible to statically determine the order in which expressions within a function are evaluated. An expression can only be evaluated after all data inputs are available. Because of non-strictness the order in which data inputs are available may depend on the context in which an expression appears. Therefore, when evaluating an expression, a lenient evaluator must first synchronize on all data inputs and then dynamically schedule the evaluation of the expression.

The examples in Figure 2.4 illustrate several important points, especially the need

```
% Example 1: non-strict function calls
def f x y = (x*x, y+y);
def g z = { a,b = (f z a) in b};      % computes (z*z)+(z*z) = 2 z^2
def h z = { a,b = (f b z) in a};      % computes (z+z)*(z+z) = 4 z^2

% Example 2: non-strict conditional
def f p z = { a,b,c = if p then (y,z,x) else (z,x,y);
              x = a+a;
              y = b*b
             in c};
def g z = f true z;
def h z = f false z;

% Example 3: non-strict function calls (callees affect caller)
def f1 x y z = (y,z,x);
def f2 x y z = (z,x,y);
def f func z = { a,b,c = (func x y z);
                 x = a+a;
                 y = b*b
                in c};
def g z = f f1 z;
def h z = f f2 z;

% Example 4: non-strict I-structures
def f A k l m n = { A[k] = A[m] * A[m];
                    A[l] = A[n] + A[n]};
def g z = { A = I_array (1,3);
            A[1] = z;
            _ = (f A 2 3 1 2);
           in A[3] };
def h z = { A = I_array (1,3);
            A[1] = z;
            _ = (f A 3 2 2 1);
           in A[3] };

% Example 5: synchronization and dynamic scheduling
def f x y z = (x+y,z*z);
def g z = { a,b = (f z z a) in b};
def h z = { a,b = (f b b z) in a};
```

Figure 2.4: *Examples illustrating the need for dynamic scheduling under non-strictness. The functions g always compute $(z * z) + (z * z) = 2z^2$ while the functions h compute $(z + z) * (z + z) = 4z^2$. In all cases a single addition and single multiplication are executed; in the functions g the multiplication occurs before the addition, in the functions h they execute in the reverse order.*

for dynamic scheduling, using a trivial, somewhat contrived computation. In the first example, the function f takes two arguments, x and y, and returns two results, $x * x$ and $y+y$. Inside the function f there is no dependence between the multiplication and addition. Therefore they can be put in any order when compiling this function for traditional strict evaluation. This is not true under non-strict evaluation. In our example, the function f is used in two different contexts which require non-strictness. In the function g the argument z is given as the first argument to the function f, while the second argument to f is taken from its first result. This requires that f first compute $z * z$, return this result, and then compute $(z * z) + (z * z) = 2z^2$. We see that in this case the multiplication is executed before the addition. In the function h just the opposite occurs, the second result of the function f is fed back in as the first argument. Here $z + z$ is computed first and then $(z + z) * (z + z) = 4z^2$. Now the addition is executed before the multiplication. The multiplication and the addition have to be scheduled independently. It is impossible to put them together into a single thread and order them statically. What is really surprising is that the scheduling is independent of the respective data values for the arguments, it just depends on the order in which the arguments arrive, *i.e.*, on the context in which the function is used and how results are fed back in as arguments.

The next example, taken from [Tra91], shows that this form of dynamic scheduling does not only occur across function calls, but also inside functions. In this example a single conditional steers the evaluation of three variables, a, b, and c. If the predicate is true then b gets the value of z, y the value $z * z$, a obtains the same value, and finally x and the result becomes $z * z + z * z = 2z^2$. In this case the multiplication is executed before the addition. If the predicate is false the variables are evaluated in a different order. First the variable a is bound the the argument z, x and b evaluate to $z * z$, and finally y and the result evaluate to $(z + z) * (z + z) = 4z^2$. Now the addition is performed before the multiplication. Again, we see that both the addition and multiplication have to be scheduled dynamically. It is surprising again that this occurs even though the operations appear outside of the scope of the conditional. The conditional affects the order in which the values a, b, and c are available.

The reader may think that, unlike in the previous example, the scheduling in this example is at least data dependent, since it is influenced by the conditional and therefore depends on the value of the predicate. While this observation is correct, we can obtain precisely the same behavior without conditionals as shown in example 3. Here the conditional is replaced with a call to a function taking three arguments and returning three results. The function which is called is determined by the argument *func*; it is $f1$ in the case of g and $f2$ in the case of h. These two functions, $f1$ and $f2$ do not perform any computation, they just shuffle the results around and thereby affect the order in which the addition and multiplication in the caller get executed. This example illustrates that not only can the caller affect the order in which operations get executed in the callee, but also the callee can affect the order in the caller. In general, it is the whole context in which a function appears which determines the order, whereby the whole context we mean the whole call tree. This example also shows the duality between conditionals and function calls via function variables. A conditional can be viewed as a call site, where one of two functions is called depending on the predicate. These two functions contain the code of the *then* side and *else* side of the conditional, respectively. In fact, this is the view we are going to take for our thread partitioning, as discussed later in Section 4.2.

The fourth example shows that dynamic scheduling is also required for I-structures. The function f takes five arguments, an I-structure A and four indices k, l, m, and n. It fetches an element from $A[m]$, multiplies this with itself and stores the result into location $A[k]$. The function also fetches from location $A[n]$, adds this element with itself and stores it into location $A[l]$. Figure 2.5 shows the dependencies for the body of function f, and what happens for the two cases where $k = n$ and $l = m$.

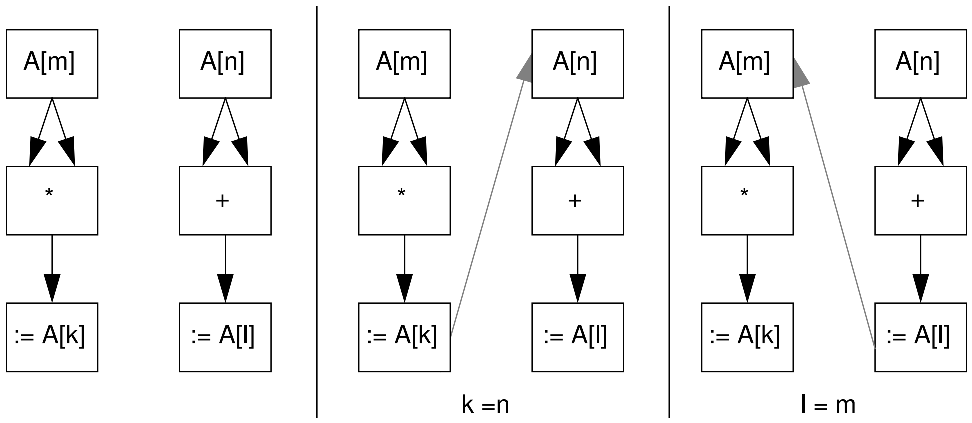


Figure 2.5: *The dependencies for function f and two possible scenarios, $k = n$ and $l = m$.*

If $k = n$, as it is the case when f is called from function g, then the store into location

$A[k]$ defines the value which is fetched from $A[n]$. Therefore there exists a dependence from the store to the fetch as indicated by the dashed line. Thus, the multiplication has to be executed before the addition. Note that this dependence is not directly present in the function, it is established through the synchronizing I-structure. Should the fetch occur before the store, it would get deferred until the store happens. If $l = m$ the operations would execute in the reverse order. In general, it is not known which function fills in an I-structure. The consumer of an I-structure element cannot name its producer. Therefore if a fetch defers, it is necessary to continue with the evaluation of any expression which does not depend on the fetch, even if the computation is in a different function activation. Any one of those could produce the value which gets stored into the location. This example illustrates that dependencies have to unravel at run time and that these may not only travel through arguments, results and internal call sites, but also through I-structure accesses. These synchronizing I-structures require tagged heap storage which is expensive without special hardware support.

The examples presented so far, with the exception of the data structure accesses, only require dynamic scheduling by not synchronization, because in these simple examples each operation depends only on a single input. The need for synchronization is illustrated by our last example which is similar to the first example. The function f takes three arguments x, y, and z, and computes two results $x + y$ and $z * z$. In order to compute the first result the first two arguments have to be available, therefore, synchronization between the two arguments has to occur before the operation can be scheduled.

2.3.2 Forms of Dynamic Scheduling

How does dynamic scheduling due to non-strictness differ from conditional branches or subroutine calls of sequential languages? As summarized in Table 2.2, the biggest differences between these forms of dynamic scheduling are when the scheduling occurs, and what memory model and primitives are required.

In some sense, conditional branches are also a form of dynamic scheduling, as the compiler cannot statically tell which code is executed next. In the case of a simple conditional branch, the dynamic scheduling is data dependent. Only the program counter has to be changed. Therefore the target code of the branch is still executing in the same environment

Form	Occurrence	Memory requirements	Primitive
Conditional	data dependent	environment frame	branch
Function call	always	stack	indirect jump, stack pointer
Non-strictness	context dependent	heap	indirect jump, continuation queue

Table 2.2: *Forms of dynamic scheduling and their memory requirements.*

(call frame).

The dynamic scheduling required for function calls and returns is slightly more complicated because both the instruction pointer and frame (stack) pointer have to he changed. Usually, the caller can name the callee and directly jump to it. The callee on the other hand, usually does not know the caller and therefore requires an indirect jump for the return. Because the caller suspends while the callee is executing a stack is sufficient to manage the call frames.

With non-strictness, a computation which suspends also cannot name the computation which has to be started next. As our previous examples showed, the dynamic scheduling is not data dependent, it is context dependent, *i.e.*, the execution order may be determined by the surrounding context from which a function is invoked, independently of data values. As discussed below, a stack is not sufficient, because a tree of frames is formed. A heap is needed to manage the call frames. In the case of I-structures, it is not necessarily the caller which must be continued if a child suspends.

2.3.3 Memory Management

In addition to synchronization and dynamic scheduling, heap management incurs a high overhead with lenient evaluation. It is not as high as with lazy evaluation where the heap may have to keep an environment for every unevaluated argument. With lenient evaluation we know that all expressions are eventually evaluated and it is therefore possible to associate a single environment, the activation frame, with every function call. But unlike sequential and strict languages, these activation frames cannot be mapped onto a single stack. The caller does not necessarily suspend after invoking a function because non-strictness may make it necessary for the caller and multiple callees to be active at the same time.[6] Therefore, the call structure at any point in time forms a tree rather than a stack. This can be expensive to manage and requires a heap.

2.3.4 Summary

In this section we presented the salient features of the non-strict language Id and compared its evaluation strategy, lenient evaluation to evaluation strategies used in other functional languages, strict and lazy evaluation. We saw that the evaluation strategy has a large effect on the expressiveness, parallelism, and overhead.

Expressiveness: Non-strictness, as present in lenient and lazy evaluation, significantly increases the expressiveness. The programmer can create circular data structures and define data structures recursively in terms of their own elements. This results in more efficient programs — both in space and time requirements [Hug88, Bir84, Joh87a] In addition, lazy evaluation provides the control structure to handle infinite data structures. Using explicit delays, programmers can obtain the same benefit under lenient evaluation. Lazy evaluation is the only strategy which completely avoids speculative computation by only evaluating expressions which contribute to the final result. The other two schemes do computation eagerly, thereby putting the burden on the programmer to avoid infinite or speculative computation.

Parallelism: Lenient evaluation results in the largest amount of parallelism. This is very important for a parallel implementation. It has been shown that for many problems an implementation with lazy evaluation cannot identify enough parallelism without user annotations or speculative computation.

Overhead: Strict evaluation has the lowest overhead for passing argument and dynamic scheduling. The overhead for lazy evaluation can be prohibitively expensive without strictness analysis, while the overhead for lenient evaluation falls between that of

[6] Independent of non-strictness, the same behavior also arises on parallel machines, when a caller forks of a child for parallel execution. The set of requirements that arise from parallelism arise under non-strictness even when executed on a sequential machine.

strict and lazy evaluation. As later chapters of this thesis show, this overhead can be reduced to a tolerable level with partitioning.

The overall efficiency of an implementation depends on a combination of all of the above issues, and also how well the compiler can deal with them. I believe that lenient evaluation offers the best tradeoff for a parallel implementation of functional languages. As we saw, it is actually required if the functional core is extended with synchronizing data structures. This is why we focus on the lenient language Id.

Having made that choice, we studied in detail the overhead of lenient evaluation, the requirements this imposes on an implementation, and how we can address these requirements. We found that is necessary to support the following:

Dynamic scheduling: The non-strict semantics of the language allows results of functions be fed back in as arguments. Therefore it may not be possible to statically order the operations within a function. The execution order may depend on the context in which a function is used, and dynamic scheduling is required.

Synchronization: Before scheduling the evaluation of an expression all required data inputs have to be available. It is necessary to synchronize on these data.

Heap management: The call structure at any point in time forms a tree and therefore cannot be mapped onto a stack. A heap is required to hold the call frames.

Tagged data structures: I-structures provide synchronization between producers and consumers. Each element is write-once. Tags, or some small state machine, have to be associated with every element to keep track of its state: empty, deferred, or full. When a fetch defers, the corresponding continuation has to be associated with the element. These data structures have to be kept on a heap because their life time may be longer than the function that allocates them.

The question is how to implement the dynamic scheduling, synchronization, heap management, and tagged data structures most efficiently? Before answering that question let us briefly study our compilation target, massively parallel machines, and understand the implementation requirements they impose.

2.4 Massively Parallel Machines

As discussed in the previous chapter, special purpose hardware, such as dataflow and parallel graph reduction machines, were specifically built to support the execution of functional languages. Even though these special purpose machines efficiently support mechanisms required by these languages, they have a difficult time competing against the technological thrust behind commodity microprocessors. The main technological reasons for this are the dramatic increase in microprocessor performance and memory capacity, combined with their excellent cost/performance.

The same technological reasons are also leading to a convergence of architectures for massively parallel machines (MPPs). Modern MPPs consist of many processing nodes which communicate through a dedicated interconnection network. Each node is essentially a full workstation-like computer, containing a powerful processor and substantial amount of memory. This structure is also present in our main experimental platform, the CM-5. Because of the significant cost of each node, typical MPPs have less than a few thousand nodes. At the hardware level, communication is implemented through sending of messages. This is even true for large shared memory machines. In the past, communication performance has always lagged behind processing performance, and interconnection networks have limited bandwidth and significant latency.

To derive the compilation requirements for MPPs is it necessary to develop an understanding of their performance characteristics. These can be captured quite accurately through a couple of simple parameters describing the communication latency, overhead, and bandwidth.

2.4.1 LogP Model

The *LogP* model [CKP+93] of parallel computation is based on distributed-memory multiprocessors in which processors communicate by point-to-point messages. The model specifies the performance characteristics of the interconnection network, but does not describe the structure of the network.

The main parameters of the model are:

L**:** an upper bound on the latency, or delay, incurred in communicating a message con-

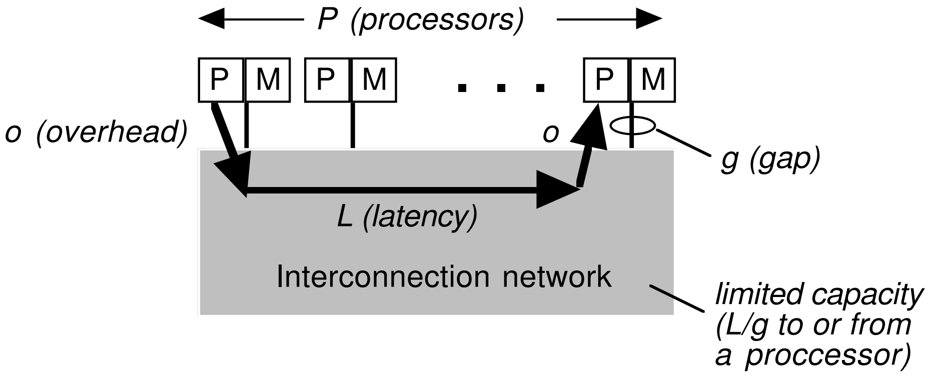


Figure 2.6: *The LogP model describes an abstract machine configuration in terms of four performance parameters:* L*, the latency experienced in each communication event,* o*, the overhead experienced by the sending and receiving processors for each communication event,* g*, the gap between successive sends or successive receives by a processor, and* P*, the number of processors/memory modules.*

taining a word (or small number of words) from its source module to its target module.

o: the overhead, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time the processor cannot perform other operations.

g: the gap, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g corresponds to the available per-processor communication bandwidth.

P: the number of processor/memory modules.

The model is asynchronous, *i.e.*, processors work asynchronously and the latency experienced by any message is unpredictable, but is bounded above by L. Because of variations in latency, the messages directed to a given target module may not arrive in the same order as they are sent. The key metrics are the maximum time and the maximum space used by any processor.

2.4.2 Compilation Requirements

To produce code which achieves good speedups on parallel machines it is necessary to distribute the work evenly among the processing nodes, place the data as to minimize the communication, reduce the communication overhead, and avoid idling processors during long latency communication. At the same time, sequential efficiency on every processor is important. Issues that need to be addressed are.

Sequential efficiency: Since massively parallel machines are built out of commodity microprocessors, good single thread performance is essential because commodity microprocessors only support a single thread of execution efficiently. Therefore all of the issues discussed in the previous section, dynamic scheduling, synchronization, heap management, and tagged data structures, have to be addressed. In addition, the compiler has to make efficient use of the storage hierarchy.

Identifying parallelism: Execution on multiple processors requires that the program be decomposed into tasks which can be executed in parallel. For sequential languages it can be difficult to identify such tasks. Usually loop parallelism or simple forms of task parallelism are exploited. Identifying the parallelism in implicitly parallel languages is easier, since it is not hidden. In Id programs parallelism is present at multiple levels: function call and loop parallelism can be used to distribute work across processors, while instruction level parallelism within individual functions can be used to hide communication latencies. The compiler produces an activation frame for every function and partitions each function into a collection of threads.

Work and data distribution: A good distribution of work and data improves parallel performance. The goal is to keep the load balanced while minimizing communication by coordinating the data placement and work distribution. For very structured problems this mapping can be predetermined by the programmer or compiler [HKT92]. In other cases run-time support or hardware support for load balancing or caching of data structures is required [LLG+90]. In our case, the non-strictness of the language makes analyzing the data structures very hard. In addition, many of the problems we study are irregular in nature, such that this mapping cannot be determined automatically by the compiler. The problem of work and data distribution by the compiler is not specifically addressed in this thesis, our implementation relies on support from the run-time system.

Communication: Communication is required because function activations and data structures are distributed across multiple processing nodes. Accesses to data structures or sending of arguments and results may involve communication. Unfortunately, this communication is fine-grained. Accesses to data structures result in small messages, and as we saw in the previous section, multiple arguments and results may have to be sent independently because of non-strictness. Fine-grain communication incurs a large overhead on existing massively parallel machines. The compiler can reduce this overhead by producing a specific handler for every possible message so that no run-time message parsing is required, *i.e.*, the message format and message processing are coded in the instruction sequence of the handler [vECGS92].

Latency hiding: Communication latencies for remote request can be substantial, even if the actual hardware latency, L, is not. The reason is that a request has to be serviced by the remote processor. Therefore the latency visible from the requesting processor between sending of a request and obtaining a reply is at least $2L + 2o$, possibly much longer, depending on the actual work required to service the request, as well as network and destination contention. During this time the requesting processor should avoid idling and work on something that does not depend on the reply. Split-phase accesses, where the request and the response are two separate operations, are essential to overlapping communication and computation. Even with split-phase accesses good processor utilization is difficult to achieve because commodity processors do not support fast context switching. Our solution is to compile every function into multiple threads such that switching among these threads is fast. An expensive context switch only occurs when switching to another function activation.

Synchronization: Communication also requires synchronization with the ongoing computation. If a processor idles during requests, synchronization is easy because there can only be a single outstanding communication event. The processor just waits until the reply returns and then continues executing. If processors switch to some other work during long latency communication, synchronization becomes more complicated. If the implementation supports a large number of outstanding communication events, it must provide a large enough synchronization name space [AI87]. With I-structure accesses, multiple fetches may defer before one is finally satisfied. Independently of the lenient language, fine grain synchronization may also be required because the network may re-order messages.

Dynamic scheduling: If processors continue working after issuing requests, dynamic scheduling is required to execute the computation dependent on the responses. The computation may be decomposed into multiple threads of control which are scheduled dynamically.

Storage hierarchy: Commodity microprocessors have a multi-level storage hierarchy consisting of a set of registers, on-chip cache, 2nd-level cache, and main memory. This hierarchy has to be exploited for efficient code. Sequential code usually exhibits good spatial and temporal locality. Multithreaded execution required for hiding latency requests may decrease the locality. Implementing this dynamic multithreaded scheduling while maintaining locality and making good use of the storage hierarchy is a major challenge.

Heap management: In a parallel call, a function activation frame is allocated on another processor and the current processor continues executing the caller. Thus, a parallel call tree is formed which is harder to manage than a conventional call stack. Non-strictness alone already requires a tree of activation frames, which has to be managed on the heap. To obtain efficiency, the compiler makes all allocation and releases of frames explicit. Allocation always occurs on a frame basis and there are no implicit storage resources such as token queues. This can be directly used to implement parallel calls, without much additional complexity.

It is surprising to see that many of the issues required for implementations of non-strict languages (dynamic scheduling, synchronization, and heap management) are also present when producing code for parallel machines, independent of the source language. Therefore, most of the compilation and run-time techniques developed to support the efficient execution of lenient languages are also applicable to parallel implementations of other languages.

A clear understanding of the performance characteristics of the parallel machine can help improve efficiency and make the right tradeoff between the various compilation

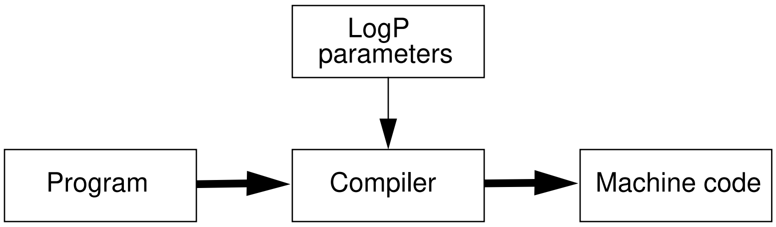


Figure 2.7: *Possible compilation model under LogP (not used in our Id implementation).*

issues. One could envision a compiler, as shown in Figure 2.7, which adapts the code depending on the machine characteristics. For example, on machines where the communication overhead is very expensive, less parallel work could be spawned or data distributed in larger blocks to reduce communication. Similarly, on machines with high latencies, more efforts would have to be invested to use instruction level parallelism to hide this latency. While this approach is certainly desirable, our source language is much too dynamic and our parallel programs much too irregular to attempt this. Instead we rely on the run-time system to make some of these decisions (*e.g.*, when to spawn off work). It should be noted though, that other parallel implementations of functional languages use the former approach, most notably SISAL implementations [BCFO91]. SISAL limits itself to completely static computation structures, and, therefore, the compiler can statically partition the program for parallel execution [SH86]. This partitioning is strongly affected by the communication requirements of the program and by the performance characteristics of the machine.

2.5 Compilation Challenge

In the previous section we have analyzed the implementation requirements of lenient languages for parallel machines. Now we present compilation and run-time techniques to implement them efficiently. In the next three chapters we focus on two of those techniques, compiler controlled multithreading and partitioning a program into sequential threads.

We saw that lenient languages and parallel machines require fine-grained dynamic scheduling, synchronization, low overhead communication, parallel heap management, tagged data structures, latency tolerance, load balancing, good data distribution, and storage hierarchy locality. For all of these issues, the compiler should first try to compile away as much as possible, and then implement whatever remains as efficiently as possible.

For example, in the case of communication, a good work distribution and data distribution eliminates much of the communication. The remaining communication is implemented efficiently by grouping multiple small messages together to reduce the overhead, and by having the compiler produce code that directly sends and handles messages. Likewise, with dynamic scheduling, grouping multiple instructions statically into threads reduces the scheduling and synchronization; scheduling between related threads first, implements the remaining dynamic scheduling more efficiently, while improving locality and providing a means for tolerating latency. Finally, avoiding heap allocation of small objects, such as continuation queues, temporary message buffers, and synchronization variables, makes the memory management faster. To obtain a good implementation of lenient languages on parallel machines, all of these techniques have to be applied. We now discuss these in more detail.

2.5.1 Partitioning

The most important compilation step for obtaining sequential efficiency is partitioning the program into sequential threads, because this is what commodity microprocessors support efficiently. The order of the instructions within a thread is determined statically, while dynamic scheduling occurs between threads. This approach to compiling non-strict functional languages was first proposed by Traub [Tra88, Tra89]. The goal is to maximize the length of the threads. Unfortunately, this criteria alone does not reflect the quality of a partitioning, because threads in Traub's framework are allowed to suspend. Every time a thread suspends, a different thread is scheduled, essentially requiring a context switch. This dynamic scheduling among threads is very expensive without special support for multithreading. Therefore, a better optimization criteria than the size of the threads is the number of thread switches.

Our framework captures this goal by disallowing thread suspension.[7] A thread forms the basic unit of work: once scheduled it runs until completion. Thus, possible suspension points form thread boundaries. This suspension arises from the non-strictness of the

[7]This definition of threads differs substantially from the notion of "operating systems" threads used in threads packages. Note also that Traub's framework only addresses the ordering requirements within each thread, while our framework deals both with ordering and dynamic scheduling.

language and may occur at synchronization points or at function call boundaries. These threads therefore capture precisely the dynamic scheduling required for reasons of non-strictness. The other advantage is that this also nicely integrates suspension due to long latency communication, by disallowing threads which issue remote request to also wait for the reply. This gives us a framework to deal both with the language constraints and machine constraints on thread generation in a uniform manner.

Extensive compiler analysis is needed to maximize the size of threads. The compiler must ensure that it does not group together operations which require dynamic scheduling. Of course, we are only interested in correctly partitioning a program for all of those inputs for which the unpartitioned program does not deadlock. With non-strictness partitioning is hard, because the compiler must assume that any result of a function could be fed back in as an argument. The partitioning can be improved with interprocedural analysis which can derive that some of the potential feedback dependencies cannot arise.

The examples presented earlier showed that even with the best analysis techniques a compiler may not be able to statically schedule all operations of a function. Non-strictness may make it necessary to make some ordering decisions at run-time. Therefore, dynamic scheduling is still required. Independent of non-strictness, the same is also true if long latency communication is involved. Thus, the two limiting factors on the thread size are non-strict semantics and interprocessor communication. Compiler-controlled multithreading implements the remaining scheduling efficiently.

2.5.2 Compiler Controlled Multithreading

In compiling lenient languages for parallel machines, the goal is not simply to minimize the number of thread switches, but to minimize the total cost of dynamic scheduling and synchronization while tolerating latency on remote reference and making effective use of processor resources, such as registers and cache bandwidth. To partition correctly, we must expose the scheduling, synchronization, and communication to the compiler. By making static ordering decisions, the compiler tries to eliminate these as much as possible. The goal is to specialize the remaining scheduling, synchronization, and communication, using the cheapest primitives available. (Hopefully, the cheapest form is also the one that occurs most frequently.) Our approach is based on the observation that the scheduling and communication primitives can be closely tied to the storage hierarchy and memory management.

2.5.3 Summary

This chapter started with the observation that lenient evaluation as present in Id forms the best basis for a parallel implementation of functional languages. It then showed that there are considerable challenges in compiling lenient languages for conventional massively parallel machines, and presented an overview of techniques to deal with them. The next chapter presents compiler controlled multithreading in more detail, while the three chapters thereafter focus on the partitioning algorithms and their theoretical framework.

Chapter 3

Compiler-Controlled Multithreading

This chapter presents the salient features of the Threaded Abstract Machine (TAM) as a compilation target for non-strict languages. TAM was specifically designed to deal efficiently with the dynamic scheduling required by non-strictness and inter-processor communication [CSS$^+$91, CGSvE93]. As shown below, TAM exposes the scheduling, synchronization, communication, and storage hierarchy to the compiler in a structured form. By making the cost of these operations explicit, the compiler can optimize against them and use the least expensive primitive. Section 3.2 discusses the various issues involved in compiling a language such as Id to TAM, including the representation of parallelism, frame management, and communication. One of the most important issues — partitioning the program into regions which can be scheduled into sequential threads — is not addressed in this chapter. The discussion is deferred to the next two chapters which focus solely on partitioning. This chapter finishes with an overview of the remaining steps which are involved in generating sequential threads, once the program is partitioned. Instructions within each thread must be linearized, operands assigned to frameslots and registers, and the control transfer and synchronization across threads is made explicit.

3.1 The Threaded Abstract Machine

This section describes the TAM execution model. TAM exposes communication, synchronization, and scheduling so that the compiler can optimize for important special cases.

3.1.1 Program Structure

A TAM program is a collection of *code-blocks*, where each code-block consists of several *threads* and *inlets*. Typically a code-block represents a function or loop body in the high level language program. Threads represent the computational body of the code-block, and inlets specify the interface through which the code-block receives messages, including arguments, return values, and heap-access responses. Each inlet is a message handler to receive a specific message. Threads and inlets are straight-line sequences of instructions, which execute from beginning to end without suspension or branching.

Each invocation of a code-block has an associated *activation frame*. As suggested by Figure 3.1, the frame provides storage for the local variables, much like a conventional stack frame. It also provides the resources required for synchronization and scheduling of threads, as explained below.

3.1.2 Storage Hierarchy

The TAM data storage hierarchy is composed of three levels: registers, local frames, and heap-allocated structures. Registers are the least expensive to access; however, their content is short-lived. Accessing locals from the frame is more expensive, essentially a local memory access; however, they persist throughout an invocation. Placement of data between frame and registers is under compiler control. Frame storage is assumed to be distributed over processors, but each frame is local to some processor and only accessed from that processor. Work is distributed over processors on a frame invocation basis. Interframe communication is potentially interprocessor communication and is realized by sending values to inlets relative to the target frame.

Heap storage contains objects that are not local to a code-block invocation, including statically and dynamically allocated arrays. Heap storage is assumed to be distributed over processors and is accessed through split-phase fetch and store operations, described below. In addition to data, each heap location holds a small number of tag bits, providing element-by-element synchronization as required for I-structures, M-structures, and thunks [ANP87, Hel89].

3.1.3 Execution Model

Invoking a code-block involves allocating a frame, depositing arguments into it, and initiating execution of the code-block relative to the newly allocated frame. The caller does not suspend upon invoking a child code-block, so it may have multiple concurrent children. Thus, the dynamic call structure forms a tree, rather than a stack, represented by a tree of frames. This tree is distributed over the processors, with frames as the basic unit of work distribution. Finer grain parallelism is not wasted; it is used to maintain high processor utilization. Thread parallelism within a frame is used to tolerate communication latency and instruction parallelism within a thread is used to tolerate processor pipeline latency.

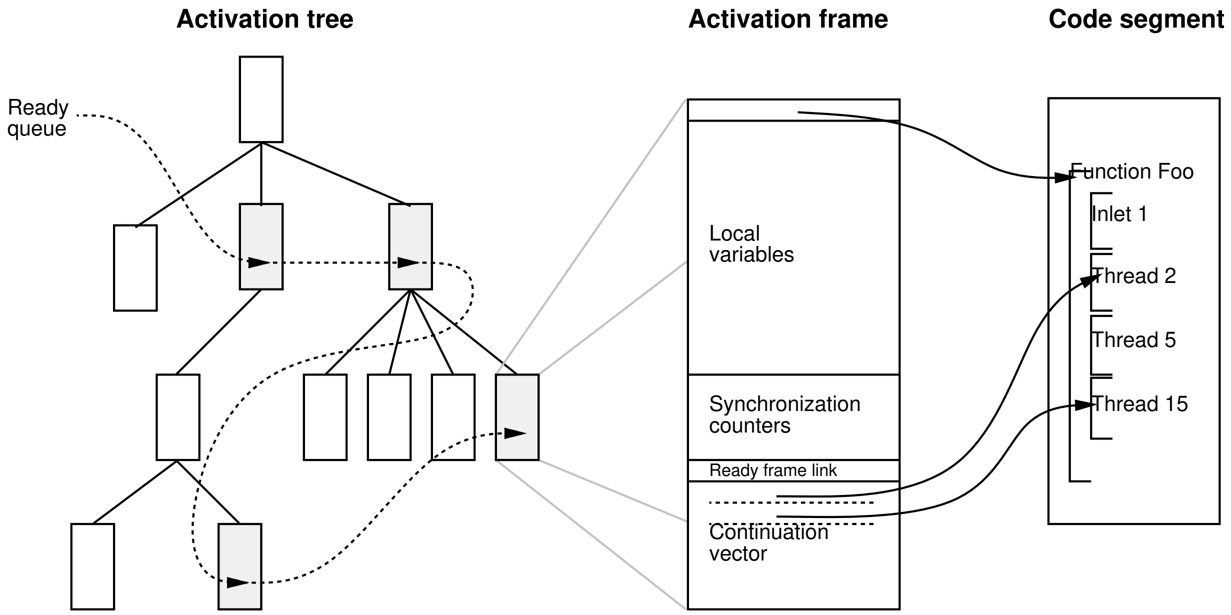


Figure 3.1: *TAM activation tree and embedded scheduling queue. For each function call, an activation frame is allocated. Each frame, in addition to holding all local variables, contains counters used to synchronize threads and inlets, and provides space for the* continuation vector — *the addresses of all currently enabled threads of the activation. On each processor, all frames holding enabled threads are linked into a ready queue. Maintaining the scheduling queue within the activation keeps costs low: enabling a thread simply consists of pushing its instruction address into the continuation vector and sometimes linking the frame into the ready queue. Scheduling the next thread within the same activation is simply a pop-jump.*

Initiating execution of a code-block means enabling threads of computation. Each of the arguments to a code-block potentially enables a distinct thread. As shown in Figure 3.1, a processor contains a linked list of ready frames (the activation tree is usually much larger than the number of processors), each with several enabled threads in a region of the frame called the *continuation vector*. When a frame is scheduled, threads are executed from its continuation vector until none remain. The last thread schedules the next ready frame. Thus, the frame defines a unit of scheduling, called a *quantum*, consisting of the consecutive execution of several threads. This scheduling policy enhances locality by concentrating on a single frame as long as possible.

Instructions in a thread include the usual computational operations on registers and local variables in the current frame. The basic control flow operation is the instruction FORK, which to enables another thread to execute in the current quantum. The SWITCH operation conditionally forks one of two threads. Threads are also enabled as a result of message arrivals. Precedence between threads, *i.e.*, data dependences and control dependences, are enforced using synchronization counters within the frame. *Synchronizing threads* have an associated *entry count* which is decremented by forks and posts of the thread. The thread is enabled when the count reaches zero. Each thread is terminated by a STOP instruction, which causes the next enabled thread in the current frame to be executed.

TAM threads are self-scheduled; there is no implicit dispatch loop in the model. Thus, the compiler can control the scheduling by how it chooses to generate forks, posts, and entry counts. There is also no implicit saving and restoring of state, so the compiler manages storage in conjunction with the thread scheduling that it specifies. Since threads do not suspend, values that are local to a thread clearly can be kept in registers. In addition, whenever the compiler can determine that a collection of threads always executes in a single quantum, it can allocate values accessed by these threads to registers. Quanta may be much larger than what static analysis could determine, because several non-strict arguments to a function may arrive together or because accesses to the global heap return quickly. Since the frame switch is performed by compiler generated threads, it is possible to take advantage of this dynamic behavior by allocating values to registers based on expected quantum sizes and saving them if an unexpected frame switch occurs.

Compared to dataflow execution models, TAM simplifies the resource management required to support arbitrary logical parallelism. A single storage resource, activation frames, that is naturally managed by the compiler as part of the call/return linkage represents the parallel control state of the program, including local variables, synchronization counters, and the scheduling queue. TAM embodies a simple two-level scheduling hierarchy that reflects the underlying storage hierarchy of the machine. Parallelism is exploited at several levels to minimize idle cycles while maximizing the effectiveness of processor registers and cache storage.

3.1.4 Efficient Communication

Communication between frames is performed by sending messages. A message is sent to a specific inlet of a particular activation, identified by its frame address. The compiler produces a specific handler for every possible message so that no run-time message parsing is required, *i.e.*, the message format and message processing are coded in the instruction sequence of the handler. An inlet receives the data of the message, stores it into specific slots in the associated frame, and enables specific threads relative to that frame. Enabling a thread from an inlet, POST, is distinct from enabling one from a thread, FORK, and has a different optimization goal. The FORK enables computation that is closely related to the current processor state and attempts to maximize the coupling between the two threads. The POST may enable computation that is unrelated to the current processor state, so it tries to affect that state as little as possible. In addition, the POST is responsible for entering the frame into the scheduling data structure if the target frame had no enabled threads. Inlets may preempt threads, but they may not preempt other inlets.

The SEND operation packs a number of data values into a message and sends it to the inlet of the target activation. Execution then proceeds with the instructions following the SEND. When the message is received, processing of the current thread is interrupted and the receiving inlet is executed. After completing the inlet, processing continues with the interrupted thread.

Accessing the global heap does not cause the processor to stall, rather it is treated as a special form of message communication. A request is sent to the memory module containing the accessed location while threads continue to execute. The request specifies the frame and inlet that will handle the response. If the response returns during the issuing quantum, the inlet integrates the message into the on-going computation by depositing the value in a frame or register and enabling a thread. However, if a different frame is active when the response returns, the inlet deposits the value into the inactive frame and posts a thread in that frame without disturbing the register usage of the currently active frame. The global heap supports synchronization on an element-by-element basis, as with I-structures. Thus, there are two sources of latency in global accesses. A hardware communication latency occurs if the accessed element is remote to the issuing processor and, regardless of placement, a synchronization latency occurs if the accessed element is not present, causing the request to be deferred. Using split-phase operations for heap accesses enables the compiler to produce latency tolerant code.

3.2 Compiling to TAM

The overall goal in compiling to TAM is to produce code that obtains processor efficiency and locality, yet is latency tolerant. TAM exposes the cost of scheduling, synchronization, and communication. This gives the compiler a clear optimization goal and allows it to map the various constructs of the parallel language to the best suited TAM primitives. On the other hand, TAM places the responsibility for correctly resolving several issues, such as management of frames, ordering of threads, and usage of local storage on the compiler. This section discusses the key aspects of the compilation process from Id down to TAM.[1]

3.2.1 Representation of Parallelism

As discussed in the previous chapter, parallelism in Id is present at various levels. At the highest level, function calls or multiple iterations of a loop can execute in parallel. Within a function, independent expressions can execute in parallel. All of these forms of parallelism have their correspondence in TAM. The coarsest grain of parallelism is represented in TAM by frames, which can be distributed over processors. Finer grain parallelism within a frame is represented by threads, which can be used to mask communication latency. Lastly, instruction level parallelism can be exploited within a thread. The compiler must manage the parallelism in the program by mapping it to the appropriate TAM level.

[1] Although the source language for our compiler is Id, the TAM parallel execution model is well suited for implementing other parallel languages.

A parallel function call can be directly represented using TAM's frame mechanism. At the time of the call, a frame is allocated on another processor, the arguments are sent to that frame, the callee starts working on the frame and the caller continues in parallel with other work. Once the callee finishes it sends the results back to the caller, which then schedules the computation dependent on the results.

Loop parallelism can also be represented using the frame mechanism. In this case, a set of frames each holding the local variables of a different iteration of the loop body may be allocated on a set of processors. The assignment of iterations to frames can be addressed by a variety of policies. A very general form of parallel loop structure, called k-bounded loops [CA88], is used in compiling Id. In this scheme, the amount of parallelism, *i.e.*, the number of frames, is determined at the time the loop is invoked, possibly depending on values within the program. The loop builds a ring of k frames and cycles through them. Each iteration detects its own completion and sends a signal to the previous frame indicating that the frame is ready for the next iteration.

When allocating a frame, it must either be allocated locally or on another processor. TAM provides mechanisms to control the placement of frames, but does not dictate how to use it. Thus, the compiler or run-time system must allocate frames in a manner that provides adequate load balancing. For highly irregular parallel problems it is difficult for the compiler to determine the mapping statically, so dynamic load balancing techniques provided by the run-time system are needed.

3.2.2 Frame Storage Management

The representation of dynamic parallelism presents a fundamental storage management challenge. For efficiency, TAM requires all allocation and releases of frames to be explicit. Allocation always occurs on a frame basis; there are no implicit storage resources such as token queues. The size of a frame is fixed at the time of allocation.

The management of frames is typically integrated with the calling convention. In a sequential language, arguments and results are deposited in predefined locations on the stack, or passed in registers. In TAM, argument and result passing is represented in terms of inter-frame communication. The caller sends arguments to predefined inlets of the callee and the callee sends results back to inlets specified by the caller. Two additional arguments are passed to the callee: the parent frame pointer and the return inlet number. If tail call optimization is performed, the caller can pass it's own return frame pointer and inlet directly to the callee. The Id compiler augments each function with code to detect the completion of all computation, so that the frame is released by the last thread in the code-block. K-bounded loops detect completion of each iteration and include additional code to release the ring of frames when the entire loop finishes.

3.2.3 Communication

Sharing of information and coordination of activity among portions of the program are represented in TAM via sends to inlets and heap requests delivered to inlets. This encourages a latency tolerant style of code-generation. When a remote access is initiated, the computation continues; the response will be received asynchronously and will enable whatever computation depends on it. The execution model places no limit on the number of outstanding messages, although architectural factors such as latency, overhead, or available bandwidth may introduce a practical limit [CSvE93]. The communication model is efficient because no buffering is required in the communication layer. Storage to receive the message is pre-allocated in the frame so that the inlet can move the data directly from the network interface to the frame [vECGS92].

The Id compiler generates a specialized message handler for each heap reference, function argument, and result in the program text. This specialization reduces the message size, since the message format is encoded in the inlet, and reduces the cost of message handling, since no parsing is required. Currently, remote references to global data structures are handled by generic remote reference message handlers. These handlers are executed on the processor on which the accessed data element resides. The handlers perform synchronization if needed, access the data element, and send the reply back. In our compilation scheme for Id, the only case where a message handler may need buffering is when a read must be deferred: a continuation consisting of the requesting processor, inlet, and frame is enqueued on the element so that the read can be completed when the data is written.

There are various techniques the compiler can employ to make the code latency tolerant. By issuing several remote requests in the same thread, the latency of multiple requests can be overlapped. By issuing remote requests as early as possible, the latency can be overlapped with computation not dependent on the reply. This non-speculative form of prefetching can be achieved by pulling remote references as far up in a thread as possible, and by ensuring that threads with remote references get scheduled first. In both cases there must be adequate parallelism available to overlap the communication latency with computation. If the program does not not have enough parallelism, techniques such as loop unrolling may be applied to introduce it. In general, the more parallelism a program exploits, the more storage resources it needs. Thus a tradeoff has to be struck between good latency tolerance and storage requirements [CSvE93].

3.2.4 Partitioning

One of the most important compilation issues is partitioning the program into regions which can be mapped into sequential threads. This process occurs in two steps. First, the instructions of a function are grouped into disjoint subsets. In a second step, every one of those subsets is linearized and the control operations and operands are made explicit. Partitioning the program into the subsets is non-trivial due to non-strictness. The next two chapters focus solely on partitioning, while the remainder of this chapter discusses the steps which are required for producing sequential threads from the subsets derived by the partitioning.

3.3 Thread Generation

Internally the compiler represents the program by a dataflow graph, where nodes represent individual operations and arcs connecting the nodes describe data dependencies. The partitioning algorithm groups this dataflow graph into disjoint subsets. After partitioning, the nodes within each of the subsets are still in a partial order. To produce sequential threads, these subsets must be linearized, arcs connecting the nodes must be associated with registers and frame slots, and control transfer and synchronization between threads must be made explicit. This is done by the following steps: instruction scheduling, lifetime and quantum analysis, frame slot and register assignment, redundant arc elimination, fork insertion and entry count initialization, and thread ordering.

Before describing these steps in more detail we present a small example to illustrate them.

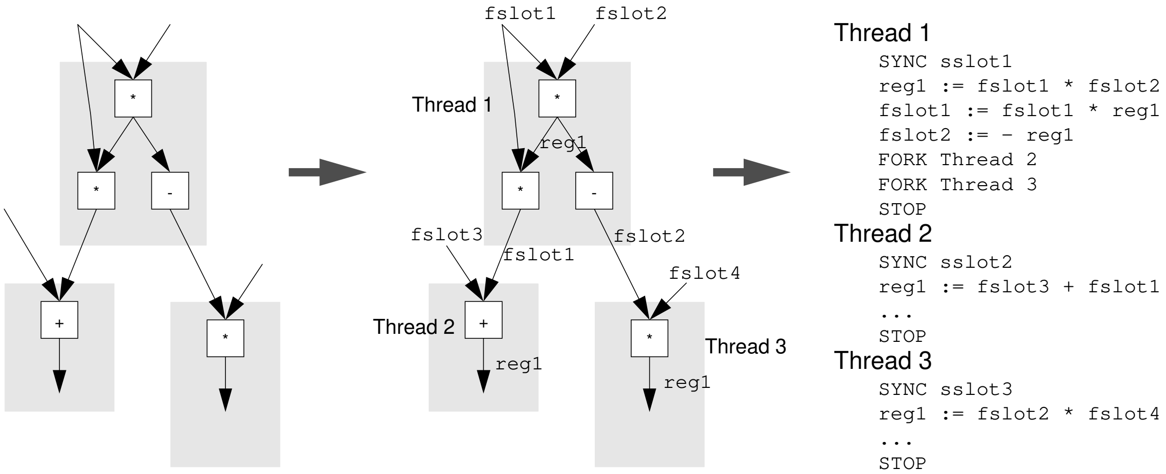


Figure 3.2: *Thread generation: the partitioned dataflow graph is first translated into a labeled dataflow graph, where all arcs representing operands are assigned to registers and frameslots. In addition, a linear ordering of all instructions within each thread is derived, the threads are ordered, and the control transfer between them is made explicit. The translation from a labeled dataflow graph into threaded code is then straightforward.*

The example, given in Figure 3.2, shows on the left a small dataflow graph where the partitioning algorithms identified three threads as indicated by the shaded regions. This graph has to be transformed into the threaded code shown to the right. First, the partitioned dataflow graph is translated into a labeled dataflow graph, where all operands and control transfers are made explicit. A linear ordering of the instructions of each thread is derived which obeys all data dependencies. Every arc representing an operand is assigned to a frame slot or register. Operands crossing synchronization points and possible suspension points are stored in the frame, while values whose lifetime is confined to a single thread can be kept in registers. The goal of register and frameslot assignment is to minimize the required storage resources. In our example we can reuse `reg1` several times, as well as `fslot1` and `fslot2`, assuming that the operands to the multiplications in Thread 1 are not used somewhere else. Finally, the control transfer and synchronization between threads is made explicit by ordering the threads and introducing fork and synchronization operations. Translating the labeled dataflow graph into threaded code is then straightforward.

3.3.1 Instruction Scheduling

Initially, the subsets determined by the partitioning process are still in a partial order and must be linearized. Linearization affects the lifetime of values and thus has a strong influence on the register and frame slot assignment. In addition, instruction level parallelism can be used to hide communication latency. Instruction scheduling depends on specifics of the target machine architecture such as pipeline structure and register availability. For a superscalar machine, for example, it might desirable to put independent instructions close to each other so that they can be scheduled together. We currently use a heuristic that tries to schedule heap accesses early and attempts to minimize the overlap of the lifetimes of values. Combined with graph coloring for register assignment, this tends to give latency tolerance while minimizing the storage used.

3.3.2 Lifetime Analysis

The compiler has to determine whether a value can be stored in a register or whether it needs to be placed into a frame slot. This is done by analyzing the lifetime of a value. The lifetime of a result is defined as the time from when it is produced until the last consumer uses it. The lifetimes of individual variables can easily be determined from the arcs of the dataflow graph. If all targets are in the same thread as the source node, the lifetime is limited to a single thread and the value can safely be placed in a register. However, under the TAM scheduling paradigm once a frame is made resident, threads are executed until no enabled threads remain. A result can safely be stored in a register if no code-block context swap can occur in its lifetime: values can be carried in registers across threads as long as the threads execute in the same quantum. We use quantum analysis to determine statically what threads will be executed together.

Dynamically, quanta sizes may actually be much larger than that predicted by static analysis. This might be the case, for example, because a potentially very long latency I-fetch operation has been served rapidly, so that the threads that use this value can also be executed before leaving the code-block activation. A speculative register assignment would try to guess these larger scheduling quanta and then assign registers accordingly. In the case that the “guess” turns out to be wrong, the registers need to be saved before leaving the current activation and restored on re-entry. TAM provides support for this by allowing the compiler to specify threads that will be executed at the end and the beginning of a code-block activation. In this way, a compromise can be struck between fast context switching and utilization of processor resources on a case-by-case basis. This is supported in the machine language for TAM, but is currently not exploited by the compiler.

3.3.3 Frame Slot and Register Assignment

In order to reduce the frame size and the number of registers required, we need to reuse frame slots and registers for operands that are guaranteed to have disjoint lifetimes. This is done by constructing and coloring an appropriate interference graph. A graph is constructed that contains a node for each operand and an edge between two nodes if their lifetimes overlap. Coloring this graph so that all vertices connected by an edge have different colors gives a valid register and frame slot assignment. We use a simple graph coloring heuristic, since finding the minimum number is NP-complete.

Whereas in sequential languages the uncertainty in interference arises because of multiple assignments under unpredictable control paths, in compiling Id the uncertainty arises because of dynamic scheduling. The compiler can only determine a partial order among the threads. The compiler thus has to make worst case assumptions when computing the lifetime of variables which cross thread boundary.

3.3.4 Synchronization

The compiler has to ensure that all forms of data and control dependencies are enforced. Due to parallel execution and non-blocking remote communication, more explicit synchronization is needed than for sequential languages. For example, a computational thread has to synchronize with the reply of a remote request before using its value. Similarly, a thread that needs values produced by two other threads, has to synchronize on both threads having completed. TAM provides two ways to enforce data and control dependencies. Inside a thread, the linear ordering of instructions determines their execution order. Across threads, the use of explicit synchronization counters and control primitives, such as FORK, SWITCH, and POST, specifies the order in which the threads are executed and allows computation to synchronize with communication. The compiler orders the instructions within a thread and produces the control transfer operations across threads such that all data and control dependencies are enforced.

A synchronization counter is associated with each synchronizing thread. The compiler determines the number of times a thread is forked, posted, or switched to, which constitutes the initial value of the corresponding synchronization counter [Sch91]. The counter must be initialized before the first fork or post is attempted.

Synchronization cost can be reduced by eliminating redundant control transfers between threads.[2] Eliminating such transfers decreases the entry count of the target thread. A control transfer from one thread to another is *redundant* if there exists another unconditional control path between the two threads. A trivial case of this is where multiple transfers cross from one thread to another. Redundant control transfers can be eliminated, since they carry no new synchronization information. This transformation is shown in Figure 3.3.

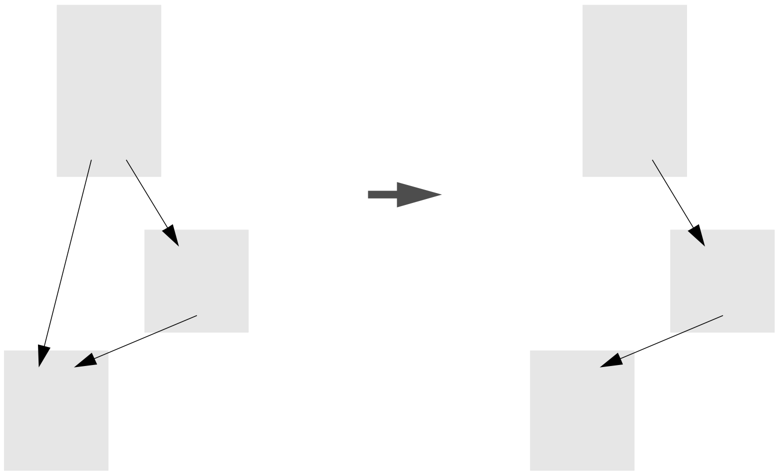


Figure 3.3: *Redundant control arc elimination rule.*

3.3.5 Thread Ordering

TAM provides a control transfer mechanism which forks threads for later execution. By placing threads contiguously, a fork and a stop can often be replaced by a simple fall-through. The translator from TAM to target machine code moves the last fork or switch

[2] This optimization is not possible for pure dataflow machines as control transfer and operands are combined into a single mechanism: the dataflow token. Separating the two, as under TAM, gives the opportunity to optimize them separately.

in a thread to the very bottom and replaces it by a branch.[3] If the target is the next thread, this becomes a fall-through. Thread ordering is determined by a depth-first search starting from root threads triggered only by inlets. An unnumbered adjacent thread is selected to be the successor using the following priorities: fork to unsynchronizing thread, switch to unsynchronizing thread, fork to synchronizing thread, switch to synchronizing thread, fork or switch to thread which is also forked by an inlet. This scheme tries to ensure that the fall-through will be executed immediately after the completion of the thread.

3.3.6 Summary

The primary goal of this section was to describe the details of thread generation. Although it was presented here in the context of TAM, thread generation for other threaded execution models is very similar. We now describe how partitioning derives the subsets of instructions which can be mapped into statically scheduled threads.

[3]This optimization is different for multithreaded processors. There it is desirable to execute a fork as soon as possible, so that the processor can pick up the thread and work on it in parallel.

Chapter 4

Intraprocedural Partitioning

This chapter describes the algorithms for partitioning non-strict functions into groups of instructions, each of which can be mapped onto a sequential thread. As discussed in the previous two chapters, these threads are completely statically scheduled and are needed for an efficient execution on conventional instruction set processors. Any dynamic scheduling required by non-strictness or long latency communication occurs only between threads.

The structure of this chapter is as follows. Section 4.1 starts by explaining the goal of the partitioning algorithm to produce non-suspending sequential threads of maximal length. Section 4.2 discusses the structured dataflow graphs used to represent the programs. Section 4.3 presents various basic block partitioning algorithms: simple, dataflow, dependence set, demand set, and iterated partitioning. These are used to develop an intuition of what is required for a correct partitioning before discussing the most powerful basic block algorithm, separation constraint partitioning in Section 4.4. In Chapter 5 partitioning is extended with interprocedural analysis, which refines the inlet and outlet annotations and thereby results in larger threads to be formed. Appendix A formally proves the correctness of the algorithms presented here.

4.1 Problem Statement

The goal of partitioning is to minimize the dynamic scheduling required by sequential and parallel execution of a non-strict program. This is achieved by partitioning the program into sequential threads of maximal length. Each of the threads is scheduled statically and dynamic scheduling occurs only between threads. As discussed in Chapter 2, the partitioning algorithm has to comply with the dynamic scheduling requirements imposed by the non-strict semantics and long latency communication.

More formally, following a definition given in [PT91], a *thread* is a subset of the instructions comprising a procedure body, such that:

1. a compile-time instruction ordering can be determined for the thread which is valid for all contexts in which the containing procedure can be invoked, and
2. once the first instruction in a thread is executed, it is always possible to execute each of the remaining instructions in the compile-time ordering, without pause, interruption, or execution of instructions from other threads.

Our solution to the partitioning problem is best described in terms of the framework we use for addressing the issues. As a starting point for the partitioning algorithm, we shall consider programs represented as structured dataflow graphs which are used in the Id compiler [Tra86]. A *structured dataflow graph* consists of a collection of *basic blocks*, one for each function and each arm of a conditional, and *interfaces* which describe how the basic blocks relate to one another. Each basic block is represented by an acyclic dataflow graph. The task of a partitioning algorithm is to take a structured dataflow graph and partition the vertices of each basic block into non-overlapping regions such that each subset can be mapped into a non-suspending sequential thread. A correct partitioning has to avoid introducing circular dependencies between threads, including both static cycles within basic blocks and dynamic cycles across basic blocks. In addition, it must ensure that requests and responses to split-phase operations are mapped to different threads.

As eluded to in the previous chapter, deriving the threads is done in two steps. First, the nodes of each basic block are partitioned into disjoint subsets. Then the instructions of each subset are linearized. Because threads do not suspend, the instruction ordering chosen for one thread does not influence the ordering for another thread and it is possible to determine the instruction ordering of the threads independently. Since the basic blocks are acyclic, any topological ordering will do. The partitioning algorithms presented here only derive the subsets of vertices and leave the actual ordering of instructions within each thread to a later stage of the compiler.[1] Even without the instruction ordering, we shall refer to each subset of vertices simply as a thread.

4.2 Program Representation

The programs to be partitioned are represented in form of a *structured dataflow graph*. Similar intermediate representations are commonly found in optimizing compilers [FOW87, BP89, SG85]. A structured dataflow graph consists of a collection of basic block and interfaces. Typically, each function and each arm of a conditional of the program is represented by a separate basic block. A basic block is a directed acyclic dataflow graph, where the nodes represent the operations, while the edges describe their data dependencies. Interfaces specify how the basic blocks relate, *i.e.*, how the basic blocks call one another.

4.2.1 Basic Blocks

A basic block is represented as a dataflow graph, a directed acyclic graph describing the operations and data dependencies of the instructions comprising a control region. The vertices of the dataflow graph describe the instructions, while the edges capture their data dependencies. These dependencies are present in every context in which the procedure can be invoked, and are therefore called certain dependencies. We distinguish two kind of certain dependencies: *direct* and *indirect*. Direct dependencies are dependencies whose scope is limited to a single basic block, while indirect dependencies may involve nodes of other basic blocks. Direct dependencies are represented by straight edges and indirect dependencies are represented by squiggly edges. Nodes connected by an indirect dependence have to reside in different threads. In addition to certain dependencies, the dataflow graph uses inlet and outlet annotations to represent potential dependencies.

[1]This is also desirable, because a good ordering may depend on specifics of the machine architecture, *e.g.*, whether it is a superscalar processor or not.

Operations

The nodes in the dataflow graph represent primitive operators. A selection of the operators used for the compilation of Id programs is shown in Figure 4.1. Each node has a number of inputs, the operands, and produces a result depending on the operation it performs, as indicated by its label.

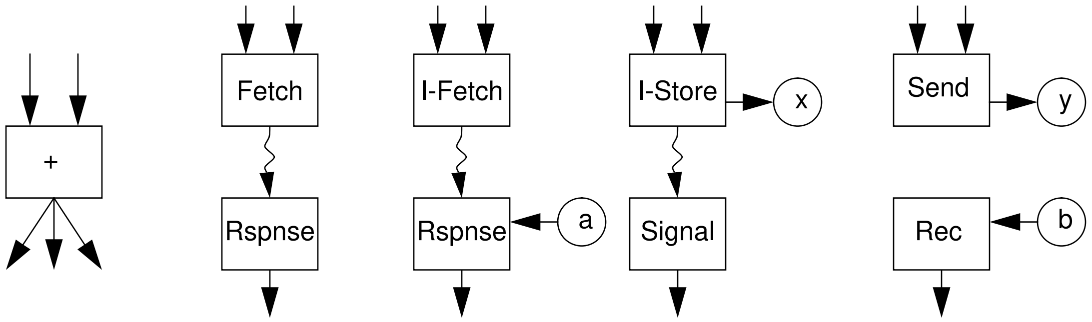


Figure 4.1: *Dataflow nodes used for the compilation of Id.*

The simplest nodes are basic *arithmetic and logic operators*, such as addition or multiplication. Since Id is a strongly typed language, all overloading has been resolved at the dataflow graph and different nodes are used for integer or floating point operations.

Operations for creating and accessing data structures are more complicated. Data structures are assumed to be distributed across multiple processors. Thus, allocation and accesses to data structures may involve communication and require split-phase operations. The simplest accesses are *fetch* operations which take a data structure and an offset and return the value of the corresponding element. In the split-phase representation, the request and reception of the response are two separate operations which are connected with a squiggly edge to indicate the long latency dependence. The *store* operation takes a data structure, offset, and value, and simply writes the value into the corresponding element, returning a signal once the operation has completed. The *I-fetch* and *I-store* operations are data structure accesses which, in addition to reading or writing some element, also perform the synchronization between the producer and consumer of an element required by I-structure semantics. There exists a dependence from the I-store node which fills an I-structure element to all of the I-fetch nodes which read. It may not be possible to determine these dependencies at compile time. Instead of using edges, *inlet* and *outlet* annotations are used to represent them. The inlet annotation, the incoming circle attached to a response node, represents all of the I-store nodes this node potentially depends on. Likewise, outgoing circles are used for outlet annotations to represent the set of I-fetch nodes this instruction potentially influences. For example, the I-fetch response node with the inlet annotation $\{a\}$ from Figure 4.1 may depend on the I-store with the outlet annotation $\{x\}$

Finally, *send* and *receive* nodes are used for passing of arguments and results. Each argument to a basic block comes in via a receive node, while each result of a basic block is passed back to the caller with a send node. The group of n receive nodes for arguments and m send nodes for results forms the *def site* of a basic block. Other basic blocks may have a corresponding *call site* with n send nodes for the arguments and m receive nodes for the results. *Interfaces* specify how the basic blocks relate, *i.e.*, which call sites can call which def sites. Since receive and send nodes may also be end points for potential dependencies, they also carry inlet or outlet annotations. We will use the term *inlet node* for any node with an inlet annotation, be it a receive node or I-fetch response node. Likewise we are going to use the term *outlet node* for any send or I-store node.

As proposed by [AA89], conditionals can also be represented by function calls. A conditional with two arms can be viewed as a function call, where depending on the result of the predicate, one of two basic blocks are called. This representation simplifies the partitioning process, as we can use the same unified mechanism to deal with function calls and conditionals.

Dependencies

We now present the three kinds of dependencies in more detail. Two of these are certain dependencies, which can be be either direct or indirect, while the third kind, potential dependencies, are always indirect.

Certain direct dependencies: A straight arc $u \longrightarrow v$ specifies that there exists a direct dependence from node u to node v which is known to exist in every context in which the basic block can appear. The instruction represented by node u has to execute before the instruction v, and instruction u has direct responsibility for scheduling v. In general, the edge also indicates a data dependence, *i.e.*, it represents the operand which travels from one instruction to the next. Sometimes, these arcs are used to indicate solely control dependencies required to ensure a correct execution order of instructions with side effects, and, in that case, the arc can be thought of as carrying a dummy argument of size zero. The partitioning algorithm looks only at the control ordering implied by the dependence edges, while later phases of the compiler, most notably frameslot and register allocation, require the data dependencies.

Certain indirect dependencies: A squiggly arc $u \rightsquigarrow v$ indicates a certain indirect dependence from node u to node v which is known to exist in every context in which the basic block can appear. The instruction represented by node u has to execute before the instruction v, and v will be scheduled as a indirect consequence of executing v. The dependence path from u to v may involve nodes which are outside of the basic block and therefore may require interprocessor communication or suspension. Thus, a squiggly arc indicates a potentially long latency dependence, and two nodes connected by a squiggly arc cannot reside in the same thread.

Potential indirect dependencies: In addition to certain dependencies, the dataflow graph also captures potential dependencies, *i.e.*, dependencies which may arise only in some of the contexts in which the basic block can appear. Since potential edges depend on the context, they must always be completed through nodes of other basic blocks and are therefore indirect dependencies. We use *inlet* and *outlet annotations* to represent the end points of potential dependencies. We attach incoming circles to nodes for inlet annotation and outgoing circles for outlet annotations. Since potential dependencies are indirect, they also indicate that the nodes which they connect cannot be placed into the same thread.

Each inlet annotation consists of a set of inlet names (a set of symbols), while each outlet annotation consists of a set of outlet names. An inlet name represents a set of outlet nodes this node definitely depends on. This set is not known at compile time, but every node which contains the same inlet name in its inlet annotation depends on this set of outlet nodes. Initially, every inlet node is given a unique singleton inlet annotation, indicating that every inlet node may depend on a different set of outlet nodes. Likewise, every outlet node is given a unique singleton outlet annotation. The rule for determining potential dependencies is the following: we assume that potential dependencies are present, unless contradicted by certain dependencies, *i.e.*, an inlet name may depend on any outlet name, unless there exists a certain dependence path in the opposite direction from the inlet to the outlet. The reason is that when the certain dependence path exists the potential dependence cannot exist, as the program would otherwise deadlock.

For example, the I-fetch response node with the inlet annotation $\{a\}$ from Figure 4.1 may depend on the I-store with the outlet annotation $\{x\}$, if there does not exist a path from a node with a in its inlet annotation to a node with x in its outlet annotation.

Through interprocedural analysis, the inlet and outlet annotations can be refined and may be given partially overlapping annotations, reflecting shared dependencies. For example, assume that we have four nodes with the inlet annotations $\{a, b\}$, $\{a, b\}$, $\{b, c\}$, and $\{a, b, c\}$, respectively. These annotations share names, as the inlet name a appears in the first two and last annotations, the name b appears in all annotations, and the name c appears in the last two annotations. Therefore, the first two nodes depend on precisely the same set of outlet nodes since their inlet annotations are the same. Similar, any node which depends on the first and third node (or second and third node), depends on the same set of outlet nodes as the fourth node. The concept of sharing between annotations is explained in more detail in the next chapter.

Even though nodes connected by a squiggly edge have to be placed into separate threads, squiggly edges are certain dependencies and thereby help to contradict potential dependencies. This in turn may improve the partitioning. Interprocedural analysis can further improve partitioning by refining the inlet and outlet annotation to express sharing, and introducing squiggly edges from a send to a receive whenever it can determine that the corresponding path always exists.

4.2.2 Interfaces

Global analysis requires the knowledge of how the basic blocks relate. This is provided by interfaces. Each basic block contains one def site and zero or more call sites. Each basic block can be viewed as a n argument and m result procedure, where the arguments come in through n receive nodes and the results leave through m send nodes. These receive and send nodes comprise the def site. Other basic blocks which call this procedure, contain a corresponding call site with n send nodes for the arguments and m receive nodes for the results. Interfaces specify how the def sites and call sites relate. The interfaces essentially describe the call graph of the program, and as we will see later, they can be used to create a grammar which can generate all possible call trees.

The simplest kind of interface is a single-call-single-def interface. Here a single call site invokes a single def site, and only this call site invokes the def site. This situation occurs if a procedure is called only at a single place in the program. Global analysis across this interface can work in two directions. It can improve the call site with the information derived at the def site (callee) by refining the inlet and outlet annotations and introducing new squiggly arcs. For example, if two argument receive nodes in the callee influence the same set of outlet nodes, then we can give the corresponding two argument send nodes at the call site the same outlet annotation. If the callee contains a dependence path from an argument receive node to a result send node, then we can introduce a squiggly arc at the call site from the corresponding argument send node to result receive node. Likewise, the def site can also be improved with the information from the call site. Annotations can be refined and squiggly arcs can be introduced in the def site from a result send back to an argument receive, if the corresponding path exists in the caller.

Slightly more complicated is a multiple-call-single-def interface. This corresponds to the case where a procedure can be invoked from multiple call sites. Again, global analysis can improve the annotations and squiggly edges of the call sites and def sites. Care has to be taken at the def site though. The new annotations and squiggly edges must be consistent with all possible callers. For example, a squiggly edge can only be introduced if the corresponding path exists in all callers. Of course, the compiler can always specialize a function by cloning it. It can then reannotate the cloned basic block and partition it such that it would only work for this one call site.

The final case, a single-call-multiple-def interface, arises in the case of conditionals or function calls through function variables. Here each arm of the conditional is viewed as an n argument and m result procedure. Depending on the value of the predicate one of the basic blocks is called. The same interprocedural optimizations as in the previous case are possible. But now, care has to be taken when refining the call site. It has to be consistent with all of the def sites.

4.2.3 Simple Example

We now present a simple example which illustrates the concepts just introduced. Figure 4.2 shows the dataflow graph for the simple function `f` which we discussed earlier in Section 2.3.1.

```
def f a b = (a*a, b+b);
```

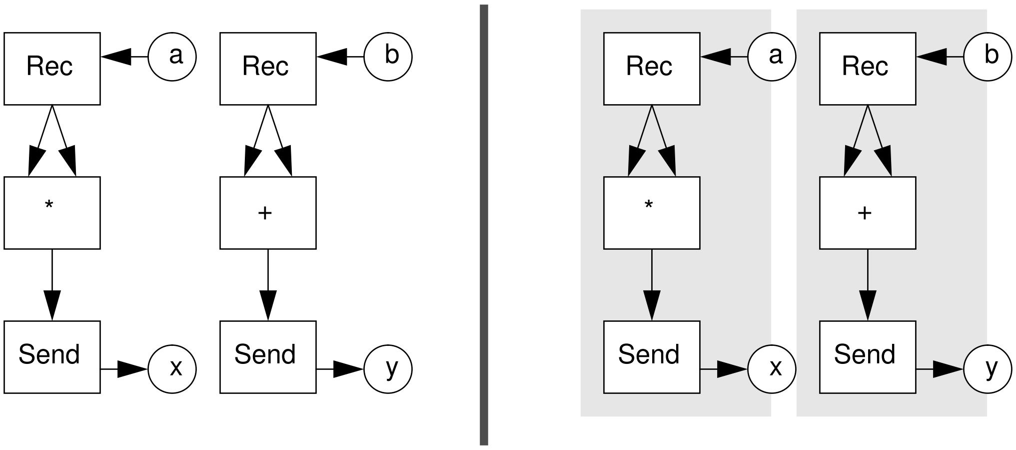


Figure 4.2: *Small example of a dataflow graph for the function* `f a b = (a*a, b+b);` *and its partitioning into two threads.*

The arguments to the function come in through two receive nodes. Once received, the first argument is multiplied with itself and sent back to caller as the first result. The second result is produced by adding the second argument to itself. As already illustrated in Section 2.3.1, the order in which the nodes execute may depend on the context in which the function appears. For example, the function `f` may be called by the following two functions.

```
def g a = { x,y = (f a x) in y};    % computes (a*a)+(a*a) = 2 a^2
def h a = { x,y = (f y a) in x};    % computes (a+a)*(a+a) = 4 a^2
```

The function `g` feeds the first result of the function `f` back in as the second argument, while the function `h` feeds the second result back in as the first argument. Thus, depending on the context in which `f` appears, there may exist a dependence from a send node back to a receive node. These dependencies are potential dependencies since they do not always exist, and they are represented by inlet and outlet annotations.

The two receive nodes are given the inlet annotation $\{a\}$ and $\{b\}$ respectively, while the two send nodes have the outlet annotation $\{x\}$ and $\{y\}$. A potential dependence may exist if there does not exist a certain dependence which contradicts the potential dependence. In this example, there may exist a potential dependence from the send node with x in its outlet annotation back to the receive node with b in its inlet annotation, because there does not exist a certain dependence path from a node with b in its inlet annotation to a node with x in its outlet annotation. Likewise, there may exist a potential dependence from the send node with y in its outlet annotation to the receive node with a in its inlet annotation. On the other hand, there cannot exist a potential dependence from the send node with the outlet annotation $\{x\}$ back to the receive node with the inlet annotation $\{a\}$, because this is contradicted by a certain dependence path. Thus, the inlet and outlet annotations correctly capture the two potential dependencies which may arise at run time. As a result, the left nodes and the right nodes have to stay in separate threads, and the partitioning algorithm can at best obtain two threads, as indicated by the shaded regions.

On the other hand, we may improve the partitioning if we know that the function `f` only gets called in the following context:

```
def foo a = { x,y = (f a a);
               z = x+y;
             in z };        % computes (a*a)+(a+a) = a^2+2a
```

If the compiler knows that `foo` is the only function which calls `f`, or if it is desirable to produce a specialized version of `f` for the call site in `foo`, then the compiler can refine the annotations for the def site of `f`. In this case, it is valid to give both receive nodes of the def site the same inlet annotation, say $\{a\}$, as both argument send nodes at the call site depend on same argument of the function `foo`. Likewise, we can refine the outlet annotations of the two result send nodes of `f` and give them a common outlet annotation, say $\{z\}$. Now, when partitioning `f`, the compiler can identify that under no circumstances can there exist a potential dependence from a result back to an argument, since under the new annotation there exists a certain dependence path from a receive node with the inlet name a to a send node with the outlet annotation z. Thus, the compiler can group all of the nodes into a single thread.

This partitioning can now be used to improve the annotations at the call site of `f` in `foo`. The call site contains two send nodes for the arguments and two receive nodes for the results. Initially, these are given the singleton inlet and outlet annotation and no squiggly edges are present. After partitioning *f* into a single thread, we can improve the call site and introduce a squiggly edge from each of the two argument send nodes to each of the two result receive nodes. Since these squiggly edges capture all of the indirect dependencies and no additional potential dependencies exist, the nodes can be given the empty inlet and outlet annotations.

4.2.4 Summary

The representation of programs in form of basic blocks and interfaces provides the framework for our partitioning algorithms. Basic blocks express three kind of dependencies. Nodes connected by an indirect dependence (either certain or potential) cannot reside in the same thread because otherwise that thread might have to suspend. Certain dependencies are used to contradict potential dependencies. Partitioning can be further improved by interprocedural analysis which further eliminates potential dependencies by refining annotations and introducing new squiggly edges. Global analysis is based on interfaces which describe how the call and def sites of the basic blocks relate.

The partitioning algorithms do not necessarily require dataflow graphs. They can operate just as well on control graphs, or graphs that separate the data and control dependencies, such as dual graphs [Sch91]. Dataflow graphs have well-defined operational semantics and can be executed directly. Data is represented by tokens traveling along the arcs. A node fires when all operands are available, *i.e.*, a token is present on every one of the inputs. Upon firing, a node computes a result based on the data values of the tokens and propagates a result token to every one of the outputs. When executing the graph directly, the dynamic scheduling overhead is very high. An important observation, captured by partitioning, is that dynamic scheduling is not required throughout. Therefore, it makes sense to group nodes which do not require dynamic scheduling into threads.

4.3 Basic Block Partitioning

In this section, a sequence of algorithms for partitioning a basic block into threads are presented. As mentioned earlier, the partitioning algorithms only determine the subsets of the vertices of a basic block; the actual ordering of instructions within each threads is left to a later stage of the compiler. The algorithms presented in this section are *simple partitioning*, *dataflow partitioning*, *dependence set partitioning*, *demand set partitioning*, and a combination of the last two, called *iterated partitioning*. Each of these partitioning schemes is more precise than its predecessor but also more complex to implement. The description of these algorithms summarizes previous work and serves to develop an intuition of what a correct partitioning has to do. The next section presents a new basic block partitioning algorithm, *separation constraint partitioning*. Separation constraint partitioning is more powerful than any of the previous partitioning schemes, but also more complicated and computationally expensive. In explaining separation constraint partitioning, we will use the concepts introduced by dependence set and demand set partitioning.

A basic block partitioning seeks to group together vertices of a basic block which can be mapped into a non-suspending sequential thread. It therefore has to avoid introducing circular dependencies among threads, both static cycles within the basic block and dynamic cycles involving other basic blocks. Without circular dependencies, it is possible to delay the scheduling of a thread until all of its predecessors have terminated and then run the thread until completion. Nodes which require dynamic scheduling have to reside in different threads. To derive a correct partitioning, these algorithms can only use the information present in the basic block, certain dependencies in the form of straight and squiggly edges, and potential dependencies as indicated by inlet and outlet annotations. Dynamic scheduling may be required between two nodes if there exists an indirect dependence between them. Certain indirect dependencies separate requests from responses and indicate that in the case of a remote access, the response has to be scheduled dynamically. Potential indirect dependencies indicate that these dependencies may arise at run time, depending on the context in which a basic block appears. Initially, the partitioning algorithms have to make worst case assumptions. There could exist a potential indirect dependence from any outlet node to any inlet node. Potential indirect dependencies can be ruled out when they are contradicted by certain dependencies. This can be improved with the refined inlet and outlet annotations and new squiggly edges obtained by global analysis.

Each of the six partitioning algorithms uses a different approach to determine that neither a certain nor a potential indirect dependence can arise between a group of nodes placed into the same thread. Both simple and dataflow partitioning are trivially correct. Dependence set and demand set partitioning are greedy partitioning algorithms: they both seek to group together nodes into maximal subsets, where the sole criteria for grouping together nodes is whether they depend on the same set of inlet or outlet annotations. To deal with split-phase, long latency communication, these algorithms are combined with subpartitioning, which ensures that all squiggly edges cross threads. Iterative partitioning iteratively applies dependence set and demand set partitioning. Although this algorithm is more powerful than dependence and demand set partitioning, it may still fail in some cases to group nodes together which can safely be merged into a single thread. Separation constraint partitioning, described later, does not exhibit this limitation. This algorithm identifies everything that could possibly get grouped together given the information present in the basic block.[2] This algorithm computes separation constraints which tell for any two nodes whether they can be merged or not. This approach deals in a unified way with both potential and certain indirect dependencies, *i.e.*, separation constraints due to non-strictness and long latency communication. Separation constraint partitioning still works greedily, but can be combined with heuristics which steer the order in which nodes get merged. These heuristics try to minimize the dynamic scheduling and synchronization overhead by first attempting to merge those nodes which result in a larger savings.

Before describing the algorithms formally, we will use a small example illustrating how the various algorithms derive the threads and their difference in power.

4.3.1 Partitioning a Small Example

The small program

```
x = a + b; y = a * b;
```

can be represented by a two-argument-two-result basic block. It is used to illustrate the different partitioning schemes. Figure 4.3 shows the dataflow graph for block, together with the threads derived by the various basic block partitioning algorithms.

Simple partitioning is the most trivial form of partitioning, here each node is placed into

[2]Separation constraint partitioning can be shown to be the most powerful annotation based partitioning algorithm, *i.e.*, at the end of the algorithm there are no two nodes which can still be merged under the present basic block annotation.

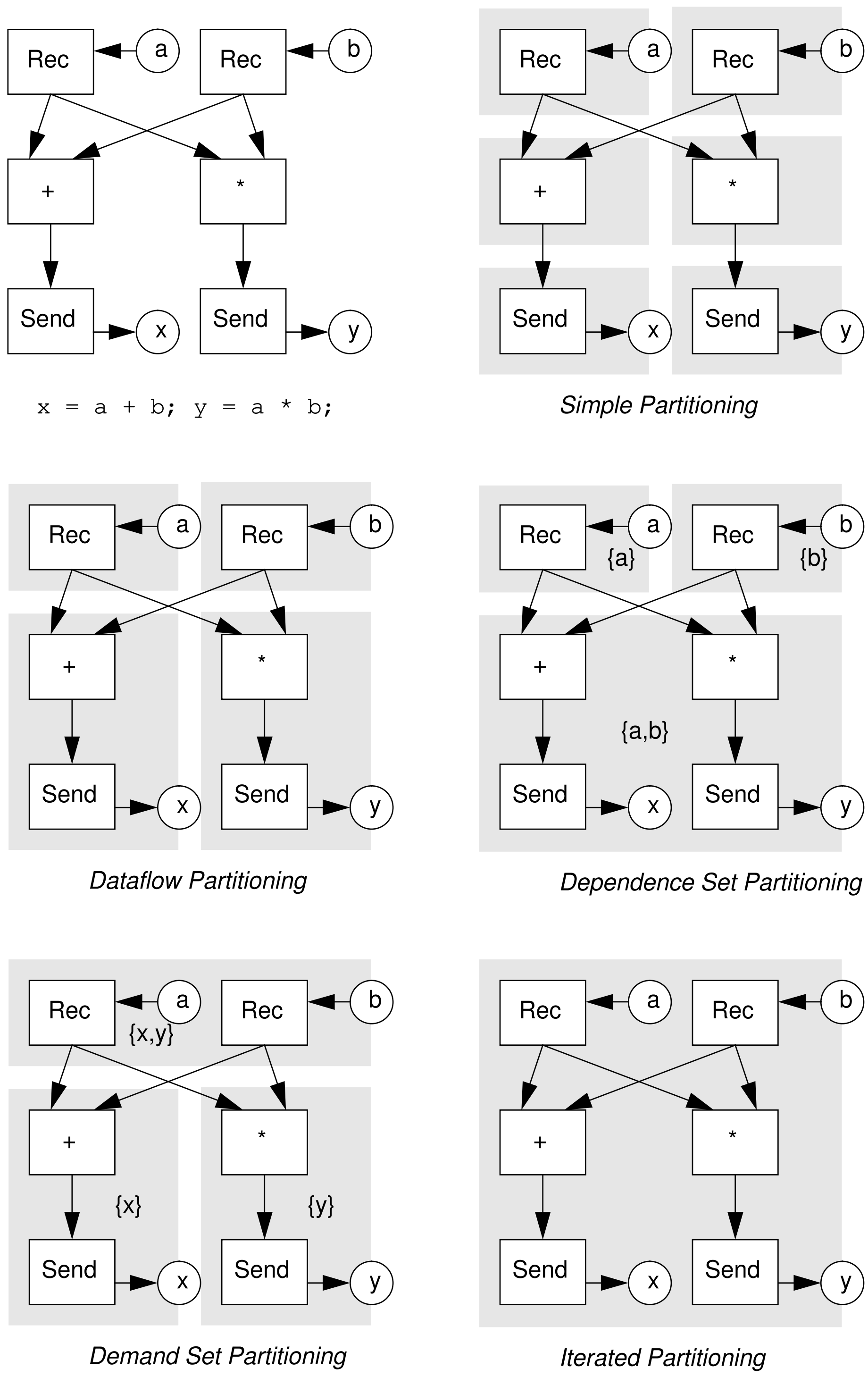


Figure 4.3: *The dataflow graph for the small example* `x = a + b; y = a * b;` *and the threads derived by various partitioning algorithms. The shaded areas indicate the threads.*

its own thread, resulting in a total of six threads. This trivially yields a correct partitioning because it does not introduce any new dependencies, but its dynamic synchronization overhead is far too large.

Dataflow partitioning recognizes the fact that unary operations never need dynamic synchronization. In this scheme, inlet nodes (receive nodes and responses for split-phase communication) and nodes requiring synchronization start a new thread. Unary nodes are placed into the thread of their predecessor. This partitioning scheme will put simple fan-out trees into a single thread. Applying this rule to our example, we obtain four threads. We call this scheme dataflow partitioning, because it reflects the limited thread capability supported by most modern dataflow machines [PC90, GH90, SYH+89].

Far more powerful is *dependence set partitioning*, which finds a correct partitioning by grouping together nodes with the same dependence set. The *dependence set* of a node is the union over the inlet annotations of all of its predecessors (all predecessors reachable over zero or more straight or squiggly edges). For our example, dependence set partitioning produces three threads. As we can see, dependence set partitioning works across fan-out trees. We have annotated each thread with its dependence set. The four bottom nodes depend on exactly the same set of inlet annotations, $\{a, b\}$, they are therefore placed into the same thread.

This partitioning is correct because there could not exist a potential indirect dependence among these nodes. The informal argument is as follows. Any such dependence would have to travel through an outlet node. The inlet annotations indicate the set of (unknown) outlet nodes on which an inlet node depends. Two nodes with the same dependence set, depend on the same set of inlet annotations, and therefore they depend on the same set of outlet nodes. Thus, there cannot exist a potential indirect dependence from one node to another, otherwise there would also exist the same dependence from a node back to itself.[3] Thus, nodes with the same dependence set can safely be placed into the same thread. There is one aspect that has been ignored so far because it does not arise in this example. Nodes connected by certain indirect dependencies (squiggly edges) may end up in the same thread. Therefore dependence set partitioning needs to extended with a *subpartitioning* algorithm to ensure that long latency operations cross thread boundary.

[3]We reiterate that while such a dependence may actually arise at run-time in some cases, the result would be that the program deadlocks. In that case, the program will deadlock no matter what the partitioning does. Since we do not care where a deadlock occurs (if it occurs) the partitioning can do anything.

Demand set partitioning works similarly to dependence set partitioning, but finds a correct partitioning by grouping together nodes with the same demand set. The *demand set* of a node is the union over the outlet annotations of all successor nodes (again, the successor relation is reflexive and transitive). We have indicated the demand sets in our example. With demand set partitioning, the top two nodes can go into the same partition since their demand sets are the same: $\{x, y\}$.

This partitioning scheme also ensures that there are no static cyclic dependencies within the basic block, nor any dynamic cyclic dependencies across basic blocks. The discussion of correctness is analogous to dependence set partitioning. Nodes with the same demand set are being demanded by the same set of (unknown) inlet nodes. Therefore there cannot exist a potential indirect dependence from one to another as one would have a cycle to itself. Demand set partitioning also has to be combined with subpartitioning to ensure that certain indirect dependencies cross threads. As this example shows, dependence set partitioning tends to group nodes at the bottom of a graph, while demand set partitioning tends to group nodes at the top.

Iterated partitioning combines the power of dependence set and demand set partitioning by applying them iteratively. First one of the two partitioning algorithms is applied. Then a reduced graph is formed which contains a node for every thread and an edge between two nodes if an edge crosses the threads. Then the other partitioning algorithm is applied on the reduced graph. This process of partitioning and forming the reduced graph is continued iteratively until no further changes occur. In our example, starting with dependence set partitioning we first obtain three threads. Forming the reduced graph and then applying demand set partitioning results in a single thread containing all of the six original nodes. If we start iterated partitioning with demand set partitioning for this example, the same final result is obtained.

While this scheme combines the power of dependence set and demand set partitioning, and therefore works across different fan-out and fan-in trees, it may still fail in some cases to produce maximal threads (an example is given later). *Separation constraint partitioning*, presented in Section 4.3.4, does not have this limitation. For our current example, separation constraint partitioning would produce the same result as iterated partitioning, a single thread.

4.3.2 Dependence Set and Demand Set Partitioning

The discussion of dependence set and demand set partitioning follows the presentation given in previous joint work [TCS92]. Dependence set partitioning derives threads by grouping together all nodes that depend on the same set of inlets. This algorithm was originally proposed by Iannucci [Ian88]. The algorithm has to be combined with subpartitioning which splits threads that enclose squiggly edges.

Definition 1 (Dependence Set) *The* dependence set *of a node is the set of inlets on which it depends:*

$$Dep(v) = \bigcup_{u \in Pred^*(v)} Inlet(u)$$

where Inlet(u) *is the set of inlet names that annotate node* u*, and Pred*$^*(v)$ *is the set of nodes from which there is a path of length zero or more to* v *(through either straight or squiggly edges).*

Algorithm 1 (Dependence Set Partitioning) *Given a dataflow graph:*

1. Compute $Dep(v)$ for all nodes v.

2. Partition the graph into maximal sets of nodes with identical dependence sets.

Computing the dependence sets can be done in a single traversal in topological order, since the graph is acyclic. Finding nodes with identical dependence sets can be implemented by lexicographically sorting the dependence sets.

Demand set partitioning works analogously to dependence set partitioning. It groups together all nodes which are demanded by the same set of outlets, *i.e.*, groups together a set of nodes on which the same set of outlets depend. This algorithm requires demand sets, which are computed in the reverse topological order. This algorithm was first discovered by Hoch *et al.*, [HDGS93] who called this rule "collapse vertices with identical exit sets." In [SCvE91] this algorithm was called "dominance set partitioning".

Definition 2 (Demand Set) *The* demand set *of a node is the set of outlets which depend on it:*

$$Dem(v) = \bigcup_{u \in Succ^*(v)} Outlet(u)$$

where Outlet(u) *is the set of outlet names that annotate node* u*, and Succ*$^*(v)$ *is the set of nodes to which there is a path of length zero or more from* v *(through either straight or squiggly edges).*

The demand set partitioning algorithm is analogous to Algorithm 1.

4.3.3 Subpartitioning

Dependence and demand set partitioning do not distinguish between straight and squiggly edges. As a result, two nodes which are connected by a squiggly edge may end up in the same thread. Squiggly arcs indicate split-phase operations; therefore, the two nodes connected by a squiggly edge should reside in two different threads. To ensure that all squiggly edges cross threads, subpartitioning splits the threads derived by dependence and demand set partitioning into smaller threads. The algorithm determines for every node its subpartition number which is the maximum distance in the number of squiggly arcs from the roots of the threads. Then smaller threads are then formed by splitting each thread into subsets with the same subpartitioning number.

Algorithm 2 (Subpartitioning)
Given a thread:

1. Visit each node v of the thread, in topological order according to intra-thread straight and squiggly edges, and compute

$$\mathit{Subpart}(v) \leftarrow \max(0, \max_{u \in \mathit{Pred}_s(v)} \mathit{Subpart}(u), 1 + \max_{u \in \mathit{Pred}_q(v)} \mathit{Subpart}(u))$$

   where $\mathit{Pred}_s(v)$ and $\mathit{Pred}_q(v)$ are the immediate predecessors of v via straight and squiggly edges, respectively.

2. Form smaller threads by grouping together nodes with identical *Subpart*$()$.

Subpartitioning does not necessarily require a separate pass over the graph. It can be combined directly with both dependence and demand set partitioning (see [TCS92] for details).

Appendix A gives the full proofs of the correctness of dependence and demand set partitioning with subpartitioning. It shows that the subsets these algorithms discover can in fact be mapped to sequential non-suspending threads, *i.e.*, that they satisfy the requirements specified in Section 4.1. Informally, one can argue correctness by showing that no static or dynamic dependence path between two nodes in a thread can be completed through a node outside the thread. No static path can exist, because nodes along such a path must have monotonically increasing dependence sets (or monotonically decreasing demand sets). No dynamic path can exist, because such a path would have to be completed through one of the inlets (outlets) in the thread's dependence (demand) set, implying a cycle. Finally, subpartitioning insures that all threads can be executed without pause or interruption since all long latency, split-phase operations cross thread boundaries. Also, subpartitioning does not introduce any new cycles because subpartition numbers are monotonically increasing along any path.

4.3.4 Iterated Partitioning

As our small introductory example from Figure 4.3 showed, dependence set partitioning tends to group nodes at the bottom of a graph, while demand set partitioning tends to group nodes at the top. Iterated partitioning combines the power of dependence and demand set partitioning by applying them iteratively. First, one of the two algorithms is applied, then the reduced graph is formed and the other algorithm is applied. This process is repeated until no further changes occur.

The *reduced graph* for a basic block contains a node for every thread and an edge between two nodes if the corresponding two threads have an edge between them. In the case where both squiggly and straight edges cross the threads, a squiggly edge is chosen. In addition, the inlet (outlet) annotation of a new node is the union of all inlet (outlet) annotations of the nodes within the corresponding thread.

Algorithm 3 (Iterated Partitioning) *Given a dataflow graph:*

1. *Apply dependence set partitioning with subpartitioning*
2. *Form the reduced graph*
3. *Apply demand set partitioning with subpartitioning*
4. *Form the reduced graph*

5. Repeat from Step 1 until no further changes

Instead of dependence set partitioning, it would also be possible to start iterated partitioning with demand set partitioning. As discussed below, the number of iterations required to find the final solution is usually very small.

4.3.5 Examples of Iterated Partitioning

In Figure 4.4 iterated partitioning is applied to another small example. The dataflow graph corresponds to a two-input, three-result basic block, consisting of two *receive* nodes and three *send* nodes. Note that the outlet annotation is not the trivial annotation, as the outlet annotation of node 5 is $\{x, y\}$ which is not a singleton. It shares the x with node 2 and the y with node 3. This annotation could have been the result of interprocedural annotation propagation, described in the next chapter. Applying first dependence set partitioning results in three threads, as indicated by the shaded regions in Part (a). The dependence sets, also shown in the figure are $\{a\}$ for nodes 1 and 2, $\{a, b\}$ for node 3, and $\{b\}$ for nodes 4 and 5. The reduced graph consisting of three nodes with the new inlet and outlet annotation is shown in Part (b). Next, demand set partitioning is applied which results in two threads, corresponding to the demand sets $\{y\}$ and $\{x, y\}$. Forming again the reduced graph and then applying dependence set partitioning results in a single thread as shown in Part (c). Part (d) shows the final reduced graph consisting of a single node. Iterated partitioning succeeded in grouping all five nodes of the original graph into single thread in three partitioning steps.

An interesting question that arises naturally with iterated partitioning is how many cycles of dependence and demand set partitioning are required before reaching a stable solution. Experimental data indicates that in practice the number of cycles required to find the final solution seems to be very small. Two cycles (*i.e.* four partitioning steps) was sufficient for the set of Id programs used in Chapter 6, the largest of which had basic blocks containing slightly over 600 nodes. Some straightforward theoretical analysis reveals that every partitioning step — with the exception of the very first — has to merge at least two nodes. The final result is reached, if at any step no nodes are merged. The first step is an exception, because the very first partitioning scheme may encounter a graph where it cannot merge anything, but where the next partitioning scheme can. For a graph with n

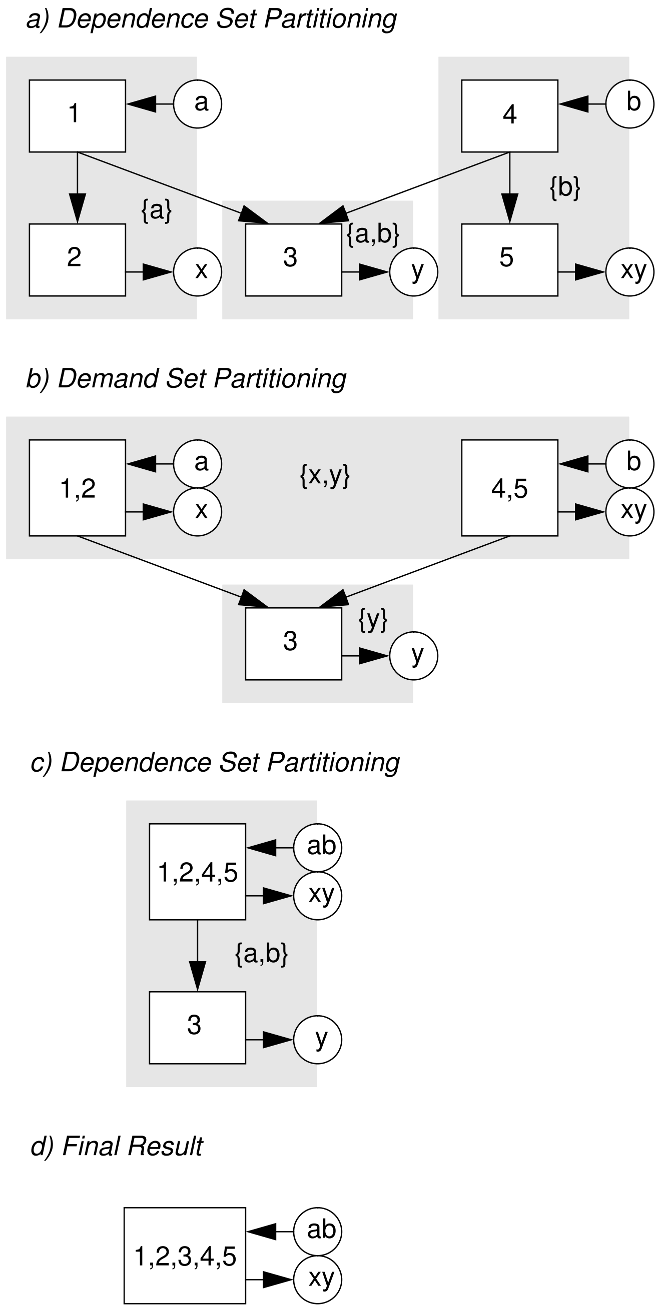


Figure 4.4: *Example of Iterated Partitioning.*

nodes at most $n-1$ mergings can occur, and therefore at most n partitioning steps are required.

Actually, it is possible to construct examples which require precisely this number of steps. Figure 4.5 shows an example with 6 nodes which requires 5 steps of dependence and demand set partitioning to find the final solution. This example was constructed using the following observation. Assume we have a graph with n nodes which requires $n-1$ partitioning steps to form a single thread. Obviously every partitioning step merges only two nodes. The two nodes being merged in the j-th partitioning step are being merged with a third node during the next partitioning step, and so on. Assume the j-th partitioning step uses dependence set partitioning. It must be the case that the two nodes being merged have the same dependence set, and that together they have a demand set which is equal to a third node. On the other hand, neither of the demand sets of the first two nodes alone can be equal to the demand set of the third node. The second node must have contained a new name in its outlet annotation, not already present in the first node.

What happens is that the dependence or demand sets of the merged nodes grows at every step. At every step, the new node that is being merged brings in a new inlet or outlet name. This results in a new dependence or demand set which is equal to another node. Applying this rule 5 times to a node with a single inlet annotation $\{a\}$ and outlet annotation $\{x\}$ gives us above example. It is trivial to extend this example so that it requires additional iterations; each additional iteration introduces a new inlet or outlet name. Although the example requires non-singleton inlet and outlet annotations, it is possible to construct equivalent (but slightly more complicated) examples using only singleton annotations which exhibit the same behavior. Therefore, iterated partitioning may require an arbitrary number of iterations, even without interprocedural annotation propagation. Using the above example as guidance we wrote an small Id function which requires six steps of iterated partitioning.

4.3.6 Limits of Iterated Partitioning

Even though iterated partitioning is much more powerful than dependence or demand set partitioning alone, it is not powerful enough to always find maximal threads. At the end of the algorithm there may still be some threads left which could safely be merged.

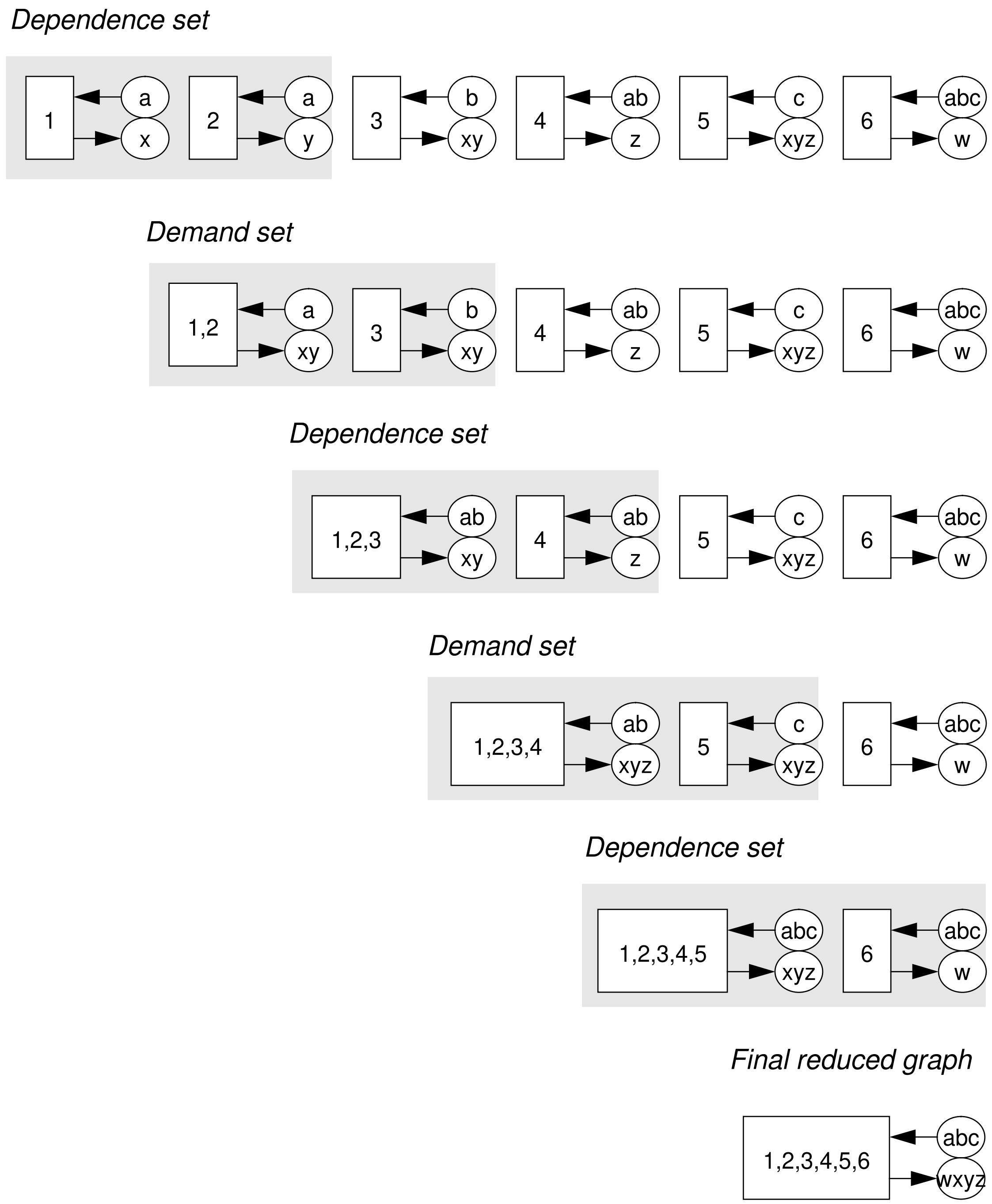


Figure 4.5: *Example with six nodes requiring five steps of dependence and demand set partitioning to obtain the final solution. If we start with dependence set partitioning, three times dependence and twice demand set partitioning is required. Starting with demand set partitioning, the very first time demand set partitioning does not merge any nodes, therefore dependence and demand sets partitioning are both required three times for a total of six steps.*

Figure 4.6 shows such an example.

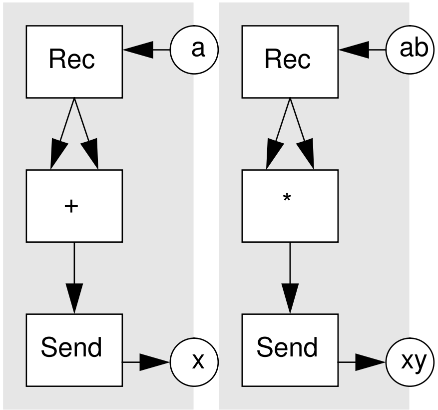


Figure 4.6: *Example where iterated partitioning fails to merge two threads.*

This example consists of six nodes and iterated partitioning forms two threads. As in the previous example, the inlet/outlet annotations are not the trivial annotation. The dependence set of the three left nodes is $\{a\}$ and their demand set is $\{x\}$. Thus iterated partitioning will group them all into a single thread. Likewise, the three right nodes are grouped into a single thread, because their dependence set is $\{a, b\}$ and their demand set is $\{x, y\}$. Iterated partitioning cannot merge the left nodes and the right nodes since their dependence and demand sets are different. They could be safely merged though, because there cannot exist a potential indirect dependence from any of the left nodes to any of the right nodes, nor can there exist one in the reverse direction. This is based on the following observation.

Remember, a name in an inlet annotation represents a set of unknown outlets that any node which contains this name in its inlet annotation depends on. Dependence sets thus represent the set of outlets a node depends on. The dependence set of the left nodes is a subset of the dependence set of the right nodes. Since the right nodes depend on a larger set of outlet nodes, they cannot influence any of the left nodes and there cannot exist a potential dependence from the right to the left nodes. Likewise, demand sets represent the set of inlets a node influences. The demand set of the left nodes is a subset of the demand set of the right nodes. Thus, the left nodes influence a smaller set of inlet nodes and cannot influence any of the right nodes. Therefore, it is possible to merge the two threads. There cannot exist an indirect dependence from the first thread to the second because the demand set of the first is a subset of the second, and there cannot exist a indirect dependence from the second thread to the first because the dependence set of the second is a superset of the the first.

To be able to merge these threads we have to develop a more powerful partitioning rule which is not solely based on equal dependence or demand sets. Above example indicates that it is safe to merge two nodes u and v if $Dep(u) \subseteq Dep(v)$ and $Dem(u) \subseteq Dem(v)$, or vice versa. Actually, we can go much further than this. It is possible to merge two nodes if all of the inlet names in $Dep(u)$ influence all of the outlet names in $Dem(v)$ and vice versa. Then there cannot exist a potential dependence from one node to the other. This observation is formalized by separation constraint partitioning.

4.4 Separation Constraint Partitioning

Separation constraint partitioning is based on the observation that it is possible to *precisely* determine for any two nodes whether they can be merged or not. The rule is very simple. Two nodes of a basic block cannot be merged, *i.e.*, have to reside in different threads, if there exists either a certain indirect dependence or an potential indirect dependence between them. Otherwise, they can be merged into the same thread. The reason is that both forms of indirect dependencies, certain and potential, may require dynamic scheduling. We will discuss later how to compute these separation constraints.

Given this separation rule, we can easily devise an effective partitioning algorithm. Starting with the unpartitioned dataflow graph, we find two nodes without a separation constraint, merge them, form the reduced graph, and iterate this process until no further nodes can be merged. This simple approach underlies separation constraint partitioning, but the actual algorithm is extended with some optimizations which address several pragmatic issues. For example, the order in which the nodes are merged is very important as it determines the final result. The algorithm is therefore combined with a heuristic to obtain an order which both reduces the complexity of the algorithm and improves the result of the partitioning. The heuristic still tries to merge all possible pair of nodes, it just attempts to do this in a better order. This algorithm is guaranteed to derive maximal (but not necessarily optimal) threads: after it has finished, every pair of threads has a separation constraint between them and therefore it is impossible to do further merging.[4] Another advantage of this algorithm is that it deals in a unified way with the partitioning constraints introduced by certain and potential indirect dependencies, and therefore does not require subpartitioning. Unfortunately, it is computationally more expensive than any of the previous partitioning algorithms.

4.4.1 Indirect Dependencies and Separation Constraints

We start with addressing the question of how the separation constraints can be computed. Separation constraints arise out of two forms of indirect dependencies, certain and potential. Certain indirect dependencies are easy to identify, they are represented by squiggly edges. Squiggly edges connect send nodes to receive nodes and are either present initially or added through interprocedural analysis. Nodes connected by a squiggly edge cannot reside in the same thread. Furthermore, any two nodes connected by a path which goes over one or more squiggly edges cannot reside in the same thread. Thus it is relatively easy to capture the separation constraints due to certain indirect dependencies.

What about potential indirect dependencies? These are obtained from the inlet and outlet annotations of the basic block. Each outlet/inlet annotation name pair represents a set of potential indirect dependencies. The pair (o, i) represents the collection of indirect edges from all nodes containing o in their outlet annotation to all nodes with i in their inlet annotation.[5] Some of the potential dependencies may be contradicted by certain dependencies, *i.e.*, the reverse path may be present in the dataflow graph. These potential dependencies are inadmissible, because they lead to a deadlock should they actually arise at run-time. For partitioning purposes it is safe to assume that they cannot arise, because if a program deadlocks, it does not matter if it deadlocks somewhere else. Therefore we only consider *admissible* potential indirect dependencies which do not lead to a circular dependence.

How do we determine whether there exists an admissible potential indirect dependence between two nodes or not? We start by computing the set of known IO dependencies from

[4] It can be shown that no other basic block partitioning scheme could do further merging. This algorithm makes the best use of the available information. The only way additional opportunities for merging could arise is by refining the squiggly edges and inlet/outlet annotations through interprocedural analysis.

[5] It is the set $\{(s, r) | o \in \mathit{Outlet}(s), i \in \mathit{Inlet}(r)\}$.

inlet names to outlet names.

$$IO = \{(i, o) | \exists s, r : s \in Succ^*(r), o \in Outlet(s), i \in Inlet(r)\}$$

where $Succ^*$ is the reflexive, transitive successor relation computed on the dataflow graph including all squiggly edges. Note that this IO set does not contain pairs of nodes of the graph; it rather contains pairs of names of the inlet and outlet annotations. For example, for the graph given in Figure 4.6, this set would be $IO = \{(a, x), (a, y), (b, x), (b, y)\}$, because for every one of these pairs there exists a path from a node with the inlet name in its inlet annotation to a node with the corresponding outlet name in its outlet annotation.

This set of known IO dependencies from inlet names to outlet names can be used to precisely determine the inadmissible potential indirect dependencies. For any $(i, o) \in IO$, the collection of potential indirect dependencies represented by (o, i) are inadmissible, as they would lead to cyclic dependencies. The complement of the IO set therefore describes the admissible potential edges. For any $(i, o) \notin IO$, the collection of potential indirect dependencies represented by (o, i) is admissible. Thus, we must assume that they could actually appear at run-time. A simple test of whether an admissible potential indirect dependence may occur from a node u to a node v is to see whether there exists an $o \in Dem(u)$ and $i \in Dep(v)$ such that $(i, o) \notin IO$. If this is the case, the two nodes cannot be merged. By combining this test with our previous test for certain indirect dependencies, we can derive a binary relation which tells us whether an indirect dependence (either certain or admissible potential) exists between a node u and node v.

$$ID(u, v) = \begin{cases} true & \text{if } \exists s \in Succ^*(u), r \in Pred^*(v) \text{ such that } s \rightsquigarrow r & \textit{(certain indirect)} \\ true & \text{if } \exists o \in Dem(u), i \in Dep(v) \text{ such that } (i, o) \notin IO & \textit{(admissible potential)} \\ false & \text{otherwise} & \textit{(no indirect dep.)} \end{cases}$$

Using this binary relation ID[6], it is easy to develop a test of when two nodes cannot be merged. This is the case if an indirect dependence may exist between them, *i.e.*,

$$sep(u, v) = ID(u, v) \text{ or } ID(v, u)$$

[6]Note that the binary relation *ID* is not reflexive, *i.e.*, $ID(u, u)$ is always *false* (assuming that we start with a correct acyclic dataflow graph). It may be symmetric though, *i.e.*, both $ID(u, v)$ and $ID(v, u)$ can be *true*. This is due to the fact that admissible potential dependencies between two nodes may exist in both directions, the graph in Figure 4.2 is a prime example. Obviously, both potential dependencies cannot co-exist at the same time.

Let's apply this rule to the example of Figure 4.6. The set of known dependencies from inlet names to outlet names is $IO = \{(a, x), (a, y), (b, x), (b, y)\}$. This set contains all possible combinations of pair of inlet and outlet names. Therefore, we see that there cannot exist an admissible potential indirect dependence between any two nodes. Since the graph is free of squiggly edges we obtain that $ID(u, v) = \mathit{false}$ for any pair of nodes. Thus, there does not exist a separation constraint, $sep(u, v)$ is always false, and any two nodes could be merged, which is the desired result.

On the other hand, for the example from Figure 4.2, we see that $IO = \{(a, x), (b, y)\}$. Thus an admissible potential indirect dependence exists from the left send to the right receive, and from the right send to the left receive. As a result, the left nodes cannot be merged with the right nodes, as we observed earlier. Let us summarize the algorithm before discussing a slightly more complicated example.

4.4.2 Separation Constraint Partitioning

The algorithm uses sep to determine whether two nodes can be merged or not. This process is iterated until all pairs of nodes have been examined.

Algorithm 4 (Separation constraint partitioning) *Given a dataflow graph:*

1. *Compute the set IO of known dependencies from inlet names to outlet names from the dataflow graph and its annotation.*
2. *Use this to find two nodes u, v which can be merged, i.e., for which sep(u, v) = false, then merge u, v into a single thread and update the graph by forming the reduced graph.*
3. *Repeat from Step 2, until no more nodes can be merged.*

So far this algorithm does not specify the order in which nodes are tested for merging. We will address this issue later. Also the algorithm looks quite expensive to implement. At most $\mathcal{O}(n^2)$ pair of nodes have to be tested for possible merging and at most $\mathcal{O}(n)$ nodes can be merged. Computing the separation constraints, merging the nodes, forming the reduced graph, and updating the data structures, all seem like very expensive operations. How can we reduce the cost of the algorithm? An important observation is that the set IO does not change during the execution of the algorithm. This has to be the case because otherwise the partitioning algorithm would create illegal threads which introduce dependencies contradicting admissible potential dependencies.

Looking at the two examples presented so far, we see there that the separation constraints also did not change during the execution of the algorithm. In the example from Figure 4.2, there always exists a separation constraint from the three left nodes to the three right nodes, while in the example from Figure 4.6, there are no separation constraints at all. Would be possible to create a separation graph and color it in a spirit similar to graph coloring used for register allocation? The nodes of the separation graph could represent the vertices of the original dataflow graph, with edges between two nodes if they cannot reside in the same thread. Coloring this graph such that no two connected vertices have a common color, would give us a partitioning: each color would represent a thread. Unfortunately, this approach is not possible, as the next example shows.

4.4.3 Example of Separation Constraint Partitioning

The example given in Figure 4.7 shows that merging two nodes may also change the separation graph. This example consists of twelve nodes. The set of known dependencies from inlet names to outlet names is $IO = \{(a, x), (a, y), (a, z), (b, x), (b, y), (b, z), (c, y), (c, z)\}$. The only pair not in IO is $\{(c, x)\}$. Therefore an admissible potential indirect dependence exists from node L_5 to node R_1. This in turn implies that the left nodes L_1, L_2, L_3, L_4, and L_5 cannot be merged with any of the right nodes R_1, R_2, R_3, R_4, and R_5, *i.e.*, $sep(u, v) = true$ for these pair of nodes. There are no separation constraints at all for the nodes M_1 and M_2. At first glance, it seems as if they could get merged with any of the other nodes. This is not completely true! If node M_2 is merged with with any of the left nodes — *e.g.*, with node L_4 as shown on the left side of the figure — then node M_1 cannot be merged with any of the right nodes. At the end M_1 and M_2 will both be merged with all of the left nodes. On the other hand, if node M_1 gets merged with any of the right nodes, so must M_2. Thus, merging one of the middle nodes affects the other and the result of the whole process.[7]

[7]The same is also true for iterated partitioning. If we had started iterated partitioning with dependence set partitioning, the final result at the left of the figure would have been the outcome; starting with demand set partitioning, the final result would be the one at the right of the figure. Therefore, under iterated partitioning, the two middle nodes always end up in the same thread. Separation constraint partitioning could also find a third solution, as node M_1 could safely get merged with the left nodes, and node M_2 with the right nodes.

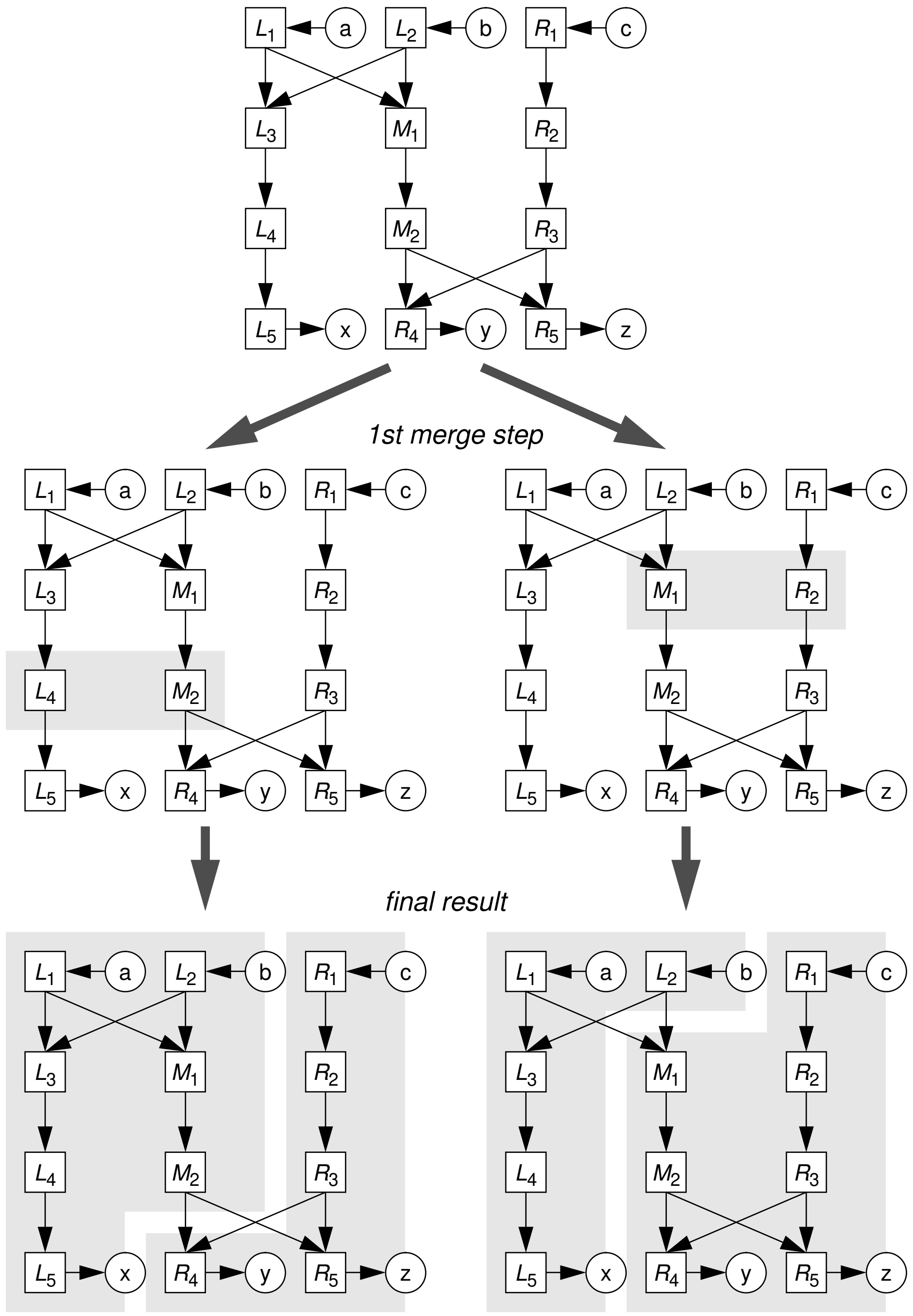


Figure 4.7: *Example showing that merging two nodes under separation constraint partitioning may introduce new separation constraints, and thereby affect the final result. If M_2 is merged with any of the left nodes, M_1 cannot be merged with any of the right nodes, and vice versa.*

Why do the separation constraints change? Merging two nodes changes the connectivity of the dataflow graph, which in turn affects the dependence and demand sets, and thereby the separation constraints. This can introduce new separation edges into the separation graph. Simple coloring of the separation graph does not work. For example, simple coloring could have merged M_2 with L_4 and M_1 with R_2 which results in an illegal partitioning. Therefore the connectivity of the graph and the separation constraints have to be updated after merging two nodes.

4.4.4 Refined Separation Constraint Partitioning

So far, the algorithm has left unspecified the actual representation of the graph, the order in which to test the nodes, how to implement the test, and how to update the graph. This section presents one solution. We observed that the set of known IO dependencies from inlet to outlet names is invariant throughout the algorithm. Therefore, existing separation constraints never disappear, and merging two nodes can only introduce new constraints. If two nodes cannot be merged the first time, they can never be merged. Differently phrased, merging nodes never creates new opportunities for merging; although it may inhibit some. As a result, every pair of nodes has to be tested at most once for merging.

For a graph with n nodes, at most n^2 pairs of nodes have to be tested, while at most $n-1$ pairs of nodes can be merged. Thus, testing occurs far more frequently than merging, and it is desirable to make computing the separation constraints cheap. Always computing the dependence and demand sets when testing for merging, or recomputing them after merging two nodes seems prohibitively expensive. We present an implementation which does not require dependence and demand sets. First, the set of known IO dependencies is computed. This requires computing the reflexive, transitive closure of the connectivity graph, and then using the inlet and outlet annotation to derive the dependencies from inlet names to outlet names. Then the set IO is used to determine all admissible potential dependencies from inlet nodes to outlet nodes. These are combined with the certain indirect dependencies to determine whether an indirect dependence may exist between two nodes or not, which is what we need to decide whether two nodes can be merged or not. After merging two nodes, the transitive closure of the graph and the indirect dependencies are updated.

Algorithm 5 (Refined separation constraint partitioning) *Given a dataflow graph with inlet and outlet annotations:*

1. Compute the reflexive, transitive closure of the successor relation Succ^* over straight and squiggly edges.

2. Determine the set of known dependencies from inlet names to outlet names.

$$\mathit{IO} = \{(i, o) | \exists r, s : (r, s) \in \mathit{Succ}^*, i \in \mathit{Inlet}(r), o \in \mathit{Outlet}(s)\}$$

3. Compute the set of admissible potential indirect dependence edges, *i.e.*, potential indirect dependencies which are not contradicted by certain dependencies.

$$\mathit{PID} = \{(s, r) | \exists i, o : i \in \mathit{Inlet}(r), o \in \mathit{Outlet}(s), (i, o) \notin \mathit{IO}\}$$

4. Set the set of certain indirect dependencies *CID* to be the set of squiggly edges.

5. Combining *PID* and *CID*, compute the set of nodes which have an indirect dependence (either certain or potential) between them.

$$\mathit{ID} = \{(u, v) | \exists r, s : (u, s) \in \mathit{Succ}^*, (r, v) \in \mathit{Succ}^*, (s, r) \in \mathit{PID} \cup \mathit{CID}\}$$

6. Find two nodes u, v without an indirect dependence between them, *i.e.*, for which $(u, v) \notin \mathit{ID}$ and $(v, u) \notin \mathit{ID}$.

   Merge u, v into a single thread and update the representation.

   (a) Derive the new set of nodes, use v as representative for the two merged nodes and discard u.

   $$V_{new} = V - \{u\}$$

   (b) Compute the new reflexive transitive closure.

   $$\begin{aligned} \mathit{Succ}^*_{new} = & \{(p, s) | (p, s) \in \mathit{Succ}^*,\ p \neq u, s \neq u\} \\ & \cup \{(p, s) | (p, u) \in \mathit{Succ}^*, (v, s) \in \mathit{Succ}^*,\ p \neq u, s \neq u\} \\ & \cup \{(p, s) | (p, v) \in \mathit{Succ}^*, (u, s) \in \mathit{Succ}^*,\ p \neq u, s \neq u\} \end{aligned}$$

(c) Compute the new set of indirect dependencies.

$$\begin{aligned} ID_{new} \;=\; & \{(p,s) | (p,s) \in ID,\ p \neq u, s \neq u\} \\ & \cup \{(p,s) | (p,u) \in ID, (v,s) \in Succ^*,\ p \neq u, s \neq u\} \\ & \cup \{(p,s) | (p,v) \in ID, (u,s) \in Succ^*,\ p \neq u, s \neq u\} \\ & \cup \{(p,s) | (p,u) \in Succ^*, (v,s) \in ID,\ p \neq u, s \neq u\} \\ & \cup \{(p,s) | (p,v) \in Succ^*, (u,s) \in ID,\ p \neq u, s \neq u\} \end{aligned}$$

(d) Set $V = V_{new}$, $Succ^* = {Succ^*}_{new}$, and $ID = ID_{new}$.

7. Repeat from Step 6, until no more nodes can be merged.

Let us take a closer look at how ID is computed and what happens when two nodes are merged. Even though it looks different, the definition of ID given in Step 5 is equivalent to the definition in Section 4.4.1. In Step 6, when two nodes u, v are merged, the reduced graph has to be formed, and the reflexive transitive closure and indirect dependencies have to be updated. We simply discard u and use the node v as a representative for the two merged nodes. When computing the new successor relation, all successors of one node are now successors of all predecessors of the other node, and vice versa. In addition, all successors pointing or emerging from u have to be redirected to v instead.

Figure 4.8 shows what needs to happen when updating the indirect dependencies. Indirect dependencies going to one of the two nodes u or v have to be extend to all successors of the other node. Likewise, all indirect dependencies emerging from one the two nodes have to be extended to all predecessors of the other node. Again, all indirect dependencies incident to or emerging from u have to the redirected to the new representative v.

What is the complexity of above algorithm? That depends on the actual representation of ID and $Succ^*$. We will do the analysis here assuming they are represented by a connectivity matrix. Assume that the problem size n is the maximum of the number of edges, number of inlet names, and number of outlet names. Since the dataflow graph is acyclic, initially computing the transitive closure in can be done in $\mathcal{O}(n^2)$. Determining the set IO in Step 2 and the PID edges in Step 3 can be done in $\mathcal{O}(n^3)$. Computing ID requires $\mathcal{O}(n^2)$. Testing whether two nodes can be merged takes only constant time, since ID is represented as a matrix. As discussed earlier, at most $\mathcal{O}(n^2)$ pairs of nodes have to

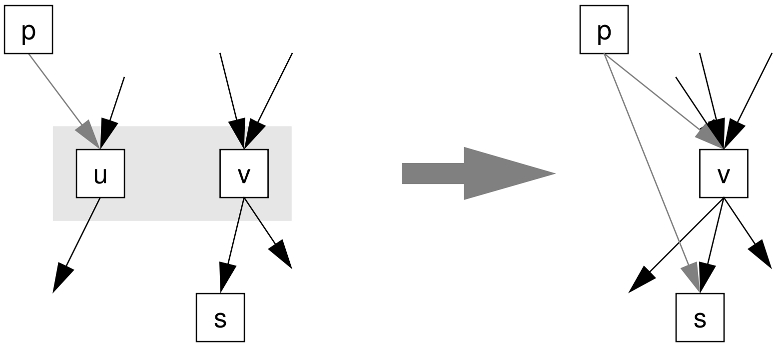


Figure 4.8: *After merging two nodes with separation constraint partitioning, the indirect dependencies (represented by dashed edges) have to be updated. As shown here, all indirect dependencies leading to u have to also point to all successors of v. The other three cases — indirect dependencies leading to v, coming from u or from v — are dealt with similarly.*

be tested, thus this part of Step 6 costs $\mathcal{O}(n^2)$. Merging occurs at most $\mathcal{O}(n)$ times and the complexity of Steps 6*(a)*–*(d)* is $\mathcal{O}(n^2)$. Overall, the total complexity of the algorithm is $\mathcal{O}(n^3)$.

In contrast, the complexity of dependence and demand set partitioning is $\mathcal{O}(n^2)$. For iterated partitioning four dependence and demand set partitioning steps are sufficient in practice. Thus it runs much faster than separation constraint partitioning for large basic blocks (we have experienced basic blocks of up to 600 nodes).

4.4.5 Correctness

Unlike dependence and demand set partitioning, it is much harder to understand intuitively the correctness of the separation constraint partitioning algorithm.

The most difficult part is the update of the of ID when two nodes u, v are merged. What is surprising in the update of ID, is that it is sufficient to focus on only one indirect dependence at a time. Assume that we have two indirect dependencies $(p, u), (v, s) \in ID$, as shown in Part (a) of Figure 4.9. Intuitively, one would expect that merging u and v should introduce an indirect dependence (p, s) as shown in Part (b) of the figure. This is not the case, because the two indirect dependencies may not co-exist, *i.e.*, in the context where (p, u) exists (v, s) does not exist and vice versa. Therefore, the indirect dependence (p, s) should not always be introduced, as doing so could introduce a circular indirect dependence as shown in Part (c) of the figure. If both indirect dependencies can co-exist, then the current algorithm introduces (p, s). To understand why, we distinguish two cases. In the case where one or both of the indirect dependencies are due to certain indirect dependencies, they are also present in the reflexive transitive closure $Succ^*$, thus (p, s) is introduced. In the other case where both (p, u) and (v, s) are solely due to potential indirect dependencies and both can co-exist, (p, s) is also admissible, and therefore already in the set of indirect dependencies. This may seem surprising, but is due to the fact that we are using inlet/outlet annotations to represent potential indirect dependencies and that we assume indirect dependencies to be present unless contradicted by certain dependencies.

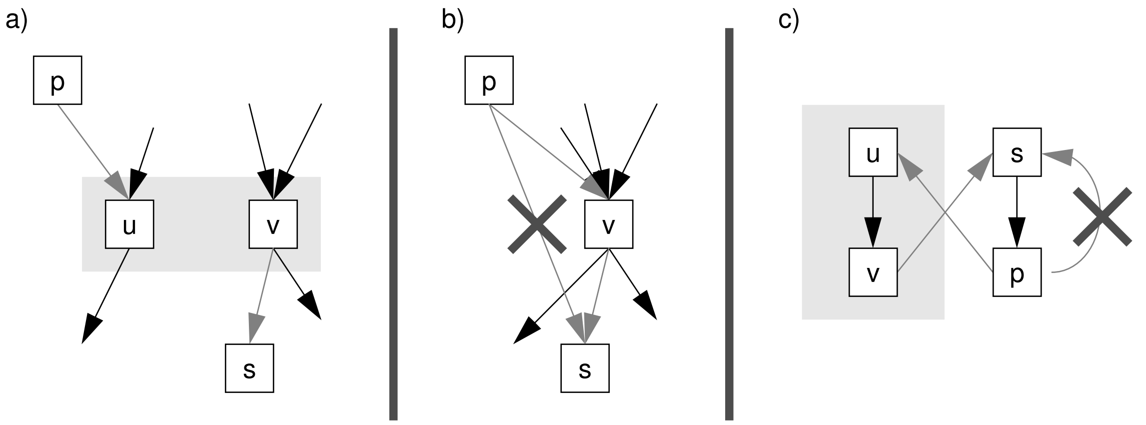


Figure 4.9: *After merging two nodes u and v, it is not necessary to follow two indirect dependencies. Assume that the dashed arcs in Part (a), $(p, u), (v, s) \in ID$, are indirect dependencies due to potential dependencies. The reason that the update algorithm does not have to consider introducing the indirect dependence (p, s) as shown in Part (b), is that if it is admissible, it will already exist. If it is not admissible, then a certain dependence from s to p exists and introducing an indirect dependence (p, s) results in a cycle, as shown in Part (c). This situation corresponds to the example from Figure 4.2.*

While we hope that our discussion so far has made it clear that this algorithm adequately deals with the separation constraints introduced by potential and certain indirect dependencies, it seems much less clear that the algorithm does not introduce static cycles within a basic block. The reason is that the nodes can get merged in any order. Consider the example in Figure 4.10, which contains one admissible potential indirect dependence, between the nodes N_1 and N_4. Initially, the nodes N_1 and N_4 are the only two nodes which cannot be merged. Assume that the algorithm first chooses to merge nodes N_1 and N_3. These two nodes do not form a correct thread because the path from N_1 to N_3 goes over the node N_2 which is not part of the thread. If on the next step N_2 is merged with N_4, it would never be possible to obtain a legal thread. Fortunately, this cannot happen. Merging nodes N_1 and N_3 introduces a new separation constraint between N_2 and N_4. The algorithm ensures that all nodes along paths between two nodes which get merged, always can, and eventually will, get merged with the two nodes. Phrased differently, at the end of the algorithm all regions will be completely convex and therefore form valid threads, even if this may not be the case during intermediate steps of the algorithm. Thus separation constraint partitioning is quite different from iterated partitioning. There the partitioning was correct after every step and we could choose to stop any time if so desired. Separation constraint partitioning, on the other hand, has to run until the end.

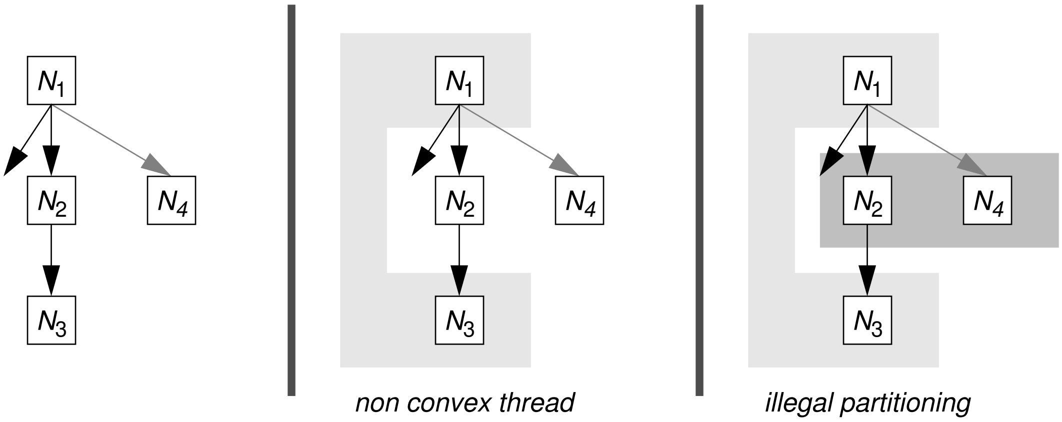


Figure 4.10: *Example showing that separation constraint partitioning may create non-convex threads during intermediate steps, but that the final result can never be an illegal partitioning.*

4.4.6 Merge Order Heuristics

The algorithm as presented so far does not specify the order in which each pair of nodes is visited and tested for merging. This flexibility is actually a big advantage, as it permits the algorithm to be combined with a heuristic that visits the nodes in an order as to minimize the communication, dynamic scheduling, and synchronization overhead. All three operations are expensive on commodity processors. On most parallel machines communication has the highest overhead, and therefore should be addressed first.

Merge order		Savings		
Node pair	Situation	Communication	Scheduling	Synchron.
Boundary Nodes	Function call	1 send, 1 receive	1 post	—
	Conditional	—	1 switch or merge	—
Interior Nodes	Directly connected	—	1 fork (implicit)	0-1 sync
	Common ancestor	—	1 fork (redundant)	0–1 sync
	Common successor	—	1 fork (redundant)	0–1 sync
	Not connected	—	—	sync var

Table 4.1: *Order in which heuristic for separation constraint partitioning tries to merge nodes.*

Our heuristic is to first try to merge nodes belonging to the same function call boundary (which reduces communication), then nodes at conditional boundaries (which reduces dynamic scheduling), and finally the remaining interior nodes of the basic block. This is summarized in Table 4.1. The intuition behind this strategy is as follows.

Function call boundary: Nodes at a function call boundary may require remote communication which is one of the most expensive operations because of communication overhead. The def site of a basic block consists of n receive nodes for the arguments and m send nodes for the results. Conversely, the corresponding call site has n send nodes for the arguments and m receive nodes for the results. If the caller and callee reside on different processors, a message has to be sent for every argument and every result. In the case where multiple send nodes and the corresponding receive nodes are grouped into a single thread, a single larger message can be created which reduces the communication overhead. This situation is shown in Figure 4.11: two argument send nodes in the caller are grouped into the same thread, as are the corresponding two receive nodes of the callee. At code generation time, these can be replaced with a single send/receive pair, which sends both arguments together. Now the communication overhead is incurred only once. Additional savings arise because the receiving thread needs to be posted only once. Note that this optimization can only be applied if the send nodes can be grouped together in all of the call sites which invoke the callee's basic block.

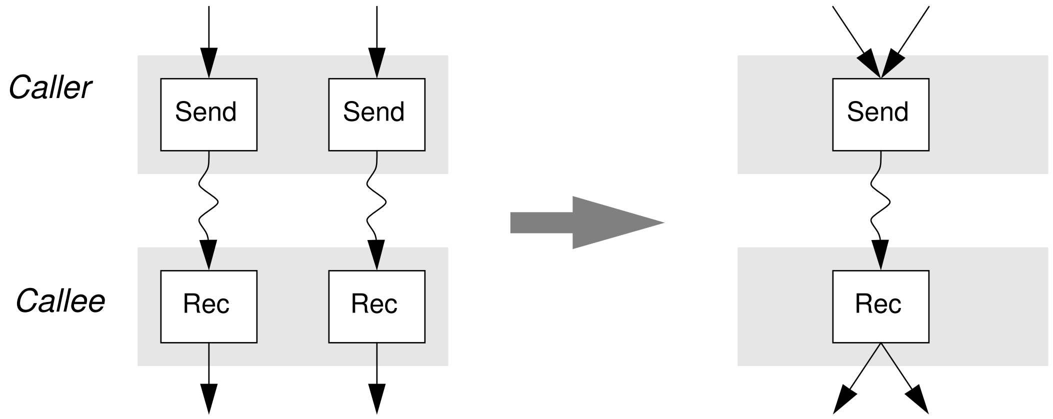


Figure 4.11: *Multiple send and receive nodes which are grouped into the same thread can be replaced with a single message, thereby decreasing the communication overhead.*

Conditional boundary: Next we focus on merging boundary nodes of conditionals. This optimization, discussed in [Sch91], minimizes the control transfer overhead in the case where the full power of non-strict conditionals is not required. So far, in our discussion of basic blocks and interfaces, we have represented conditionals as function calls, with send and receive nodes. In practice, each of the send/receive pairs at the top of a conditional represents a dataflow *switch* node which takes the input value and, depending on the predicate, steers it to one of the two arms of the conditional. Likewise, a send/receive pair at the bottom of the conditional represents a dataflow *merge* node, which takes the data value from one of the two arms and forwards it to the output. The optimizations for conditionals are similar to function calls, as shown in the Figure 4.12. Multiple switches that end up in the same thread, can be replaced with a single switch, thereby minimizing the control transfer overhead, as only a single conditional branch instruction needs to be created. Similarly, multiple merges can get grouped into a single merge, also saving a control transfer operation.

After nodes belonging to a basic block boundary have been merged, the heuristic focuses on interior nodes. The merge order for interior nodes is also given in Table 4.1 and is illustrated in Figure 4.13.

Directly connected nodes: Merging two nodes which are directly connected also reduces the control overhead. In general, edges crossing thread boundaries turn into control operations. The thread from which the edge emerges has to fork the thread to which

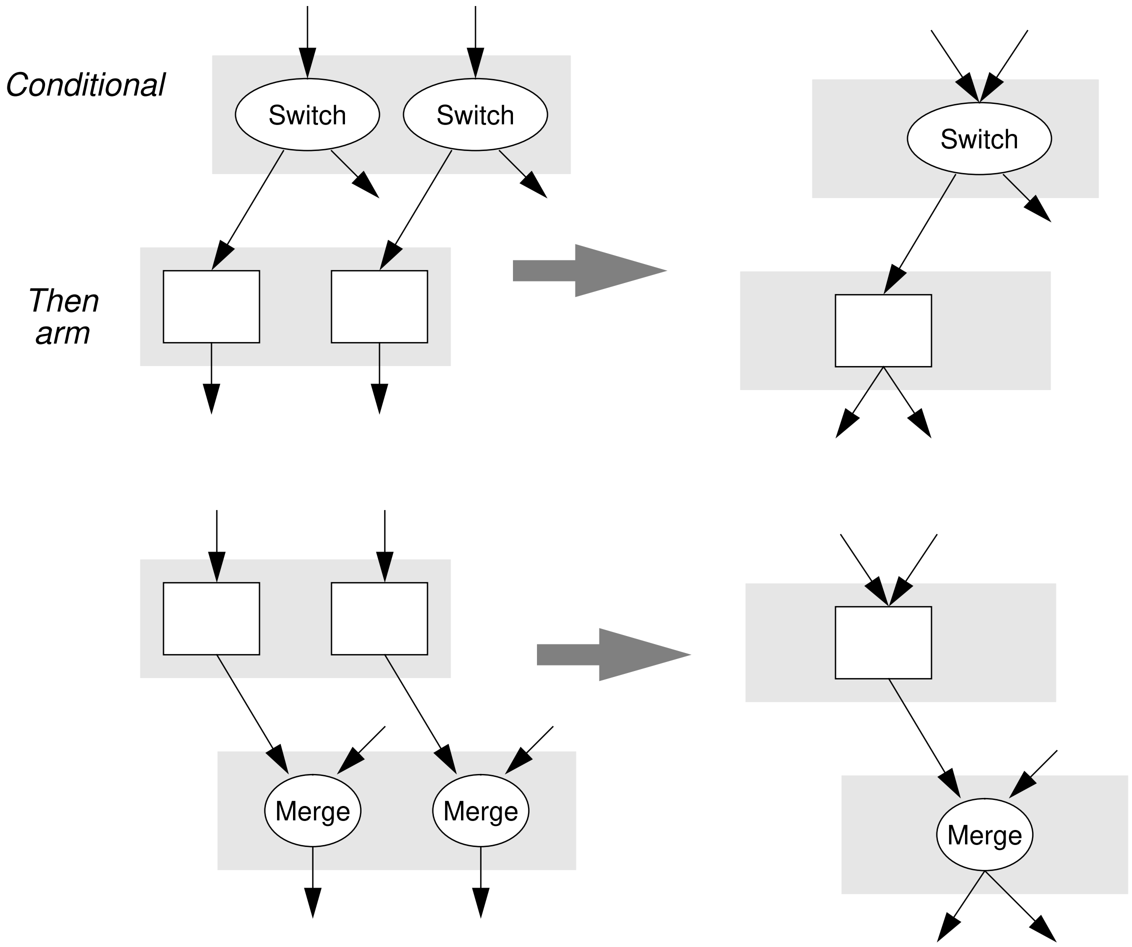


Figure 4.12: *Merging of switch and merge nodes at conditional boundary reduces control transfer overhead.*

it is connected. If there are also other threads forking the destination thread, then the fork also involves synchronization. If two nodes connected by a straight edge are placed into the same thread, then the control operation becomes implicit through the instruction ordering of the thread. This saves a fork and, in the case where multiple threads fork the destination thread, also saves synchronization. Another advantage in this situation is that updating the reflexive transitive closure and indirect dependencies is less expensive because one node is already a predecessor of the other.

Nodes with common ancestor: When two nodes with a common ancestor are merged, similar savings in control overhead can be obtained. For this to be case, the common ancestor has to be a direct predecessor of one of the nodes, as shown in Part (b) of Figure 4.13. The control represented by the edge connecting the ancestor node N_1 to node N_3 becomes redundant, as there exists another control path from N_1 to the merged node N_4. Thus a fork and synchronization may be saved.

Nodes with common successor: In the analog case, where two nodes with a common successor are merged, the savings are the same.

Unconnected nodes: Merging unconnected nodes only reduces the number of synchronization variables. On the other hand, merging two such nodes could create a new ancestor or successor for other nodes which can also be merged. In that case, the same gains as in the previous two cases may be obtained.

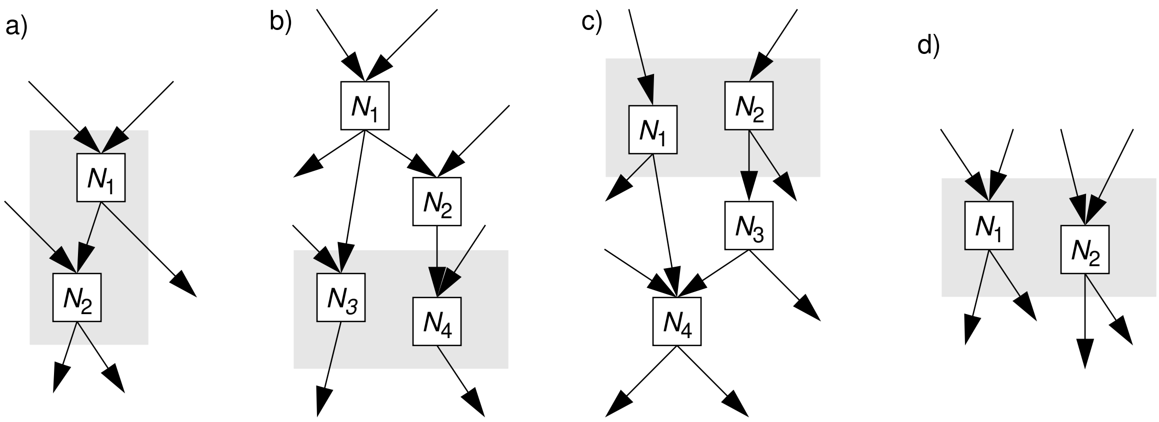


Figure 4.13: *Order in which the heuristic for separation constraint partitioning tries to merge interior nodes. These different situations reduce the control overhead to a varying degree. In Part (a) two directly connected nodes are merged. As a result, the control represented by the edge connecting N_1 to N_2 becomes implicit through the instruction ordering of the thread. In Part (b) two nodes with a common ancestor are merged. Here the control represented by the edge from N_1 to N_3 becomes redundant, because there is another path from N_1 to N_4. The same occurs when two nodes with a common successor are merged, as in Part (c). Merging nodes which are completely unconnected, as in Part (d), only reduces the number of synchronization variables which are required. On the other hand, merging two such nodes could lead to new situation of form (b) or (c).*

4.4.7 Summary

Separation constraint partitioning iteratively merges two nodes until no more nodes can be merged. A simple rule is used to determine whether two nodes can be merged or not. A separation constraint exists between two nodes if there can exist a indirect dependence between them. In that case, the nodes cannot be merged. This rule deals in a unified way with the partitioning constraints introduced by certain and potential indirect dependencies. This algorithm is guaranteed to derive maximal threads. After it has finished, it is impossible to do further merging without refining the annotation with interprocedural analysis.

The order in which the nodes are tested for merging is very important, and can affect the final result. Merging some nodes may inhibit the merging of other nodes. We have presented a heuristic which first tries to merge nodes which result in a larger savings in terms of communication and control overhead. Before merging the interior nodes, we should first try to refine the inlet and outlet annotations using interprocedural analysis to ensure that communication is reduced. Finally, after obtaining the best possible partitioning at the function call boundaries, we partition the interior of basic blocks.

Unfortunately, the computational complexity of separation constraint partitioning is much higher than iterated partitioning. This makes a big difference for large basic blocks. Our current strategy is to use separation constraint partitioning to group the most important nodes first, the nodes at function call and conditional boundaries. Thereafter, we use iterated partitioning to group the remaining interior nodes.

Chapter 5

Interprocedural Partitioning

The basic block partitioning algorithms presented so far are limited in their ability to derive threads, because without global analysis they must assume that every send in a basic block may feed back to any receive, unless contradicted by certain dependencies. This is captured by the singleton inlet and outlet annotations given initially to send and receive nodes. Global analysis can determine that some of these potential dependencies cannot arise and thereby improve the partitioning. For example, the information gained while partitioning a procedure can be used to improve the inlet and outlet annotations of its call sites. These refined annotations may share names, reflecting the sharing among dependence and demand sets present in the procedure. In addition, squiggly edges from argument send nodes to result receive nodes can be introduced if the procedure has the corresponding paths from the argument receives to result sends. Both the refined annotations and the squiggly edges help to better approximate the potential indirect dependencies and thereby improve subsequent partitioning.

The same optimizations are also possible in the reverse direction. The annotations of the def site of a procedure can be improved with the information present at its call site. Dependence and demand sets at the call site determine the new sharing in inlet and outlet annotations at the def site. Squiggly edges can be introduced from result send nodes back to argument receive nodes, if the corresponding paths from result receive nodes to argument send nodes exist in the call site. This optimization is more complicated if a procedure has more than one call site, in which case the new annotations and squiggly edges must be compatible with all of the call sites. We will first focus on single-def-single-call interfaces and discuss later how to extend this to multiple-call and multiple-def interfaces.

The structure of this chapter is as follows. Before giving the formal description of the interprocedural partitioning algorithm, Section 5.1 illustrates it using a simple example with two procedures. Section 5.2 presents the global partitioning algorithm, while Section 5.3 focuses on the global analysis for single-def-single-call interfaces. Section 5.4 extends the global analysis algorithm for multiple-call and multiple-def interfaces. This algorithm is illustrated in Section 5.5 using a conditional example. Section 5.6 discusses some limitations of interprocedural partitioning, while Section 5.7 shows how the analysis can be extended in the case of mutually recursive functions. Finally, Section 5.8 discusses additional refinements to the global partitioning algorithm.

5.1 Example of Interprocedural Partitioning

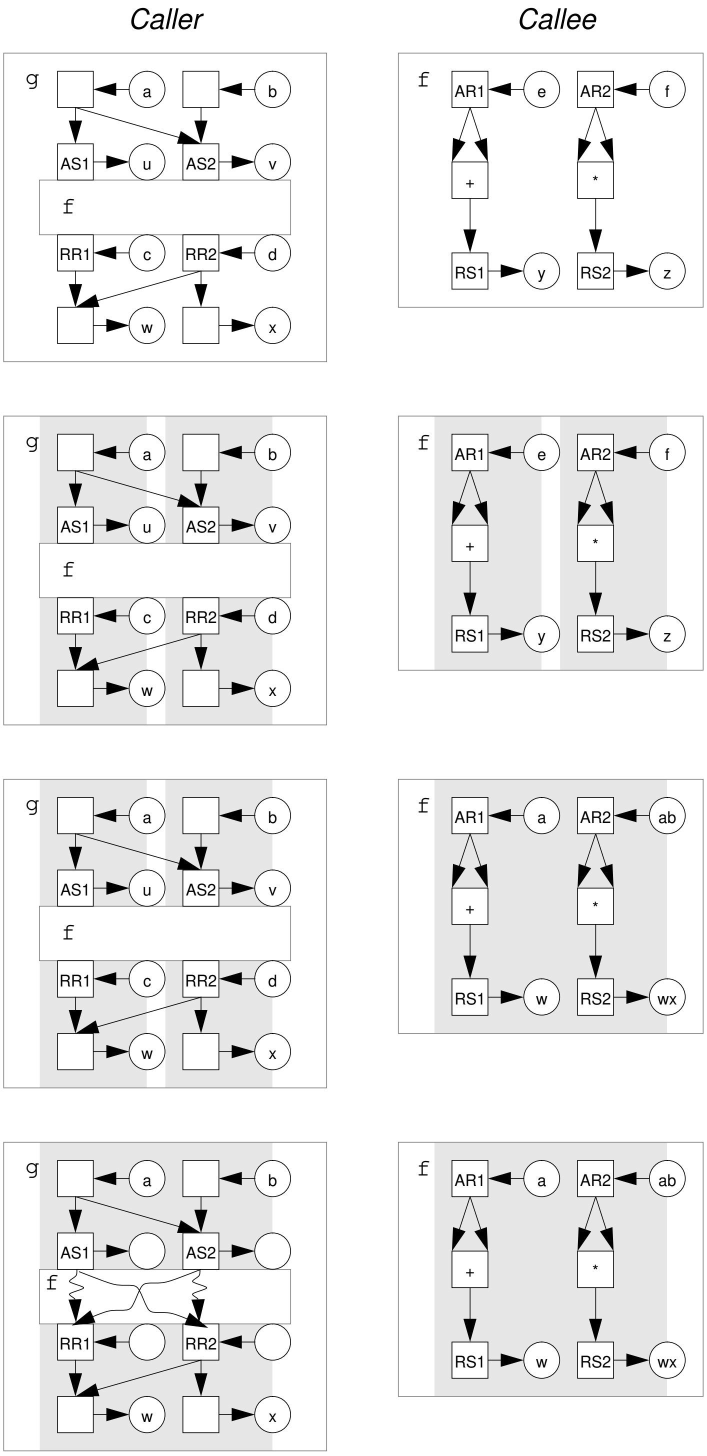
The interprocedural partitioning algorithm, especially the annotation propagation part of it, is fairly complex. Before presenting the algorithm formally, we discuss a small example which should help illustrate it. The example, given in Figure 5.1, consists of two basic blocks, a caller and callee, connected by a single-def-single-call interface. The left part of the figure shows the dataflow graph for the caller, the function `g`, while the right part shows the dataflow graph for the callee, the function `f`. Both procedures receive two arguments and return two results. The procedure `g` contains a call site of the procedure `f`, as indicated by the interior dashed rectangle, the two argument send nodes (AS1 and AS2), and result receive nodes (RR1 and RR2). The corresponding def site of the procedure `f` consists of the two argument receive nodes (AR1 and AR2) and two result send nodes (RS1 and RS2).

As shown in Part (a) of the figure, the algorithm starts by initially giving all receive and send nodes a unique singleton inlet or outlet annotation. This indicates that every inlet potentially depends on every outlet node, unless contradicted by certain dependencies. With this annotation, both procedures can be partitioned using any of the basic block partitioning algorithms presented in the previous chapter. As shown by the shaded regions in Part (b), partitioning the caller results in four threads, while partitioning the callee results in two threads. This is the best partitioning possible under the trivial annotation. The top four nodes of the caller cannot be placed into a single thread because there does not exist a dependence from a node with b in its inlet annotation to a node with u in its outlet annotation. Therefore, the partitioning algorithm has to assume that a potential indirect dependence may exist from the node with the outlet annotation $\{u\}$ back to the node with the inlet annotation $\{b\}$, which implies that the top left nodes nodes cannot be placed into the same thread as the top right nodes. Analogous arguments can be made for why the other threads have to stay separate.

To improve the partitioning, we must apply interprocedural analysis which propagates information across the interface. There are two possible ways to propagate information, from the callee to the caller and from the caller to the callee. Let us first explore what happens when propagating from the caller to the callee. Propagation involves introducing squiggly edges and refining inlet and outlet annotations. In this case, no squiggly edge is introduced, since the caller does not have a certain dependence path from a result receive node to an argument send node. The new inlet annotations given to the argument receive nodes at the def site reflect the dependence sets of the argument send nodes at the call site. As shown in Part (c) of the figure, the node AR1 gets the new inlet annotation $\{a\}$, while the node AR2 gets the inlet annotation $\{a, b\}$. Likewise, the new outlet annotations given to the result send nodes reflect the demand set of the corresponding result receive nodes, which are $\{w\}$ and $\{w, x\}$, respectively.

After reannotation, the new annotations may express sharing, *i.e.*, may have overlapping sets. In the example, the inlet name a appears in both inlet annotations, and the outlet name w appears in both outlet annotations. The new annotations correspond precisely to the situation previously shown in Figure 4.6. As discussed in Section 5.6, these annotations indicate that for any given context, the set of outlet nodes on which AR1 depends is a subset of the set of outlet nodes on which AR2 depends. Likewise, the set of inlet nodes which RS1 influences is a subset of the set of inlet nodes which RS2 influences. Now we can partition the callee again. Using separation constraint partitioning, we determine that $\{(a, w), (a, x), (b, w), (b, x)\}$ are all in IO. Therefore, there cannot exist a potential indirect dependence between any pair of nodes, and separation constraint partitioning can group all nodes of the callee into a single thread, as indicated by the shaded region in Part (d) of the figure.

As the example so far illustrates, capturing the sharing between annotations is key for successful partitioning. However, annotation propagation must be careful not to



a) Initial Annotation
Every receive is annotated with a unique singleton inlet name, and every send with a unique singleton outlet name.

b) Initial Partitioning
Partitioning of the caller results in four threads, while the callee gets two threads.

c) Reannotation of Callee
Annotation propagation from the caller to the callee results in the new inlet and outlet annotations.

d) Partitioning of Callee
With the new annotation separation constraint partitioning can obtain a single thread.

e) Reannotation of Caller
Annotation propagation from the callee to the caller introduces four squiggly edges and eliminates the annotations.

f) Partitioning of Caller
Partitioning now obtains two threads. Further reannotation and partitioning does not improve this.

Figure 5.1: *Example of interprocedural partitioning with annotation propagation across a single-def-single-call interface.*

introduce unintended sharing across different sites. For example, when a basic block contains multiple call sites to the same function, the new annotations given to the different call sites should be distinct. Otherwise it may seem as if internal inlets or outlets in different call sites are related, which may result in an illegal partitioning. Therefore, the annotation propagation algorithm may have to rename the new inlet and outlet annotations to avoid unintended sharing among different call sites.

Returning to the example we next propagate annotations from the callee to the caller. This time we can introduce four squiggly edges at the call site, one from every argument send node to every result receive node, since the corresponding certain dependence paths are present in the callee now that it consists of a single thread. These squiggly edges capture all of the dependencies which can arise at this call site. Therefore, we can give the argument send and result receive nodes at the call site empty inlet and outlet annotations as shown in Part (e) of the figure. Applying separation constraint partitioning, we obtain now that $IO = \{(a, w), (a, x), (b, w), (b, x)\}$, which indicates that there cannot exist a potential indirect dependence between any pair of nodes. The new squiggly edges and the improved annotations help to eliminate the potential dependencies present before annotation propagation. Therefore, the two top threads in the caller can be merged into a single thread, as shown in Part (f) of the figure. Likewise, the bottom two threads can be grouped into a single thread. Because the top and the bottom thread are connected by squiggly edges, they have to remain separate. Thus, partitioning the caller results in two threads, the best partitioning that can be obtained for this example. Further reannotation and partitioning does not improve this. Note that the resulting threads are the same as for a strict sequential program.

The last reannotation step shows the importance of squiggly edges. Since squiggly edges represent certain dependencies they help contradict potential dependencies. In addition, they may also result in smaller inlet and outlet annotations, as dependencies captured by squiggly edges do not have to be captured anymore by potential dependencies.

We are now going to discuss several subtle but important points about the interprocedural partitioning algorithm. After repartitioning a procedure, annotations which were derived from the procedure prior to repartitioning may become invalid. The reason is that partitioning changes the connectivity of the graph, and thereby also the dependence and demand sets which were used to derive the annotations. In the above example, after repartitioning the caller, the dependence sets encountered at the two argument send nodes of the call site are $\{a, b\}$ and $\{a, b\}$, which is different from the annotations $\{a\}$ and $\{a, b\}$ which were previously assigned to the argument receive nodes of the def site of `f`. In this particular case, it would still be safe to use the old annotations, as the old annotations have less sharing than the new annotations.[1] In general, this may not the case, and it may be necessary to invalidate the old annotations. Our partitioning algorithm solves this problem, by always resetting to the trivial annotation all of those annotations which use information from a procedure which is partitioned. This ensures that the annotations are always valid.

Because of the interaction between annotation propagation and partitioning, it is not possible to partition multiple basic blocks in parallel. For example, at first glance it seems quite natural to partition all procedures in parallel, then simultaneously propagate from callees to callers and from callers to callees, and, using the new annotations, partition all procedures again. This does not work, because partitioning a caller may invalidate the annotations at the callee, and vice versa. This also brings up another subtle point. A procedure whose annotation uses information about the procedure itself may have its own annotation invalidated during partitioning. Our partitioning algorithm therefore disallows partitioning the procedure in this case.

These subtle points will be discussed in much more detail later in this chapter; they are only mentioned here to increase the readers awareness and help in the understanding of the formal description of the global partitioning and annotation propagation algorithm presented in the next two sections.

5.2 Global Partitioning

The global partitioning algorithm starts by initially giving all receive and send nodes the trivial annotation, *i.e.*, each receive node is assigned a unique singleton inlet annotation and each send node is assigned a unique singleton outlet annotation. Then the algorithm

[1] The notion of an annotations safely approximating the sharing will be explained in more detail later in Section 5.4. It is always safe to use an annotation with less sharing, specifically it is always safe to use the trivial annotation which exhibits no sharing at all since every node is annotated with a different name. For our example, it should be obvious that the new sets $\{a, b\}$ and $\{a, b\}$ are identical and therefore express more sharing than $\{a\}$ and $\{a, b\}$. Thus it is safe to keep $\{a\}$ and $\{a, b\}$.

iteratively applies one of following three transformations until no further changes occur: (a) a basic block is partitioned using its annotations, (b) the annotations of a site are refined by annotation propagation, or (c) the annotation of some send or receive node is set back to its initial trivial annotation. Any of the basic block partitioning algorithms presented in the previous chapter may be used for partitioning a basic block. When a basic block is partitioned, all annotations which use some information from this basic block have to be recomputed. To ensure that the annotations are always valid, the algorithm resets them to the trivial annotation. To keep track of which annotations have to be reset, the algorithm associates with every node a variable, $Uses$, which indicates the set of basic blocks which were used to compute its annotation.

Algorithm 6 (Global partitioning) *Given a structured dataflow graph:*

1. *(Initial annotation)* Give every send and receive node a trivial annotation, *i.e.*,
   (a) assign every receive node r a unique singleton inlet name and set *Uses*$(r) = \emptyset$, and
   (b) assign every send node s a unique singleton outlet name and set *Uses*$(s) = \emptyset$.
2. Apply one of the following three transformations.
   (a) *(Reset an annotation)* Select an arbitrary send or receive node u, give it its initial trivial annotation and set *Uses*$(u) = \emptyset$.
   (b) *(Annotation propagation)* Select an arbitrary interface which connects a site S in a basic block *BB* to a site S' in a basic block *BB*$'$. Propagate new annotations from S to S' following the steps described below in Algorithm 7.
   (c) *(Partitioning)* Select an arbitrary basic block *BB* for which *BB* $\notin$ *Uses*(u) for all nodes $u \in$ *BB*. Perform the following two steps.
      i. Partition the basic block with any legal partitioning algorithm.
      ii. Give all send and receive nodes v for which *BB* $\in$ *Uses*(v) the initial trivial annotation and set *Uses*$(v) = \emptyset$.
3. Repeat from Step 2 until no further changes.

The algorithm has to be combined with a strategy for choosing the order in which to visit the basic blocks and do the partitioning and reannotation. The algorithm offers great flexibility. In the case where the call graph forms a tree, a natural strategy is to do alternating passes up and down the tree. The algorithm can start with the basic blocks at the leaves of the tree, partition each separately, propagate the annotations up one level, partition those basic blocks, and continue this until the root of the tree is reached. Next, the algorithm can work its way down the tree. This process can be repeated until no further changes occur.

In the case where the call graph contains cycles or forms a dag, we need to be more careful. The reason is that we have to avoid propagating information about the structure of a basic block back to itself. This can happen if annotations are propagated around a cycle of the call graph, or from one call site of a procedure into another. As discussed before, in that case, partitioning of the basic block becomes impossible because partitioning a basic block may invalidate its own annotation. This is captured in above algorithm by the prerequisite in Step 2c, which inhibits partitioning of a basic block in the case where some of its annotations use information about the basic block itself. Our approach is to limit the algorithm to some spanning tree of the call graph and use the traversal strategy described above.

Actually, the algorithm does not need to run until no further changes occur. The structured dataflow graph is correctly partitioned at any point, so the algorithm can stop prematurely without affecting correctness. It need only run until the end to find a solution where no further thread grouping is possible. Obviously, this algorithm still works greedily. The final result depends on the order in which annotations are propagated and basic blocks are partitioned.

We now describe in more detail the annotation propagation step.

5.3 Global Analysis

The reannotation step propagates information across a single-def-single-call interface. The interface connects site S in the basic block BB to the site S' in BB'. As shown in Figure 5.2, it relates the send nodes $s_1, \ldots, s_n$ in site S to the receive nodes $r'_1, \ldots, r'_n$ in site S'. It also relates the send nodes $s'_1, \ldots, s'_m$ in site S' to the receive nodes $r_1, \ldots, r_m$ in site S. In the case where S is a call site and S' is a def site, the nodes s_i and r'_i handle the arguments and the nodes s'_j and r_j handle the results. In the case where S is a def site and S' is a call site, the nodes s_i and r'_i handle the results, while the nodes s'_j and r_j handle the arguments.

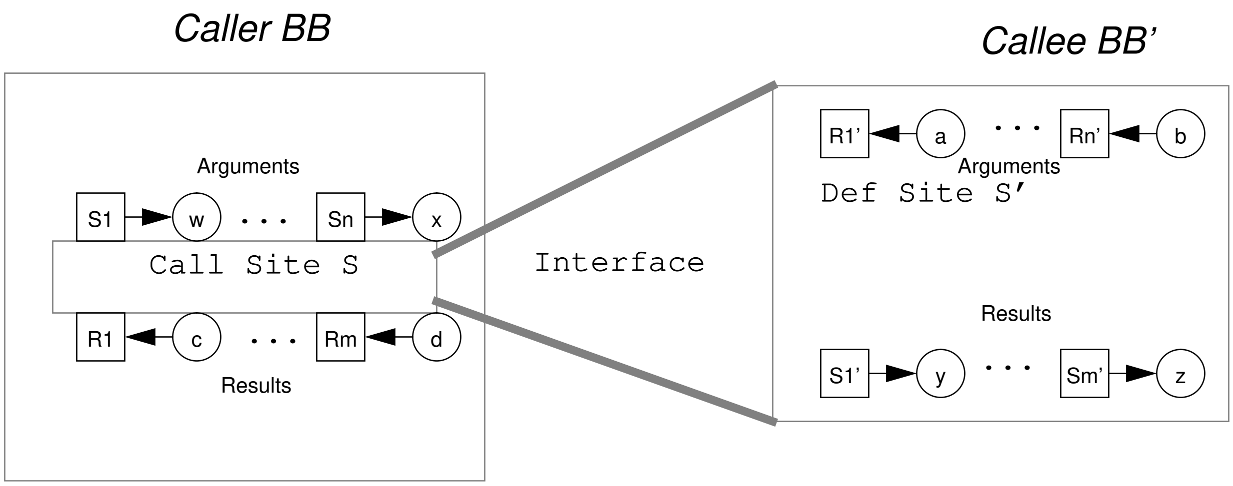


Figure 5.2: *Nodes comprising a single-def-single-call interface.*

The annotation propagation from the site S in the basic block BB to the site S' in BB' involves introducing squiggly edges and refining inlet and outlet annotations. Each receive node of S' is assigned a new inlet annotation which reflects the dependence set of the corresponding send node in S. Likewise, each send node of S' gets a new outlet annotation which reflects the demand set of the corresponding receive node in S. A squiggly edge is introduced from a send to a receive node in S', if the corresponding path exists in BB. In the subsequent algorithm, we use $\theta(u)$ to denote the thread which contains the node u of the original graph.

Algorithm 7 (Annotation Propagation) *Let the site S in the basic block BB be connected to the site S' of the basic block BB$'$ by a single-call-single-def interface. Let $(r_1, \ldots, r_n)$ and $(s_1, \ldots, s_m)$ denote the collection of receive and send nodes for the site S. Let $(r'_1, \ldots, r'_m)$ and $(s'_1, \ldots, s'_n)$ denote the collection of receive and send nodes for the site S'. The following steps compute the new annotation at the site S'. In the subsequent steps, the indices i and j run over the intervals $1 \leq i \leq n$ and $1 \leq j \leq m$.*

1. Let $\mathit{Dem}_i = \mathit{Dem}(\theta(r_i))$, where this demand set is computed in the reduced graph of the basic block *BB*, but where the send outlets of site S are given new unique names $\mathit{Outlet}(s_j) = \{\sigma_j\}$ for the purpose of computing these demand sets.

2. Let $\mathit{Dep}_j = \mathit{Dep}(\theta(s_j))$, where this dependence set is computed in the reduced graph of the basic block *BB*, but where the receive inlets of site S are given new unique names $\mathit{Inlet}(r_i) = \{\rho_i\}$ for the purpose of computing these dependence sets.

3. For each i, j such that $\sigma_j \in \mathit{Dem}_i$ (equivalently, such that $\rho_i \in \mathit{Dep}_j$, both hold if and only if there exists a path from $\theta(r_i)$ to $\theta(s_j)$), do

   (a) Add a squiggly arc (s'_i, r'_j) in site S'.

   (b) Set $\mathit{Dem}_i \leftarrow \mathit{Dem}_i - \{\sigma_j\}$.

   (c) Set $\mathit{Dep}_j \leftarrow \mathit{Dep}_j - \{\rho_i\}$.

4. Set $\mathit{Outlet}(s'_i) \leftarrow \mathit{Dem}_i$ in site S' (where Dem_i is renamed if necessary to avoid unintended sharing with other sites in *BB*′).

5. Set $\mathit{Inlet}(r'_j) \leftarrow \mathit{Dep}_j$ in site S' (renamed if necessary to avoid unintended sharing).

6. Set $\mathit{Uses}(s'_i) = \{\mathit{BB}\} \cup \bigcup_{u \in \mathit{Succ}^*(\theta(r_i))} \mathit{Uses}(u)$, where Succ^* is computed in the reduced graph of the basic block *BB*, and where $\mathit{Uses} = \emptyset$ for the send nodes of site S and receive nodes of *BB* for the purpose of computing these sets.

7. Set $\mathit{Uses}(r'_j) = \{\mathit{BB}\} \cup \bigcup_{u \in \mathit{Pred}^*(\theta(s_j))} \mathit{Uses}(u)$, where Pred^* is computed in the reduced graph of the basic block *BB*, and where $\mathit{Uses} = \emptyset$ for the receive nodes of site S and send nodes of *BB* for the purpose of computing these sets.

Note that this algorithm works the same, whether we are propagating from a call site to a def site, or from a def site to a call site.

To understand the algorithm, the reader should first apply it in all details to the example from Figure 5.1, and convince himself that it produces the desired result. We will only give a brief discussion of the step which propagates annotations from the caller to the callee and results in the refined annotations shown in Part (c) of the Figure 5.1. First, the algorithm computes the demand sets for the result receive nodes, replacing the outlet annotations $\{u\}$ and $\{v\}$ with $\{\sigma_1\}$ and $\{\sigma_2\}$. We obtain that $Dem_1 = Dem(\theta(\text{RR1})) = \{w\}$ and $Dem_2 =$

$Dem(\theta(\text{RR2})) = \{w, x\}$. Likewise we compute the dependence sets for the argument send nodes and obtain $Dep_1 = Dep(\theta(\text{AS1})) = \{a\}$ and $Dep_2 = Dep(\theta(\text{AS2})) = \{a, b\}$. Since no σ_j appears in a Dem_i we do not introduce any squiggly edge at the callee. The first argument receive node of the callee is given the inlet annotation $\{a\}$ and the second argument receive node the inlet annotation $\{a, b\}$, reflecting the dependencies present in the caller. Likewise the first result send node is given the outlet annotation $\{w\}$ and the second result send node the outlet annotation $\{w, x\}$. Finally we set the $Uses$ set of all receive and send nodes in the callee to contain the caller.

This algorithm has several subtle points, some of which we hinted at during the description of the example of interprocedural partitioning. First, the new annotations may have to be renamed. As discussed earlier, this is to avoid unintended sharing between different call sites, which in turn could lead to an illegal partitioning. Another subtle point is that inlet and outlet annotations of the send and receive nodes at site S are given new unique names for the purpose of computing the demand and dependence sets. There are two reasons for this. It makes deriving the squiggly edges convenient, as we just have to check whether $\sigma_j \in Dem_i$ or equivalently $\rho_i \in Dep_j$. This only holds if there exists a path from $\theta(r_i)$ to $\theta(s_j)$, in which case we can introduce a a squiggly arc from s'_i to r'_j in site S'. More importantly, it ensures that information about a basic block is not immediately propagated back to itself. The nodes at site S will contain BB' in their $Uses$ set if their annotations were previously refined by annotation propagation from S' to S. As a result, any subsequent propagation from S to S', may carry information about BB' back into annotations of S'. This is not desirable, as in that case it is not possible to partition BB'.

The last difficulty, is how the algorithm keeps track of the $Uses$ variable. Computing these is analogous to computing dependence or demand sets. The new inlet annotation for a receive node is derived from the dependence set of the corresponding send node. Therefore, its $Uses$ set should be the union over the $Uses$ set of all nodes which contributed to computing the dependence set. Similarly, demand sets are used for computing the new outlet annotations, and the new $Uses$ set is the union over the $Uses$ set of all nodes which contribute towards the demand set.

5.4 Multiple Callers and Conditionals

The annotation propagation algorithm must be refined for the cases where multiple call sites are connected to a single def site (a procedure being called from many places), and where multiple def sites connect to a single call site (conditionals). In the case where information is propagated from a single source site to multiple destination sites, the algorithm described in the previous section can be used, it just is applied to each destination site individually. In the other case, where information is propagated from multiple source sites into a single destination site, the new annotations have to be consistent with all of the source sites, *i.e.*, the sharing between the inlet and outlet annotations has to safely approximate the sharing encountered at each of the source sites and squiggly edges can only be introduced if the corresponding paths exist in all of the source sites.

How do we extend Algorithm 7 to safely approximate the sharing? Let us first understand what the sharing expresses. In the case of a single-def-single-call site interface, the new annotations given to the receive nodes reflect the dependence sets of the corresponding send nodes at the source site, and the new annotations given to the send nodes reflect the demand sets of the corresponding receive nodes. Multiple dependence or demand sets may contain common inlet or outlet names, *i.e.*, they may share names. This sharing also happens at the new annotations. The names themselves are not important, but rather the way they relate. For example, in Figure 5.1 we gave the callee the new inlet annotations $\{a\}$ and $\{a, b\}$. We can just as well rename these annotations and use $\{c\}$ and $\{c, d\}$ instead. We can also use the annotations $\{c, d\}$ and $\{c, d, e\}$, even though this is not just a simple renaming of the original set. The important feature which characterizes the sharing is that the first set is a proper subset of the second set and they are both different from the empty set. Note that the annotations $\{a\}$ and $\{b\}$ do not exhibit the same sharing; as should be obvious, their degree of sharing is less.

What does it mean to safely approximate the sharing? During reannotation it is always safe to assign new annotations which exhibit less sharing. When using annotations which exhibit less sharing, annotation based partitioning will over approximate the potential dependencies, because fewer *IO* dependencies from inlet names to outlet names can be proven to exist. This in turn may only result in more separation constraints which keep thread sizes small and, as a consequence, it is never possible to form larger illegal threads. The trivial annotations exhibit no sharing at all, since every node is annotated with a different singleton annotation. Therefore, it is always correct to assign the trivial annotation.

The mathematical properties of the sharing among annotations can be captured formally by *congruence syndromes* which are defined in Appendix A. Congruence syndromes characterize the degree of sharing for a set of annotations. These congruence syndromes form a lattice. Annotations with more sharing have a congruence syndrome higher up in the congruence syndrome lattice. The trivial annotation has no sharing at all; its congruence syndrome is the bottom element in the congruence syndrome lattice. It is always safe to use an annotation which exhibits less sharing, *i.e.*, has a congruence syndrome which is lower in the lattice.

In the case where annotations are propagated from multiple sites to a single site, it is necessary to derive an annotation which is a safe approximation of the annotations of each individual site. This can be obtained easily by first renaming the individual annotations and then taking their union. This new annotation is correct because it exhibits less sharing than any of the individual annotations. Phrased more formally, the congruence syndrome of the new annotation is the greatest lower bound of the congruence syndromes of all individual annotations. Intuitively, this is the case, because after renaming the individual annotations and taking their union, two sets of the new annotations will only be identical if the corresponding two sets are identical for all individual annotations.

5.4.1 Propagation Rules

Our discussion shows how Algorithm 7 has to be modified in order to propagate annotations from multiple sites. A squiggly edge is only introduced if the corresponding path exists in all source sites. The new outlet annotations are the union of all renamed demand sets from the source sites, and the new inlet annotation are the union of all renamed dependence sets from the source sites. Likewise, the new $Uses$ sets are just the union of the uses sets from every source site.

Algorithm 8 (Multiple Site Annotation Propagation) *Let the site S in the basic block BB be connected to sites $S_1, \ldots, S_l$ of the basic blocks BB$_1, \ldots,$ BB$_l$. Let $(r_1, \ldots, r_m)$ and $(s_1, \ldots, s_n)$ denote the collection of receive and send nodes for the site S. Let $(r_1^k, \ldots, r_n^k)$ and $(s_1^k, \ldots, s_m^k)$*

denote the collection of receive and send nodes for the site S_k. The following steps are used to compute the new annotations at the site S. The indices i, j, and k run over the intervals $1 \leq i \leq n$, $1 \leq j \leq m$, and $1 \leq k \leq l$.

1. Let $Dem_i^k = Dem(\theta(r_i^k))$, where this demand set is computed in the reduced graph of the basic block BB_k, but where the send outlets of site S_k are given new unique names $Outlet(s_j^k) = \{\sigma_j^k\}$ for the purpose of computing these demand sets.

2. Let $Dep_j^k = Dep(\theta(s_j^k))$, where this dependence set is computed in the reduced graph of the basic block BB_k, but where the receive inlets of site S_k are given new unique names $Inlet(r_i^k) = \{\rho_i^k\}$ for the purpose of computing these dependence sets.

3. For each i, j such that $\sigma_j^k \in Dem_i^k$ for all k (equivalently, such that $\rho_i^k \in Dep_j^k$ for all k), do

   (a) Add a squiggly arc (s_i, r_j) in site S.

   (b) Set $Dem_i^k \leftarrow Dem_i^k - \{\sigma_j^k\}$.

   (c) Set $Dep_j^k \leftarrow Dep_j^k - \{\rho_i^k\}$.

4. Set $Outlet(s_i) \leftarrow \bigcup_k Dem_i^k$ in site S (where the Dem_i^k are renamed if necessary to avoid unintended sharing with other sites in BB).

5. Set $Inlet(r_j) \leftarrow \bigcup_k Dep_j^k$ in site S (renamed if necessary to avoid unintended sharing).

6. Set $Uses(s_i) = \bigcup_k(\{BB_k\} \cup \bigcup_{u \in Succ^*(\theta(r_i^k))})Uses(u)$, where $Succ^*$ is computed in the reduced graph of the basic block BB_k, and where $Uses = \emptyset$ for the send nodes of site S_k and receive nodes of BB_k for the purpose of computing these sets.

7. Set $Uses(r_j) = \bigcup_k\{BB_k\} \cup \bigcup_{u \in Pred^*(\theta(s_j^k))} Uses(u)$, where $Pred^*$ is computed in the reduced graph of the basic block BB_k, and where $Uses = \emptyset$ for the receive nodes of site S_k and send nodes of BB_k for the purpose of computing these sets.

5.5 Conditional Example

Figure 5.3 illustrates the behavior of this algorithm for a conditional. The conditional has three inputs and three outputs. First, every send and receive node is annotated with

a) Initial Annotation
Every receive is annotated with a unique singleton inlet name, and every send with a unique singleton outlet name.

b) Initial Partitioning
Partitioning the caller results in six threads, while each of the callees gets three threads.

c) Reannotation of Caller
Annotation propagation from the callees to the caller results in the new inlet and outlet annotations and introduces three squiggly edges.

d) Partitioning of Caller
With the new annotation all nodes at the bottom of the caller can be placed into a single thread.

e) Reannotation of Callees
Annotation propagation from the caller to the callees introduces the new annotations.

f) Partitioning of Callees
Since all outlet annotations are the same, partitioning of the callees results in a single thread each.

e) Reannotation of Caller
Annotation propagation from the callees to the caller introduces nine squiggly edges and removes all annotation names.

f) Partitioning of Caller
Partitioning now obtains two threads. Further reannotation and partitioning does not improve this.

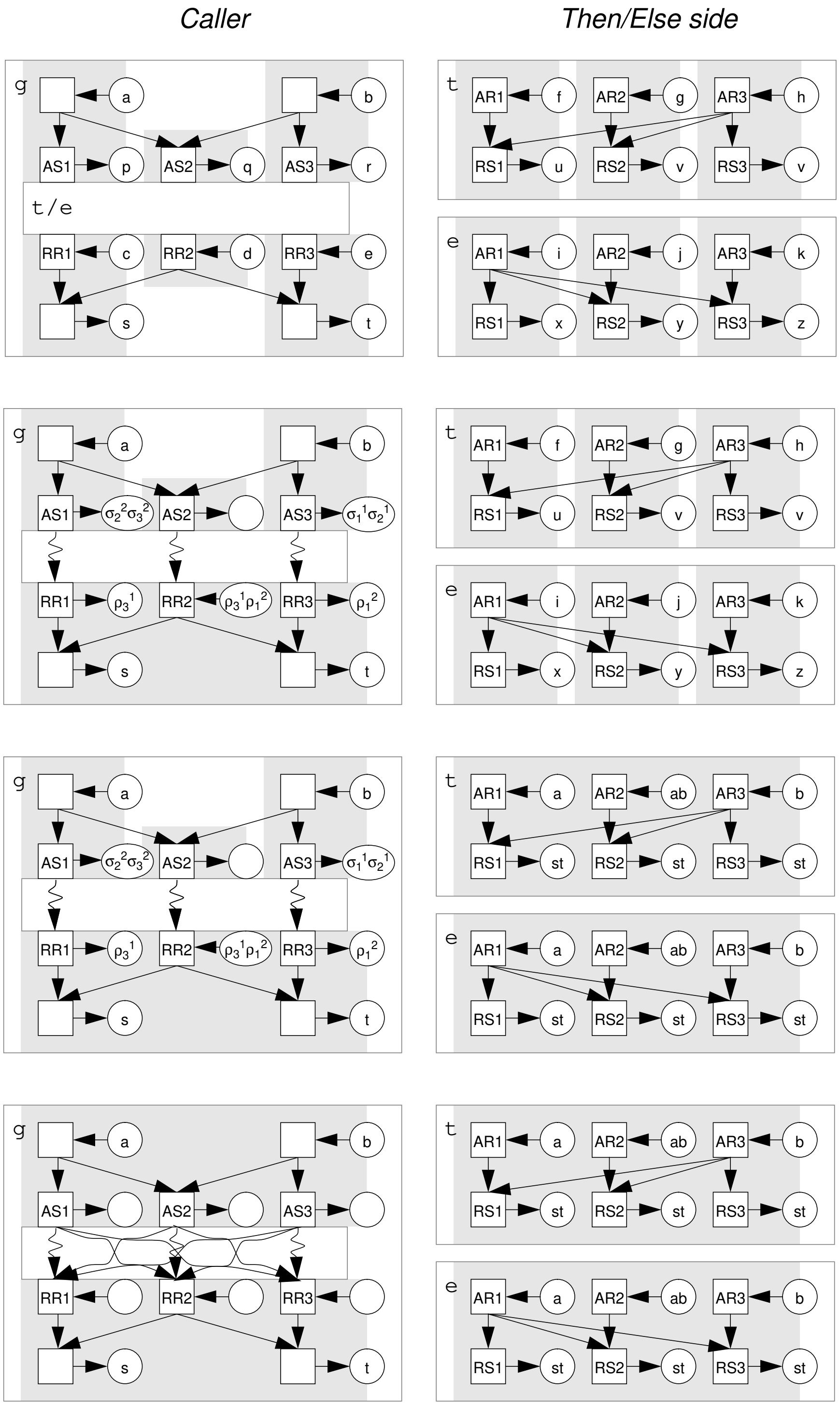


Figure 5.3: *Example of interprocedural partitioning across a conditional.*

a unique singleton inlet and outlet name. Under this annotation, partitioning the caller results in six threads, while each arm of the conditional results in three threads (Part b).

Next, we propagate annotations from both arms of the conditional into the caller. In the callees, we rename the outlet annotations of the send nodes to σ_j^k $(1 \leq j \leq 3, 1 \leq k \leq 2)$ and compute the demand sets for the receive nodes. For the "then" side, we obtain $Dem_1^1 = \{\sigma_1^1\}$, $Dem_2^1 = \{\sigma_2^1\}$, and $Dem_3^1 = \{\sigma_1^1, \sigma_2^1, \sigma_3^1\}$. Likewise, for the "else" side, we obtain $Dem_1^2 = \{\sigma_1^2, \sigma_2^2, \sigma_3^2\}$, $Dem_2^2 = \{\sigma_2^2\}$, and $Dem_3^2 = \{\sigma_3^2\}$. In the same fashion, we compute the dependence sets following the rule given in Step 2 of the algorithm. For the "then" side, we obtain $Dep_1^1 = \{\rho_1^1, \rho_3^1\}$, $Dep_2^1 = \{\rho_2^1, \rho_3^1\}$, and $Dep_3^1 = \{\rho_3^1\}$. For the "else" side, we obtain $Dep_1^2 = \{\rho_1^2\}$, $Dep_2^2 = \{\rho_1^2, \rho_2^2\}$, and $Dep_3^2 = \{\rho_1^2, \rho_3^2\}$. From these sets, we derive the squiggly edges. Since $\sigma_1^1 \in Dem_1^1$ and $\sigma_1^2 \in Dem_1^2$, the prerequisite of Step 3 is satisfied for both sites, and we can introduce a squiggly edge in the caller from AS1 to RR1. Similarly, we can introduce two other squiggly edges in the caller, from AS2 to RR2, and from AS3 to RR3. When inserting a squiggly edge from ASi to RRj, we also have to remove the corresponding σ_j^k from Dem_i^k and ρ_i^k from Dep_j^k. Finally, we take the union over the remaining dependence and demand sets and obtain $\{\sigma_2^2, \sigma_3^2\}$, $\{\}$, and $\{\sigma_1^1, \sigma_2^1\}$ as the new outlet annotations of the send nodes of the caller, and $\{\rho_3^1\}$, $\{\rho_3^1, \rho_1^2\}$, and $\{\rho_1^2\}$ as the new inlet annotation for the receive nodes. This situation is shown in Part (c) of the figure. Subsequent partitioning of the caller results in a single thread for the five bottom nodes, it does not improve the partitioning of the top nodes.

Now we propagate annotations in the opposite direction, from the caller to the callees. This is done by applying the single-def-single-call interface propagation rule described in Algorithm 7 to each arm of the conditional individually. The inlet annotations of the callees become $\{a\}$, $\{a, b\}$, and $\{b\}$, while the outlet annotations are all the same $\{s, t\}$. Partitioning under this new annotation groups the nodes of each arm into a single thread.

Finally, we propagate once more from callees to the caller. Since both arms consist of a single thread, we introduce nine squiggly edges in the caller and and assign new empty inlet and outlet annotations. Partitioning the caller leads now to two threads, the best possible for this example. Additional propagation and partitioning does not improve it.

5.6 Limitations of Interprocedural Partitioning

A prerequisite for partitioning a basic block BB in Algorithm 6 is that none of its annotations use any information about the basic block, *i.e.*, $BB \notin Uses(u)$ for all $u \in BB$. This severely limits the global analysis in the case of mutually recursive functions and in the case where a function has multiple call sites, because the analysis cannot be done across these. This section explains why the restriction is required. We start by giving an intuitive argument and presenting an example which highlights the problem. This problem actually went unnoticed in prior work. Developing the proof of correctness for our algorithm made it visible. In the next section, we discuss refinements to the partitioning algorithm which work in these cases.

Global partitioning, like many other interprocedural program optimizations, consists of three phases. First, information captured by analyzing each individual procedure is summarized. Then global analysis propagates this information across procedure boundaries and combines them into refined annotations. Finally, these annotations are used to determine the actual program transformation, in our case the partitioning of instructions into threads. In order for the program transformation to be correct, it has to be conservative and cannot perform transformations which invalidate the result of the analysis. The analysis and the actual program transformation have to be compatible and work hand in hand. Unfortunately, this may not be the case if a basic block is partitioned based on an annotation that contains information about the structure of the basic block; that annotation may not be valid after partitioning, because partitioning may change the structure. In this case partitioning may derive incorrect threads which introduce circular dependencies. To be more specific we now present an example which exhibits this problem.

The dataflow graph representation of the example is shown in Figure 5.4. It corresponds to the following Id definition:

```
def f a b = { x = a*a;
               y = b+b;
              in (x,y) };
def g c = { x1,y1 = f c x2;
            x2,y2 = f c x1;
            z = y1+y2;
           in z };
```

The function `f` takes two arguments and returns two results. It is being called twice by the function `g`. Both calls receive `c` as their first argument, while the second argument is provided by the first result of the other call site. Even though the call sites are mutually dependent, the program does not have a circular dependence, as the dataflow graph representation illustrates. Initially every receive node is given a unique singleton inlet annotation and every send node is given a unique singleton outlet annotation. Part (b) of the figure shows the result of propagating annotations from `f` to `g`. For both call sites, squiggly edges are inserted from the first argument send to the first result receive and from the second argument send to the second result receive, reflecting the dependencies present in `f`. Because `f` does not have any internal I-fetch or I-store operations, the new inlet and outlet annotations at the call site are simply empty sets. Propagating from `g` back to `f` now results in the annotations shown in Part (c). In both call sites, the dependence set for the argument send nodes is $\{c\}$ and the demand set for the result receive nodes is $\{z\}$. For each call site, we rename the dependence and demand sets and then take their union, resulting in the inlet annotation $\{c_1, c_2\}$ for the argument receive nodes and the outlet annotation $\{z_1, z_2\}$ for the result send nodes. Note that our global partitioning algorithm cannot partition `f` in this situation, because the set $Uses$ of the nodes at the def site now contains `f`. Without that restriction, a partitioning of `f` could derive a single thread, because the dependence and demand sets of all nodes are the same. Producing a single thread for `f` is obviously a mistake, as it leads to a circular dependence between the two call sites in `g`.

The reason that we cannot partition in this situation is that partitioning a function simultaneously affects all of its call sites. In this example the call sites are mutually dependent, and therefore partitioning could lead to a circular dependence. This potential circular dependence is not captured by the interprocedural analysis, because it does not exist at the time of analysis. Partitioning of one instance of a function may change the dependencies observed at another instance. The interprocedural analysis does not take this into account. Note that partitioning only one of the call sites would be fine, *i.e.*, it would be correct to group together the nodes for only one instance of `f`, as long as it is not done in the other. Thus a natural — but fairly expensive — solution is to produce different versions of the function by cloning it. Then each occurrence of the function can be partitioned differently, and producing a single thread in one of the occurrences of the

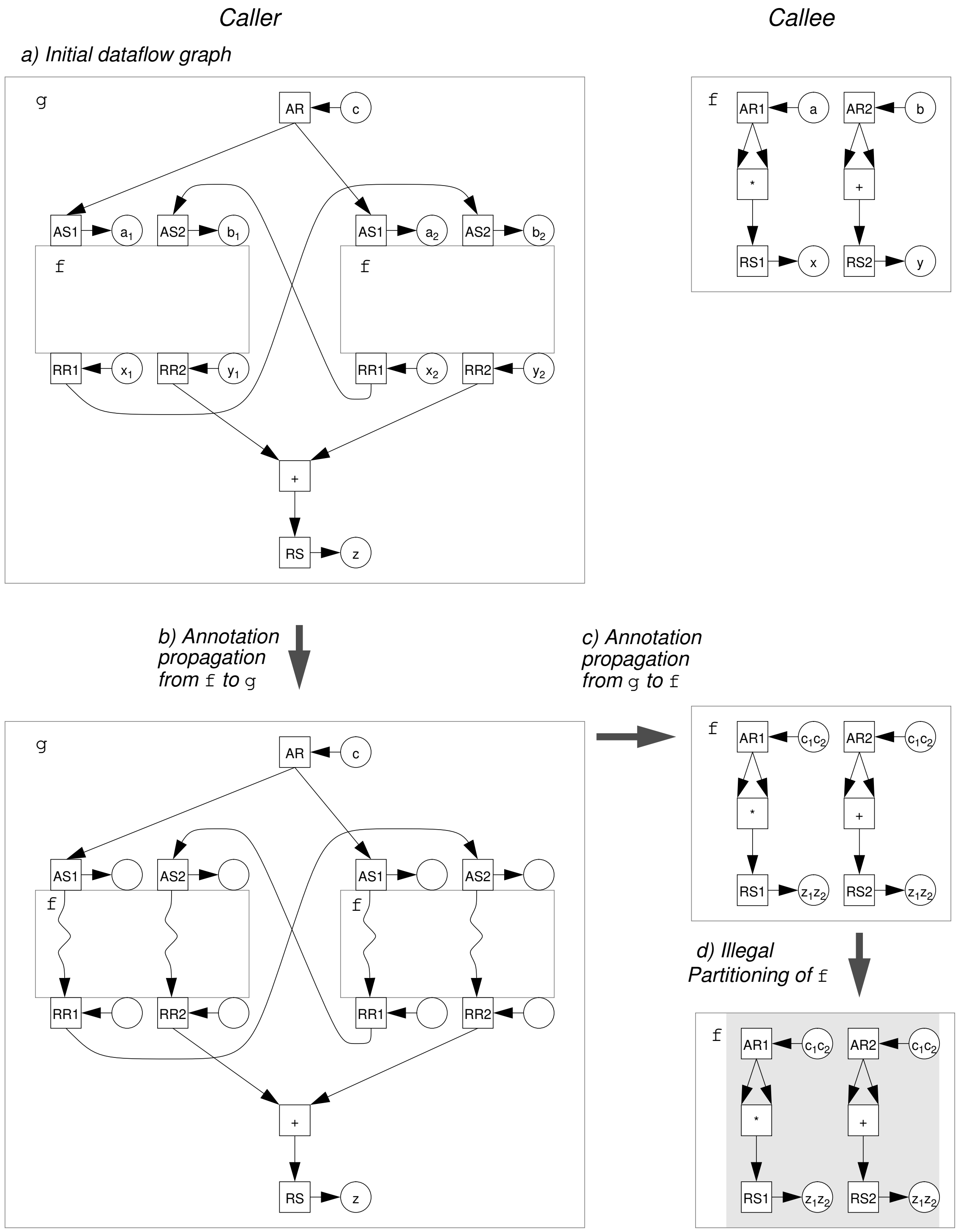


Figure 5.4: *Example showing that partitioning of a basic block can yield illegal threads if it uses annotations from global analysis which contain information about the structure of the basic block.*

function ensures that this cannot be done in the other. This approach may not only increase the code size substantially, but may also result in larger compile time, as the partitioner and code-generator has to be invoked on every cloned function. Another solution is to make the program analysis and program transformation compatible by either using a weaker program analysis or a weaker program transformation. This is essentially what our algorithm does by disallowing partitioning in the case where the annotation of a site of a basic block contains information about the basic block itself.

This subtle problem remained unobserved in previous work on thread partitioning, and is present in the partitioning algorithms presented in [Tra91, TCS92]. In his thesis, Traub developed a partitioning scheme based on function dependence graphs [Tra88]. A function dependence graph captures all of the certain and potential dependencies which may exist between the individual operations of a function. A certain dependence exists between two operations if one *always* has to execute before the other. A potential dependence exists if one operation *sometimes* has to be executed before the other. The goal is to safely approximate this function dependence graph, *i.e.*, to underestimate the certain dependencies and to overestimate the potential dependencies. Traub claims that this approximation can then be used for partitioning. His rule is that two nodes cannot be grouped into the same thread if there exists potential dependencies between the two nodes in both directions. Otherwise they can be placed into the same thread. Unfortunately, this rule breaks down if a function can have multiple occurrences.

Let us apply this rule to the above example and derive the function dependence graph for the function `f`, assuming that the only two calls to `f` are the ones inside of `g`. Let us analyze the dependencies between the two nodes representing the result x and y. There does not exist any dependence (neither certain nor potential) from x to y because for any given instance of `f` it is possible to produce the result y before the result x. Likewise, there does not exist a dependence in the other direction. Following Traub's partitioning rule, the two nodes can be placed into the same thread. As discussed before, this introduces a circular dependence between the two call sites. Nevertheless, this constitutes a correct partitioning in Traub's original framework, because there threads may suspend. With suspension, it is possible to find a correct ordering of x and y in the case where both are placed into the same thread. A possible execution order would first produce x_1, the first result of the first call site, then have the thread suspend. Now the thread in the second call site can produce x_2 and also y_2. Finally, we can resume the first thread to produce its second result, y_1. As a consequence, the thread has to be ordered such that the result

x is produced before y. Even when threads are allowed to suspend, we can trigger the partitioning problem by making the example slightly more complicated:

```
def f a b = { x = a*a;
              y = b+b;
             in (x,y) );
def h c = { x1,y1 = f c x2;
            x2,y2 = f c x1;
            x3,y3 = f y4 c;
            x4,y4 = f y3 c;
            z = y1+y2+x3+x4;
           in z };
```

The function `h` contains two additional calls to `f`. Again, the function dependence graph does not contain any dependencies between x and y and the corresponding instructions can be grouped into the same thread with Traub's partitioning rule. The dependencies among the two new call sites are just reversed, requiring that the result y be produced before the result x. Now it is impossible to find an instruction ordering among x and y which satisfies the combined requirements of both pairs of call sites.

Why don't other global program optimizations, such as strictness analysis, encounter this problem? There are two reasons. First, these analysis usually only derive properties of functions themselves and not of the context in which they can appear. Second, applying a transformation does not change the result of the analysis. Strictness analysis asks the question whether an argument is always required for producing the result of a function. Strict arguments can be evaluated before calling the function. For example, if strictness analysis determines that a function is strict in arguments x and y, then the compiler can decide to either evaluate x, or y, or both x and y before calling the function.[2] One decision does not influence the other. This is not the case with partitioning. Assume that the analysis determines that nodes u, v and r, s can be grouped together. If the compiler decides to put u and v into the same thread, it may not be able group r and s anymore! Thus, the two optimizations may be mutually exclusive. Most global program transformations, such as transforming call-by-need into call-by-value with strictness analysis, do not exhibit such

[2]Actually, if the arguments are more complex than simple scalar values, it is only possible to start the evaluation before calling the function. Stronger analysis techniques are required to show that arguments can be evaluated completely before calling the function.

mutually exclusive behavior.

5.7 Handling Recursion

Now that we understand why interprocedural partitioning of functions with multiple instances (this includes mutually recursive functions) is hard, we will discuss refinements to the partitioning algorithm which deal with these difficulties. The key is to have the global analysis only derive information which is guaranteed not to change during partitioning.

One strategy is to limit the global analysis to only propagate information from def sites to the corresponding call sites, but never in the reverse direction. This way it only derives information which is a property of the function and not of context in which it is being used.[3] This small change is sufficient to avoid the problem discussed in the previous section and also works in the presence of cycles in the call graph, as it is the case with mutually recursive functions. It is possible now to allow annotations of the basic block to use information about the basic block itself. Therefore we can drop the $Uses$ variable in Algorithm 6 and the prerequisite for partitioning a basic block which required that none of its annotations use any information about the basic block.

Unfortunately, the information which can be obtained in this way may not be very good for mutually recursive functions. The reason is that the algorithm starts with no knowledge about the program (the trivial annotations) and then iteratively refines this. This refinement may fail to improve at recursive function calls because propagation rule are conservative. Fortunately, techniques developed for dataflow analysis [ASU86, WZ85] provide a solution: abstract interpretation combined with techniques for finding fixed points [CC77]. We illustrate this concept by showing how abstract interpretation can be used by global analysis to derive squiggly edges.

5.7.1 Squiggly Edges

Each function and each arm of a conditional is represented by a basic block with n-inputs for the arguments and m-outputs for the results. The annotation propagation

[3] Similarly we could limit the global analysis to only propagate from the call sites of a basic block to its def site, but never in the reverse direction. This would only derive information about the context in which a function appears.

algorithm introduces a squiggly edge at a call site from an argument send to a result receive, if the callee contains a corresponding dependence path from the argument receive to the result send. Squiggly edges are very important for partitioning. They represent certain dependencies and are used to contradict potential dependencies, which in turn can improve the partitioning. The algorithm has to safely approximate squiggly edges, *i.e.*, it can only introduce a squiggly edge at the call site, if the certain dependence path is actually present in the callee. Our partitioning algorithm ensures that threads can only grow, *i.e.*, once produced our algorithms will never split a thread. Therefore dependence paths, once present, will never disappear, and a squiggly edge, once derived, never has to be removed. Our goal is to find the best possible approximation of the squiggly edges which we introduce at the call sites of a function.

We start the discussion with an example illustrating how to determine squiggly edges in the most difficult case, in the presence of recursion.

Example

Let us take a look at the recursive function shown in Figure 5.5. This example consists of three basic blocks, the function `f` and the two arms of the conditional `t` and `e`. Each of the basic blocks receives two arguments and returns two results. The basic block for `f` contains a call site for the conditional. The "then" side contains a recursive call to `f` and a dependence from the first argument to the first result, while the "else" side terminates the recursion. Let us now try to derive the squiggly edges at the call sites. We do this by deriving for every basic block the certain dependence paths from arguments to results. We start with a basic block which does not contain any call sites, in our example the "else" side of the conditional. Here there exists a dependence path from the first argument to the first result and from the second argument to the second result. Next we consider the "then" side, which contains a call site of `f`. Since we have not inspected `f` yet, we have to assume that it contains no certain dependence paths from arguments to results, and we cannot insert any squiggly edges at the call site. As a consequence, we can only derive a dependence from the first argument of the "then" side to its first result. Next we look at `f` which contains a call site for the conditional. To approximate the squiggly edges at the call site of the conditional correctly, we only introduce a squiggly edge if the corresponding dependence path exists in both arms. The only dependence which is common to both arms is from the first argument to the first result. Instantiating this squiggly edge into the call site of the conditional, we can derive that there exists a certain dependence path from the first argument to the first result of the function `f`. Now we can use this new information for `f` to improve our previous assumptions about the call site of `f` in the "then" side, and we can introduce a single squiggly edge from the first argument send to the first result receive. This does not improve the analysis anymore. Unfortunately this is the best we can do if we require that the dependencies be always safely approximated at every step.

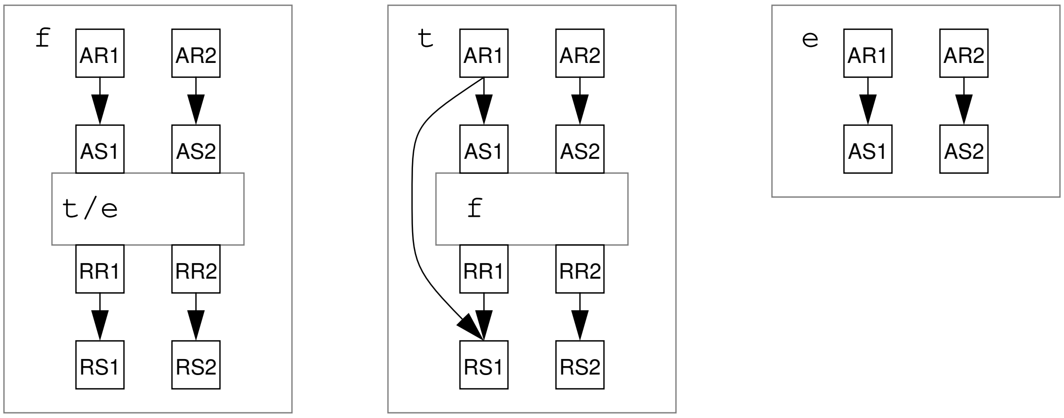


Figure 5.5: *Example for determining dependencies from arguments to results.*

It should be obvious, that in this example, there also exists a dependence path from the second argument of `f` to the second result. The reason is that — unless the computation never finishes — the function `f` will eventually call the "else" side to terminate the recursion, which then establishes this dependence. We can derive this by applying techniques developed for global dataflow analysis. Initially we over approximate the squiggly edges at call sites for mutually recursive functions by assuming that there exits a dependence from every argument to every result. Using these squiggly edges we compute for every function the dependencies from arguments to results. These determine the new set of squiggly edges at the call sites used for the next round. We repeat this process until the squiggly edges do not change anymore. We have now found the correct set off suiggly edges. Applying this process to the above example we see that the "else" side remains unchanged, there is a dependence from the first argument to the first result and another from the second argument to the second result. For the "then" we derive that there exists a dependence from every argument to every result, because we assume this to be the case for the call site of `f`. Next we refine the squiggly edges for the conditional call site in `f`. The only dependencies present in both arms of the conditional are those from the first argument to the first result and from the second argument to the second result, therefore we introduce the corresponding two squiggly edges at the call site in `f`. As consequence we will derive two dependencies for `f`, from the first argument to the first result and from the second argument to the second result. Now we use these dependencies for the call site of `f` in the "then" side and derive that the "then" also has a dependence from the first argument to the first result and from the second argument to the second result. The reader may verify that we have now found the fixed point. Subsequent iterations will not change the dependencies.

This example shows how we can derive the dependencies. We use abstract interpretation combined with techniques for finding the fixed point. A possible domain for the abstract interpretation is to use a $n \times m$ matrix for each basic block with n arguments and m results. This matrix contains a 1 in entry (i, j) if there exits a dependence from the i-th argument to the j-th result, and a 0 otherwise. We can then define an abstract (non-standard) semantics which essentially does dependence analysis to derive for every basic block this matrix of dependencies from arguments to results. For basic blocks without call sites it is straightforward to derive this matrix. Basic blocks containing one or more call sites use the callee's matrices to obtain the dependencies present in the call site, and then derive their own matrix. In the case of a conditional call site we just have to take the "logical and" of the matrices of the two arms, *i.e.*, the call site for a conditional should only reflect a dependence if it is present in both arms. For mutually recursive functions we initially set their matrices to contain a 1 everywhere, and iteratively refine them by propagating from a def site to all of its call sites until we find a fixed point.

Following this rule, we reach the fixed point in a finite number of steps since we are operating with a finite domain. Of course, it is necessary to find this fixed point before using the result of the analysis for partitioning the program. The squiggly edges at intermediate steps may be over approximated and therefore may not be valid. As a consequence we cannot partition a basic block at intermediate steps.

Algorithm

Now that we understand what the algorithm needs to do, let us formalize it. The algorithm takes a program represented as a structured dataflow graph and computes a matrix for every basic block indicating the dependencies from arguments to results. For a basic block BB it derives a matrix M_{BB} which contain a 1 at location $[i, j]$ if there exits a dependence path from the i-th argument to the j-th result. Otherwise it contains a 0. The program also maintains a variable $ToVisit$ which indicates the set of basic blocks for which the dependence matrix may have to be updated. Initially the algorithm sets the dependence matrix entries for every basic block to contain all 1's. It then starts with the basic blocks without any call sites, refines their matrices, and iteratively refines any other basic block which calls them, until a fixed point is found.

For a basic block BB we will use $SITES(BB)$ to denote the set of call sites in the basic block. For a call site S we use $CALLS(S)$ to denote the set of basic blocks which can be called at this site.

Algorithm 9 (Squiggly edges) *Given a structured dataflow graph.*

1. For every basic block *BB* in the structure dataflow graph, set the dependence matrix $M_{BB}[i, j] = 1$ for all $1 \leq i \leq n, 1 \leq j \leq m$, where n is the number of arguments and m the number of results of the basic block.

2. Set *ToVisit* to be the set of all basic blocks.

3. Select a $BB \in ToVisit$ and perform the following steps.

   (a) Remove *BB* from the *ToVisit* set.

   (b) For each call site $S \in SITES(BB)$ do the following steps.

      i. Remove all squiggly edges at the call site S.

      ii. For all i, j for which $M_{BB'}[i, j] = 1$ for all $BB' \in CALLS(S)$, add a squiggly edge to site S from the i-th argument send to the j-th result receive.

   (c) Using the dependencies present in the basic block and the squiggly edges introduced by the previous step, compute the new dependence matrix M_{new} of *BB*, such that $M_{new} = 1$ if there exits a dependence path from the i-th argument to the j-th result, otherwise 0.

(d) If the dependence matrix has changed, *i.e.*, if $M_{new} \neq M_{BB}$, set $ToVisit = ToVisit \cup \{BB' | \exists S \in SITES(BB') : BB \in CALLS(S)\}$.

(e) Set $M_{BB} = M_{new}$.

4. Repeat from Step 3 until $ToVisit = \emptyset$.

It is obvious that this algorithm will find a fixed point after a finite number of steps. Of course the algorithm should employ techniques developed for efficiently finding these fixed points [ASU86, CPJ85], for example, we want to select the basic blocks in Step 3 in the "right" order starting with basic blocks which do not contain any basic blocks, and only update those parts of matrix which change and trace only those effects on other basic block.

5.7.2 Annotations

Deriving squiggly edges is only one aspect of the global analysis algorithm. The second aspect is the new inlet and outlet annotations which are propagated into a site by Algorithm 8. We can also employ abstract interpretation for deriving inlet and outlet annotations in the case of mutually recursive functions.

The goal is to obtain a safe approximation of the inlet and outlet annotations required at the call sites. These annotation should capture the internal I-fetch and I-store nodes. In the case of mutually recursive functions, we start with the best possible annotation and then iteratively refine it until we find a fixed point. The best possible annotation assigns the nodes at every call site the empty annotation assuming that the corresponding callee contains no internal I-fetch or I-store operation. To refine these annotations we propagate from def sites to all of their call sites following the annotation propagation rule given in Algorithm 8. There is one subtle point here: we have to find the fixed point for the annotations at call sites in the congruence syndrome lattice and not in the actual annotations. For example, if the two result receive nodes at a call site get the inlet annotations $\{a\}$ and $\{a, b\}$, and in the subsequent iteration they obtain the annotations $\{c, d\}$ and $\{c, d, e\}$ their sharing syndrome has not changed as discussed earlier. Even though the annotation sets have changed in this case, their congruence syndrome has not and we do not need to propagate the change any further. We are again guaranteed to find a fixed point in a finite number of steps, since the congruence syndrome lattice is finite. Unfortunately this lattice is very large, since the congruence syndrome for n annotation sets is represented by a $2^n \times 2^n$ matrix. Just computing the congruence syndrome for the collection of inlet or outlet annotations at a call site is a very expensive operation, finding the fixed point by iteration is even more expensive. Therefore this approach is not feasible with the exception of some important special cases, such as the case where all annotations are empty or all are the same.

On the other hand, it always safe to go the opposite direction. Instead of starting with empty annotations we start with trivial unique singleton annotations and refine them. Now we may not be able to derive the best possible annotation in the case of mutually recursive functions, but it is not necessary anymore to find a fixed point, and we can take the approximation derived at any point in time for partitioning.

5.8 IO Sets Refinements

Another simple extension to the algorithm is to propagate IO set information together with the annotations. Remember, the IO set is constructed by separation constraint partitioning and contains the known dependencies from inlet names to outlet names. An inlet/outlet name pair (i, o) appears in IO if there exists a path from a node with i in its inlet annotation to a node with o in its outlet annotation. Otherwise the pair does not appear in IO. Even though such a path does not exist within a basic block, it may be that some other basic block can demonstrate that such a dependence exists, *i.e.*, it can show that (i, o) can be safely put into IO. Phrased differently, even if the edges inside a basic block are not sufficient to prove a dependence from an inlet name to some outlet name, it may be possible that some edges in some other basic block can provide such a proof. It seems natural to make use of this additional information. All we need to do is to extend the annotation propagation to also propagate the current knowledge about dependencies from inlet names to outlet names.

This can happen in two ways. When propagating new annotations, it is possible to also propagate the known IO dependencies among the new inlet and outlet names. Separation constraint partitioning then improves these with the dependencies present in the new basic block. Thus, it is also desirable to propagate IO information back. We have to be careful when propagating from multiple sites; a known dependence can only be propagated back if is present in all of the source sites. Also note that sometimes it is necessary to rename inlet and outlet names to avoid sharing with other sites. We have to keep track of this renaming when propagating IO back and forth. Formalizing the algorithm for propagating IO sets along with annotations is fairly complicated, as it is necessary to keep track of the renamed inlet and outlet annotations and their original names. Instead, we will limit ourselves to presenting an example to illustrate how it works and that it can make a difference in the quality of the partitioning.

Figure 5.6 shows an example which requires IO sets to be propagated across the interfaces to obtain a better partitioning. The example shows the three basic blocks of a conditional, the caller, then side, and else side, connected by multiple-def-single-call interface. Initially, all send and receive nodes are given the trivial annotation. Partitioning the caller results in four threads, while partitioning of the two arms of the conditional results in two threads each. In this example, propagating information from the two arms of the conditional into the caller does not lead to a better annotation, and subsequent partitioning will result in no changes. The reason is that it is impossible to introduce squiggly edges, since no path from a receive node to a send node is present in both arms.

Propagating from the caller to the two arms of the conditional leads to the inlet annotation $\{a\}$, $\{b\}$ and the outlet annotation $\{u, v\}$, $\{v\}$ as shown in Part (c). This new annotation alone is insufficient to improve the partitioning of the conditional arms. There is no path from the node containing b in its inlet annotation to a node with u in its outlet annotation in the "then" arm. Thus $(b, u) \notin IO$ and separation constraint partitioning has to assume that there may exist a potential indirect dependence from the first result send back to the second argument receive. For the "else" arm, the situation is similar. Here $(a, u) \notin IO$ and the partitioning has to assume that there may exist a potential indirect dependence from the first result send back to the first argument receive. Notice, however, that in the caller we can prove both $(b, u) \in IO$ and $(a, u) \in IO$. The corresponding paths are present in the caller, they just happen to go around the conditional! Therefore, the potential indirect dependencies — which the partitioning algorithms thinks may exist in the arms of the conditional — cannot exist. If we just propagate this additional knowledge $(b, u), (a, u) \in IO$ from the caller to the two arms, separation constraint partitioning can use it to derive a single thread for both arms as shown in Part (d).

a) Initial Annotation
Every receive is annotated with a unique singleton inlet name, and every send with a unique singleton outlet name.

b) Initial Partitioning
Partitioning the caller results in four threads, while each of the callees gets two threads.

c) Reannotation of Callees
Annotation propagation from the caller to the callee results in the new inlet and outlet annotations, in addition the set IO = {(a,u),(b,u)} is passed in.

d) Partitioning of Callees
With the new annotation separation constraint partitioning can obtain one thread for each callee. Note that is only possible because of the IO set.

e) Reannotation of Caller
Annotation propagation from the callee to the caller introduces four squiggly edges and empty annotations.

f) Partitioning of Caller
Partitioning now obtains two threads. Further reannotation and partitioning does not improve this.

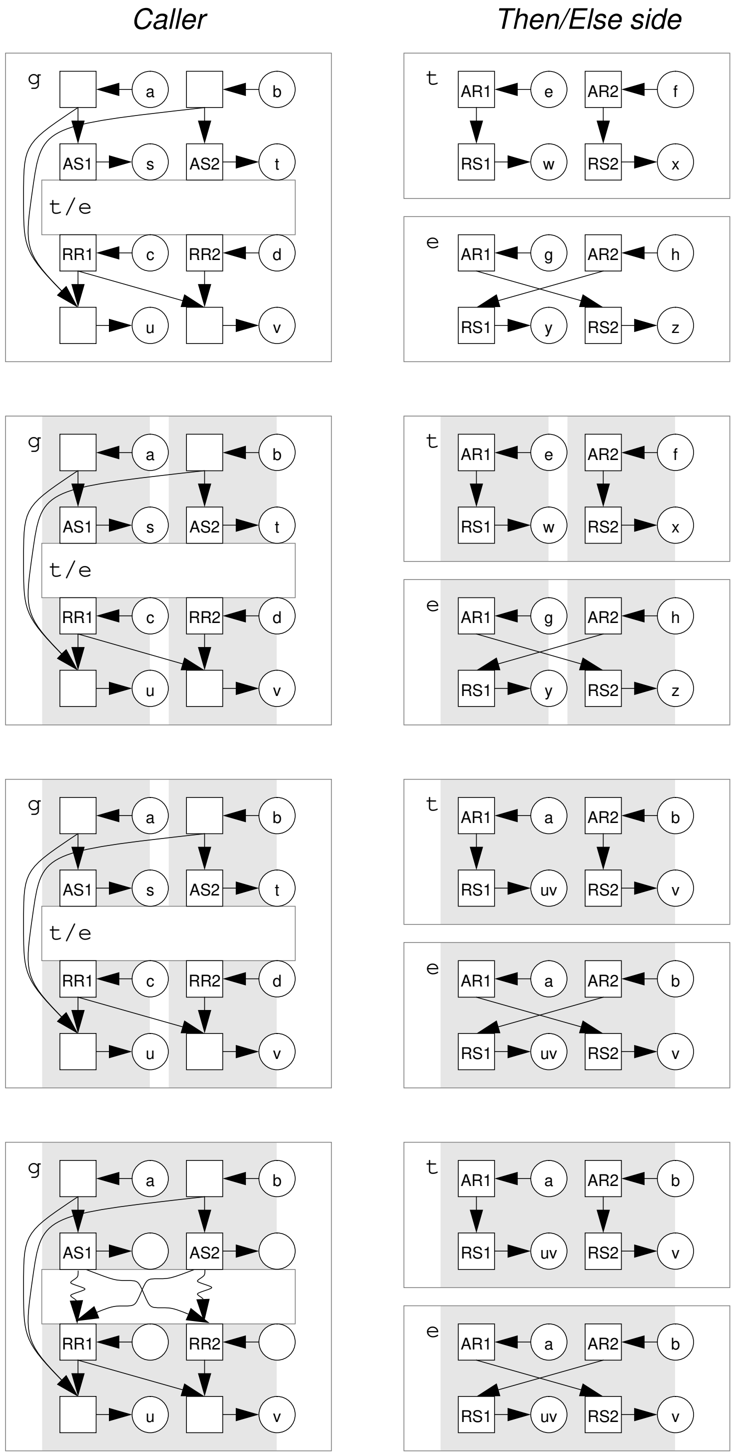


Figure 5.6: *Example of interprocedural partitioning propagating IO sets with new annotations.*

Propagating in the opposite direction, from the callees to the caller, introduces four squiggly edges and assigns empty inlet and outlet annotation at the call site, reflecting the fact that both arms consist of a single thread and that neither arm contains any call sites, I-fetch, or I-store nodes. The new annotation results in two threads for the caller, again, the best we can obtain for this example. This result was only possible because we propagated IO set information along with the annotations.

5.9 Summary

This finishes our presentation of the algorithms for partitioning non-strict functions into sequential threads. We first presented several basic block partitioning algorithms — dataflow, dependence set, demand set, iterated, and separation constraint partitioning — each one more sophisticated than its predecessor but also more complex. We then discussed how extend basic block partitioning with interprocedural analysis. The basic block partitioning algorithms are limited in their ability to derive threads, because without global analysis they must assume that every result of a basic block may feed back in to any argument, unless contradicted by certain dependencies. Global analysis can determine that some of these potential dependencies cannot arise and thereby improve the partitioning. The next chapter presents measurements showing the effectiveness of the partitioning algorithms in reducing the control overhead.

Chapter 6

Experimental Results

This chapter evaluates our compilation paradigm and presents data on the code quality produced. Previous to our work, execution of Id programs was limited to dataflow architectures or dataflow graph interpreters. By compiling via TAM to commodity microprocessors, we have achieved an performance improvement of more than two orders of magnitude over dataflow graph interpreters, making this Id implementation competitive with specialized machines supporting dynamic instruction scheduling in hardware. To better identify the specific gains, we can constrain how programs are partitioned. (In fact, we can generate code that closely models the spectrum of dataflow architectures.) It can be seen that the partitioning described in the previous chapters substantially reduces the control overhead. Most of the gain comes from sophisticated basic block partitioning. Interprocedural analysis reduces communication for argument passing, improves scheduling behavior, and further reduces control, although the gain in control is not as large as that obtained by basic block partitioning. That the gain is not so large is not due to a lack of the effectiveness of our analysis techniques; we verify effectiveness by compiling programs strictly. Additional improvements to the interprocedural partitioning algorithm would not further reduce the control overhead significantly.

This chapter consists of five parts. Section 6.1 gives an overview of our experimental framework, and discusses the compiler, backends, benchmark programs, and machines used for the measurements. Section 6.2 presents data showing the effectiveness of partitioning in reducing the control overhead, while Section 6.3 shows the effectiveness of TAM in handling the dynamic scheduling which remains. Section 6.4 combines the observations of the previous sections and looks at the overall efficiency of the programs. Finally, Section 6.5 summarizes our experimental findings.

6.1 Experimental Framework

This section describes our experimental framework, giving an overview of the Id compiler, the benchmark programs, as well as the machines used for the data collection.

6.1.1 Id Compiler and TAM backend

Figure 6.1 gives an overview of the structure of the Id to TAM compiler. Id programs are first translated into program graphs by an Id front-end developed at MIT [Tra86]. Program graphs are a hierarchical graphical intermediate form which allow a representation of the various basic operations as well as conditionals, function definition, and application. This facilitates powerful high-level optimizations, such as motion of arbitrarily large program constructs across loops or conditionals. The meaning of program graphs is given in terms of a dataflow firing rule.

Program graphs are very similar to the structured dataflow graphs used by our partitioning algorithm. The only difference is that some program graph nodes denote complex operations, such as make-functional-tuple, which have to be mapped to a collection of simpler primitives. This mapping is a local transformation and can be achieved by following expansion rules for the individual program graph nodes. The structured dataflow graph is then partitioned into threads, using the algorithms presented in the previous two chapters. As discussed below, we have implemented several versions which perform different degrees of partitioning. This enables us to study the effectiveness of the partitioning algorithms. After partitioning, the compilation steps described in Section 3.3 — instruction scheduling and linearization, lifetime analysis, register and frame-slot allocation, redundant arc elimination, and fork insertion — are required for producing TAM code.

TAM code is currently represented in TL0 (which stands for Thread Language Zero). TL0 is a concrete implementation of the abstract ideas embodied in TAM. TL0 can be translated in a second straightforward translation step to actual machines, and has been

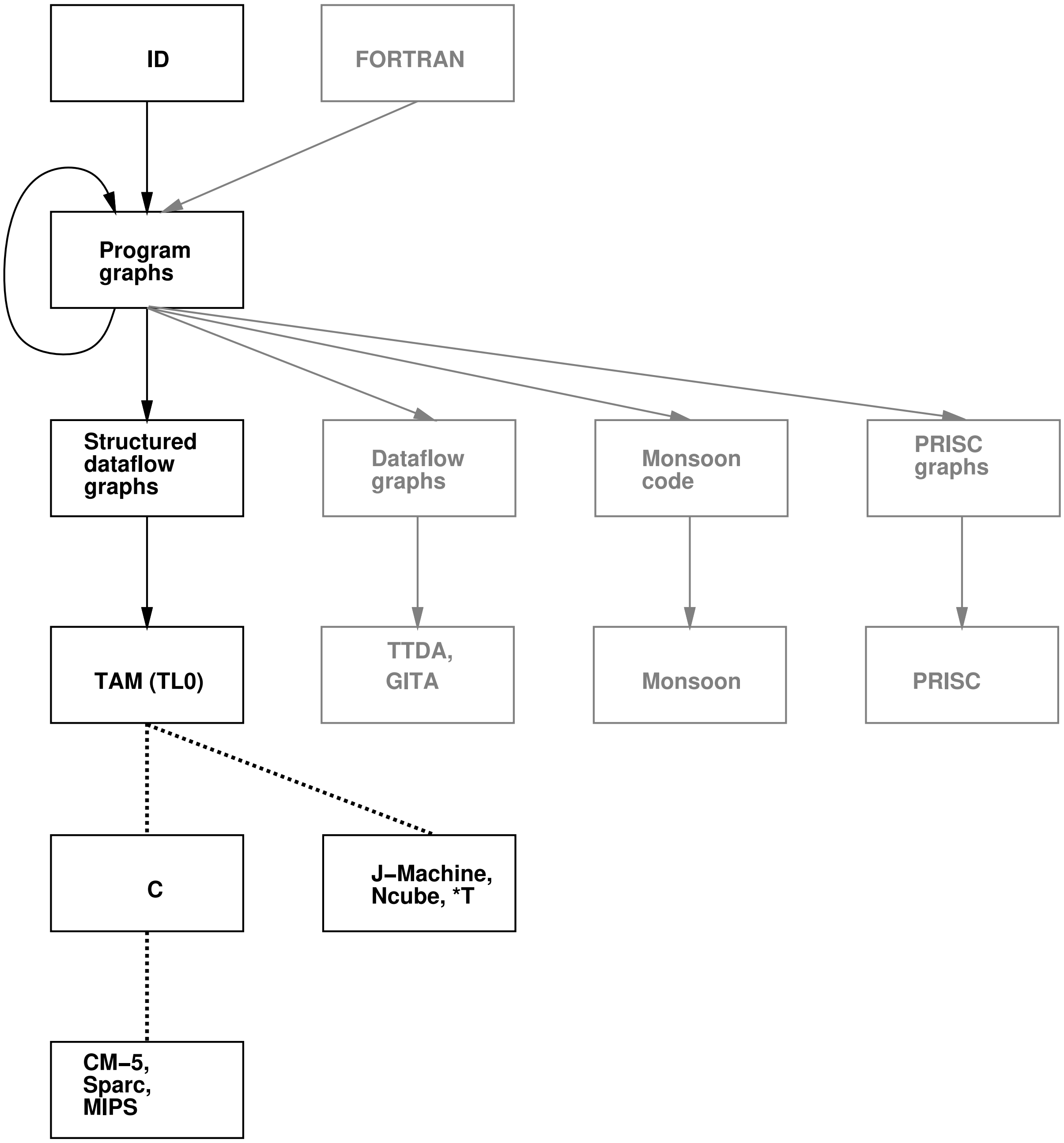


Figure 6.1: *Overview of Id to TAM compiler. Shown in gray are other compilation approaches.*

implemented on sequential machines, shared-memory machines, as well as on message passing machines. Our translation path uses C as a portable "intermediate form" and is producing code for the CM-5, as well as for various standard sequential machines [Gol94]. It is this implementation that we used for statistics collection and measurements. Other backends translate TAM directly into machine code, such as the backend developed for the J-Machine [Spe92].

Figure 6.1 also shows in gray other compilation approaches that chose program graphs as their intermediate form. The original MIT compiler translates program graphs into dataflow graphs which can then be executed on the Tagged Token Dataflow Architecture (TTDA) [ACI$^+$83] or be interpreted by a graph interpreter (GITA). Other backends translate program graphs for the Monsoon dataflow machine [PC90] and for the P-RISC machine [NA89]. Some other approaches compile FORTRAN programs into representations similar to program graphs [BP89, FOW87].

6.1.2 Machine Characteristics

Our main experimental platforms are the Thinking Machines CM-5 and sequential machines, especially Sparc-based Sun workstations.

The CM-5 is a massively parallel MIMD computer based on the Sparc processor. Each node consists of a 33 Mhz Sparc RISC processor chip-set (including FPU, MMU and 64 KByte direct-mapped write-through cache), 32 MByte of local DRAM memory and a network interface. The nodes are interconnected in two identical disjoint incomplete fat trees, and a broadcast/scan/prefix control network. The CM-5 also contains vector units, which are not used in the code generation from TAM.

For most sequential measurements a SparcStation 10 was used. This workstation consists of a 50 MHz superscalar Sparc processor with first-level on-chip instruction and data cache. The instruction cache is a 20 KByte five-way set associative cache, while the data cache is a 16 KByte 4-way set associative cache. In addition, the system also has a 1 MB off-chip second level cache, and 128 MByte of DRAM main memory.

6.1.3 Benchmark Suite

We use six benchmark programs, as shown in Table 6.1, ranging up to 1,100 source code lines. Most of these programs were developed by other researchers in the context of other platforms, especially the GITA dataflow graph interpreter and Monsoon dataflow machines. It should be noted that the code was taken as is, compiled for TAM and executed on standard workstations or the CM-5 without any modifications. Therefore the fine-grain nature of the programs was retained.

Program	Code Size (Lines)	Short Description	Input Size
Quicksort	55	Quick sort on lists	10,000
Gamteb	649	Monte Carlo neutron transport	40,000
Paraffins	185	Enumerate isomers of paraffins	19
Simple	1105	Hydrodynamics and heat conduction	1 1 100
Speech	172	Speech processing	10240 30
MMT	118	Matrix multiply test	500

Table 6.1: *Benchmark programs and their inputs.*

Quicksort is a simple quick-sort using accumulation lists. The input is a list of 10,000 random numbers. *Gamteb* is a Monte Carlo neutron transport code [BCS$^+$89]. It is highly recursive with many conditionals. The work associated with a particle is unpredictable, since particles may be absorbed or scattered due to collisions with various materials, or may split into multiple particles. Splitting is handled by recursive calls to the trace particle routine. Particles are independent, but statistics from all particle traces are combined into a set of histograms represented as M-structures. The input consists of 40,000 initial particles. *Paraffins* [AHN88] enumerates the distinct isomers of paraffins up to size 19. *Simple* [AE88, CHR78] is a hydrodynamics and heat conduction code widely used as an application benchmark, rewritten in Id. It integrates the solution to several PDEs forward in time over a collection of roughly 25 large rectangular grids. Each iteration consists of several distinct phases that address various aspects of the hydrodynamics and heat conduction. Simple is irregular, due partly to the relationship between the phases, which traverse the data structures in different ways. In addition, table look-ups are performed inside of the grid-point calculation and boundaries are handled specially. The problem size is a 100x100 grid. *Speech* determines cepstral coefficients for speech processing. *MMT* is a simple matrix operation test using 4x4 blocks; two double precision matrices of size 500x500 are created, multiplied, and then the result matrix is reduced to a single number.

First, we were interested in finding out how much non-strictness is used in these programs. As the examples presented in Chapter 2 showed, there are different forms of non-strictness. Function calls can be non-strict, which allow results of functions to be fed back in as arguments. This makes the order of evaluation inside the function depend on the context in which it appears. Conditionals can also be non-strict and a conditional can affect the order in which expressions outside of the conditional are evaluated. Finally, data structures may be non-strict. Elements of a data structure may be recursively defined in terms of other elements.

To check how much non-strictness our programs require, we extended the compiler to treat function calls and conditionals in a strict fashion. Under this mode, a function is only invoked after all of its arguments have been evaluated, and results are only returned after they have been completely computed. Much to our surprise, all of our benchmark programs, and most of the other Id programs we came across, still run correctly. This is an indication that programmers are not making much use of the additional expressiveness provided by non-strictness of functions or conditionals. It should be noted, though, that several of the benchmark programs make use of data structure non-strictness. Being able to define array elements in term of others is certainly an important feature, as it can result in more efficient programs. It is convenient that many programs run under strict compilation, because we can use it as a kind of upper bound on what is achievable for interprocedural partitioning.

6.2 Effectiveness of Partitioning

In this section we present empirical data on the effectiveness of partitioning in reducing the control overhead. To measure the effectiveness of partitioning we compare the following four different partitioning schemes.

Dataflow partitioning (DF): The simplest form we consider is dataflow partitioning, which puts unary nodes into the thread of their predecessor. Nodes requiring synchronization or receiving the responses of split-phase operations start a new thread.

This partitioning scheme reflects the limited thread capabilities supported by modern dataflow machines. This partitioning scheme has previously been studied in [SCvE91].

Iterated partitioning (IT): Far more powerful is iterated partitioning, which iteratively applies dependence and demand set partitioning to each basic block. All inlet and outlet nodes are annotated with the trivial annotation. This partitioning scheme partitions each basic block in separation and does not do any analysis across basic blocks. The power of iterated partitioning is similar to the algorithm in [HDGS93] and dependence set partitioning with merging presented in [SCvE91].

Interprocedural partitioning (IN): Interprocedural partitioning applies the global analysis techniques discussed in the previous chapter. These refine the inlet and outlet annotations and introduce squiggly edges at call sites. The interprocedural algorithm first tries to group nodes at def and call site boundaries using separation constraint partitioning, before merging any interior nodes.

Strict partitioning (ST): Finally, strict partitioning represents an upper bound on what is achievable. Here, all functions and conditionals are compiled strict, *i.e.*, all boundary nodes of def and call sites are grouped into a single partition. Then iterated partitioning is used for grouping the interior nodes. Obviously, no interprocedural partitioning algorithm could do better than strict partitioning. It should be noted that this scheme does not generally represent a correct partitioning algorithm, as it fails to produce correct threads for programs requiring non-strict functions or conditionals.

Although implemented, we do not study dependence set partitioning nor demand set partitioning alone. When every basic block is partitioned in separation, we may just as well immediately apply iterated partitioning. The reason is that iterated partitioning is not much slower than dependence or demand set partitioning because only a small number of iterations are required. We also do not study separation constraint partitioning in isolation. Some of the basic blocks contain around 600 nodes, and separation constraint partitioning having a complexity of $\mathcal{O}(n^3)$ just runs to slow, and without interprocedural analysis does not really result in a better partitioning. On the other hand, separation constraint partitioning is being used by interprocedural partitioning algorithm to first group the boundary nodes at the def and call sites. Here we exploit the capability of separation constraint partitioning to determine for any pair of nodes whether they can be merged or not.

Dataflow partitioning and strict partitioning represent the two extremes of the spectrum: dataflow partitioning does very little partitioning, reflecting the limited thread capabilities supported by dataflow machines, while strict partitioning represents the best any interprocedural partitioning algorithm could hope to do. One can view this as providing partitioning with an "oracle" which indicates that it is safe to group the boundary nodes. For programs which require non-strictness, it leads to an incorrect partitioning. Iterated and interprocedural partitioning represent the two real partitioning schemes. Iterated partitions every basic block in isolation, while interprocedural performs the analysis across basic blocks.

There are several ways to evaluate the effectiveness of these partitioning schemes. The simplest is to compare the number of dynamically executed threads or the average thread size. Of course, since threads can be synchronizing, this may not give a clear picture about the total number of control operations. A more detailed study measures the dynamic distribution of TAM instructions and compares the relative occurrence of control, communication, heap access, and ALU operations. By looking at the instruction distributions we can tell how good a partitioning scheme is in reducing the control overhead and see how effective TAM is in implementing the remaining scheduling. Of course, it is necessary to develop an understanding of the machine complexity of the individual instructions. Finally, we can directly measure the running time of each benchmark under the partitioning schemes.

At first glance timing measurements would seem to be the bottom line. But timings alone are not sufficient, because they are very dependent on the actual hardware and implementation of the run-time system. Timings also make it hard to understand the causes for performance differences. TAM instruction distributions help in understanding these differences, but they still depend on details of the abstract machine specification. Dynamic thread counts, while being the crudest form of performance evaluations should be valid across a larger variety of threaded execution models.

In the next sections, we present all three sets of measurements. Starting with dynamic thread counts, we gradually refine our observations, present abstract machine instruction distributions, and finally show the actual run time results. The numbers were collected both on a SparcStation 10 and CM-5.

6.2.1 Thread Characteristics

Program	Partition.	Threads	Thread Instr.	IPT	Inlets	Inlet Instr.	IPI
QUICKSORT	DF	710762	1390427	2.0	115221	455719	4.0
	IT	345856	922268	2.7	115221	420699	3.7
	IN	257617	686629	2.7	112218	399681	3.6
	ST	231616	601544	2.6	93213	315658	3.4
GAMTEB	DF	8176392	17538350	2.1	1582900	9799799	6.2
	IT	3433592	11127349	3.2	1582900	7902417	5.0
	IN	2533148	9031862	3.6	1448550	6326447	4.4
	ST	1542497	6481295	4.2	1013791	3621609	3.6
PARAFFINS	DF	1500412	3688261	2.5	246902	752992	3.0
	IT	690539	2070406	3.0	246902	747856	3.0
	IN	591334	1871503	3.2	246732	746822	3.0
	ST	581569	1800064	3.1	245144	740951	3.0
SIMPLE	DF	4684461	12417200	2.7	1565609	6830577	4.4
	IT	731427	5792943	7.9	1565609	5173584	3.3
	IN	574693	5176756	9.0	1372036	4455107	3.2
	ST	517980	4959568	9.6	1308679	4183034	3.2
SPEECH	DF	4954265	14068380	2.8	1602581	4935663	3.1
	IT	1296170	8297689	6.4	1602581	4879195	3.0
	IN	1281817	8242855	6.4	1591108	4825433	3.0
	ST	1264065	8174672	6.5	1576198	4755616	3.0
MMT	DF	443091	1821732	4.1	111260	335175	3.0
	IT	67523	1195103	17.7	111260	334391	3.0
	IN	67475	1194857	17.7	111151	333930	3.0
	ST	67292	1194381	17.7	110993	333226	3.0

Table 6.2: *Dynamic thread count, TL0 thread instruction count, average number of thread instructions per thread (IPT), inlet count, TL0 inlet instruction count, and average number of inlet instructions per inlet (IPI) for the benchmark programs under various partitioning schemes.*

Table 6.2 shows the dynamic thread count, TL0 thread instruction counts, and average number of TL0 instructions per thread for the set of benchmark programs under the four partitioning schemes. It also contains the equivalent set of numbers for inlets. The thread and inlet counts are also represented graphically in Figure 6.2, where they are normalized with respect to dataflow partitioning.

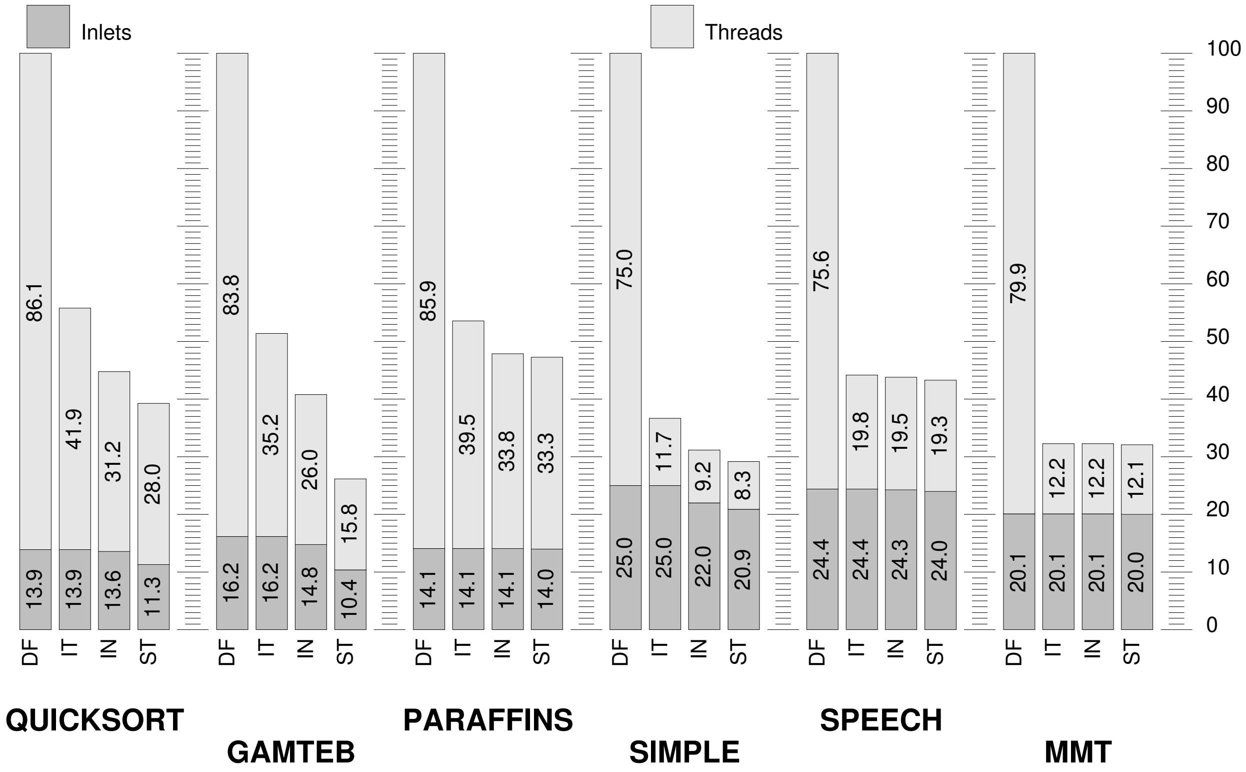


Figure 6.2: *Dynamic thread and inlet distributions for the benchmark programs under various partitioning schemes (normalized to dataflow partitioning).*

As the graphs show, the dynamic thread count drops dramatically with more sophisticated partitioning. Under iterated partitioning, the programs Quicksort, Gamteb, and Paraffins execute less than half of the number of threads than under dataflow partitioning. For the more coarser-grained programs, Simple, Speech, and MMT, the gain is even larger: under iterated partitioning only 1/7 as many threads are executed.

The next step, going to interprocedural partitioning, results in an additional, but smaller gain. Actually, for two of the programs, Speech and MMT, there is close to no improvement at all. The reason is that for these two programs most threads are executed in an inner loop, and the number of threads at function call boundaries which interprocedural partitioning reduces is not significant.[1] Quicksort and Gamteb, two programs with many function

[1] Id offers two kind of loops, regular 1-bounded loops and k-bounded loops. Id semantics treat 1-bounded loops as being strict. Therefore our Id compiler compiles all 1-bounded loops strict, which results in fairly efficient code for both Speech and MMT. If a parallel execution of multiple loop iterations is desired, k-bounded loops or recursive functions are used.

calls and conditionals, show the largest improvements. Here the number of threads are reduced by another 25%.

Why does interprocedural partitioning not reduce the number of threads by more? Are the analysis techniques presented in the previous chapter not powerful enough or do they break down in the presence of data structures? To answer this question we also compiled all functions and conditionals in a strict fashion. Surprisingly, this does not reduce the number of threads by much (with the exception of Gamteb), which is an indication that there is little room for improvement to the interprocedural partitioning. In addition, interprocedural partitioning and strict partitioning also decrease the number of inlets slightly, as multiple arguments or results can be combined into a single larger message which is handled by a single inlet. The same optimization cannot be applied to inlets due to I-fetches, as different I-structure elements may reside on different processors. Therefore, the number of inlets only decreases slightly. The largest gain is for Gamteb, where the number of inlets is reduced by about 10% for interprocedural partitioning and an additional 30% for strict partitioning. It should be noted that under the best partitioning, inlets outnumber the threads by a factor of two for some of the programs. The fact that inlet outnumber threads is another indication that focusing on reducing the number of threads may not yield much additional benefits. Our interprocedural partitioning is nearly as good as can be achieved through thread partitioning.

The picture is very similar when looking at average thread sizes given in Table 6.2. Iterated partitioning alone already obtains most of the gain. Adding interprocedural analysis increases the thread sizes only slightly. Actually, the increase in thread size is less than the decrease in number of threads, because the total number of instructions also decreases as less control operations are required. As expected, with dataflow partitioning, the average thread length is fairly small, between 2.0 instructions per thread for Quicksort and 4.1 instructions per thread for the blocked matrix multiply. Iterated partitioning increases these sizes to 2.7 instructions per thread for Quicksort and 17.7 instructions per thread for MMT. The thread sizes highlight the different nature of the programs. Quicksort is a very fine-grained program with lots of recursive function calls and conditionals, while the blocked matrix multiply is a much coarser-grained program. Although the number of instructions per thread seems to be very small, the reader should not forget that under TAM any control flow operation or split-phase access forms a thread boundary. TAM threads essentially correspond to basic blocks in conventional languages.[2] In addition, most TL0 instructions are expanded into several RISC instructions by the final translation step, so the number of Sparc instructions per thread is substantially larger. The limiting factors to increasing the thread sizes does not seem to be the interprocedural analysis, but rather long latency operations and conventional control flow.

6.2.2 TL0 Instruction Distributions

Now let us study the performance gain in more detail. Figure 6.3 shows the dynamic TL0 instruction distribution for the benchmark programs under the four partitioning schemes, each normalized to dataflow partitioning. Instructions are classified into one of four categories: ALU operations, heap accesses, communication, and control operations. The programs toward the left of the figure exhibit very fine-grain parallelism and are very control intensive. The moderate blocking and regular structure of MMT shows a significant contrast.

Under dataflow partitioning the majority of TL0 instructions are control instructions, reflecting the fine-grained nature of dataflow programs. The only exception is blocked matrix multiply, where ALU operations constitute 46% of all instructions. As expected, iterated partitioning substantially reduces the control operations. All of the other instruction categories remain equal. The reduction of control operations roughly corresponds to the decrease of threads observed earlier. For most programs, iterated partitioning reduces the number of control operations by more than a factor of 2; for Simple and MMT the reduction is much larger. Interprocedural partitioning further reduces the control operations over iterated partitioning for the more finely grained programs, while for the coarse grained programs the improvement is insignificant. Under the best partitioning scheme, the control instructions have reached a tolerable level, constituting between 10% and 30% of all instructions.[3] As already discussed earlier, interprocedural partitioning also slightly decreases the number of instructions related to communication, as the grouping of arguments and results reduces the number of messages.

[2] Note, that here the term basic block is used to describe a sequence of instructions without any control transfer [ASU86], which is different from our previous use of basic block for structured dataflow graphs.

[3] This is similar to the fraction of control instructions observed for imperative programs on commodity processors. Patterson and Hennessy report that, depending on the architecture, between 13% and 25% are control instructions [HP90].

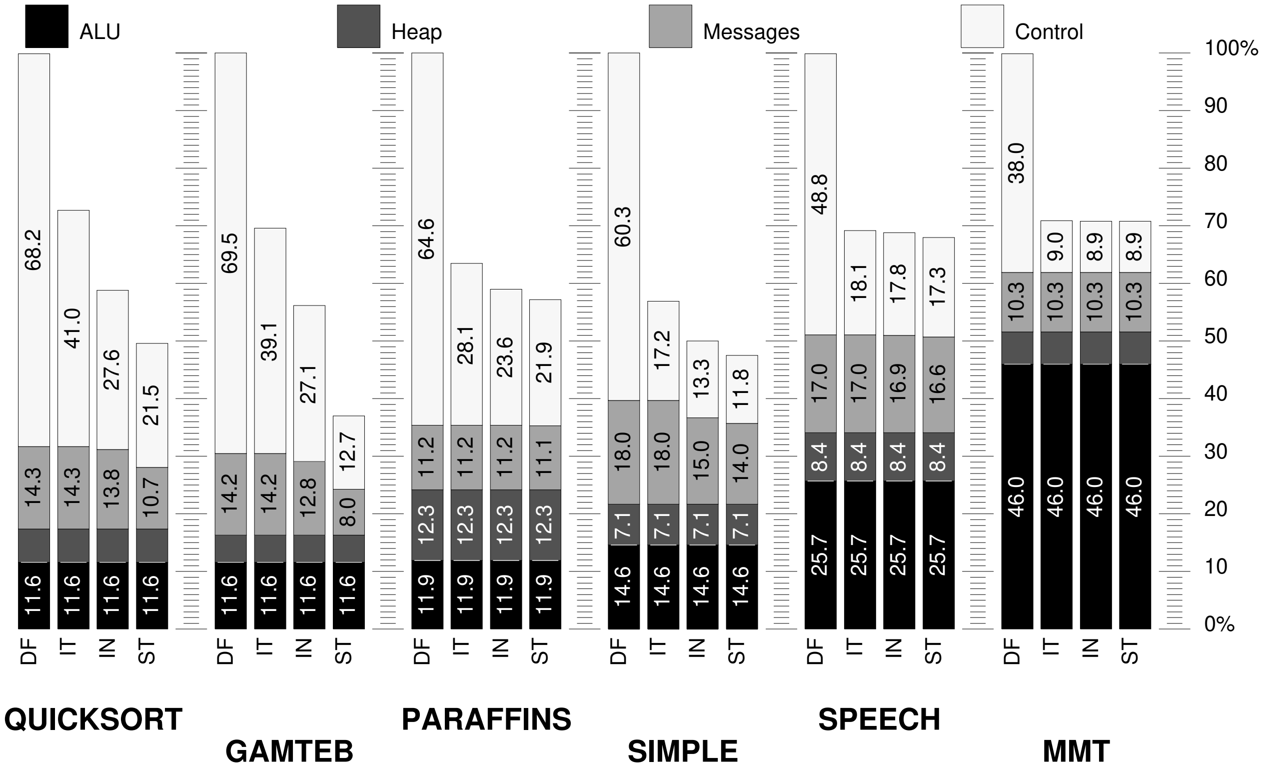


Figure 6.3: *Dynamic TL0 instruction distribution for the benchmark programs under various partitioning schemes (normalized to dataflow partitioning).* ALU *includes integer and floating-point arithmetic,* messages *includes instructions executed to handle messages,* heap *includes global I-structure and M-structure accesses, and* control *represents all control-flow instructions including moves to initialize synchronization counters.*

Obviously, this distribution of TL0 instruction does not correspond to the actual breakdown of execution time because the individual instructions have varying execution costs associated with them. For example, most ALU instructions are much cheaper to implement than most control and heap related instructions. Likewise, on most parallel machines sending and receiving of messages is extremely expensive, and communication related TL0 instructions account for a large fraction of the execution time. Before discussing run-time measurements on sequential and parallel machines, we present further data showing the effectiveness of partitioning.

6.2.3 Grouping of boundary nodes

One distinguishing feature about partitioning across basic block interfaces is that it may group nodes at basic block boundaries. For example, multiple send nodes residing in the same thread can be grouped into a single send node if the corresponding receive nodes also reside in a single thread. A similar optimization also occurs at boundaries of conditionals. Here multiple switch operations can be replaced by a single switch node, or multiple merge nodes can be replaced by a single merge node.

Recall that dataflow partitioning and iterated partitioning alone do not work across basic block boundaries, and these optimizations are not applied. Interprocedural partitioning attempts these optimizations, but has to be careful not to group nodes when non-strictness is required, as otherwise the program may deadlock. Strict partitioning goes to the extreme, it always groups all boundary nodes, which results in an incorrect partitioning when non-strictness is used. Focusing on this grouping at basic block boundaries and comparing it for the various partitioning schemes gives us a good measure of success for the interprocedural partitioning, and what further improvement is possible.

Table 6.3 contains the dynamic counts of function calls, send operations (for arguments and results), and switch operations for the benchmark programs under the various partitioning schemes. The table also contains two columns with normalized values, the average number of send operations per function call and the average number of switch instructions per executed conditional. These two columns are displayed graphically in Figures 6.4 and 6.5.

Let us first take a look at the average number of send operations per function call. This number should be at least two; one for sending in the arguments and another for returning the results. As the table and graph show, this is precisely the case under strict partitioning. In our current implementation of the compiler every function takes its n arguments and returns a single result (which may be a tuple). In addition, another pseudo argument, the trigger, is required to start computation which does not depend on any of the arguments, and a second result, the signal, is produced to indicate that all computation within the function has terminated. Thus, without any grouping every function call requires precisely $3 + n$ sends: one send for the trigger, n sends for the arguments, one send for the result, and a final send for the signal.

Program	Partit.	Calls	Sends	Sends/Call	Switches	Switches/Cond.
QUICKSORT	DF	6004	28013	4.7	178486	2.1
	IT	6004	28013	4.7	178486	2.1
	IN	6004	25008	4.2	93244	1.1
	ST	6004	12007	2.0	86243	1.0
GAMTEB	DF	77389	646499	8.4	1848248	3.9
	IT	77389	646499	8.4	1848248	3.9
	IN	77389	512147	6.6	953534	2.0
	ST	77389	154777	2.0	478329	1.0
PARAFFINS	DF	855	2614	3.1	298630	1.5
	IT	855	2614	3.1	298630	1.5
	IN	855	2442	2.9	199156	1.0
	ST	855	1709	2.0	194460	1.0
SIMPLE	DF	37240	294171	7.9	181671	1.8
	IT	37240	294171	7.9	181671	1.8
	IN	37240	100596	2.7	118096	1.2
	ST	37240	74479	2.0	99405	1.0
SPEECH	DF	3618	30002	8.3	439822	1.0
	IT	3618	30002	8.3	439822	1.0
	IN	3618	18527	5.1	431098	1.0
	ST	3618	7235	2.0	419836	1.0
MMT	DF	37	305	8.2	25145	1.0
	IT	37	305	8.2	25145	1.0
	IN	37	194	5.2	25115	1.0
	ST	37	73	2.0	25040	1.0

Table 6.3: *Dynamic number of function calls, send operations for arguments and results, the ratio of sends per call, number of switches, and the ratio of switches per conditional for the benchmark programs under various partitioning schemes.*

Using this formula we can determine the average number of arguments for each benchmark. For example, Quicksort has on average 4.7 sends per function call, thus the average function call takes 1.7 arguments. Likewise, Gamteb, Simple, Speech, and MMT all have around 8 sends per function call, which indicates that for these programs the average function takes around 5 arguments. The reason for this high number is that compiler optimizations, such as lambda lifting, introduce additional arguments which are not present in the source code. Detupling and multiple return value optimizations, which currently are not applied, would further increase this number. As expected, iterated partitioning has the same number of sends per call as dataflow partitioning. Both schemes do not work

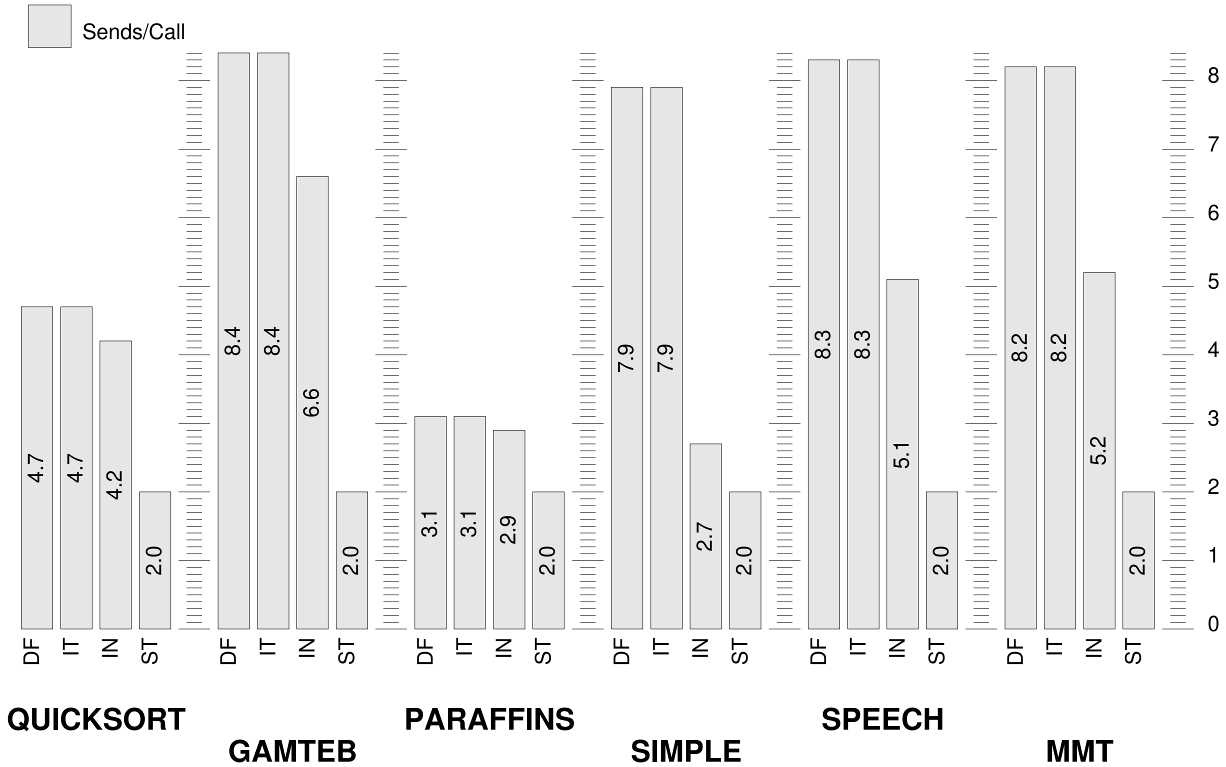


Figure 6.4: *Average number of argument and result send operations per function call.*

across basic blocks and do not attempt to merge any send or switch nodes. The average number of sends per call varies from 3.1 for paraffins to 8.4 for Gamteb. Interprocedural partitioning reduces the number of sends per function call to 2.9 for Paraffins and 6.6 for Gamteb. The largest gain is for Simple, where interprocedural partitioning reduces this number from 7.9 to 2.7, which is close to the required minimum of 2 sends per function call obtained by strict partitioning.

A similar picture exists for conditionals although here the average conditional takes far fewer arguments. Under dataflow partitioning each argument to a conditional requires a switch, while each result requires a merge. For conditionals which contain computation that does not depend on any of the arguments, the compiler adds another switch for the trigger. Likewise, an additional merge may be required for the signal. Unfortunately, it is not possible to determine the savings in the number of merge instructions, as every merge instruction turns into a fork, which is indistinguishable from regular forks. Therefore

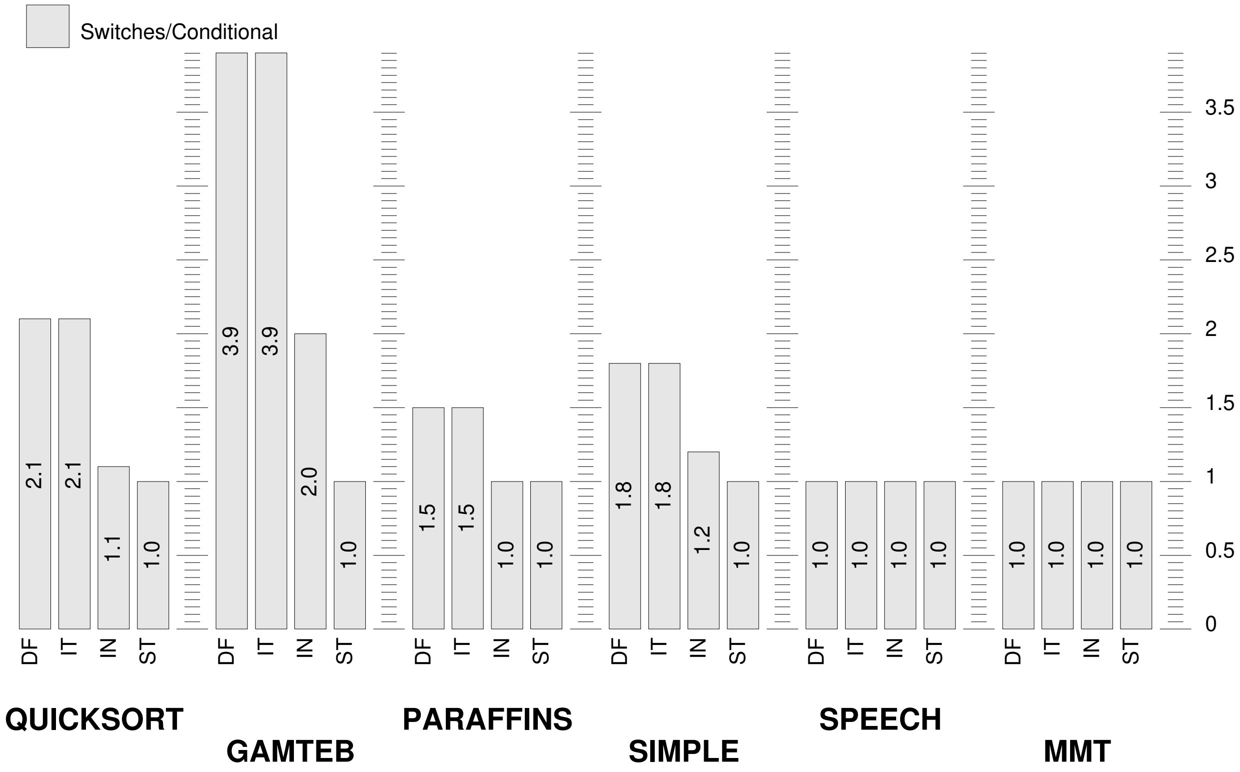


Figure 6.5: *Average number of switch instructions per conditional.*

we focus on the number of switches.[4] As the numbers from Figure 6.5 indicate, under dataflow partitioning the average number of switch operations per conditional is slightly above 2, with the exception of Speech and MMT where is it close to 1. This is due to the inner loops which also use switches, but which are compiled as strict by default. Interprocedural partitioning is very successful and can eliminate most of the possible switches for Quicksort, Gamteb, and Simple, getting very close to the minimum of one switch instructions per conditional obtained under strict partitioning. The only program where interprocedural partitioning does not perform as well, is Gamteb, where it reduces the number of switch instructions per conditionals by 50% from 3.9 to 2.0. This program, contains lots of unstructured conditionals which access many I-structures, and the analysis, being conservative, fails to recognize that it is actually safe to group the switches.

[4]Actually, for switches we have a similar problem. Besides regular conditionals, switches are also being used in the code generation of loops. Since the purpose of this switch is to steer control depending on the result of the loop predicate, we treat it like a conditional. Remember, that 1-bounded loops are strict, they therefore only produce a single switch. This may artificially decrease the average number of switches per conditional in the case where some tight inner loop without conditionals is frequently executed.

Looking at the programs from this perspective highlights the effectiveness of interprocedural partitioning in grouping boundary nodes. The reason that we did not see this substantial difference between iterated and interprocedural partitioning previously, is that it got buried among all of the other instructions which partitioning does not affect. The reason that our interprocedural partitioning algorithm did not come closer to the solution obtained by the "oracle" strict partitioning, is because of I-structure accesses. Our algorithm does not analyze I-structure accesses, so it must assume that every I-fetch may potentially depend on any I-store. Integrating subscript analysis, pointer analysis, and escape analysis into the algorithm should alleviate this problem. As we will see in the next section, our execution vehicle, TAM, effectively deals with ungrouped messages in the case where the compiler fails to group them together statically. The two level scheduling hierarchy of TAM essentially does this "grouping dynamically" as multiple arguments may accumulate before a function is scheduled. TAM provides the fall-back mechanism when the compiler fails to group arguments statically.

6.3 Compiler-controlled Multithreading

Our approach in viewing the dynamic scheduling required by non-strict programs as a compilation problem, rather than a fundamental architectural problem, has two facets. First, the compiler tries to eliminate as much dynamic scheduling as possible by partitioning the program into threads. We studied the effectiveness of partitioning in the previous section. Second, the remaining dynamic scheduling is implemented as efficiently as possible, by specializing it and using the cheapest form available in each specific context. This is the goal of the two level scheduling hierarchy provided by TAM, which runs all enabled threads of the same function activation before moving to a different function. This section analyzes the success of TAM's scheduling hierarchy. The next section puts both parts together and presents actual run-time performance.

6.3.1 Scheduling Hierarchy

The most distinguishing feature of TAM is its two level scheduling hierarchy. This scheduling hierarchy is very important for the execution of non-strict programs, because, even if the compiler does not group instructions into a single static thread — either because it has to be conservative or because of potentially long latency accesses — the scheduling hierarchy often may execute these instructions together, grouping them dynamically into a single quantum. Table 6.4 shows several dynamic scheduling statistics collected on a SparcStation 10 for the benchmark programs under the different partitioning schemes. The first three columns contain the average number of instructions per thread (IPT), threads per quanta (TPQ), and quanta per activation (QPA). As discussed in the previous section, the average thread size increases as the partitioning scheme improves. Dataflow partitioning obtains anywhere from 2.0 to 4.1 instructions per thread. Iterated partitioning improves the thread sizes from 2.7 up to 17.7 instructions per thread. For most of the programs, interprocedural partitioning and strict partitioning does not increase thread sizes by much, because as the number the threads decrease so does the number of control related instructions.

If TAM's scheduling hierarchy were to work perfectly, one would expect the number of quanta per activation not to change across the different partitioning schemes. In the case where partitioning fails to group instructions into a single thread, TAM's scheduling hierarchy would compensate for it and ensure that they nevertheless execute in the same quantum. While this is the case for some of the programs, *e.g.*, Paraffins, which always has 2.0 quanta per activation independently of the partitioning scheme, some programs show a big difference in quanta per activation, most notably Quicksort and MMT. For Quicksort the quanta per activations drops from 7.5 for dataflow partitioning, to 2.4 for iterated partitioning, 2.0 for interprocedural partitioning, and finally to 1.8 for strict partitioning. For MMT the difference is not quite as large. Here the quanta per activation drops from 3.8 for dataflow partitioning down to 2.5 for iterated, increases again to 2.9 for interprocedural partitioning, and drops to 1.6 for strict partitioning. Why does TAM's scheduling hierarchy not keep the number of quanta per activation fairly constant? As we discovered, the reason is that non-strict execution enables consumer/producer parallelism, which may result in a different scheduling behavior. Let us illustrate this behavior using Quicksort and MMT.

The Quicksort program consists of three parts: a function which produces a list of random numbers, the actual quicksort routine, and finally a function which checks whether the result list is sorted. The situation is similar for MMT. Here the main function calls four other functions: two which produce the initial two matrices, the actual matrix multiply

Program	Partitioning	IPT	TPQ	QPA	RCV size	TPDQ	TFDQ
QUICKSORT	DF	2.0	15.8	7.5	1.4	1.2	13.2
	IT	2.7	23.8	2.4	1.1	3.2	19.5
	IN	2.7	21.5	2.0	1.2	3.8	16.4
	ST	2.6	21.0	1.8	1.0	4.1	16.0
GAMTEB	DF	2.1	35.7	3.0	3.7	3.2	28.8
	IT	3.2	17.1	2.6	1.9	2.0	13.3
	IN	3.6	14.8	2.2	1.6	2.4	10.8
	ST	4.2	10.0	2.0	1.0	2.7	6.2
PARAFFINS	DF	2.5	864.3	2.0	2.0	140.2	722.1
	IT	3.0	402.2	2.0	1.0	61.3	339.8
	IN	3.2	344.6	2.0	1.0	61.4	282.2
	ST	3.1	343.3	2.0	1.0	59.9	282.4
SIMPLE	DF	2.7	50.3	2.5	3.7	13.1	33.5
	IT	7.9	9.4	2.1	1.6	2.4	5.3
	IN	9.0	7.7	2.0	1.1	2.5	4.1
	ST	9.6	7.0	2.0	1.0	2.5	3.5
SPEECH	DF	2.8	593.7	2.3	4.0	188.0	401.6
	IT	6.4	202.4	1.8	1.0	65.0	136.3
	IN	6.4	199.4	1.8	1.0	64.9	133.5
	ST	6.5	196.6	1.8	1.0	64.9	130.7
MMT	DF	4.1	3142.5	3.8	2.4	786.6	2353.4
	IT	17.7	718.3	2.5	1.2	182.5	534.6
	IN	17.7	636.6	2.9	1.2	161.7	473.7
	ST	17.7	1160.2	1.6	1.0	295.8	863.4

Table 6.4: *Dynamic TAM scheduling statistics: thread instructions per thread (IPT), threads per quanta (TPQ), quanta per activation (QPA), average size of the remote continuation vector (RCV size), threads posted during a quantum (TPDQ), and threads forked during a quantum (TFDQ).*

which performs the computation, and finally a function which reduces the result matrix to a single number. Unfortunately, with non-strictness, the functions can, and actually do, execute in the wrong order. Initially, the main function sends a trigger to all functions: the producer functions, the consumer function which does most of work, and finally the function which checks the result. Now all functions are ready to execute. At this point, the scheduler — not knowing which function is more important to work on first — picks up the function which checks the result. As the result is not yet available this function immediately returns. Next the consumer function is called, which also cannot perform much work without its other arguments. Finally, the producer is scheduled which creates the initial data structure; thereafter, the consumer and checker are invoked again, this time in the right order.

These two programs highlight a drawback of non-strict execution. While it may increase the parallelism in a program, it may also lead to a much larger scheduling overhead. In these examples, functions are scheduled in the wrong order, which results in a higher number of quanta per activation than with optimal scheduling. Nevertheless, we want to emphasize, that without the scheduling hierarchy of TAM, the anomaly just observed may be further aggravated. For example, in the case of MMT, the scheduling hierarchy ensures that the functions which produce the initial matrices, once scheduled, run until the whole matrix is produced. These observations also highlight the importance of interprocedural partitioning. Even if it does not increase the actual thread sizes by much, it may still improve the scheduling behavior by ensuring that a function is only invoked after several required arguments are available. For our programs, the only exception is MMT, where interprocedural partitioning does not completely manage to group all arguments and all results at function calls, and due to arbitrary scheduling the behavior under iterated partitioning is somewhat better.

What about the remaining dynamic scheduling characteristics? Looking at the second column of the table, we see that the number of threads per quanta is surprisingly high, from 7 for Simple under strict partitioning up to 3,100 for MMT under dataflow partitioning. Multiplying this number with the average thread length, we see that a substantial number of instructions are executed per quanta, anywhere from 40 for Gamteb to several thousand for MMT. Thus, the most expensive scheduling operation, the frame switch at quanta boundary, is not executed very frequently, and its overhead is amortized among many instructions. For most of the programs, the number of threads per quanta decreases as the partitioning improves. This is due to the fact that the total number of threads executed drops dramatically with improved partitioning (*cf*. Table 6.2), while the number of quanta does not change that much. Again, the two exceptions are Quicksort and MMT. In the case of MMT, the number of threads executed drops by a factor of 7 when going from dataflow partitioning to iterated partitioning, but does not change much thereafter. As a consequence, the number of threads per quanta first decreases from 3142 under dataflow partitioning to 718 under iterated partitioning, but later increases again to 1160 under strict partitioning because there is a drop in the number of quanta per activations.

Where does the fairly large number of threads per quanta come from? As the last three columns of the table indicate, there are three sources. First, threads can be enabled for execution while a function is not running. These threads are accumulated in the remote continuation vector (RCV) of the corresponding activation frame. Column four of the table indicates the average size of the RCV when a frame is scheduled. Under dataflow partitioning, this number varies from 1.4 for Quicksort up to close to 4 for Gamteb, Speech, and Simple. Due to TAM's scheduling hierarchy, multiple arguments accumulate and the corresponding threads are enabled before the function is actually started. With improved partitioning, the average RCV size drops substantially. This is also the case under iterated partitioning, even though this scheme does not group multiple arguments into a larger message. Under iterated partitioning the RCV sizes drop to 1.1 for Quicksort and to 1.9 for Gamteb. The reason is that multiple arguments synchronize at inlet level on the receiving end, and therefore a thread is only posted after they all have arrived. With interprocedural partitioning, this effect is further strengthened, and finally, under strict partitioning the RCV only contains a single item when a function is scheduled! This is an indication that for strict programs, the generality of an RCV — which accumulates work while a function is not running — is not required. Current on-going research is studying how to further refine TAM's execution model for this case.

The threads enabled on the RCV only account for a small fraction of the total number of threads which are executed during a quantum. Of the remaining threads, on average about 1/4 are posted from an inlet while the frame is currently running, and the other 3/4 are forked from other threads. For example, for Gamteb under interprocedural partitioning in addition to the 1.6 threads initially on the RCV, 2.4 threads are posted during a quantum and 10.8 threads are forked, resulting in a total of 14.8 threads per quantum. For MMT under interprocedural partitioning, 1.2 threads are initially on the RCV, 161.7 threads are posted, and an additional 636.6 threads are forked, giving us a total of 473.7 threads per quantum. The substantial number of threads posted during a quantum is due to I-structure accesses which return immediately. Since this set of numbers was collected on a uniprocessor, no communication latency occurs, and accesses to data structure return immediately unless the corresponding element is not present, in which case they defer. For these program only a very small fraction of I-structure accesses actually defer. TAM's scheduling hierarchy takes advantage of this and integrates I-structure accesses returning immediately into the on-going computation.

6.3.2 Scheduling Hierarchy Under Parallel Execution

Table 6.5 indicates the effectiveness of the two-level scheduling hierarchy under long remote access latency and parallel execution. These statistics were collected on a 64 processor CM-5, and the programs were compiled using iterated partitioning. For the sequential measurements presented so far, we have compiled all loops as 1-bounded loops. On a parallel machine, we compile k-bounded loops such that k frames are allocated, which allows k iterations of the loop to be executed in parallel. Therefore, the total number of executed threads and the average thread length may differ from the sequential statistics. We see that depending on the application the compiler generates threads of about 3 to 17 TL0 instructions. However, anywhere from 7 to 530 threads execute in each quantum, which is smaller than the equivalent sequential numbers. But still, a substantial amount of work is executed in each quanta, thus amortizing the cost of posting and swapping the frame. As the quanta per activations show, a typical frame experiences about four periods of activity during its lifetime. This is about twice as much as under sequential execution. The difference is especially high for Speech; compare an average 21.7 quanta per activation on the parallel machine to only 1.8 on the sequential machine. The reason is that under parallel execution remote references do not always return in time, so processors schedule some other enabled frame, resulting in a new quantum. In the case of Speech, the functions are fairly small and do not contain sufficient parallel slackness to tolerate the long communication latencies.

The origin of the threads comprising a quantum is given in the bottom middle rows of Table 6.5. We see that when a frame is run, it has usually accumulated multiple threads. Since each successful post of a thread may require multiple posts (*e.g.*, in Gamteb 2.5 posts are required before a thread is pushed onto the RCV), a sizable amount of data will have been accumulated in the frame before it is scheduled. As a result, several potential synchronization events are passed without suspension and many threads are forked while the frame is running.

Typically, more than one message response arrives during the quantum in which it

	Qsort	Gamteb	Paraffins	Simple	Speech	MMT
Ave TL0 Insts. per Thread	2.6	3.2	3.1	5.3	6.3	17.6
Threads per Quanta	11.5	13.5	215.5	7.5	16.7	530.0
Quanta per Invocation	4.1	3.4	2.7	4.8	21.7	3.4
RCV Size when Scheduled	1.1	1.6	1.3	1.4	1.0	1.6
Threads forked dur. Quant.	8.8	10.2	168.4	4.1	11.7	406.6
Threads posted dur. Quant.	1.5	1.6	45.7	1.9	4.0	121.9
Non-synch. Threads	60.7%	40.6%	66.5%	45.9%	65.0%	73.4%
Average entry	2.4	2.5	3.0	3.7	4.4	7.0

Table 6.5: *Dynamic scheduling characteristics under TAM for the programs on a 64 processor CM-5 (for iterated partitioning). The bottom row gives the average entry count for synchronizing threads.*

was issued, triggering further activity. Notice, I-fetches that are serviced locally and not deferred will complete during the issuing quantum. In fact, due to the amount of time it takes to issue a remote I-fetch, if the requests are not deferred, any series of four requests will generally cause a response in the same quantum.

Latency tolerance occurs in two ways: within a quantum and across quanta. Within the quantum, multiple requests are issued and before the quantum ends more than one of them will have completed and returned. Between quanta there is sufficient parallelism that when one frame finishes its quantum other frames have accumulated work to do.

6.3.3 Handling Remote Accesses

The large quanta size is achieved in spite of a fairly high remote reference rate. Table 6.6 shows the breakdown of split-phase operations for the two largest programs, Gamteb and Simple. For heap accesses, I-fetch and I-store operations are divided further to show the fraction of accesses to an element that is local to the processor issuing the access. The I-structure allocation policy currently implemented recognizes two I-structure mappings. I-structures that are smaller than some threshold are allocated within a single processor, while those larger than the threshold are interleaved across processors. Small structures are allocated on the processor that requests the allocation, if space is available. We see that in Gamteb almost all the I-stores are local under this policy, as are one quarter of the I-fetches. Gamteb allocates many small tuples dynamically as particles are traced through the geometry. Under this policy, these are created and filled in on one node, but only a quarter of the I-fetches are local. Simple, on the other hand, operates mostly on large grids interleaved over the machine. No correspondence is established between the data structures and the computations that access them. As a result, the cost of an average remote reference for Simple is higher than for Gamteb (which is reflected in Figure 6.9).

Split-Phase Type	Gamteb 8192	Simple 128
I-Fetch	33.3%	66.3%
Local	26.4%	2.1%
Remote	73.6%	97.9%
I-Store	20.1%	9.1%
Local	99.1%	15.2%
Remote	0.9%	84.8%
Send	33.7%	19.1%
other	12.9%	5.5%

Table 6.6: *Percentages of split-phase operations into instruction types and locality. In Gamteb local allocation of small structures allows most of the stores and many of the fetches to be serviced without network access. Localization in Simple is much less successful, since the data structures are large and no correlation is established between program and data mapping.*

By extending our attention to another layer in the memory hierarchy, the heap, we can try to avoid latency as well as tolerate it. In order to reduce the number of remote references and the communication overhead, we introduce caching of remote references. For programs like Simple, where the remote reference rate is high and the number of deferred I-fetches is low (23%), the cache works remarkably well. Besides seeing the immediate effect of lowering remote references from 97% to 21%, it boosts the quantum size from 7.6 threads to 12.6.

The numbers presented so far highlight the value of TAM's scheduling hierarchy, both in dealing with potential non-strictness and the potential long latency nature of data structure accesses. For the case where partitioning fails to group arguments or results statically, the scheduling hierarchy still ensures that multiple arguments accumulate before starting the function. Similarly, since most data structure accesses do not defer and return within the same quantum, they can be integrated without delay into the on-going computation.

6.4 Overall Efficiency

In the previous two sections we have studied the effectiveness of partitioning in reducing the control overhead and the effectiveness of TAM in implementing the remaining dynamic scheduling. In this section we put both parts together and present run time measurements.

6.4.1 Timing Measurements

Until now we have only used abstract performance characteristics — number of threads executed and TL0 instruction distributions — to analyze the effectiveness of partitioning. The distribution of TL0 instructions alone does not reflect the breakdown in execution time, as each TL0 primitive has different costs associated with it. Table 6.7 shows the actual run time measurements for the benchmark programs on a SparcStation 10 under the various partitioning schemes. Figure 6.6 shows the same timings graphically, normalized to dataflow partitioning.

Program	Input Size	DF	IT	IN	ST
QUICKSORT	10,000	6.6	2.2	1.6	1.5
Gamteb	40,000	573.8	373.7	245.1	169.5
Paraffins	19	3.4	2.5	2.2	2.2
Simple	1 1 100	7.1	3.9	2.9	2.7
Speech	10240 30	1.4	0.6	0.6	0.6
MMT	500	130.2	66.1	61.0	61.0

Table 6.7: *Dynamic run-time in seconds on a SparcStation 10 for the benchmark programs under various partitioning schemes.*

As we can see, improved partitioning directly translates into improvements in execution time. Surprisingly, this improvement approximately matches the dynamic thread and inlet distributions presented earlier in Figure 6.2. Iterated partitioning obtains most of the gain, reducing the runtime by around a factor of two. Interprocedural partitioning results only in a modest additional gain, the largest is for Gamteb where it reduces execution time by another 30% over iterated partitioning. Further improvements for strict partitioning are insignificant for most of the programs, an indication that both the interprocedural

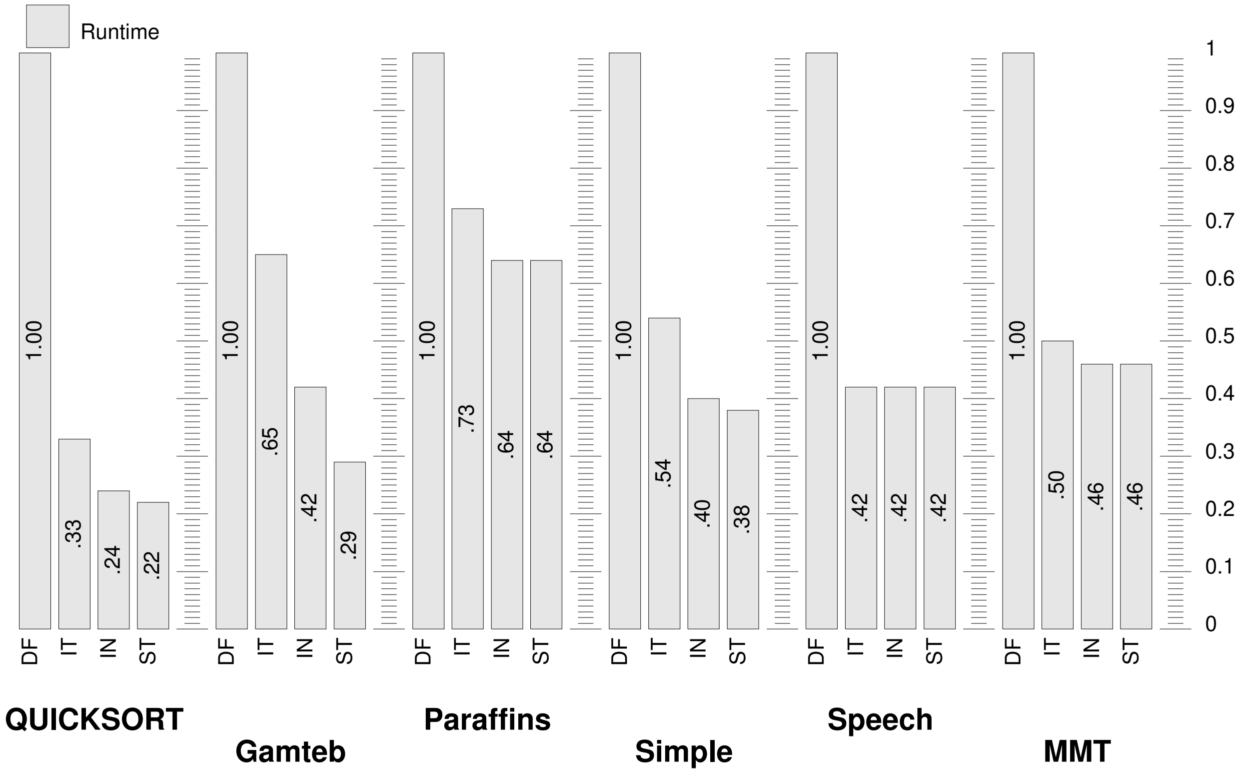


Figure 6.6: *Dynamic run-time on a SparcStation 10 for the benchmark programs under various partitioning schemes (normalized to dataflow partitioning).*

partitioning and TAM's scheduling hierarchy are quite effective. Gamteb is the only program for which improvements to the interprocedural partitioning algorithm could further reduce the run time (by at most 30%).

6.4.2 Cycle Distributions

Run time measurements alone do not tell us were the execution time is being spent. To better understand this, we compute the execution cost in cycles for each of the TL0 primitives and multiply this by their execution frequency. Most of the TL0 primitives can be specialized depending on the context in which they appear, and they may have different costs associated with the various cases. For example, a FORK of a thread, can be either a fall-through, jump, or real fork, depending on whether it is the last instruction in a thread or not, and whether it jumps to the immediately following thread or not. In addition, we can distinguish whether a fork is synchronizing or not, and in the case where it is synchronizing, whether the synchronization succeeds or fails. Similar specializations can be derived for the other TL0 instructions, *e.g.*, SEND (local/remote), POST (idle/ready/running, sync/no-sync, succ/fail), I-FETCH (present/deferred, local/remote), *etc.* For a extended discussion of the methodology, including a description of the special cases of the various TL0 instruction and their precise cycle counts, see [CGSvE93, SGS+93]. We have analyzed the implementation of each of the primitives for a Sparc processor, by studying the best possible implementation in Sparc assembly code. For statistics collection, the code-generator inserts a few instructions into the threads and inlets to collect roughly one hundred dynamic statistics on each processor. At the end of the program, the overall counts are accumulated. This provides TL0-level dynamic instruction frequencies, which characterize the requirements of Id90 programs. The frequency of dynamic scheduling events is obtained as well. Multiplying these frequencies with their cost give us a good estimate how much each component contributes to the overall execution time.

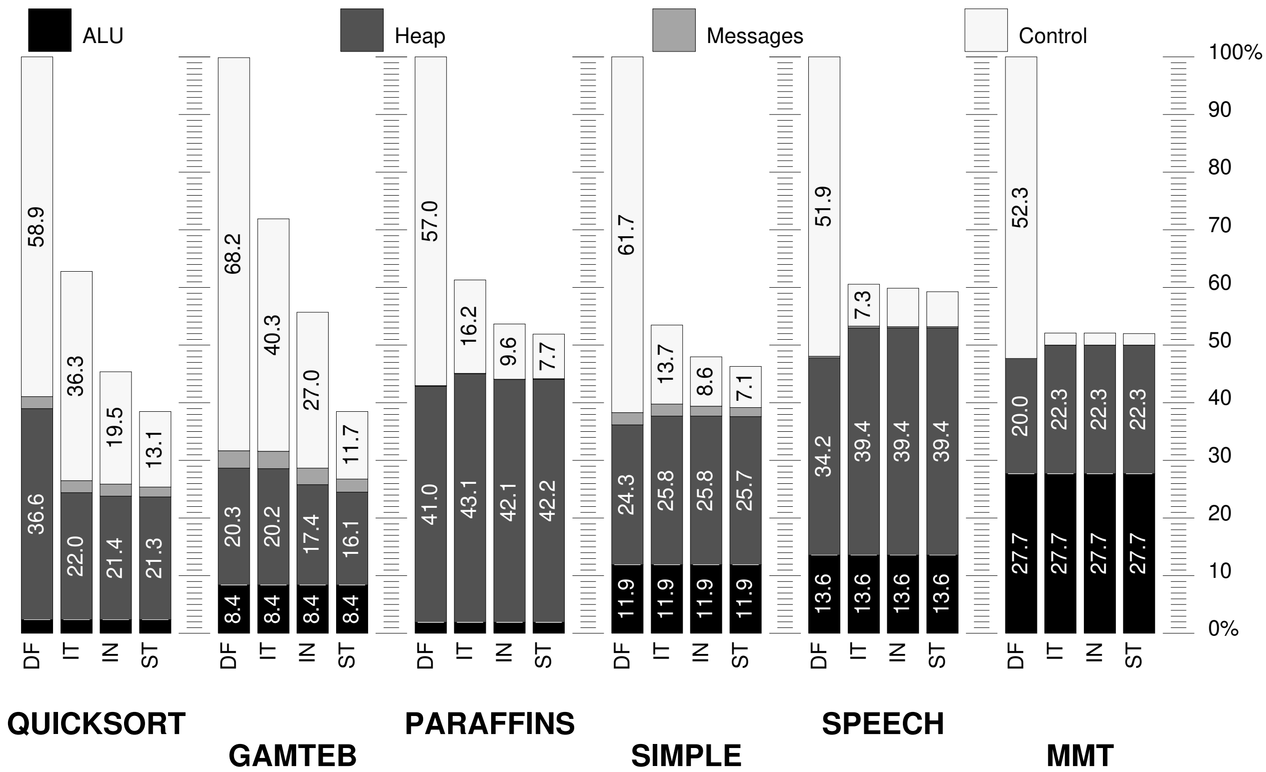


Figure 6.7: *Dynamic cycle distribution of TL0 instructions for a Sparc processor for the benchmark programs under various partitioning schemes (normalized to dataflow partitioning).*

The resulting cycle distributions for a Sparc processor are shown in Figure 6.7, normalized to dataflow partitioning. This distribution shows what fraction of machine cycles are spent in each of the four categories. As before, *ALU* includes integer and floating-point arithmetic, *heap* includes global I-structure and M-structure accesses, *messages* includes instructions executed to handle messages, and *control* represents all control-flow instructions including moves to initialize synchronization counters. The one major difference is that the cycles spent executing inlet instructions for handling heap accesses are accounted for under the heap category instead of the message category. The reason is that we want to distinguish them from messages required for passing of arguments and results. One note of caution: our current TAM backend maps the TL0 instructions to C code, therefore some categories, especially control operations, are likely to be more expensive than indicated here.

Observe how different the cycle distribution is from the TL0 instruction breakdown presented in Figure 6.3. Obviously, the relative ratio of the heap and message categories have changed, since we are accounting for heap related message handling in the heap category. But the relative ratios of the other categories has also changed. For example, ALU operations do not consume as many cycles as more complicated heap or control operations. Observe, that the programs with tight inner loops — Paraffins, Speech, and MMT — call functions relatively infrequently, and therefore have an insignificant message category for passing of arguments.

The decrease in control instructions is much higher than previously indicated by plain TL0 instruction distributions. The reason is that improved partitioning mostly eliminates expensive control operations. For example, in MMT, the TL0 instructions required for control drops by a factor of 4.2 when going from dataflow partitioning to iterated partitioning, while the cycle count drops by a factor of 25.9. Under dataflow partitioning most of the forks are expensive synchronizing forks. The other partitioning schemes optimize most these away, leaving only non-synchronizing branches. These can be implemented very efficiently, as they are directly supported by the Sparc instruction set.

Not surprisingly, after good partitioning most of the execution time is spent in heap related instructions. I-structures require emulating presence bits in software. I-fetches may defer if the element is not present. In addition, on a parallel machine communication may be necessary because I-structures can reside on some other processor or be spread across the machine. Even though most I-fetches do not defer, just testing for these cases and manipulating the tag bits increases the cost of heap accesses.

The cycle distributions show that the gain obtained from partitioning is diluted by the fairly high overhead present in the execution of these programs. This overhead is independent of the partitioning schemes and arises out of the implicitly parallel nature of Id, which requires expensive I-structure accesses, heap management for parallel calls, and communication for sending of arguments and results. Reducing this overhead by special hardware support or additional compilation techniques makes partitioning a much more important optimization, as the relative performance difference between best and worst partitioning scheme will be much larger.

6.4.3 Sequential Efficiency

Observing this high overhead in heap accesses we already see that the sequential efficiency of Id currently cannot match that of imperative languages. For some of our benchmark programs we had an equivalent C or FORTRAN version available, so a comparison seems natural. We compiled all of those programs with -O4, using gcc 2.3.3 or f77, and measured their execution times on the same SparcStation10. The timing results are shown in Figure 6.8. The table also contains the run-time measurements collected on a 1PE Monsoon dataflow machine, as reported in [HCAA93].

Program	Input Size	Id (Monsoon)	Id (Sparc)	C/FORTRAN	Ratio
Quicksort	10,000		1.6	0.6	2.7
Gamteb 9-cell	40,000	336	245.1		
Gamteb 2-cell	40,000	59	32.9	5.8	5.7
Paraffins	19	2.4	2.2	0.2	11
Simple	1 1 100	6	2.9	0.5	5.8
Simple	100 1 100		270.2	43.2	6.3
MMT	500	106	61.0	18.0	3.4

Table 6.8: *Dynamic run-time in seconds on Monsoon and on a SparcStation 10 for several of the Id benchmark programs and equivalent C or FORTRAN programs. Gamteb and Simple are FORTRAN programs, while Quicksort, Paraffins, and MMT are C programs. The first column shows the measurements on a 1PE Monsoon. The last column reports the ratio of execution times between Id and C/FORTRAN.*

As expected, the performance of our Id implementation is still far from C or FORTRAN — between 2.7 and 11 times slower. For the two largest programs, the ratio between Id and FORTRAN is about 5.8. Even the blocked matrix multiply is a factor of 3.4 slower, although the partitioning algorithm produces large threads, making the control overhead insignificant. On the other hand, this implementation is more than a factor of 300 faster than previously available dataflow graph interpreters, showing the significant reduction in control overhead that can be obtained by going to an threaded execution model [Sch91].

When comparing against the dataflow machine, we see that the Id programs always run faster on the SparcStation 10 than on Monsoon, which is an additional indication that lenient languages do not really require special hardware support for dynamic scheduling. The SparcStation is between 1.1 to 2.1 times faster, depending on the program. One has to take into account that the two machines are vastly different. The Monsoon is an experimental prototype which became operational in 1990, while the Sparcstation is a mature product which became operational in 1992. They were developed with completely different goals in mind. Monsoon not only provides support for dynamic scheduling in hardware, but also integrates the network interface into the processor design. We have yet to see the benefits resulting from fast inter-processor message handling, since so far we compared only sequential measurements. We will address this in the next section. The dataflow machine is a board-level design, consisting of one 10 MHZ 64-bit Monsoon processor with 256 K word store and one I-structure board with 4 M word store. The processor and memory were connected by a packet communication network. The Monsoon processor uses a 10 MFLOPS ECL floating point co-processor. The SparcStation 10 workstation consists of a 50 MHz superscalar Sparc CMOS processor with a 20 KByte first-level on-chip instruction and 16 KByte first-level on-chip data cache. In addition, the system also has a 1 MB off-chip second level cache, and 128 MByte of DRAM main memory. The SparcStation achieves 50 MFLOPS peak performance, and 27 MFLOPS on Linpack N=1000. Thus, if we were to normalize for cycle time or for peak MFLOPS, the Monsoon is between 2.9 and 4.6 times faster on above programs.

Discounting these differences, it is noteworthy that a software solution on continually improving hardware runs faster than the specialized hardware which was specifically built to support the fine-grained dynamic scheduling of implicitly parallel languages. A hardware prototype will never be able to compete with the exponential performance improvement (about 1.5$\times$ faster per year) that conventional microprocessors experience. The prototype is also unlikely to employ the latest technology. Thus, the performance difference must be substantially higher to warrant a special purpose hardware solution. As discussed before, additional improvements in compilation techniques which address the remaining overhead, *e.g.* the heap accesses, will further strengthen the software solution, making the special hardware solution even less attractive.

For sequential execution, conventional RISC based workstations are certainly the desired platform. Next, let us focus on parallel execution.

6.4.4 Parallel Efficiency

In this section, we report on the performance observed for Id programs on the CM-5 multiprocessor. The machine consists of 64 Sparc processors running at 33MHz, each with a 32 MByte memory, a 64 KByte cache, and a floating-point co-processor. The processors are connected as an incomplete fat-tree of degree four, so the maximum distance between processors is 6 hops and the average distance is 5.4 hops. Communication is supported by Active Messages[vECGS92] driving a custom network interface via the memory bus. Sending or receiving a message takes approximately 50 cycles.

We focus on our two largest application benchmark programs: Gamteb and Simple, which are the most interesting of our applications in this context, since they have the most unstructured form of parallelism. We have not modified the source programs to tune for our environment, although such tuning is certainly possible (and desirable). We have not yet collected the run-time measurements for the other benchmark programs, although we will do so in the near future. It is likely that they will perform much worse than the two larger problems. In particular, for very structured problems, such as MMT, our performance will not compare favorably to explicit parallel implementations written in C or FORTRAN. The reason is that no correlation between work and data is currently established, resulting in a much higher remote reference rate than obtained by explicitly parallel programs. Not only will our implementation lose a factor of 3 in sequential performance, but will also lose a substantial factor in parallel efficiency.

Table 6.9 gives the executions times for Gamteb for 1 to 64 processors, while Table 6.10 gives the numbers for Simple. Figure 6.8 shows the speedup obtained on the CM-5

Processors	Input	k-Bounds	Time	Scaled Speedup
1	128	32	10.5065	1.00
2	256	64	15.9207	1.31
4	512	128	16.4318	2.56
8	1024	256	17.3968	4.83
16	2048	512	17.3444	9.69
32	4096	1024	19.4104	17.32
64	8192	2048	19.8333	33.90

Table 6.9: *Dynamic run-time in seconds on the CM-5 for Gamteb. The input size (number of particles) is maintained constant per processor N = 128*P, the k-bounds are set to k = 32*P. The last column gives the scaled speedup.*

Processors	k-bounds	Time	Speedup
1	2	65.0058	1.0
2	2	66.5908	0.97
4	4	44.6631	1.46
8	8	26.2929	2.47
16	16	12.9114	5.03
32	32	9.18252	7.08
64	64	3.95517	16.44

Table 6.10: *Dynamic run-time in seconds on the CM-5 for Simple, running 1 iteration on a 128x128 grid. The k-bounds for k-bounded loops are varied from 2-64, depending on the number of processors. The last column shows the speedup.*

for the two applications benchmarks. Before commenting on the data, we must state the fairly large set of conditions under which it is collected, including problem scaling, parallelism scaling (k-bounds), frame allocation policy, and I-structure management policy. For Gamteb, the problem size is scaled to maintain a constant number of initial particles per processor ($N = 128 * P$). The parallelism is scaled by adjusting the k-bound on the outer-most loop in proportion to the number of processors ($k = 32 * P$). With k-bounded loops, k iterations of a loop run in parallel. The frame allocation policy wraps k-bounded loop frames around the processors (*i.e.*, processor p allocates the next frame on processor $(p+1) \bmod P$) and function call frames are allocated alternatingly in a local neighborhood of three (one frame for the left neighbor, one for me, one for the right neighbor). I-structures of size 32 or more are allocated horizontally across processors (interleaved), while smaller

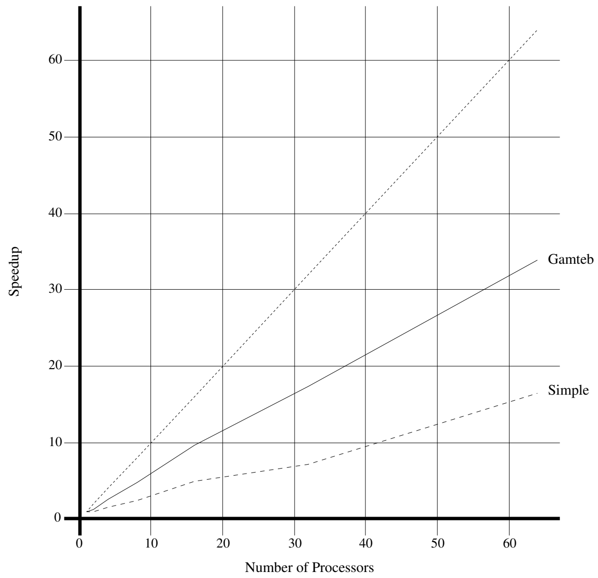


Figure 6.8: *Speedup for Gamteb and Simple on the CM-5 from 1 to 64 processors.*

ones are allocated on the processor that requests the allocation. No caching is employed. For Simple, the problem size is fixed ($N = 128$ for one timestep) and the parallelism is scaled by adjusting the k-bound for the outer loop of several loop nests that iterate the grids ($k = 2 * P$). The frame allocation policy is essentially the same as in Gamteb, except that the function call frame policy is a little more "selfish" (one for me, one for left, one for right, one for me). The I-structure allocation policy only differs from Gamteb in that caching is employed. For a more detailed discussion about the effect of parallelism scaling (k-bounds), frame allocation policy, and I-structure management policy on the execution time, see [CGSvE92].

In both applications, we observe a linear speedup beyond a small number of processors. Going from one processor to two, we see the effect of message handling overhead. Roughly half the time is lost to message handling overhead in Gamteb and three-quarters of the time in Simple. The difference is arises from the remote reference rates in the two programs, as discussed in Section 6.3.3. On the high end, this correlates with the speedup obtained on the full machine; on the low end, it correlates with the number of processors required before any significant speedup is obtained. Although the programs could be tuned to obtain better performance on this particular machine, our goal is to evaluate TAM in the regime of implicit parallelism and implicit data placement.

So far, we have not established how the sequential performance on a workstation relates the performance of one CM-5 processor. The SparcStation 10 is a 50 MHz superscalar Sparc processor, while the CM-5 has 33 MHz regular Sparc processor. As indicated in Table 6.11, we have measured the performance of one CM-5 processor using a naive 400×400 matrix multiply and found it to lie between that of a SparcStation IPX and that of a SparcStation 1+. We also measured the execution time of Gamteb on the sequential machines, and compared it with the execution time of the CM-5 version. Using this, we were able to determine that our CM-5 implementation of TL0 is a about a factor of two or three slower than the sequential version. The reason is that the sequential version does some optimizations, such as I-structure access inlining, which are currently not attempted in the CM-5 backend.

Machine	Gamteb (Id)	Matrix Multiply (C)
SparcStation 10	205.4	43.0
SparcStation IPX	439.0	60.8
SparcStation 1+	758.8	112.8
CM-5 (1 proc.)	1537.4	98.0

Table 6.11: *Dynamic run-time in seconds of Gamteb (9-cell), running 40,000 particles on a variety of machines (the same TL0 code). Also shown are the run-times for a C version of naive Matrix Multiply for size 400 x 400.*

	Gamteb		simple	
Processors	Time	Speedup	Time	Speedup
1	59.0	1.0	468.1	1.0
2	30.3	1.95	251.8	1.86
4	15.5	3.81	135.5	3.45
8	7.35	7.35	74.7	6.27

Table 6.12: *Dynamic run-time in seconds on Monsoon for Gamteb (2-cell), running 40,000 particles, and Simple, running 100 iterations on a 100x100 grid.*

Table 6.12 show the speedup for the Gamteb (2-cell) and Simple on Monsoon, varying the number of processors from 1 to 8. These timings were reported in [HCAA93]. The version of Gamteb that was used for these measurements is a 2-cell version (the equivalent to the FORTRAN program), which runs substantially faster than the 9-cell version we have been using until now. The largest Monsoon multiprocessor constructed is made up of eight processing element and eight I-structure boards, connected by a two-stage, packet-switched butterfly network composed of 4×4 switches. The Monsoon processor has an 8-stage pipeline. Thus, to be fully utilized, the program has to show at least 64-fold parallelism.

As the speedup tables show, Monsoon achieves a near perfect speedup on 8 processors. The main reason is that Monsoon supports very efficient communication. Thus, unlike the CM-5, Monsoon is very tolerant to a bad data mapping. For parallel execution of implicitly parallel programs, it is a benefit to have fast communication and I-structure accesses.

6.4.5 Parallel Cycle Distributions

Of course, we are also interested to know where the execution time is being spent on the parallel machine, but because real communication is involved for heap instructions, we break the heap category up into its three components: the time required for heap related messages, the time for manipulating the tag bits, and the time spend in heap related control. Figure 6.9 shows the breakdown of Cycles Per TL0 instruction (CPT) on the CM-5 for each benchmark program under iterated partitioning. The CPT is the work involved in an average TL0 instruction. The CPT is broken down into bars showing the contribution resulting from each TL0 instruction class. In addition, the Operand bar at the top reflects the memory access penalty in bringing data into registers for ALU instructions, assuming a 5% miss rate. The Atomicity bar at the bottom accounts for the overhead introduced by polling the network. The Heap bar has been split into three distinct implementation components. We see that the focus on efficient thread scheduling pays off, as control represents only 10 to 30 percent of the total costs. By comparison, one half to three quarters of the time is spent in the processor network interface. Even though the handling of these messages is very efficient (we are using Active Messages [vECGS92]), the simple fact that the network interface is on the memory bus, rather than the cache bus, accounts for almost all of this cost. Given the compilation techniques represented by TAM, the most important architectural investment for supporting emerging parallel languages is simply to bring the network interface closer to the processor. However, this must be accomplished without unduly increasing the memory access and atomicity costs.

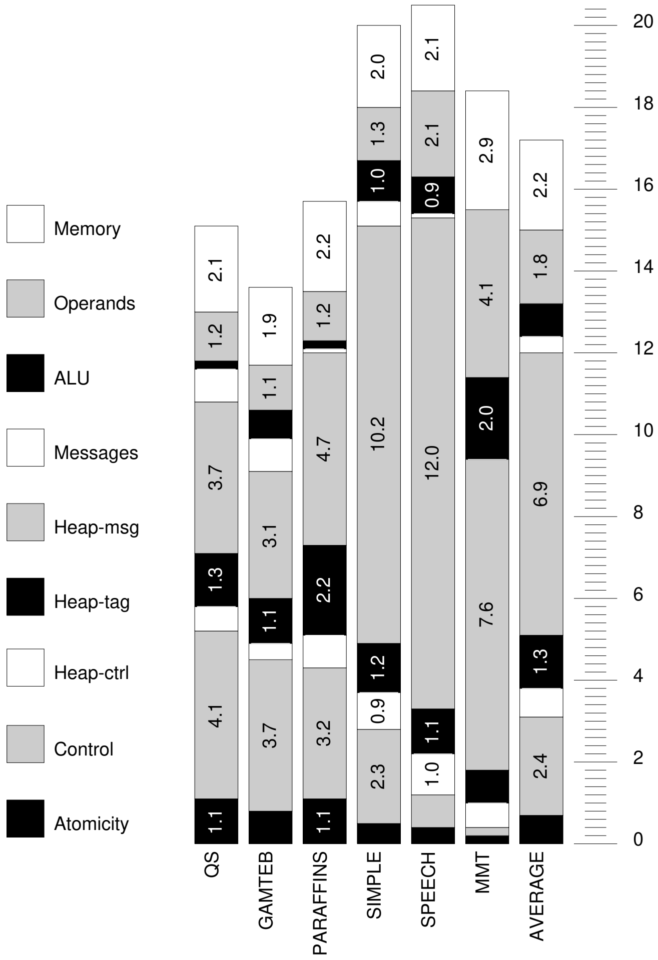


Figure 6.9: *Distribution of processor time for the benchmark programs. The metric used is Sparc cycles per TL0 instruction (CPT); bars show the contribution to the CPT resulting from each TL0 instruction class.*

6.5 Summary

The measurements presented in this chapter clearly demonstrate the effectiveness of our partitioning algorithms in reducing the dynamic scheduling required for both sequential and parallel execution of Id programs. We compared four different partitioning strategies: dataflow, iterated, interprocedural, and strict. Dataflow partitioning reflects the limited thread capabilities of dataflow machines, while strict partitioning represents the best possible. Programs compiled with iterated partitioning execute significantly fewer threads than under dataflow partitioning. A similar picture exists when looking at other performance metrics, such as TL0 control instructions or execution time. Interprocedural partitioning results only in modest additional gains, mostly because the gain is hidden under the large remainder of the program not affected by partitioning. Looking at a more precise indicator, the number of send instructions per call and the number of switch instructions per conditional, we see that there is actually a significant difference. We also compared interprocedural partitioning to strict partitioning, and saw that, with few exceptions, improvements to the interprocedural partitioning algorithm would only result in a small further gain.

We then studied the effectiveness of TAM's scheduling hierarchy with respect to dealing with the remaining scheduling; it works well, even in the presence of long latency communication on the CM-5. On the CM-5; we observed a linear speedup after the first couple of processors, but nevertheless most of the execution time is spent on communication. Even with the best partitioning algorithm and the TAM scheduling hierarchy, the overall efficiency is still disappointing when compared to program written in imperative sequential languages C or FORTRAN. The greatest part of the inefficiency is introduced by the large fraction of I-structure accesses which incur a high overhead on conventional machines. If this overhead were to be reduced, either by special hardware support or by additional compilation techniques, then good partitioning would play an even more important role, as the difference between the partitioning schemes would become more pronounced.

Chapter 7

Conclusions and Future Work

This thesis presents novel compilation techniques for implicitly parallel languages, focusing on the lenient functional language Id. The language allows an elegant formulation of a broad class of problems while exposing substantial parallelism. However, its non-strict semantics require fine-grain dynamic scheduling and synchronization, making an efficient implementation on conventional parallel machines challenging. This dissertation shows that it is possible to efficiently implement this language by partitioning the program into regions that can be scheduled statically as sequential threads and realizing the remaining dynamic scheduling through compiler-controlled multithreading. Partitioning the program into sequential threads requires substantial compiler analysis because, unlike in imperative languages, the evaluation order is not specified by the programmer. Care has to be taken to generate threads which obey all dynamic data dependencies and avoid deadlock.

We now present related work, future directions, and finish with our conclusions.

7.1 Related Work

Over the past two decades much research has been done in compiling lenient languages for dataflow architectures [ACI$^+$83, Tra86, AN90, GKW85, Cul90, GDHH89]. Only in recent years have researchers come to understand that language and architecture can be studied separately and have begun to study compilation aspects of these languages for commodity processors.

Our own research was inspired by Traub's seminal theoretical work on partitioning of non-strict programs into sequential threads [Tra91] and Nikhil's work on the P-RISC, an abstract machine for executing such languages [NA89]. Besides these two approaches, we briefly review other research in partitioning non-strict functional languages.

Serial combinators (Hudak, Goldberg 1985)

Serial combinators by Hudak and Goldberg are one of the first attempts to improve the execution of lazy functional programs by increasing the granularity of the program. Their approach is to group several combinators into larger serial combinators. These techniques were developed to reduce the overhead of very fine-grained programs on parallel machines. Their approach was implemented on several parallel machines [Gol88].

Macro-dataflow partitioning (Sarkar, Hennessy 1986)

Sarkar and Hennessy describe an algorithm for partitioning SISAL programs into macro-actors which are then mapped onto different processors [SH86]. Each macro-actor, since executed on a single processor, essentially corresponds to a sequential thread. Their goal is to improve parallel efficiency by executing the functional program at coarser granularity. Their algorithm attempts to minimize the communication overhead between macro-actors, while exposing sufficient parallelism. This tradeoff is captured by a simple analytical performance model which is used for the partitioning algorithm. Since SISAL is a strict functional language, the macro-actors and the ordering of instructions can be determined statically; therefore the partitioning problem is different from ours. Similar to the requirements of our threads, the macro-actors must be able to run to completion once all their inputs are available.

Partitioning into threads (Traub 1988)

Traub was the first to view the compilation of non-strict languages for sequential execution as a process of partitioning the program into sequential threads [Tra88, Tra91]. Traub develops an elegant (although not quite bug free) theoretical framework, which uses dependence analysis to characterize when instructions can be grouped into the same thread. In addition, he describes code-generation issues for producing threads after a partitioning of the program has been derived. Traub's original framework allows threads to suspend; thus, his threads capture the sequential ordering which is required between instructions, but not the dynamic scheduling which may occur between the threads.

Dependence set partitioning (Iannucci 1988)

Iannucci, developed the first dependence set partitioning algorithm, when looking for a simple but effective strategy for producing threads (he called them scheduling quanta) from Id code for his von-Neuman-dataflow Hybrid [Ian88]. This partitioning scheme, which is part of our iterated partitioning, groups nodes which depend on the same set of inputs [Ian88].

P-RISC execution model (Nikhil 1989)

Although originally developed as an extended RISC like architecture with limited multithreaded hardware support for dynamic scheduling [NA89], the P-RISC execution model can be used as an abstract machine for the compilation of Id [Nik93]. In his current compilation approach, Nikhil employs P-RISC graphs, parallel control-flow graphs that model locality in distributed memory machines. P-RISC graphs are implemented on conventional networks of workstations. While there are many similarities between P-RISC graphs and TAM, the largest difference is that the P-RISC system does not have a two level scheduling hierarchy.

Compiling for locality (Rogers, Pingali 1989)

Rogers and Pingali also start with a functional language very similar to Id [RP94]. They require the programs to be annotated with data distributions, from which they derive a work distribution for distributed memory machines, following the owner computes rule. Their system produces C code which can be executed on message passing machines, such as the iPSC/2. They do not deal with non-strictness; thus, the focus of their work is quite different from thread partitioning.

Entry and exit set partitioning (Hoch, Davenport, Grafe, Steele 1991)

Hoch *et al.*, developed an Id implementation for Sandia's second generation dataflow machine, the multithreaded Epsilon-2 [GH90]. In addition to dependence set partitioning (entry sets in their terminology), they introduce exit sets partitioning (the equivalent of demand sets partitioning), which groups nodes which influence the same set of outputs [HDGS93]. The power of entry sets and exit sets partitioning is combined by applying them iteratively.

Dependence and dominance set partitioning (Schauser, Culler, von Eicken 1991)

Schauser *et al.*, proposed extending the two basic partitioning schemes (which here were called dependence and dominance set partitioning) with local "merge up" and "merge down" rules, thus achieving essentially the same degree of grouping as iterated partitioning [SCvE91]. Their code-generation approach is based on the Threaded Abstract Machine (TAM), the first threaded execution model to refine Traub's definition of threads by disallowing suspension [CSS+91, CGSvE93]. This notion, which has been picked up by most other researchers, has the advantage of capturing the cost of switching between threads. This work resulted in the first full-fledged Id compiler for conventional architectures, running on various sequential as well as parallel machines.

Interprocedural partitioning (Traub, Culler, Schauser 1992)

A year later, Traub *et al.*, presented a method of extending partitioning with interprocedural analysis to obtain larger threads. The algorithm is based on structured dataflow graphs, which captures the original control structure of the program. This provides a clean framework for describing the partitioning algorithm and performing the interprocedural analysis. One limitation of this interprocedural algorithm is that the analysis does not work across recursive functions. This work was essentially the starting point for this thesis.

Semantics of partitioning (Coorg 1994)

Concurrently to this thesis, Coorg developed extensions to the above interprocedural algorithm which can handle recursive functions using analysis based on path semantics [Coo94]. He presents a proof of the correctness of the algorithm, based on the semantics of the simple parallel intermediate functional language P-TAC [AA89]. The algorithm has not been implemented, so its effectiveness cannot be compared to our approach.

7.2 Future Work

There are many fruitful avenues for future work. We list just a few.

I-structure analysis

Currently, the largest remaining inefficiency are I-structure accesses, which incur a high overhead on conventional machines and expensive communication on parallel machines. This overhead could be addressed by providing special hardware support, as proposed by the recent *T project [NPA92]. On the other hand, the overhead can also be attacked by compilation techniques, similar in spirit to partitioning. For example, the compiler may be able to determine that in some cases the generality of I-structures is not required and that regular arrays can be used with plain reads and writes. Of course, the compiler must ensure that the array accesses are ordered correctly, so that reads always return the correct value. The compiler can also reduce the amount of interprocessor communication by deriving a good data layout. Developing compilation techniques for addressing these issues is beyond the scope of this thesis and should be a fruitful area for future research. In any case, reducing the base overhead currently present in these programs will make partitioning an even more important optimization, as the relative performance difference between best and worst partitioning schemes will be much larger.

Work distribution and data mapping

Probably the hardest problem, especially for irregular applications, is the mapping of work and data onto the processors. The goal is to keep the load balanced while minimizing communication by coordinating the data placement and work distribution. A good distribution of work and data improves parallel performance, as it reduces communication and idle time. For very structured problems this mapping can be predetermined by the programmer or compiler [HKT92]. In our case, the non-strictness of the language makes analyzing the data structures very hard even for regular problems, and it is not obvious how compilation techniques developed for FORTRAN or data parallel languages carry over. There should be substantial common techniques between the I-structure analysis proposed in the previous paragraph and this problem. Since we are mostly interested in problems with irregular forms of parallelism, our compiler does not derive the work and data distribution automatically; rather our implementation relies on support from the run-time system. Nonetheless, this issue should be addressed properly and is probably the most critical to obtaining good performance on parallel machines.

Other languages

A new recent development is parallel Haskell (pH), which integrates many concepts of Id into the functional language Haskell. The new language, pH, is also based on lenient evaluation, therefore all of the techniques developed in this thesis directly carry over.

The compilation techniques developed in this thesis are not limited to lenient languages; they can also be adapted to other languages which require either non-strictness or parallel execution. For example, lazy functional languages can be partitioned if basic block partitioning and interprocedural analysis is limited to demand sets. This correctly places operations which are proven to be demanded together into the same thread. This scheme is more powerful than conventional strictness analysis as it may group multiple arguments of a function into a single thread, even if the function is not strict in any of its arguments [Tra89].

In addition to functional or implicitly parallel languages, our compilation techniques may also be applied to imperative languages. By choosing structured dataflow graphs as our intermediate representation, we can the apply our techniques to both parallel and sequential languages. Structured dataflow graphs are a fully parallel intermediate representation. Other researchers have shown, that it is possible, and also desirable, to translate sequential languages, such as FORTRAN, into representations very similar to ours [BP89, FOW87]. Similarly, the compilation techniques can be applied to new emerging parallel object oriented languages, such as parallel dialects of C++, which offer virtual parallelism and do not have a single locus of control.

7.3 Conclusions

The main contribution of this thesis is the development of improved thread partitioning algorithms, which go substantially beyond what was previously available [Tra88, HDGS93, Sch91]. The algorithms presented in [TCS92] served as a starting point. This thesis extends that work in several ways:

- It presents a new basic block partitioning algorithm, *separation constraint partitioning*, which is more powerful than iterated partitioning, the previously best known basic block partitioning.
- It shows how separation constraint partitioning can be used successfully as part of the interprocedural partitioning algorithm for the partitioning of call and def site nodes.
- It extends the interprocedural analysis to deal with recursion and mutually dependent call sites which previous analysis could not handle.
- It develops a theoretical framework for proving the correctness of our partitioning approach. Developing the correctness proofs revealed problems with prior partitioning approaches.
- It implements the partitioning algorithms, resulting in a running execution vehicle of Id for workstations and several parallel machines.
- It quantifies the effectiveness of the different partitioning schemes on whole applications.

We first present several basic block partitioning algorithms — dataflow, dependence set, demand set, iterated, and separation constraint partitioning — each one more precise than its predecessor but also more complex to implement. Then, an interprocedural partitioning algorithm is presented which uses global analysis to form larger threads. The development of these algorithms requires a thorough understanding of the underlying correctness criteria. The thesis shows how a correct partitioning can be derived by analyzing each basic block as one piece and summarizing everything outside of this basic block with annotations. Even with the most sophisticated partitioning algorithm some dynamic scheduling remains, largely due to asynchronous inter-processor communication. This scheduling is handled efficiently by the compiler-controlled multithreading embodied in TAM.

In addition to addressing the theoretical aspects of partitioning, this thesis provides a complete path from the language down to commodity sequential processors and real parallel machines, making it possible to quantitatively study the effectiveness of the compilation techniques on whole applications. The partitioning algorithms have been implemented in our Id compiler which, by compiling via TAM, produces code for workstations and the CM-5 multiprocessor. This experimental platform allows us to verify the effectiveness of the partitioning algorithm and compiler-controlled multithreading. Our implementation of Id is more than two orders of magnitude faster than existing dataflow graph interpreters. As our experimental data shows, most of the gain is due to sophisticated partitioning, which reduces the control overhead substantially compared to dataflow partitioning. Powerful basic block partitioning obtains most of the gain, while interprocedural partitioning achieves a modest additional improvement. By also compiling programs strictly, we are able to verify that this is not due to an inherent limitation of our algorithm. The small thread sizes do not seem to be due to a limitation of the interprocedural analysis, but rather due to long latency operations and conventional control flow.

Measurements on the CM-5 show the effectiveness of TAM's scheduling hierarchy in dealing with the remaining scheduling; it works well, even in the presence of long latency communication. However, even with the best partitioning algorithm and the TAM scheduling hierarchy, the overall sequential efficiency is still disappointing when compared to programs written in imperative sequential languages, such as C or FORTRAN. This overhead is independent of the partitioning schemes and arises out of the implicitly parallel nature of Id, which requires expensive I-structure accesses, heap management for parallel calls, and communication for sending of arguments and results. The greatest inefficiencies are introduced by the large fraction of I-structure accesses which incur a high overhead on conventional machines and expensive communication on parallel machines.

This thesis successfully tackles the hard problem of dynamic scheduling required by implicitly parallel languages. We have placed the basis of our approach — partitioning into sequential threads — on a theoretically sound foundation, and, in doing so, have established a framework for proving the correctness of future improvements to partitioning algorithms. We have empirically shown the effectiveness of our compilation approach in producing fast executables and quantify its strengths and weaknesses on real programs running on real parallel machines.

Bibliography

[AA89] Z. Ariola and Arvind. P-TAC: A parallel intermediate language. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, pages 230–242, September 1989.

[ACI$^+$83] Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali, and R. E. Thomas. The Tagged Token Dataflow Architecture. Technical Report FLA memo, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, August 1983.

[AE88] Arvind and K. Ekanadham. Future Scientific Programming on Parallel Machines. *Journal of Parallel and Distributed Computing*, 5(5):460–493, October 1988.

[AHN88] Arvind, S. K. Heller, and R. S. Nikhil. Programming Generality and Parallel Computers. In *Proc. of the Fourth Int. Symp. on Biological and Artificial Intelligence Systems*, pages 255–286. ESCOM (Leider), Trento, Italy, September 1988.

[AI87] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proc. of DFVLR - Conf. 1987 on Par. Proc. in Science and Eng.*, Bonn-Bad Godesberg, W. Germany, June 1987.

[AN90] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[ANP87] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. Technical Report CSG Memo 269, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, February 1987. (Also in *Proc. of the Graph Reduction Workshop*, Santa Fe, NM. October 1986.).

[App92] A. W. Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Pub. Co., Reading, Mass., 1986.

[Bar61] R. S. Barton. A new approach to the functional design of a computer. In *Proc. Western Joint Computer Conf.*, pages 393–396, 1961.

[BCFO91] A.P.W. Bohm, D.C. Cann, J.T. Feo, and R.R. Oldehoeft. Sisal 2.0 reference manual. Technical Report CS-91-118, Colorado State University, November 12 1991.

[BCS+89] P. J. Burns, M. Christon, R. Schweitzer, O. M. Lubeck, H. J. Wasserman, M. L. Simmons, and D. V. Pryor. Vectorization of Monte-Carlo Particle Transport: An Architectural Study using the LANL Benchmark "Gamteb". In *Proc. Supercomputing '89*. IEEE Computer Society and ACM SIGARCH, New York, NY, November 1989.

[Bir84] R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(4):239–250, 1984.

[BP89] M. Beck and K. Pingali. From Control Flow to Dataflow. Technical Report TR 89-1050, CS Dep., Cornell University, October 1989.

[CA88] D. E. Culler and Arvind. Resource Requirements of Dataflow Programs. In *Proc. of the 15th Ann. Int. Symp. on Comp. Arch.*, pages 141–150, Hawaii, May 1988.

[CC77] P. Cuosot and R. Cuosot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. on Principles of Programming Languages, Los Angeles, CA*, January 1977.

[CGSvE92] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. Empirical Study of Dataflow Languages on the CM-5. In *Proc. of the Dataflow Workshop, 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[CGSvE93] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[CHR78] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, February 1978.

[CKP+93] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[Coo94] S. R. Coorg. Partitioning Non-strict Languages for Multi-threaded Code Generation. Master's thesis, Dept. of EECS, MIT, Cambridge, MA, May 1994.

[CPJ85] C. Clack and S. L. Peyton-Jones. Strictness analysis - a practical approach. In *Proc. Functional Programming Languages and Computer Architecture, Nancy, France, September 1985*, pages 35–49, September 1985. Springer-Verlag LNCS 201.

[CSS+91] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. of 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa-Clara, CA, April 1991. (Also available as Technical Report UCB/CSD 91/591, CS Div., University of California at Berkeley).

[CSvE93] D. E. Culler, K. E. Schauser, and T. von Eicken. Two Fundamental Limits on Dataflow Multiprocessing. In *Proceedings of the IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, Orlando, FL.* North-Holland, January 1993.

[Cul90] D. E. Culler. Managing Parallelism and Resources in Scientific Dataflow Programs. Technical Report 446, MIT Lab for Comp. Sci., March 1990. (PhD Thesis, Dept. of EECS, MIT).

[Dob87] T. Dobry. *A High Performance Architecture for Prolog*. PhD thesis, Computer Science Div., University of California at Berkeley, April 1987.

[FH88] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and its Use in Optimization . *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GDHH89] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes. The Epsilon Dataflow Processor. In *Proc. of the 16th Ann. Int. Symp. on Comp. Arch.*, Jerusalem, Israel, May 1989.

[GH90] V. G. Grafe and J. E. Hoch. The Epsilon-2 Hybrid Dataflow Architecture. In *Proc. of Compcon90*, pages 88–93, San Francisco, CA, March 1990.

[GKW85] J. Gurd, C.C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the Association for Computing Machinery*, 28(1):34–52, January 1985.

[Gol88] B. Goldberg. *Multiprocessor Execution of Functional Programs*. PhD thesis, Department of Computer Science, Yale University, 1988.

[Gol94] S. C. Goldstein. Implementation of a Threaded Abstract Machine on Sequential and Multiprocessors. Master's thesis, Computer Science Division — EECS, U.C. Berkeley, 1994. (to appear as UCB/CSD Technical Report).

[HB85] P. Hudak and A. Bloss. The Aggregate Update Problem in Functional Programming Systems. In *ACM Symposium on the Principles of Programming Languages*, pages 300–314, January 1985.

[HCAA93] J. Hicks, D. Chiou, B. S. Ang, and Arvind. Performance Studies of Id on the Monsoon Dataflow System. *Journal of Parallel and Distributed Computing*, 18:273–300, July 1993.

[HDGS93] J. E. Hoch, D. M. Davenport, V. G. Grafe, and K. M. Steele. Compile-time Partitioning of a Non-strict Language into Sequential Threads. In *Proc. 3rd Symp. on Parallel and Distributed Processing*, December 1993.

[Hel89] S. K. Heller. *Efficient Lazy Data-Structures on a Dataflow Machine*. PhD thesis, Dept. of EECS, MIT, February 1989.

[Hen80] P. Henderson. *Functional Programming: Application and Implementation*. Prentice-Hall International, 1980.

[HKT92] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[Hug88] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):187–208, 1988.

[Hut92] G. Hutton. Higher-order Functions for Parsing. *Journal of Functional Programming*, 3(2), July 1992.

[Ian88] R. A. Iannucci. A Dataflow/von Neumann Hybrid Architecture. Technical Report TR-418, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, May 1988. (PhD Thesis, Dept. of EECS, MIT).

[Joh87a] T. Johnsson. Attribute grammars as a functional programming paradigm. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, (Springer-Verlag LNCS 274).*, February 1987.

[Joh87b] T. Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Goteborg, Sweden, 1987.

[Kra88] D. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Department of Computer Science, Yale University, 1988.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th Annual Int. Symp. on Comp. Arch.*, pages 148–159, Seattle, Washington, May 1990.

[MF93] S. Murer and P. Färber. Code Generation for Multi-threaded Architectures from Dataflow Graphs. In *Architecture and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 77–90. North-Holland, January 1993.

[Moo85] D. A. Moon. Architecture of the symbolics 3600. In *Proc. 12th IEEE Int'l. Symposium on Computer Architecture*, April 1985.

[NA89] R. S. Nikhil and Arvind. Can Dataflow Subsume von Neumann Computing? In *Proc. of the 16th Annual Int. Symp. on Comp. Arch.*, Jerusalem, Israel, May 1989.

[Nik91] R. S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. In *Proc. Workshop on Massive Parallelism, Amalfi, Italy, October 1989*. Academic Press, 1991.

[Nik93] R. S. Nikhil. A Multithreaded Implementation of Id using P-RISC Graphs. In *Proc. Sixth Ann. Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.

[NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *t: A multithreaded massively parallel architecture. In *Proc. 19th. Annual Intl. Symp. on Computer Architecture, Queensland, Australia*, pages 156–167, May 1992.

[PA85] K. Pingali and Arvind. Efficient demand-driven evaluation. part 1. *ACM TOPLAS*, 7(2):311–333, April 1985.

[PC90] G. M. Papadopoulos and D. E. Culler. Monsoon: an Explicit Token-Store Architecture. In *Proc. of the 17th Annual Int. Symp. on Comp. Arch.*, Seattle, Washington, May 1990.

[Pey92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *J. Functional Programming*, April 1992.

[PT91] G. M. Papadopoulos and K. R. Traub. Multithreading: A Revisionist View of Dataflow Architectures. In *Proc. of the 18th Int. Symp. on Comp. Arch.*, pages 342–351, Toronto, Canada, May 1991.

[PvE93] R. Plasmeijer and M van Eekelen. *Functional programming and parallel graph rewriting*. Addison-Wesley, 1993.

[RP94] A. Rogers and K. Pingali. Compiling for Distributed Memory Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):281–298, March 1994.

[Rug87] C. A. Ruggiero. *Throttle Mechanisms for the Manchester Dataflow Machine*. PhD thesis, University of Manchester, Manchester M13 9PL, England, July 1987.

[Sch91] K. E. Schauser. Compiling Dataflow into Threads. Technical report, Computer Science Div., University of California, Berkeley CA 94720, 1991. (MS Thesis, Dept. of EECS, UCB).

[SCvE91] K. E. Schauser, D. Culler, and T. von Eicken. Compiler-controlled Multithreading for Lenient Parallel Languages. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, Cambridge, MA, August 1991.

[SG85] S. Skedzielewski and J. Glauert. If1: An intermediate form for applicative languages. Technical Report M170, Lawrence Livermore National Laboratory, Livermore, California, July 31 1985.

[SGS+93] E. Spertus, S. C. Goldstein, K. E. Schauser, T. von Eicken, D. E. Culler, and W. J. Dally. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. In *Proc. of the 20th Int'l Symposium on Computer Architecture*, San Diego, CA, May 1993.

[SH86] V. Sarkar and J. Hennessy. Compile-time Partitioning and Scheduling of Parallel Programs. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, pages 117–26, June 1986.

[Spe92] E. Spertus. Execution of Dataflow Programs on General-Purpose Hardware. Master's thesis, Department of EECS, Massachusetts Institute of Technology, August 1992.

[SYH+89] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. of the 16th Annual Int. Symp. on Comp. Arch.*, pages 46–53, Jerusalem, Israel, June 1989.

[TCS92] K. R. Traub, D. E. Culler, and K. E. Schauser. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads. In *Proc. of the ACM Conf. on LISP and Functional Programming*, pages 324–334, San Francisco, CA, June 1992.

[Tra86] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report TR-370, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, August 1986. (MS Thesis, Dept. of EECS, MIT).

[Tra88] K. R. Traub. Sequential Implementation of Lenient Programming Languages. Technical Report TR-417, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, September 1988. (PhD Thesis, Dept. of EECS, MIT).

[Tra89] Kenneth R. Traub. Compilation as partitioning: A new approach to compiling non-strict functional languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, London*, pages 75–88, September 1989.

[Tra91] K. R. Traub. *Implementation of Non-strict Functional Programming Languages*. MIT Press, 1991.

[TS88] M. R. Thistle and B. J. Smith. A Processor Architecture for Horizon. In *Proc. of Supercomputing '88*, pages 35–41, Orlando, FL, 1988.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[WZ85] M.N. Wegman and F.K. Zadeck. Constant propagation with conditional branches. In *Proc. 12th ACM Symp. on Principles of Programming Languages, New Orleans, LA*, pages 291–299, January 1985.

Appendix A

Partitioning: The Theory

This chapter presents the theoretical formalism for proving the correctness of the basic block and interprocedural partitioning algorithms.

Our partitioning algorithm relies only on structural properties of the program and does not require information about actual execution or semantic properties. The proof of the correctness of the partitioning algorithms is held in the same spirit, making only arguments about structural properties of the program. The key to the proof is the definition of a correct partitioned program. Intuitively, a partitioning of a program is correct if it does not introduce a deadlock for all of those invocations of the program in which the unpartitioned program does not deadlock. The equivalent structural definition states that a partitioning of a program is correct if for all instances of the program[1] which can be generated structurally for which the unpartitioned dependence graph does not contain any cycles, the partitioned dependence graph is also acyclic. The set of structurally generated program instances forms a superset of all call graphs that could arise out of actual invocation from the user. In addition to avoiding circular dependencies, our definition of correct partitioning also ensures that threads do not suspend and that all long latency operations cross threads, as required by our execution model. This chapter proves that all of the partitioning algorithms fulfill the above definition

[1] An instance of a program specifies the dependence graph formed by a corresponding invocation call tree, including a set of possible feedback arcs from I-store to I-fetch nodes.

A.1 Basic Definitions

Definition 3 (Program) A *program* P is represented as a *Program Dependence Graph*, consisting of a set of basic blocks and a site connection specification which describes the possible call graph structure among these basic blocks.

$$
\begin{aligned}
P &= (\mathit{BBS}, \mathit{CON}) \\
\mathit{BBS} &= \{\mathit{BB}_1, \mathit{BB}_2, \ldots\}
\end{aligned}
$$

Each *basic block* is represented as a seven tuple, containing the vertices, dependence edges, inlet nodes, outlet nodes, I-fetch nodes, I-store nodes, the def site (S_0) and call sites $(S_1, S_2, \ldots)$. A site specifies the m input and n output nodes that form the m-input-n-output interface to other basic blocks. The nodes and the dependence edges form the *dependence graph* of the basic block. Edges connecting an outlet node with an inlet node are called certain indirect dependence edges and are represented by squiggly edges, indicating their long latency nature.

$$
\begin{aligned}
\mathit{BB} &= (V, E, \mathit{IN}, \mathit{OUT}, \mathit{FE}, \mathit{ST}, \mathit{SITES}) \\
E &\subseteq V \times V \\
\mathit{IN} &\subseteq V \\
\mathit{OUT} &\subseteq V \\
\mathit{FE} &\subseteq \mathit{IN} \\
\mathit{ST} &\subseteq \mathit{OUT} \\
\mathit{SITES} &= (S_0, \ldots, S_k) \\
S_i &= ((r_1, \ldots, r_m), (s_1, \ldots, s_n)), r_j \in \mathit{IN}, s_j \in \mathit{OUT} \\
G_{\mathit{BB}} &= (V, E) \\
E_q &= \{(s, r) \in E | s \in \mathit{OUT}, r \in \mathit{IN}\}
\end{aligned}
$$

The *site connection specification* defines the possible call graph structure, *i.e.*, the possible connectivity among sites of the basic blocks. For every call site it specifies the set of def sites it may be connected to. In the case of conditionals or procedure calls through function variables, a call site may be connected to multiple def sites. In addition, the site annotation specification lists all basic blocks that may be invoked by a user as a top-level function.

The arity of connected sites must be consistent, *i.e.*, an m-input-n-output call site is only allowed to be connected to n-input-m-output def sites. The only exception is the top-level site which may call functions of any arity.

$$\begin{aligned} CON &: (BB_i, S_j) \rightarrow \{(BB_k, S_0), \ldots)\} \\ CON &: TOP \rightarrow \{(BB_k, S_0), \ldots)\} \end{aligned}$$

When clear from the context we will use the notation $u \in BB$ instead of $u \in V(BB)$ for a node in the vertex set of a basic block. Likewise we use the notation $S \in BB$ as shorthand for a site $S \in SITES(BB)$. Finally, for a graph $G = (V, E)$ and a set of edges $E' \subseteq V \times V$, we may use the notation $G' = G \cup E'$ as shorthand for the graph $G' = (V, E \cup E')$.

The site connection specification can be used to construct a *tree grammar* that can generate all structurally possible call graphs of basic blocks, which is a superset of all call graphs that could arise out of actual invocations from the user.

Definition 4 (Call Tree Grammar) The following expansive tree grammar defines the call tree grammar of a program $P = (BBI, CON)$.

$$\begin{aligned} G_{CT} &= (V_T, V_N, \Pi, S) \text{ where} \\ V_T &= \{BB_i | BB_i \in P\} \\ V_N &= \{S\} \cup \{S_i^j | BB_i \in P \text{ and } S_j \in BB_i\} \\ \Pi &= \{S \rightarrow \begin{array}{c} BB_l \\ \diagup\ \diagdown \\ S_l^1 \ldots S_l^k \end{array} | (BB_l, S_0) \in CON(TOP), SITES(BB_l) = (S_0, \ldots, S_k)\} \cup \\ & \quad\ \{S_i^j \rightarrow \begin{array}{c} BB_l \\ \diagup\ \diagdown \\ S_l^1 \ldots S_l^k \end{array} | (BB_l, S_0) \in CON(BB_i, S_j), SITES(BB_l) = (S_0, \ldots, S_k)\} \end{aligned}$$

Definition 5 (Program Instance) An *instance IP* of a program P is a set of basic block instances forming a call tree and a set of dependence edges from I-store to I-fetch nodes.

$$\begin{aligned} IP &= (BBI, SF) \\ BBI &= (BB_k^{(1)}, \ldots, BB_m^{(n)}) \\ SF &\subseteq \{(s, BB_i^{(j)}), (r, BB_k^{(l)}) | BB_i^{(j)}, BB_k^{(l)} \in BBI, s \in ST(BB_i), r \in FE(BB_k)\}) \\ CT &= \{(j, BB_i) | BB_i^{(j)} \in BBI\} \end{aligned}$$

A program instance has a corresponding call tree *CT* that can be generated by the call tree grammar G_{CT}. $BB_i^{(j)}$ is called an instance of the basic block BB_i, and the superscript (j) denotes its position in the call tree. A basic block may have multiple instances. The store/fetch dependence set is a set of dependence edges from I-store nodes to I-fetch nodes. Because the basic block instances form a call tree, each call site connects to precisely one def site of another basic block, which is consistent with the site connection specification of the program. Only one basic block def site (the root of the tree) is not connected, corresponding to the top-level function being invoked by the user. We will use the notation $BB_i^{(j)} \in IP$ as a shorthand for $BB_i^{(j)} \in BBI(IP)$.

Definition 6 (Dependence Graph of a Program Instance) The *dependence graph* of a program instance contains, for every instance of a basic block, all of its nodes and dependence edges (straight and squiggly). In addition it also contains the edges connecting the nodes of a call sites to the corresponding nodes in the def site, and the store/fetch dependencies.

$$
\begin{aligned}
G_{IP} &= (V_{IP}, D_{IP}) \\
V_{IP} &= \{(u, BB_i^{(j)}) | BB_i^{(j)} \in IP, u \in BB_i\} \\
D_{IP} &= \{((u, BB_i^{(j)}), (v, BB_i^{(j)})) | BB_i^{(j)} \in IP, (u, v) \in E(BB_i)\} \cup SF \cup \\
&\quad \{((s_1, BB_i^{(j)}), (r_1', BB_k^{(l)})), \ldots, ((s_1', BB_k^{(l)}), (r_1, BB_i^{(j)})), \ldots | \\
&\quad \text{for which } l \text{ is the } m\text{-th successor of } j \text{ in the tree domain, } (j.m) = (l), \\
&\quad \text{and } S_m(BB_i) = ((r_1, \ldots), (s_1, \ldots)), S_0(BB_k) = ((r_1', \ldots), (s_1', \ldots))\}
\end{aligned}
$$

Definition 7 (Admissible Program Instance) A program instance *IP* is called *admissible* if the dependence graph G_{IP} is acyclic.

Note, that in G_{IP} the top-level def site is not connected. This is correct under the assumption that a user can call a top-level function only in a *strict* fashion. The user must first provide all arguments to the top-level function before receiving any results. Thus the user cannot introduce any new dependencies not already present in the dependence graph. We can also contemplate other *user interfaces*, specifying other ways to invoke a function.

Definition 8 (User Interface) A *user interface* describes the way in which top-level functions can be invoked. Possible user interfaces include:

```
def main x = fib n;
def fib n = if (n<2) then 1
                     else (fib (n-1) + fib (n-2));
```

4 basic blocks: BB_1 (main), BB_2 (fib), BB_3 (fib-then), BB_4 (fib-else)

$G_{CT} = (V_T, V_N, \Pi, S)$

$V_T = (BB_1, BB_2, BB_3, BB_4)$

$V_N = (S, S_1^1, S_2^1, S_4^1, S_4^2)$

$\Pi = \{S \rightarrow BB_1, S_1^1 \rightarrow BB_2, S_2^1 \rightarrow BB_3, S_2^1 \rightarrow BB_4, S_4^1 \rightarrow BB_2, S_4^2 \rightarrow BB_2\}$

$S_1^1 \quad S_2^1 \quad S_4^1 \; S_4^1 \quad S_2^1 \quad S_2^1$

(main 2)

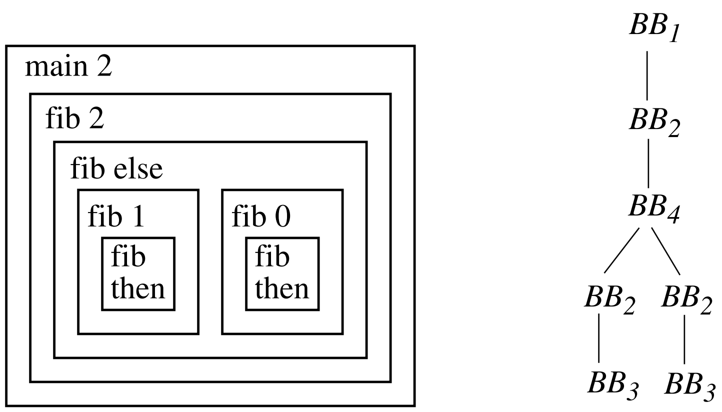


Figure A.1: *Example showing the use of the call tree grammar. This example presents two functions* `fib` *and* `main`*. These functions result in 4 basic blocks. Only the basic block corresponding to* `main` *can be invoked from the top-level. The example shows the corresponding call tree grammar and the basic block call tree that is generated for the call* `(main 2)`.

1. *Strict:* all of the arguments must be provided before any result can be used.

2. *Scalar non-strict:* some scalar results may be read before providing all of the arguments.

3. *Data structure non-strict (read or write):* elements of non-strict data structures may be read (or written into) before providing all of the arguments.

In a non-strict user interface the user would be allowed to read some scalar results, and feed them back in as arguments. To represent this correctly in the dependence graph of the program instances, we must allow for all possible feedback arcs from results to arguments that do not result in circular dependencies. We can express this by extending the store/fetch dependencies with these feedback arcs. Other even more complex calling interfaces allow the user to read elements of non-strict data structures returned, or even store some values into them. Again, these have to be captured by extending the store/fetch dependencies with the appropriate feedback edges.

Any function the user can invoke without causing deadlock has a corresponding admissible (and thereby acyclic) program instance that reflects the call tree structure and dependencies from I-store to I-fetch nodes. Admissible program instances form the foundation for the definition of correct partitioning of a program, given below. To verify that a partitioning is correct it is sufficient to check whether it introduces circular dependencies only for the admissible program instances.

A.2 Partitioning

Definition 9 (Partitioning) A *partitioning* of a basic block *BB* is a set $\Theta_{BB} = \{\theta_1, \theta_2, \ldots\}$ of subsets of the nodes $V(BB)$ such that the θ_i are pairwise disjoint, and their union is equal to $V(BB)$. A partitioning of a program P is a set Θ, which is the union of a partitioning of all basic blocks.

For a partitioning Θ and a node $v \in BB$, $\Theta(v)$ denotes the partition that contains v. The *dependence graph for a basic block under a partitioning* Θ is the reduced graph $G_{BB}{\downarrow}\Theta$ which contains a node for every partition of the basic block and the edges that cross these. For a program instance *IP* and a node $(v, BB^{(j)}) \in IP$, the expression $\Theta((v, BB^{(j)})) = (\Theta(v), BB^{(j)})$

denotes an instance of the corresponding partition. The *dependence graph for a program instance under a partitioning* Θ is the reduced graph $G_{IP}{\downarrow}\Theta$ which contains a node for every instance of a partition and the edges crossing these.

Definition 10 (Scheduled Partitioning) A *scheduled partitioning* of a program is a pair (Θ, S) of a partitioning and a schedule for each partition $\theta_i \in \Theta$ such that $S(\theta_i)$ is a linear ordering of the elements of θ_i. $S(\theta_i) = (v_{i,1}, \ldots, v_{i,|\theta_i|})$. A scheduled partition is also called a thread.

A scheduled partitioning captures the static instruction ordering for each thread.

Definition 11 (Imputed Dependencies) The *imputed dependence* edges D_S for a scheduled partitioning (Θ, S) represent the additional dependencies that the linear ordering of the schedule introduces:

$$D_S = \bigcup_{1 \le i \le |\Theta|} D_S(\theta_i) \text{ where } D_S(\theta_i) = \bigcup_{1 \le j \le |\theta_i|-1} \{(v_{i,j}, v_{i,j+1})\}$$

The *imputed dependence* edges of a program instance *IP* for a scheduled partitioning (Θ, S) are the additional dependencies introduced by the linear ordering of the schedules:

$$D_{(IP,S)} = \{((u, BB_i^{(j)}), (v, BB_i^{(j)}))|BB_i^{(j)} \in IP, (u, v) \in D_S\}$$

The *dependence graph for a program instance under a scheduled partitioning* adds the imputed dependence edges to the dependence graph of the program instance

$$G_{(IP,S)} = G_{IP} \cup D_{(IP,S)}$$

Definition 12 (Correct Scheduled Partitioning) A scheduled partitioning (Θ, S) of a program P is correct, if for all admissible program instances *IP*

1. the graph $G_{(IP,S)}$ is acyclic

2. the reduced graph $G_{(IP,S)}{\downarrow}\Theta$ is acyclic

3. $\Theta(s) \neq \Theta(r)$ for all $BB_i^{(j)} \in P$ and all $(s, r) \in E_q(BB_i)$

4. $\Theta((s, BB_i^{(j)})) \neq \Theta((r, BB_k^{(l)}))$ for all $((s, BB_i^{(j)}), (r, BB_k^{(l)})) \in SF$

This definition captures all of the scheduling requirements imposed by the non-strictness of the language, long latency of remote operations, and the threaded execution model. Under the threaded execution model, a correct scheduled partitioning has to avoid creating any circular dependencies that would result in a deadlock. It also has to ensure that partitions do not suspend or wait for long latency operations. This is being captured by the above four conditions. The first condition ensures that the linear thread schedule does not create a circular dependence within a partition. The second condition ensures that the partitioning does not introduce any circular dependence among threads. Finally, the third and fourth condition ensure that threads do not have to suspend or wait for long latency operation. All potentially long latency operations, as indicated by squiggly arcs or store fetch dependencies, have to cross partitions.

Note, that although this definition is talking about all admissible program instances, our proof of a partitioning being correct will not have to compute all of these. We will rather show that our initial partitioning satisfies these conditions for all admissible program instances, and that subsequent re-partitioning maintains this invariant. Only the first condition actually uses the partitioning schedule. Because the imputed dependencies are completely within partitions, the second condition is identical to the reduced graph $G_{IP}{\downarrow}\Theta$ being acyclic. We now define a correct partitioning as one that satisfies the last three conditions.

Definition 13 (Correct Partitioning) A partitioning Θ of a program P is correct, if for all admissible instances *IP* of the program

1. the reduced graph $G_{IP}{\downarrow}\Theta$ is acyclic
2. $\Theta(s) \neq \Theta(r)$ for all $BB_i^{(j)} \in P$ and all $(s, r) \in E_q(BB_i)$
3. $\Theta((s, BB_i^{(j)})) \neq \Theta((r, BB_k^{(l)}))$ for all $((s, BB_i^{(j)}), (r, BB_k^{(l)})) \in SF$

The next proposition shows that for every correct partitioning we can find a schedule which results in a correct scheduled partitioning. This allows us to first derive the partitioning independently of the thread schedule. The schedule can be fixed at a later stage of the compiler. This is desirable, because for different machine architectures (*e.g.*, pipelined, super-scalar) good instruction schedules may differ. Thus we just need to show that our partitioning algorithms satisfy the simpler definition of correct partitioning.

Proposition 1 (Schedule Independence) For every correct partitioning Θ there exists a schedule S such that (Θ, S) is a correct scheduled partitioning.

Proof: Let Θ be a correct partitioning of the program. We start by constructing a schedule $S(\theta_i)$ for each partition $\theta_i \in \Theta$. By the definition of partitioning, individual partitions don't cross basic blocks. For a $\theta_i \in \Theta$, let BB be the basic block that contains all of the nodes of θ_i. Let $G_{BB}|\theta_i$ denote the restriction of the graph G_{BB} to the subset of nodes $\theta_i \subseteq V(BB)$, as $G_{BB}|\theta_i = (\theta_i, E \cap (\theta_i \times \theta_i))$. If the restricted graph $G_{BB}|\theta_i$ is acyclic, then define $S(\theta_i)$ as any topological ordering of $G_{BB}|\theta_i$, which must exist because $G_{BB}|\theta_i$ is acyclic. Otherwise define $S(\theta_i)$ to be any ordering of the nodes θ_i. This completely defines the schedule S, we now have to show that (Θ, S) forms a correct scheduled partition, *i.e.*, that (Θ, S) satisfies the four conditions of a correct scheduled partitioning.

Since Θ is a correct partitioning, the conditions c) and d) of a correct scheduled partitioning are trivially satisfied. Condition b) is also satisfied, because imputed dependence edges do not cross partition boundaries the two reduced graphs $G_{IP}{\downarrow}\Theta$ and $G_{(IP,S)}{\downarrow}\Theta$ are equal. The only condition remaining to be proved is a) which states that $G_{(IP,S)}$ must be acyclic for all admissible program instances IP. Let IP be an admissible program instance and assume that $G_{(IP,S)}$ contains a cycle. We can distinguish two cases:

1) The cycle contains nodes that belong to different partitions. Then obviously the reduced graph $G_{(IP,S)}{\downarrow}\Theta$ also contains a cycle. Because the schedule only adds imputed dependence edges completely within partitions and never across partitions, the reduced graph $G_{IP}{\downarrow}\Theta$ must also contain a cycle. This violates condition a) of the definition of correct partitioning and therefore contradicts the assumption that Θ is a correct partitioning.

2) The cycle is completely contained within a partition $\theta_i \in \Theta$. Let BB be the corresponding basic block that contains the nodes of θ_i. If $G_{BB}|\theta_i$ is acyclic, then $S(\theta_i)$ is defined as a topological ordering of the nodes in $G_{BB}|\theta_i$, and therefore could not have introduced any circular dependencies. Thus from our construction of $S(\theta_i)$ it follows

that $G_{BB}|\theta_i$ must contain a cycle. But then G_{IP} must also have a cycle since it contains the subgraph $G_{BB}|\theta_i$. This is a contradiction to the assumption that IP is admissible. □

Definition 14 (Trivial Partitioning) In the trivial partitioning Θ_0 of a program, every node forms its own partition.

Lemma 1 (Trivial Partitioning is Correct) The trivial partitioning Θ_0 is correct.

Proof: Let *IP* be an admissible program instance. We show that the three conditions of correct partitioning are satisfied for the trivial partition Θ_0.

a) In Θ_0 every node forms its own partition. Therefore $G_{IP}\downarrow\Theta_0 = G_{IP}$, which is acyclic.

b) Since $\Theta_0(u) \neq \Theta_0(v)$ if $u \neq v$, all edges, including all squiggly edges, trivially cross partitions.

c) The same also holds trivially for all store/fetch dependencies. □

Although the trivial partitioning is correct, it is certainly not very good and leads to a high cost of dynamic scheduling at run-time. Our interprocedural algorithm starts with the trivial partitioning to obtain a correct partitioning. This is then iteratively improved by applying three steps: (i) first the current partitioning of each basic block is summarized in form of annotations that essentially describe the relationship between outputs and inputs; (ii) then these annotations are refined by iteratively propagating annotations across basic block boundaries; (iii) finally a basic block is repartitioned using these annotations. We have to ensure that our analysis techniques for deriving the annotations and our strategies for repartitioning work together and do not lead to inconsistencies in form of circular dependencies.

Two nodes of a basic block cannot be placed into the same partition, if there may exist an indirect dependence from one to the other. We are going to use annotations to capture possible indirect dependencies from outlet nodes to inlet nodes.

A.3 Annotation Based Repartitioning

Two forms of indirect dependencies from outlet nodes to inlet nodes can be distinguished: *certain indirect dependencies* which are known to be present for all invocations of the program and *potential indirect dependencies* which may exist only for some program instances. Annotations are used to represent these indirect dependencies. In order to be correct, annotations have to overestimate the potential indirect dependencies and underestimate the certain indirect dependencies. In fact, as the example in Section 5.6 showed, potential indirect dependencies must represent more than the actual dependencies that may be present in the program to be useful for repartitioning a basic block. Certain dependencies, both indirect and direct, are valuable because they rule out potential dependencies, thereby improving the partitioning.

Definition 15 (Annotation) An *annotation* for a collection of inlet and outlet nodes, $IN = \{r_1, \ldots, r_n\}$, $OUT = \{s_1, \ldots, s_m\}$, is a 5 tuple $A = (\Sigma_i, Inlet, \Sigma_o, Outlet, CID)$, where Σ_i is the inlet alphabet, $Inlet : IN \to Pow(\Sigma_i)$ maps each inlet node to a set of inlet names, Σ_o the outlet alphabet, $Outlet : OUT \to Pow(\Sigma_o)$ maps each outlet node to a set of outlet names, and $CID \subseteq (OUT \times IN)$ are the *certain indirect dependencies* from outlet nodes to inlet nodes.

An annotation for a call site or def site is an annotation for all of the inlet and outlet nodes comprising that site. An annotation for a basic block is annotation for the collection of all of its inlet and outlet nodes, including all call and def site nodes, as well at all store and fetch response nodes. The CID edges for a basic block annotation also contain the initial set of squiggly edges ($E_q \subseteq CID$).

These annotations are used to derive the admissible potential indirect dependencies.

Definition 16 (Potential Indirect Dependence Set) A *potential dependence set DS* for an annotation A of a basic block is a subset of the cross product of outlet and inlet names. A *potential indirect dependence set* for a potential dependence set contains for all inlet/outlet name pairs an edge from every outlet node with this outlet name to every inlet node with the corresponding inlet name. $PIDS$ is the set of all potential indirect dependence sets for a basic block under an annotation A.

$$\begin{aligned} PIDS(A) &= \{PID(DS) | DS \subseteq \Sigma_o \times \Sigma_i\} \textit{ where} \\ PID(DS) &= \{(s, r) | \exists o \in Outlet(s), \exists i \in Inlet(r) \text{ such that } (o, i) \in DS\} \end{aligned}$$

When obvious from the context we will omit A and just use the notation $PIDS$ and CID.

Definition 17 (Admissible Potential Indirect Dependence Set) A potential indirect dependence set $PID \in PIDS(A)$ for a basic block annotation A is called *admissible* under a partitioning Θ if

1. the reduced graph $(G_{BB} \cup CID \cup PID){\downarrow}\Theta$ is acyclic, and
2. $\Theta(s) \neq \Theta(r)$ for all $(s,r) \in PID$

Most of the potential indirect dependence sets in $PIDS(A)$ will not be admissible. A potential indirect dependence set that is not admissible contains some cyclic dependence and therefore never can be part of an admissible program instance. Thus, when partitioning a basic block with annotations, it is sufficient to check only for the admissible potential indirect dependence sets whether the partitioning introduces some cycle or not.

Definition 18 (Repartitioning) For a partitioning Θ of the program P, a repartitioning Θ'_{BB} of a basic block BB is a partitioning of the nodes $V(BB)$ such that only larger partitions are formed, *i.e.*, for all $u, v \in BB$ if $\Theta(u) = \Theta(v)$ then $\Theta'_{BB}(u) = \Theta'_{BB}(v)$. The resulting new partitioning Θ' of the program is defined as

$$\Theta'(v) = \begin{cases} \Theta'_{BB}(v) & \text{if } v \in BB \\ \Theta(v) & \text{otherwise} \end{cases}$$

The next lemma specifies the properties a repartitioning of a basic block must have to maintain the correctness of a correctly partitioned program.

Definition 19 (Legal Repartitioning) A repartitioning Θ'_{BB} of a basic block is *legal* under a correct partitioning Θ and an annotation A if for all admissible $PID \in PIDS(A)$ the following three conditions are satisfied

1. the reduced graph $(G_{BB} \cup CID \cup PID){\downarrow}\Theta'_{BB}$ is acyclic
2. $\Theta'_{BB}(s) \neq \Theta'_{BB}(r)$ for all $(s,r) \in PID$
3. $\Theta'_{BB}(s) \neq \Theta'_{BB}(r)$ for all $(s,r) \in CID$.

The first condition ensures that the repartitioning does not introduce any new circular dependencies. The second and third condition ensure that all potentially indirect dependencies and squiggly edges cross partitions.

A.4 Basic Block Partitioning Algorithm

Now we will prove that the basic block partitioning algorithms are correct. Before showing the correctness of dependence and demand set partitioning with subpartitioning, iterated partitioning, and separation constraint partitioning we will prove two auxiliary propositions which facilitate later proofs.

Proposition 2 (PID transitivity) Let Θ be a correct partitioning. Let $PID \in PIDS(A)$ be an admissible potential indirect dependence set under the partitioning Θ such that $(s_1, r_1) \in PID$ and $(s_2, r_2) \in PID$. If there exists a path from $\Theta(r_1)$ to $\Theta(s_2)$ in $(G_{BB} \cup CID \cup PID)\downarrow\Theta$ then there also exists an admissible $PID' \supseteq PID$ such that $(s_1, r_2) \in PID'$.

> **Proof:** Let Θ be a correct partitioning. Let DS be a potential dependence set such that $PID = PID(DS) \in PIDS(A)$ is an admissible potential indirect dependence set under the partitioning Θ with $(s_1, r_1) \in PID$ and $(s_2, r_2) \in PID$. This implies that there exists an $(o_1, i_1) \in DS$, $i_1 \in Inlet(r_1)$ and $o_1 \in Outlet(s_1)$ such that for all r'_1, s'_1 with $i_1 \in Inlet(r'_1)$ and $o_1 \in Outlet(s'_1)$ there does not exist a path from $\Theta(r'_1)$ to $\Theta(s'_1)$ in $(G_{BB} \cup CID \cup PID)\downarrow\Theta$. Likewise we know that there exists an $(o_2, i_2) \in DS$, $i_2 \in Inlet(r_2)$ and $o_2 \in Outlet(s_2)$ such that for all r'_2, s'_2 with $i_2 \in Inlet(r'_2)$ and $o_2 \in Outlet(s'_2)$ there does not exist a path from $\Theta(r'_2)$ to $\Theta(s'_2)$ in $(G_{BB} \cup CID \cup PID)\downarrow\Theta$.
>
> Choose DS' to be $DS \cup \{(o_1, i_2)\}$ and $PID' = PID(DS')$. Then certainly $(s_1, r_2) \in PID'$ and $PID \subseteq PID'$. We now prove that PID' is admissible, *i.e.*, that $(G_{BB} \cup CID \cup PID')\downarrow\Theta$ is acyclic. We know that $(G_{BB} \cup CID \cup PID)\downarrow\Theta$ is acyclic because PID is admissible. For all nodes s'_1, r'_2 for which $o_1 \in Outlet(s'_1)$ and $i_2 \in Inlet(r'_2)$ we know that $(s'_1, r_1) \in PID$ and $(s_2, r'_2) \in PID$. Since there exists a path from $\Theta(r_1)$ to $\Theta(s_2)$ in $(G_{BB} \cup CID \cup PID)\downarrow\Theta$. We know that adding the edges s'_1, r'_2 to PID will not create any new cycle. These are exactly the edges which have to be added for PID', *i.e.*, $PID' = PID \cup \{(s'_1, r'_2) | o_1 \in Outlet(s'_1), i_2 \in Inlet(r'_2)\}$. Therefore $(G_{BB} \cup CID \cup PID')\downarrow\Theta$ is acyclic and PID' is admissible. $\square$

Proposition 3 (PID arcs) Let Θ be a correct partitioning. If there exists an admissible $PID \in PIDS(A)$ under the partitioning Θ such that $(s_1, r_1) \in PID$ and $(s_2, r_2) \in PID$ then there also exists an admissible $PID' \supseteq PID$ such that either $(s_1, r_2) \in PID'$ or $(s_2, r_1) \in PID'$.

This may seem surprising at first glance but is due to the fact that we use inlet/outlet annotations to represent potential indirect dependence edges and assume that potential indirect dependence edges are present unless contradicted by certain dependencies.

Proof: Let Θ be a correct partitioning. Let *DS* be an potential dependence set such that $PID = PID(DS) \in PIDS(A)$ is admissible and $(s_1, r_1) \in PID$ and $(s_2, r_2) \in PID$. This implies that there exists an $(o_1, i_2) \in PID$, $i_1 \in Inlet(r_1)$ and $o_1 \in Outlet(s_1)$ such that for all r'_1, s'_1 with $i_1 \in Inlet(r'_1)$ and $o_1 \in Outlet(s'_1)$ there does not exist a path from $\Theta(r'_1)$ to $\Theta(s'_1)$ in $(G_{BB} \cup CID \cup PID){\downarrow}\Theta$. Likewise we know that there exists an $(o_2, i_2) \in DS$, $i_2 \in Inlet(r_1)$ and $o_2 \in Outlet(s_2)$ such that for all r'_2, s'_2 with $i_2 \in Inlet(r'_2)$ and $o_2 \in Outlet(s'_2)$ there does not exist a path from $\Theta(r'_2)$ to $\Theta(s'_2)$ in $(G_{BB} \cup CID \cup PID){\downarrow}\Theta$. We distinguish two cases.

1) There exists a r'_1, s'_2 with $i_1 \in Inlet(r'_1)$ and $o_2 \in Outlet(s'_2)$ such that there exists a path from $\Theta(r'_1)$ to $\Theta(s'_2)$ in $(G_{BB} \cup CID \cup PID){\downarrow}\Theta$. By definition also $(s_1, r'_1) \in PID$ and $(s'_2, r_2) \in PID$. Applying Proposition 2 we obtain that there exists an admissible PID' such that $(s_1, r_2) \in PID'$ and $PID \subseteq PID'$ which is the desired result.

2) For all r'_1, s'_2 with $i_1 \in Inlet(r'_1)$ and $o_2 \in Outlet(s'_2)$ there does not exist a path from $\Theta(r'_1)$ to $\Theta(s'_2)$ in $(G_{BB} \cup CID \cup PID){\downarrow}\Theta$. Choose DS' to be $DS \cup \{(o_2, i_1)\}$ and $PID' = PID(DS')$. Then certainly $(s_2, r_1) \in PID'$ and $PID \subseteq PID'$. We now prove that PID' is admissible, *i.e.*, that $(G_{BB} \cup CID \cup PID'){\downarrow}\Theta$ is acyclic. The new potential indirect dependence set is $PID' = PID \cup \{(s'_2, r'_1) | o_2 \in Outlet(s'_2), i_1 \in Inlet(r'_1)\}$. We know that $(G_{BB} \cup CID \cup PID){\downarrow}\Theta$ is acyclic because *PID* is admissible. Adding all of the edges s'_1, r'_2 to *PID* will not create any new cycle because the reverse path does not exist. Thus $(G_{BB} \cup CID \cup PID'){\downarrow}\Theta$ has to be acyclic and PID' is admissible. □

In the next proposition we show that if repartitioning a dataflow graph results in a cycle for a reduced graph with an admissible potential indirect dependence set, then we can always an admissible potential indirect dependence set and cycle involving at most one potential indirect dependence edge.

Proposition 4 (Single PID Arc) Let Θ be a correct partitioning of the program, and let Θ'_{BB} be a repartitioning of the basic block *BB*. If there exists an admissible $PID \in PIDS(A)$ under Θ and a potential indirect dependence edge $(s, r) \in PID$ such that there exists a path from $\Theta'_{BB}(r)$ to $\Theta'_{BB}(s)$ in $(G_{BB} \cup CID \cup PID){\downarrow}\Theta'_{BB}$, then there exists an admissible $PID' \in PIDS(A)$ and a potential indirect dependence edge $(s', r') \in PID'$ such that there exists a path from $\Theta'_{BB}(r')$ to $\Theta'_{BB}(s')$ in $(G_{BB} \cup CID){\downarrow}\Theta'_{BB}$.

Proof: (By induction over the number of potential indirect dependence edges along the cycle.) Let Θ'_{BB} be a repartitioning of the basic block *BB*. Let $PID \in PIDS(A)$ be an admissible potential indirect dependence set such that the reduced graph $(G_{BB} \cup CID \cup PID) \downarrow \Theta'_{BB}$ contains a cycle. Let i denote the number of potential indirect dependence edges along the cycle.

Base case ($i = 0$, or $i = 1$): here the proposition is trivially fulfilled.

Induction step: Assume the cycle goes over $i + 1$ potential indirect dependence edges. Let (s_1, r_1) and (s_2, r_2) be two potential indirect dependence edges which are adjacent in the cycle, *i.e.*, there exists a path from $\Theta'_{BB}(r_1)$ to $\Theta'_{BB}(s_2)$ in $(G_{BB} \cup CID) \downarrow \Theta'_{BB}$.

From Proposition 3 we obtain that there exists an admissible $PID' \supseteq PID$ such that either $(s_1, r_2) \in PID'$ or $(s_2, r_1) \in PID'$.

If $(s_1, r_2) \in PID'$ we obtain a cycle with precisely one potential edge less than the original cycle because $PID \subseteq PID'$. Applying the induction hypothesis gives us the desired result.

If $(s_2, r_1) \in PID'$ then we directly obtain a cycle with only one potential indirect dependence edge, since there exists a path from $\Theta'_{BB}(r_1)$ to $\Theta'_{BB}(s_2)$ in $(G_{BB} \cup CID) \downarrow \Theta'_{BB}$.
□

Lemma 2 (Correctness of Dependence Set Partitioning) Dependence set partitioning with subpartitioning is a legal repartitioning.

Proof: We will first show that dependence set partitioning alone satisfies both part 1) and 2) of the definition of a legal repartitioning. We will then show that subpartitioning preserves 1) and 2), and additionally satisfies part 3).

1) (By contradiction.) Assume that the reduced graph $(G_{BB} \cup CID \cup PID) \downarrow \Theta'_{BB}$ contains a cycle. Because *PID* is admissible and Θ is a correct partitioning, we know that $(G_{BB} \cup CID \cup PID) \downarrow \Theta$ is acyclic. We can distinguish two cases.

1a) The cycle is present in $(G_{BB} \cup CID) \downarrow \Theta'_{BB}$. We can then identify nodes u, v such that $\Theta'_{BB}(u) \neq \Theta'_{BB}(v)$ and there exists a path from $\Theta'_{BB}(u)$ to $\Theta'_{BB}(v)$ and a path from $\Theta'_{BB}(v)$ to $\Theta'_{BB}(u)$. Since there exists a path from $\Theta'_{BB}(u)$ to $\Theta'_{BB}(v)$ we know from the definition of dependence set partitioning that $Dep(u) \subseteq Dep(v)$. Likewise, the path from $\Theta'_{BB}(v)$ to $\Theta'_{BB}(u)$ tells us that $Dep(v) \subseteq Dep(u)$. Therefore $Dep(u) = Dep(v)$ which is a contradiction to $\Theta'_{BB}(u) \neq \Theta'_{BB}(v)$.

1b) The cycle contains some potential indirect dependence edges. From Proposition 4 we know that we can find an admissible $PID \in PIDS(A)$ and a cycle which contains a single potential indirect dependence edge. Let (s,r) be this potential indirect dependence edge, *i.e.*, there exists an $o \in Outlet(s), i \in Inlet(r)$ such that $PID(\{o,i\}) \subseteq PID$, $\Theta'_{BB}(u) \neq \Theta'_{BB}(v)$ and there exists a path $\Theta'_{BB}(r)$ to $\Theta'_{BB}(s)$ in $(G_{BB} \cup CID){\downarrow}\Theta'_{BB}$. The path implies that $Dep(r) \subseteq Dep(s)$. Therefore $i \in Dep(s)$ and there exists an node r' (possibly r) with $i \in Inlet(r')$ such that there exists a path from $\Theta(r')$ to $\Theta(s)$ in $(G_{BB} \cup CID){\downarrow}\Theta$. But $(s,r') \in PID$ thus $(G_{BB} \cup CID \cup PID){\downarrow}\Theta$ is not acyclic and PID is not admissible. This is a contradiction to above assumption.

Now we show that subpartitioning preserves property a). Assume that subpartitioning creates a cycle. Since dependence set partitioning alone is correct, we know that such a cycle can only occur completely within a partition derived by dependence set partitioning. Let u, v be two nodes along the cycle such that $\Theta'_{BB}(u) \neq \Theta'_{BB}(v)$ and $Dep(u) = Dep(v)$. From the cycle we obtain that $Subpart(u) \leq Subpart(v) \leq Subpart(u)$. This implies that $Subpart(u) = Subpart(v)$, which is a contradiction to the assumption that u and v end up in different threads.

2) We must show that $\Theta'_{BB}(s) \neq \Theta'_{BB}(r)$ for all $(s,r) \in PID$. Assume that this is not the case, *i.e.*, there exists an admissible PID such that $(s,r) \in PID$ and $\Theta'_{BB}(s) = \Theta'_{BB}(r)$. Therefore $Dep(r) = Dep(s)$, which implies analogous to the argument given above in case a2) that PID cannot be admissible. Subpartitioning only splits each partition into smaller threads, and therefore trivially preserves b).

2) We need to show that $\Theta'_{BB}(s) \neq \Theta'_{BB}(r)$ for all $(s,r) \in CID$. Subpartitioning ensures that all certain indirect dependence edges (squiggly edges) cross partitions. □

Lemma 3 (Correctness of Demand Set Partitioning) Demand set partitioning with subpartitioning is a legal repartitioning.

Proof: The proof is analogous to the correctness proof for dependence set partitioning, arguing with demand sets instead of dependence sets. □

Lemma 4 (Correctness of Iterated Partitioning) Iterated partitioning is a legal repartitioning.

Proof: Iterated partitioning iteratively applies dependence set and demand set partitioning with subpartitioning. From Lemma 2 and Lemma 3 we know that both dependence and demand set partitioning with subpartitioning are legal repartitioning. All that remains to be shown is that applying a legal repartitioning multiple times also gives a legal repartitioning. This is the case, because any $PID \in PIDS(A)$ which is admissible before repartitioning is also so after repartitioning. □

Lemma 5 (Correctness of Separation Constraint Partitioning) Separation constraint partitioning is a legal repartitioning.

Proof: Let Θ be a correct partitioning of the program and let A be a valid basic block annotation for *BB* with respect to Θ. Let Θ'_{BB} be a repartitioning for *BB* derived by separation constraint partitioning. We need to show that it satisfies the three conditions of a legal repartitioning. Let $PID \in PIDS(A)$ be an admissible potential indirect dependence set.

1) We have to show that the reduced graph $(G_{BB} \cup CID \cup PID){\downarrow}\Theta'_{BB}$ is acyclic. From Proposition 4 we know that it is sufficient to check that no cycle exists involving at most one potential indirect dependence edge.

We first show that while the algorithm runs, there does not exist a cycle which involves certain indirect dependence edges or a single potential indirect dependence edge. We then show that when the algorithm finishes, all partitions are convex, *i.e.*, do not have cycles involving only straight edges.

While the algorithm runs, there does not exist a cycle which involves certain or potential indirect dependence edges. This is certainly true at the beginning of the algorithm, since Θ is a correct partitioning and *PID* is admissible. Separation constraint partitioning first computes the set *ID* of indirect dependencies. If there exists a path from $\Theta(u)$ to $\Theta(v)$ which goes over a multiple *CID* or one *PID* edge, then the two nodes $(\Theta(u), \Theta(v))$ is in *ID*. This set of *ID* only grows when separation constraint merges nodes. The update of *ID* maintains the invariant that two partitions have an indirect dependence between them if there exists a path from one to the other which goes over multiple *CID* edges or a single *PID* edge. Therefore, the two partitions containing u and v cannot be merged by separation constraint partitioning.

Now we show that all partitions are convex when the algorithm finishes. Assume that $(G_{BB} \cup CID \cup PID){\downarrow}\Theta'_{BB}$ contains only cycles which go over straight edges. Let $\Theta'_{BB}(u)$ and $\Theta'_{BB}(v)$ be two partitions along such a cycle. Since there does not exist a *CID* nor a *PID* edge along the cycle we know that neither $(\Theta'_{BB}(u), \Theta'_{BB}(v))$ nor $(\Theta'_{BB}(v), \Theta'_{BB}(u))$ is in *ID*. Therefore separation constraint partitioning could merge the two nodes, a contradiction to the assumption that the algorithm has finished.

2) We have to show that $\Theta'_{BB}(s) \neq \Theta'_{BB}(r)$ for all $(s, r) \in$ *PID*. Let $(s, r) \in$ *PID* be a potential indirect dependence edge. Since *PID* is admissible, separation constraint partitioning will initially put $(\Theta(s), \Theta(r))$ into *ID*. Therefore subsequent steps can never merge the two partitions which contains s and r. Thus we obtain that at the end $\Theta'_{BB}(s) \neq \Theta'_{BB}(r)$.

3) We have to show that $\Theta'_{BB}(s) \neq \Theta'_{BB}(r)$ for all $(s, r) \in$ *CID*. The proof is analogous to part 2), arguing about a *CID* edge instead of a *PID* edge. □

This proof not only shows that separation constraint partitioning is correct, but it also shows that at the end there is a separation constraint between any pair of partitions. This implies that not further merging is possible, even when other partitioning schemes are used. Therefore separation constraint partitioning finds a maximal solution.

Now we know that the individual basic block partitioning algorithms are correct with respect to the basic block annotations. Still missing is a proof that the actual inlet/outlet annotations which our algorithms derives are valid, *i.e.*, that they overestimate the potential indirect dependencies and underestimate the certain indirect dependencies. Before being able to define whether an inlet and outlet annotation of a basic block is valid or not we must understand what they represent. The actual inlet and outlet names used are not important, but the sharing that exists between them is. This sharing is captured by congruence syndromes.

A.5 Congruence Syndromes

The inlet and outlet annotations that our partitioning algorithm uses assigns each inlet or outlet node a set of names. These names capture sharing properties, which are expressed by congruence syndromes.

Definition 20 (Congruence Syndrome) The *congruence syndrome* of a collection of sets of names $C = (S_1, \ldots, S_n)$ is a $2^n \times 2^n$ matrix defined as

$$\begin{aligned} Syn(C) &: Pow(\{1,\ldots,n\}) \times Pow(\{1,\ldots,n\}) \rightarrow \{0,1\} \text{ where} \\ Syn(C)(I_1, I_2) &= 1 \text{ iff } \bigcup_{i \in I_1} S_i = \bigcup_{i \in I_2} S_i \end{aligned}$$

A partial order on congruence syndromes is defined as follows: for two collection of sets of names $C = (S_1, \ldots, S_n)$ and $C' = (S'_1, \ldots, S'_n)$

$$Syn(C) \sqsubseteq Syn(C') \text{ iff } Syn(C)(I_1, I_2) \leq Syn_2(C')(I_1, I_2) \;\; \forall I_1, I_2 \subseteq \{1, \ldots n\}$$

Proposition 5 (Syndrome name overlap) Let $C = (S_1, \ldots, S_n)$ and $C' = (S'_1, \ldots, S'_n)$ be two collections of sets of names over the finite alphabet Σ, $S_i, S'_i \subseteq \Sigma$ for $1 \leq i \leq n$, such that $Syn(C') \sqsubseteq Syn(C)$. For a name $\alpha \in \Sigma$, let $I(\alpha)$ denote the index set $I(\alpha) = \{i | \alpha \in S_i\}$. Then for all $i \in \{1, \ldots, n\}$ and for all $\alpha \in S_i$ there exists an $\alpha' \in S'_i$ such that $I(\alpha') \subseteq I(\alpha)$.

Proof: (due to Steve Lumetta) We prove this by contradiction.

Assume that $Syn(C') \sqsubseteq Syn(C)$ and that there exists an $i \in \{1, \ldots, n\}$ and $\alpha \in S_i$ such that for all $\alpha' \in S'_i$ we have $I(\alpha') \not\subseteq I(\alpha)$. Thus for all $\alpha' \in S'_i$ there exist a $j \neq i$ for which $\alpha' \in S'_j$ and $\alpha \notin S_j$.

Let I_1 be the set of all of those indices j and let $I_2 = I_1 \cup \{i\}$. Then $S'_i \subseteq \bigcup_{j \in I_1} S'_j$ because for every $\alpha' \in S'_i$ there exists a $j \in I_1$, such that $\alpha' \in S'_j$. Thus $\bigcup_{j \in I_1} S'_j = \bigcup_{j \in I_2} S'_j$. On the other hand we know that $S_i \not\subseteq \bigcup_{j \in I_1} S_j$ because $\alpha \in S_i$ and $\alpha \notin S_j$ for all $j \in I_1$. Thus $\bigcup_{j \in I_1} S_j \neq \bigcup_{j \in I_2} S_j$, and therefore $Syn(C') \not\sqsubseteq Syn(C)$ which is a contradiction to above assumption. □

Proposition 6 (Syndrome alpha-renaming) Let $C = (S_1, \ldots, S_n)$ be a collection of sets of names over the alphabet $A = \{\alpha_1, \ldots, \alpha_k\}$, *i.e.*, $S_i \subseteq A$ for $1 \leq i \leq n$. Let $B = \{\beta_1, \ldots, \beta_k\}$ be an alphabet disjoint from A, *i.e.*, $A \cap B = \emptyset$. Let ρ be an alpha-renaming of A, $\rho(\alpha_i) = \beta_i$ such that $\beta_i \neq \beta_j$ if $i \neq j$, with the natural extension $\rho(S_i) = \{\rho(\alpha_j) | \alpha_j \in S_i\}$. Let $C' = (S'_1, \ldots, S'_n)$ be another collection of sets of names, with an arbitrary number of sets being renamed, *i.e.*, $S'_i = S_i$ or $S'_i = \rho(S_i)$ for $1 \leq i \leq n$. Then $Syn(C') \sqsubseteq Syn(C)$.

Proof: Show that for arbitrary $I_1, I_2 \subseteq \{1, \ldots, n\}$ the property $Syn(C)(I_1, I_2) = 0$ implies

Syn$(C')(I_1, I_2) = 0$. Assume *Syn*$(C)(I_1, I_2) = 0$, thus $\bigcup_{i \in I_1} S_i \neq \bigcup_{i \in I_2} S_i$. Without loss of generality, there must exist an index $i \in I_1$ and a name $\alpha \in S_i$ such that $\alpha \notin S_j$, for all $j \in I_2$. (Otherwise we can just exchange I_1 and I_2 to make this argument.) There are two cases:

1) S_i did not get alpha-renamed. Thus $\alpha \in S'_i$ and since $A \cap B = \emptyset$ we know that $\alpha \notin S'_j$ for all $j \in I_2$. This implies $\bigcup_{i \in I_1} S'_i \neq \bigcup_{i \in I_2} S'_i$, and therefore *Syn*$(C')(I_1, I_2) = 0$.

2) S_i got alpha-renamed, and $\beta = \rho(\alpha) \in S'_i$. Since $\rho(\alpha) \neq \rho(\alpha')$ if $\alpha \neq \alpha'$ and $A \cap B = \emptyset$ we know that $\beta \notin S'_j$ for all $j \in I_2$. This implies $\bigcup_{i \in I_1} S'_i \neq \bigcup_{i \in I_2} S'_i$, and therefore *Syn*$(C')(I_1, I_2) = 0$. □

Proposition 7 (Syndrome glb) Let $C^1 = (S^1_1, \ldots, S^1_n)$ and $C^2 = (S^2_1, \ldots, S^2_n)$ be two name disjoint collections of sets of names over the finite alphabet Σ, $(\bigcup_{i=1}^n S^1_i) \cap (\bigcup_{i=1}^n S^2_i) = \emptyset$, and let $C^{12} = (S^1_1 \cup S^2_1, \ldots, S^1_n \cup S^2_n)$.

Then *Syn*$(C^{12}) = glb($*Syn*$(C^1),$ *Syn*$(C^2))$, *i.e.*,

1. *Syn*$(C^{12}) \sqsubseteq$ *Syn*(C^1)
2. *Syn*$(C^{12}) \sqsubseteq$ *Syn*(C^2)
3. for all C^3 for which *Syn*$(C^3) \sqsubseteq$ *Syn*(C^1) and *Syn*$(C^3) \sqsubseteq$ *Syn*(C^2) we have *Syn*$(C^3) \sqsubseteq$ *Syn*(C^{12}).

Proof: We show that C^{12} satisfies the above three conditions.

1) and 2): Select arbitrary $I_1, I_2 \subseteq \{1, \ldots, n\}$ and assume $\bigcup_{i \in I_1} S^{12}_i = \bigcup_{i \in I_2} S^{12}_i$. Thus $(\bigcup_{i \in I_1} S^1_i) \cup (\bigcup_{i \in I_1} S^2_i) = (\bigcup_{i \in I_2} S^1_i) \cup (\bigcup_{i \in I_2} S^2_i)$ and because C^1 and C^2 are name disjoint we obtain that $\bigcup_{i \in I_1} S^1_i = \bigcup_{i \in I_2} S^1_i$ and $\bigcup_{i \in I_1} S^2_i = \bigcup_{i \in I_2} S^2_i$. This implies *Syn*$(C^{12}) \sqsubseteq$ *Syn*(C^1) and *Syn*$(C^{12}) \sqsubseteq$ *Syn*(C^2).

3): Assume there exists a C^3 for which *Syn*$(C^3) \sqsubseteq$ *Syn*(C^1) and *Syn*$(C^3) \sqsubseteq$ *Syn*(C^2). Select arbitrary $I_1, I_2 \subseteq \{1, \ldots, n\}$ and assume $\bigcup_{i \in I_1} S^3_i = \bigcup_{i \in I_2} S^3_i$. Thus we obtain $\bigcup_{i \in I_1} S^1_i = \bigcup_{i \in I_2} S^1_i$ and $\bigcup_{i \in I_1} S^2_i = \bigcup_{i \in I_2} S^2_i$, which implies $(\bigcup_{i \in I_1} S^1_i) \cup (\bigcup_{i \in I_1} S^2_i) = (\bigcup_{i \in I_2} S^1_i) \cup (\bigcup_{i \in I_2} S^2_i)$ and $\bigcup_{i \in I_1} S^{12}_i = \bigcup_{i \in I_2} S^{12}_i$, so we obtain *Syn*$(C^3) \sqsubseteq$ *Syn*(C^{12}). □

Proposition 8 (Syndrome Subsets) Let $C = (S_1, \ldots, S_n)$ and $C' = (S'_1, \ldots, S'_n)$ be two collections of sets of names, $S_i, S'_i \subseteq \Sigma$. Let $\Pi = \{I_1, \ldots, I_k\}$ be a partitioning of the index set $\{1, \ldots, n\}$ for which C and C' fulfill the following two properties:

1. partitions of C' do not share any names, *i.e.*,

$$\forall I_i, I_j \in \Pi, I_i \neq I_j, (\bigcup_{m \in I_i} S'_m) \cap (\bigcup_{m \in I_j} S'_m) = \emptyset$$

2. for all $I_i = \{i_1, \ldots, i_m\} \in \Pi$ we have $Syn(S'_{i_1}, \ldots, S'_{i_m}) \sqsubseteq Syn(S_{i_1}, \ldots, S_{i_m})$

Then $Syn(C') \sqsubseteq Syn(C)$.

Proof: Show that for arbitrary $J_1, J_2 \subseteq \{1, \ldots, n\}$ if $\bigcup_{j \in J_1} S'_j = \bigcup_{j \in J_2} S'_j$ then $\bigcup_{j \in J_1} S_j = \bigcup_{j \in J_2} S_j$. Assume $\bigcup_{j \in J_1} S'_j = \bigcup_{j \in J_2} S'_j$.

Let $\Pi_1 = \{I_i^1 | I_i^1 = J_1 \cap I_i, 1 \leq i \leq k\}$ be the partitioning Π restricted to index set J_1, and let $\Pi_2 = \{I_i^2 | I_i^2 = J_2 \cap I_i, 1 \leq i \leq k\}$ be the partitioning Π restricted to index set J_2.

Because of property 1) the corresponding partitions of C' do not share any names, *i.e.*, for all $I_i^1, I_j^1 \in \Pi_1, I_i^1 \neq I_j^1$ we have $(\bigcup_{m \in I_i^1} S'_m) \cap (\bigcup_{m \in I_j^1} S'_m) = \emptyset$ and for all $I_i^2, I_j^2 \in \Pi_1, I_i^2 \neq I_j^2$ we have $(\bigcup_{m \in I_i^2} S'_m) \cap (\bigcup_{m \in I_j^2} S'_m) = \emptyset$.

This combined with above assumption $\bigcup_{j \in J_1} S'_j = \bigcup_{j \in J_2} S'_j$ implies that the partitions must be equal, *i.e.*, for all $1 \leq i \leq k$ $\bigcup_{m \in I_i^1} S'_m = \bigcup_{m \in I_i^2} S'_m$. Applying property 2) we get for all $1 \leq i \leq k$ $\bigcup_{m \in I_i^1} S_m = \bigcup_{m \in I_i^2} S_m$, which implies $\bigcup_{j \in J_1} S_j = \bigcup_{j \in J_2} S_j$ and therefore $Syn(C') \sqsubseteq Syn(C)$. □

A.6 Valid Inlet/Outlet Annotations

With congruence syndromes we now have the mechanisms in place to address the question of when a basic block annotation is valid. Essentially the annotation has to correctly capture the certain indirect dependencies and the congruence sharing of dependencies between all inputs and outputs of instances of the basic block.

Definition 21 (Output Dependence Sets / Input Demand Sets / Indirect Dependencies) The *output dependence set* with respect to a basic block *BB*, partitioning Θ, and annotation A, for an input node of a program instance captures all of the output nodes of instances of *BB* that this input node directly depends on (*i.e.*, where the dependencies do not go through any other instances of *BB*). Excluded are those dependencies which are already captured by the certain indirect dependencies of the annotation. For an instance $BB_k^{(j)}$ of BB_k and an $r \in IN(BB_k)$

$$ODEP_{(IP,BB,A,\Theta)}(r, BB_k^{(j)}) = \{(s, BB^{(i)}) | s \in OUT(BB) \text{ for which there exists a non-trivial path from } \Theta(s, BB^{(i)}) \text{ to } \Theta(r, BB_k^{(j)}) \text{ in } G_{IP}{\downarrow}\Theta \text{ which does not go through other nodes of instances of } BB \text{ and } (s,r) \notin CID(A) \text{ if } BB_k^{(j)} = BB^{(i)}\}.$$

Similarly, the *input demand set* with respect to a basic block *BB*, partitioning Θ, and annotation A, for an output node of a program instance captures all of the input nodes of instances of *BB* that directly demand this output node. Excluded are those dependencies which are already captured by the certain indirect dependencies of the annotation. For an instance $BB_k^{(j)}$ of BB_k and an $s \in OUT(BB_k)$

$$IDEM_{(IP,BB,A,\Theta)}(s, BB_k^{(j)}) = \{(r, BB^{(i)}) | r \in IN(BB) \text{ for which there exists a non-trivial path from } \Theta(s, BB_k^{(i)}) \text{ to } \Theta(r, BB^{(j)}) \text{ in } G_{IP}{\downarrow}\Theta \text{ which does not go through other nodes of instances of } BB \text{ and } (s,r) \notin CID(A) \text{ if } BB_k^{(j)} = BB^{(j)}\}.$$

A valid annotation for a basic block, call site, or def site must faithfully capture the sharing between these sets

Definition 22 (Valid Annotation) An annotation $A = (\Sigma_i, Inlet, \Sigma_o, Outlet, CID)$ for a collection of inlet $(r_1, \ldots, r_m)$ and outlet nodes $(s_1, \ldots, s_n)$ of a basic block BB_k is called *valid with respect to a basic block BB and a partitioning* Θ, if for all admissible program instances *IP* and for all instances $BB_k^{(j)} \in IP$ of the basic block BB_k the following three conditions are satisfied

1. $Syn(Inlet(r_1), \ldots, Inlet(r_m)) \sqsubseteq$
   $Syn(ODEP_{(IP,BB,A,\Theta)}(r_1, BB_k^{(j)}), \ldots, ODEP_{(IP,BB,A,\Theta)}(r_m, BB_k^{(j)}))$,
2. $Syn(Outlet(s_1), \ldots, Outlet(s_n)) \sqsubseteq$
   $Syn(IDEM_{(IP,BB,A,\Theta)}(s_1, BB_k^{(j)}), \ldots, IDEM_{(IP,BB,A,\Theta)}(s_n, BB_k^{(j)}))$, and
3. for all $(s, r) \in CID$ there exists a path from $\Theta(s, BB_k^{(j)})$ to $\Theta(r, BB_k^{(j)})$ in $G_{IP}{\downarrow}\Theta$.

An annotation for a site S of a basic block is called a *valid site annotation with respect to a basic block BB and a partitioning* Θ, if it is valid for the collection of input and output nodes comprising this site. An annotation for a basic block *BB* is called a *valid basic block annotation under a partitioning* Θ if the annotation for the collection of all input and output nodes of the basic block is valid with respect to *BB* and Θ.

This definition of a valid annotation with respect to a *BB* and Θ essentially treats the instances of the basic block *BB* as being opaque as the sets $IDEM_{(IP,BB,A,\Theta)}$ and $ODEP_{(IP,BB,A,\Theta)}$ do not trace dependencies across instances of *BB*.

Definition 23 (Trivial Annotation) The trivial annotation A_0 for a collection of inlet and outlet nodes assigns every inlet node a unique singleton inlet annotation, every outlet node a unique singleton outlet name, and introduces no additional certain indirect dependence edges.

The trivial annotation for a basic block also assigns each I-store node a unique singleton outlet annotation and each I-fetch response node a unique singleton inlet annotation and includes all of the original squiggly edges in the set of certain indirect dependencies.

Lemma 6 (Trivial Annotation is Valid) The trivial annotation for a collection of inlet and outlet nodes is valid with respect to any basic block and any correct partitioning of the program.

Proof: Let $S = ((r_1, \ldots, r_m), (s_1, \ldots, s_n))$ be a collection of inlet and outlet nodes. Let A_0 be the trivial annotation for this collection of nodes. Let $BB \in P$ be an arbitrary basic block, and let Θ be a correct partitioning of the program. We prove that A_0 is a valid annotation with respect to *BB* and Θ, by showing that it satisfies the three properties of a valid annotation.

1) We show that
$Syn(Inlet(r_1), \ldots, Inlet(r_m)) \sqsubseteq Syn(ODEP_{(IP,BB,A,\Theta)}(r_1), \ldots, ODEP_{(IP,BB,A,\Theta)}(r_m))$. Let $C = (Inlet(r_1), \ldots, Inlet(r_m))$ be the collection of sets of inlet names for the inlet nodes. The trivial annotation assigns every inlet node a unique singleton inlet name. Thus we know that for any $I_1, I_2 \subseteq \{1, \ldots, m\}$ that $Syn(C)(I_1, I_2) = 1$ iff $I_1 = I_2$. Thus C forms the bottom element in the syndrome lattice, which implies that for any other collection of sets $C' = \{S_1, \ldots, S_m\}$ we have $Syn(C) \sqsubseteq Syn(C')$. This gives us the desired property.

2) Analogous to 1), arguing about *Outlet* and $IDEM_{(IP,BB,A,\Theta)}$ instead.

3) Since the trivial annotation does not introduce any new certain indirect dependence edges this property is still true. □

Proposition 9 (Valid Annotations) If all sites in a basic block *BB* have a valid site annotation with respect to *BB* and a partitioning Θ, and the annotations are disjoint (*i.e.*, different site annotations use different inlet and outlet names) then all of the site annotations together with the annotations for the I-store and I-fetch response nodes and initial squiggly edges form a valid annotation for the basic block under Θ.

Proof: Assume that every site in the basic block *BB* has a valid site annotation with respect to *BB* and the partitioning Θ. Assume also that every site has an annotation which is disjoint from the others, *i.e.*, different sites have different inlet and outlet alphabets and certain indirect dependencies do not cross sites. We need to show that all of these site annotations together with the annotations given the I-store and I-fetch response nodes form a valid annotation for the basic block.

Let $\mathit{SITES}(\mathit{BB}) = (S_0, \ldots, S_k)$ be the collection of def and call sites of the basic block, including a site which contains all of the I-store and I-fetch response nodes (which are annotated with the trivial annotation). Let $(r^i_1, \ldots, r^i_m)$ and $(s^i_1, \ldots, s^i_n)$ be the collection of nodes forming site S_i.

The collection of inlet and outlet nodes for the basic block $(r^0_1, \ldots, r^k_m)$ and $(s^0_1, \ldots, s^i_n)$ is obtained by appending the inlet and outlet nodes of all sites. The annotation of the basic block is formed by taking the union over all sites for each of the five components of the annotations.

$A(\mathit{BB}) = (\Sigma_i, \mathit{Inlet}, \Sigma_o, \mathit{Outlet}, \mathit{CID})$, where
$\Sigma_i(\mathit{BB}) = \bigcup_{S \in \mathit{SITES}(\mathit{BB})} \Sigma_i(S)$,
$\mathit{Inlet}(\mathit{BB}) = \bigcup_{S \in \mathit{SITES}(\mathit{BB})} \mathit{Inlet}(S)$,
$\Sigma_o(\mathit{BB}) = \bigcup_{S \in \mathit{SITES}(\mathit{BB})} \Sigma_o(S)$,
$\mathit{Outlet}(\mathit{BB}) = \bigcup_{S \in \mathit{SITES}(\mathit{BB})} \mathit{Outlet}(S)$,
$\mathit{CID}(\mathit{BB}) = \bigcup_{S \in \mathit{SITES}(\mathit{BB})} \mathit{CID}(S) \cup E_q$.

This annotation satisfies the three conditions of a valid annotation.

1) For each of the sites S_i we have
$\mathit{Syn}(\mathit{Inlet}(r^i_1), \ldots, \mathit{Inlet}(r^i_m)) \sqsubseteq \mathit{Syn}(\mathit{ODEP}_{(\mathit{IP},\mathit{BB},A,\Theta)}(r^i_1), \ldots, \mathit{ODEP}_{(\mathit{IP},\mathit{BB},A,\Theta)}(r^i_m))$.

Because the site annotations do not share any names, we immediately obtain from Proposition 8 that this property must also hold for all of the inlet nodes of the basic block, *i.e.*,

$Syn(Inlet(r_1^0), \ldots, Inlet(r_m^k)) \sqsubseteq Syn(ODEP_{(IP,BB,A,\Theta)}(r_1^0), \ldots, ODEP_{(IP,BB,A,\Theta)}(r_m^k))$.

2) analogous to 1).

3) We know that all sites satisfy that for all admissible program instances $\Theta(s, BB^{(j)})) \neq \Theta(r, BB^{(j)})$ for all $(s, r) \in CID(S_i)$. Therefore this property is also valid for $CID(BB)$. $\square$

Our partitioning algorithm starts with the trivial annotation and then iteratively improves it by propagating annotations across call and def site interfaces. This interprocedural analysis must be conservative to be safe. It is therefore important that the congruence syndrome of a newly annotated site connected to multiple other sites corresponds to the greatest lower bound of the syndrome of the annotations as derived from each of the other sites. In addition, the algorithm only derives certain indirect dependencies if the corresponding path exists in all of the other sites. This is achieved by the annotation propagation Algorithm 8 from Section 5.4.

Lemma 7 (Annotation Propagation is Correct) Let Θ be a correct partitioning of the program and let S be a site in BB , and let $S_1, \ldots, S_l$ in $BB_1, \ldots, BB_l$ be all of the sites connected to S. Let BB' be a basic block such that for all $BB_k, 1 \leq k \leq l, BB_k \neq BB'$ and all sites in BB_k have a valid annotation with respect to the basic block BB' and the partitioning Θ. Then propagating a new annotation to the site S using Algorithm 8 results in an annotation for S which is also valid with respect to BB' and Θ.

Proof: Let Θ be a correct partitioning of the program, let S be a site in BB, and $S_1, \ldots, S_l$ of the basic blocks $BB_1, \ldots, BB_l$ be all of the sites connected to S. Let BB' be a basic block such that for all $BB_k, 1 \leq k \leq l$, $BB_k \neq BB'$ and all sites in BB_k have a valid annotation with respect to BB' and Θ. Let $A'(S)$ be the new annotation for S produced by Algorithm 8. We need to show that $A'(S)$ is a valid annotation with respect to BB' and Θ, *i.e.*, that the three conditions of a valid annotation are satisfied for all admissible program instances. Let IP be an admissible program instance and $BB^{(p)} \in IP$ an instance of BB. Let $(S_k, BB_k^{(q)})$ be the site connected to $(S, BB^{(p)})$ for this program instance. Let $((r_1, \ldots, r_m), (s_1, \ldots, s_n))$ be the collection of inlet and outlet nodes forming the site S, and $((r_1^k, \ldots, r_n^k), (s_1^k, \ldots, s_m^k))$ the nodes for site S_k.

1) We need to show that
$Syn(Inlet(r_1), \ldots, Inlet(r_m)) \sqsubseteq$
$Syn(ODEP_{(IP,BB',A',\Theta)}(r_1, BB^{(p)}), \ldots, ODEP_{(IP,BB',A',\Theta)}(r_m, BB^{(p)}))$.

From the way that the Dep_j^k are computed we obtain that in Step 5 in Algorithm 8 the following holds
$Syn(Dep_1^k, \ldots, Dep_m^k) \sqsubseteq Syn(ODEP_{(IP,BB',A',\Theta)}(r_1, BB^{(p)}), \ldots, ODEP_{(IP,BB',A',\Theta)}(r_m, BB^{(p)}))$.
Thus, since $(Dep_j^{k_1} \cap Dep_j^{k_2}) = \emptyset$ for $k_1 \neq k_2$ (*i.e.*, for different k all of the Dep_j^k have non-overlapping names), and we know that $Inlet(r_j) = \cup_k Dep_j^k$, we obtain from Proposition 6 and Proposition 7 that $Syn(Inlet(r_1), \ldots, Inlet(r_m)) \sqsubseteq Syn(Dep_1^k, \ldots, Dep_m^k)$. Because of transitivity of "$\sqsubseteq$" we obtain the desired property.

2) We need to show that
$Syn(Outlet(s_1), \ldots, Outlet(s_n)) \sqsubseteq$
$Syn(IDEM_{(IP,BB',A',\Theta)}(s_1, BB^{(p)}), \ldots, IDEM_{(IP,BB',A',\Theta)}(s_n, BB^{(p)}))$. The proof is analogous to part a) arguing with outlet annotations and demand sets instead of inlet annotations and dependence sets.

3) We need to show that for all certain indirect dependencies $(s_i, r_j) \in CID'(S)$ there exists a path from $\Theta(r_i^k, BB_k^{(q)})$ to $\Theta(s_j^k BB_k^{(q)})$ in $G_{IP} \downarrow \Theta$. The algorithm introduces a squiggly edge (s_i, r_j) only if $\sigma_j^k \in Dem_i^k$ for all k. If this is the case we know that there exists a path from $\Theta(r_i^k, BB_k^{(q)})$ to $\Theta(s_j^k, BB_k^{(q)})$. Since $(s_i, BB_k^{(q)})$ is connected to $(r_i^k, BB_k^{(q)})$ and $(s_j^k, BB_k^{(q)})$ is connected to $(r_j, BB_k^{(q)})$ in G_{IP} we get that there also exists the desired path $\Theta(s_i, BB_k^{(q)})$ to $\Theta(r_j, BB_k^{(q)})$ in $G_{IP} \downarrow \Theta$. □

It follows immediately that if V_i denotes the set of basic blocks BB' such that all sites in the basic block BB_i are valid with respect to BB' and Θ, then the annotation $A(S)$ is valid with respect to all basic blocks in $\bigcap_{1 \leq i \leq k} V_i$ and Θ.

It is also clear that after annotation propagation the partitioning Θ is still correct, since the partitioning has not been changed.

Proposition 10 (Cycle Detectable) Let Θ be a correct partitioning of the program, let A be a valid annotation of the basic block BB under Θ, and let Θ'_{BB} be a repartitioning of

BB and Θ' the resulting partitioning. If there exists a cycle in $G_{IP}\downarrow\Theta'$ for some admissible program instance IP, then there exists also an admissible $PID \in PIDS(A)$ such that $(G_{BB} \cup CID \cup PID)\downarrow\Theta'_{BB}$ contains a cycle.

Proof: Let Θ be a correct partitioning of the program, let A be a valid annotation of the basic block BB under Θ, and let Θ'_{BB} be a repartitioning of BB and Θ' the resulting partitioning. Let IP be an admissible program instance and assume $G_{IP}\downarrow\Theta'$ has a cycle.

Because IP is an admissible program instance and Θ is a correct partitioning we know that $G_{IP}\downarrow\Theta$ is acyclic. The repartitioning Θ' only changes the partitioning of instances of BB. Therefore, the cycle involves at least one instance of BB. As depicted in Figure A.2, the cycle consists of parts (segments) which are completely within instances of the basic block BB (just following edges of $G_{BB}\downarrow\Theta'$), and parts which are outside (following edges of other basic blocks or SF edges). We can distinguish two cases:

1) The cycle is completely within a single instance $BB^{(j)}$ of the basic block BB. Thus the cycle is also present in $G_{BB}\downarrow\Theta'$ and choosing the admissible potential indirect dependence set $PID = \emptyset$ gives us that $(G_{BB} \cup CID \cup PID)\downarrow\Theta'_{BB}$ contains a cycle.

2) The cycle involves some edges outside of a single instance of BB. As shown in Figure A.2, each piece of the path inside of an instance of BB enters through some inlet node r and leaves through some outlet node s, such that there exists a path from $\Theta'(r)$ to $\Theta'(s)$ in $G_{BB}\downarrow\Theta'$. Again we can distinguish two cases.

2a) There exists a segment along the cycle from $\Theta'(r, BB^{(j)})$ to $\Theta'(s, BB^{(j)})$ such that $(s, r) \in CID$. Thus there also exists the corresponding path from $\Theta'(r)$ to $\Theta'(s)$ in $G_{BB}\downarrow\Theta'_{BB}$, and $\Theta'(s)$ to $\Theta'(r)$ in $(G_{BB} \cup CID)\downarrow\Theta'_{BB}$. Thus choosing the admissible $PID = \emptyset$ gives us that $(G_{BB} \cup CID \cup PID)\downarrow\Theta'_{BB}$ also contains a cycle.

2b) For every segment of the cycle inside an instance of the basic block $\Theta'(r, BB^{(j)})$ to $\Theta'(s, BB^{(j)})$, it is the case that $(s, r) \notin CID$.

We know that $G_{IP}\downarrow\Theta$ does not contain a cycle because Θ is a correct partitioning. Therefore, as shown in Figure A.2, there must exists a a segment $\Theta'(r, BB^{(j)})$ to $\Theta'(s, BB^{(j)})$ along the cycle in $G_{IP}\downarrow\Theta'$ such that there does not exist a path from $\Theta(s^*, BB^{(k)})$ to $\Theta(r^*, BB^{(l)})$ in $G_{IP}\downarrow\Theta$, where $(s^*, BB^{(k)})$ is the first predecessor node of $\Theta'(r, BB^{(j)})$ along the cycle which belongs to an instance of BB, and likewise $(r^*, BB^{(l)})$ of $\Theta'(s, BB^{(j)})$ is the first successor node along the cycle which belongs to an instance of BB.

This implies that for all successors nodes $\Theta(r', BB^{(j)})$ of $\Theta(s^*, BB^{(k)})$ there does not exist a path in $G_{IP}{\downarrow}\Theta$ to any of the predecessors nodes $\Theta(s', BB^{(j)})$ of $\Theta(r^*, BB^{(l)})$.

From the definition of $ODEP_{(IP,BB,A,\Theta)}$ and $IDEM_{(IP,BB,A,\Theta)}$ and because $(s, r) \notin CID$ we get that $(s^*, BB^{(k)}) \in ODEP_{(IP,BB,A,\Theta)}(r, , BB^{(j)})$ and $(r^*, BB^{(l)}) \in IDEM_{(IP,BB,A,\Theta)}(s, , BB^{(j)})$. Since A is a valid basic block annotation under Θ we know that

$$Syn(Inlet(r_1), \ldots, Inlet(r_m)) \sqsubseteq Syn(ODEP_{(IP,BB,A,\Theta)}(r_1), \ldots, ODEP_{(IP,BB,A,\Theta)}(r_m))$$

and

$$Syn(Outlet(s_1), \ldots, Outlet(s_n)) \sqsubseteq Syn(IDEM_{(IP,BB,A,\Theta)}(s_1), \ldots, IDEM_{(IP,BB,A,\Theta)}(s_n)),$$

where $(r_1, \ldots, r_m)$ and $(s_1, \ldots, s_n)$ are the collection of inlet and outlet nodes of the basic block BB respectively. Let $S = \{s' | (r^*, BB^{(l)}) \in IDEM_{(IP,BB,A,\Theta)}(s', BB^{(j)})\}$ and $R = \{r' | (s^*, BB^{(k)}) \in ODEP_{(IP,BB,A,\Theta)}(r', BB^{(j)})\}$. From the definition of R and S we know that $r \in R, s \in S$ and that $(s^*, BB^{(k)}) \in ODEP_{(IP,BB,A,\Theta)}(r', BB^{(j)})$ for all $r' \in R$ and $(r^*, BB^{(k)}) \in IDEM_{(IP,BB,A,\Theta)}(s', BB^{(j)})$ for all $s' \in S$. Therefore, from Proposition 5 we obtain that there exists an $\alpha \in Inlet(r)$ such that for all r' for which $\alpha \in Inlet(r')$ we have $r' \in R$, and there exists a $\beta \in Outlet(s)$ such that for all s' for which $\beta \in Outlet(s')$ we have $s' \in S$. Let $DS = \{(\alpha, \beta)\}$ and $PID = PID(DS) = \{(s', r') | \beta \in Outlet(s'), \alpha \in Inlet(r')\}$. From this definition we get that $(s, r) \in PID$ and $PID \subseteq S \times R$. PID is admissible because, as observed above, there does not exist a path from $\Theta(r')$ to $\Theta(s')$ in $G_{IP}{\downarrow}\Theta$ for any of the $r' \in R$, $s' \in S$.

We also know that there exists a path from $\Theta'(r)$ to $\Theta'(s)$ in $G_{BB}{\downarrow}\Theta'_{BB}$. The admissible PID introduces the reverse path, therefore we obtain that $(G_{BB} \cup CID \cup PID){\downarrow}\Theta'_{BB}$ also contains a cycle. □

Lemma 8 (Correctness of Valid Annotation and Legal Repartitioning) If A is a valid basic block annotation under a correct partitioning Θ of the program and Θ'_{BB} is a legal repartitioning under A then the resulting partitioning Θ' is a correct partitioning.

Proof: We have to show that the three conditions of a correct partitioning are satisfied for Θ' for all admissible program instances IP.

1) We have to show that $G_{IP}{\downarrow}\Theta'$ is acyclic. We show this by contradiction. Assume that Θ' is a correct repartitioning and that there exists an admissible program instance IP

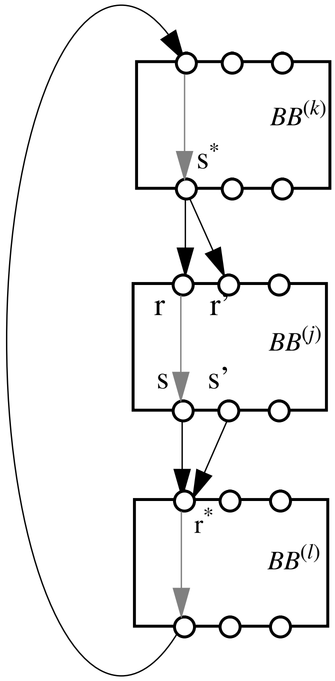


Figure A.2: *This graph shows the situation for the proof of Proposition 10. The dashed arcs indicate paths present under Θ'_{BB}, but not necessarily under Θ.*

such that the reduced graph $G_{IP}{\downarrow}\Theta'$ contains a cycle. Then, according to Proposition 10, there exists an admissible $PID \in PIDS(A)$ such that $(G_{BB} \cup PID){\downarrow}\Theta'_{BB}$ contains also a cycle. This is a contradiction to property a) of legal repartitioning which states that for all admissible PID we have $(G_{BB} \cup CID \cup PID){\downarrow}\Theta'_{BB}$ is acyclic.

2) We have to show that for all $BB_i^{(j)} \in P$ and for all $(s, r) \in E_q(BB_i)$ it is the case that $\Theta'(s) \neq \Theta'(r)$. Since the partitioning has not changed for other basic blocks $BB_i \neq BB$ is still the case that for all squiggly edges $(s, r) \in E_q(BB_i)$ we have $\Theta'(s) \neq \Theta'(r)$.

Because A is a valid basic block annotation for BB we know that $E_q(BB) \subseteq CID$. Condition 3) of correct repartitioning ensures that for all $(s, r) \in CID$ we have $\Theta'_{BB}(s) \neq \Theta'_{BB}(r)$, thus also $\Theta'(s) \neq \Theta'(r)$ which gives us the desired result for all instances of BB.

3) We have to show that $\Theta'((s, BB_i^{(j)})) \neq \Theta'((r, BB_k^{(l)}))$ for all $((s, BB_i^{(j)}), (r, BB_k^{(l)})) \in SF$. Select a $((s, BB_j^{(i)}), (r, BB_l^{(k)})) \in SF$. We distinguish three cases.

3a) Both end points are outside of instances of BB. Since the partitioning outside has not changed it must still be true that $(\Theta'(s), BB_j^{(i)}) \neq (\Theta'(r), BB_l^{(k)})$.

3b) One end point of the store/fetch dependence is inside an instance of BB , the other one is outside. Because the new partitioning Θ_{BB} does not cross basic block boundary we trivially get $(\Theta'(s), BB_j^{(i)}) \neq (\Theta'(r), BB_l^{(k)})$.

3c) Both end point of the store/fetch dependence are inside an instance of BB, *i.e.*, $BB_j = BB_l$. Since Θ is a correct partitioning we know that $G_{IP}\downarrow\Theta$ is acyclic. Since $((s, BB_j^{(i)}), (r, BB_j^{(k)})) \in SF$ we know that there does not exist a path from $\Theta(s, BB_j^{(i)})$ to $\Theta(r, BB_j^{(k)})$ in $G_{IP}\downarrow\Theta$. Our algorithm assigns every store node a unique singleton outlet annotation, and every fetch response a unique singleton inlet annotation. Let i be the inlet annotation for r and o the outlet annotation for s. Then $PID = PIDS(\{o, i\}) = \{s, r\}$ is an admissible potential indirect dependence set. Since Θ' is a correct repartitioning we know that $\Theta'_{BB}(s) \neq \Theta'_{BB}(r)$, and thus also $(\Theta'(s), BB_j^{(i)}) \neq (\Theta'(r), BB_l^{(k)})$. $\square$

A.7 The Partitioning Algorithm

So far we have only proven the correctness of the individual steps used in the interprocedural partitioning algorithm: trivial partitioning, trivial annotation, annotation propagation, and repartitioning. To fit all of those pieces together the algorithm keeps with every inlet/outlet annotation a set $Uses$, describing the set of basic blocks from which the annotation may have used information. Thus repartitioning one of those basic blocks may invalidate this annotation, and our algorithm therefore resets it to the trivial annotation.

The structure of the algorithm is shown in Figure A.3. The partitioning algorithm starts with the trivial partitioning and trivial annotation which establishes the partitioning invariant. In subsequent steps, the algorithm either reannotates a site with a trivial annotation, propagates annotations across basic block interfaces, or repartitions some basic block. Each of these steps maintains the correctness. Therefore the final partitioning of the program is also correct.

Definition 24 (Partitioning state) A partitioning state of the interprocedural partitioning algorithm is a four tuple $(P, \Theta, A, Uses)$ consisting of a partitioning Θ of the program P, an annotation A for all nodes of the program, and a function $Uses : S \rightarrow \{BB_k, \ldots, BB_l\}$ which specifies the set of basic blocks from which a site annotation may have used information.

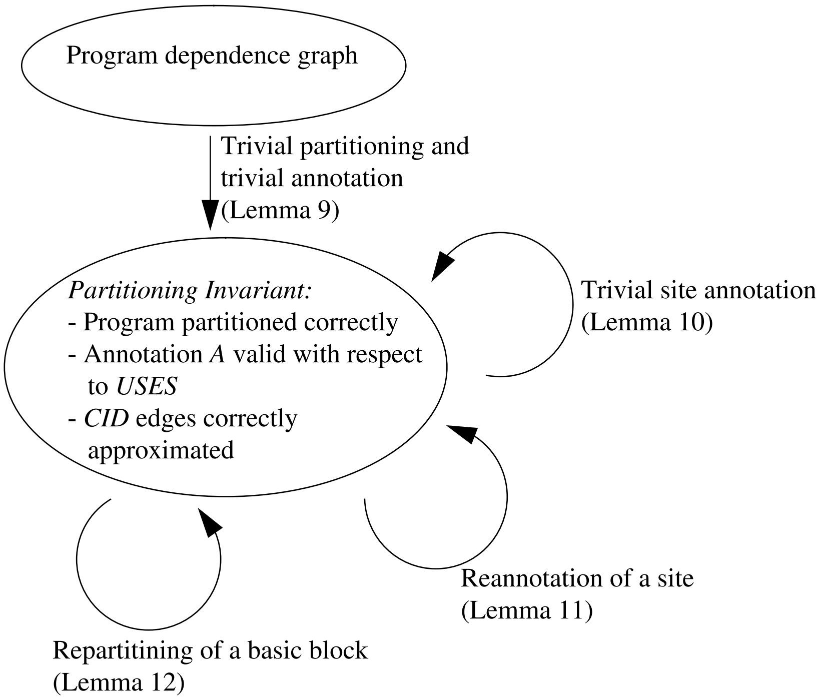


Figure A.3: *Structure of correctness proof of interprocedural partitioning algorithm. The trivial partitioning establishes the partitioning invariant and all of the subsequent steps, trivial annotation, annotation propagation and repartitioning maintains it. As a result the program is partitioned correctly at the end.*

Definition 25 (Partitioning Invariant) A partitioning state $(P, \Theta, A, \mathit{Uses})$ of the interprocedural partitioning algorithm satisfies the partitioning invariant, if

1. Θ is a correct partitioning of P,
2. the annotation A is valid with respect to *Uses*, *i.e.*,
   (a) for all sites $S \in P$ the site annotation $A(S)$ is valid with respect to Θ and all $BB_i \in P$ for which $BB_i \notin \mathit{Uses}(S)$.
   (b) for all sites $S \in P$ the site annotation $A(S)$ is valid with respect to Θ' and all $BB_i \in P$ for which $BB_i \notin \mathit{Uses}(S)$, where Θ' is derived from Θ by repartitioning an arbitrary set of basic blocks BB_i for which $BB_i \notin \mathit{Uses}(S)$.
   (c) The certain indirect dependence edges introduced by the annotations are approximated correctly, *i.e.*, the annotation $A(S)$ of a site $S = ((r_1, \ldots, r_m), (s_1, \ldots, s_n))$ in the basic block *BB* only introduces a squiggly edge $(s_i, r_j) \in \mathit{CID}$ if for all

admissible program instances *IP* and all instances of the basic block $BB^{(k)} \in IP$ there exist a path from $\Theta(s_i, BB^{(k)})$ to $\Theta(r_j, BB^{(k)})$ in $G_{IP}{\downarrow}\Theta$.

Lemma 9 (Trivial Partitioning and Trivial Annotation establishes Invariant) The initial partitioning state
$(P, \Theta_0, A_0, Uses_0)$ satisfies the partitioning invariant, where Θ_0 is the trivial partitioning, and for all basic blocks $A_0(BB)$ is the trivial annotation and $Uses_0(S) = \{\}$ for all $S \in P$.

Proof: We need to show that the properties of the partitioning invariant are satisfied.

1) From Lemma 1 we know that trivial partitioning results in a correct partitioning of the program.

2a) and 2b) From Lemma 6 we know that the trivial annotation for a collection of inlet and outlet nodes is valid with respect to any basic block and any correct partitioning of the program. Thus this property holds trivially.

2c) Since the set of certain indirect dependence edges only contains the set of original squiggly edges ($CID = E_q$), this property is also trivially true. □

Lemma 10 (Trivial Annotation maintains Invariant) Assigning a site S the trivial annotation, keeping the set of *CID* edges, setting $Uses(S) = \{\}$, maintains the partitioning invariant, *i.e.*, a partitioning state which satisfies the partitioning invariant still satisfies it after a site is assigned the trivial annotation.

Proof: Assume that the current partitioning state satisfies the partitioning invariant and that a site S is assigned the trivial site annotation. We need to show that the conditions of partitioning invariant are still satisfied.

1) Changing an annotation does not affect the partitioning, therefore the partitioning of the program must still be correct.

2a) and 2b) Reannotation of the site S does not affect the validity of any other site annotation. We just need to show that $A_0(S)$ satisfies this property. This follows directly from Lemma 6 which says that a trivial annotation of site is valid with respect to any basic block and any partitioning.

2c) This condition was true before the trivial annotation. Because neither the *CID* set nor the partitioning change, this condition must still be true. □

Proposition 11 (Certain Indirect Dependencies) Repartitioning a basic block never invalidates any certain indirect dependence edges from the annotations, *i.e.*, for all sites S the squiggly edges $CID(S)$ are still correctly approximated after repartitioning any basic block.

> **Proof:** A certain indirect dependence edge $(s, r) \in CID(S)$ is correctly approximated under a partitioning Θ, if for all admissible program instances IP and all basic block instances $BB^{(j)} \in IP$ there exists a path from $\Theta(s, BB^{(j)})$ to $\Theta(r, BB^{(j)})$ in $G_{IP}{\downarrow}\Theta$. Because a repartitioning Θ'_{BB} is only allowed to enlarge partitions, all paths present in $G_{IP}{\downarrow}\Theta$ will still be present in $G_{IP}{\downarrow}\Theta'$. Thus all CID edges are still correctly approximated. □

Lemma 11 (Annotation Propagation maintains Invariant) Let S in the basic block BB be a site which is connected to the sites $S_1, \ldots, S_l$ in the basic blocks $BB_1, \ldots, BB_l$. If the current partitioning state satisfies the partitioning invariant, then reannotating S using Algorithm 8 results in a new partitioning state which also satisfies the partitioning invariant.

> **Proof:** Let $(P, \Theta, A, Uses)$ be a partitioning state which satisfies the partitioning invariant. Let S be a site in the basic block BB and let $S_1, \ldots, S_l$ in $BB_1, \ldots, BB_l$ be all of the sites connected to S. Let A' and $Uses'$ be the resulting annotation and uses sets after annotation propagation as computed by Algorithm 8. The new partitioning state is $(P, \Theta, A', Uses')$ where A' and $Uses'$ are the same as A and $Uses$ for sites different from S.
>
> We now show that this new partitioning state satisfies the three properties of the partitioning invariant.
>
> 1) Reannotation does not affect the partitioning, therefore the program is still partitioned correctly.
>
> 2a) Reannotation does not affect the validity of any other site annotations, therefore we only need to show that the annotation $A'(S)$ is valid with respect to Θ and all $BB_i \in P$ for which $BB_i \notin Uses'(S)$.
>
> Let $BB_i \in P$ such that $BB_i \notin Uses'(S)$. We need to show that $A'(S)$ is valid with respect to Θ and BB_i. We know from the definition of $Uses'(S)$ that $BB_i \neq BB_k$ and $BB_i \notin Uses(S')$ for all $S' \in BB_k, 1 \leq k \leq l$. Thus we know that $A(S')$ is valid with respect to BB_i and Θ for all $S' \in BB_k, 1 \leq k \leq l$. Applying Lemma 7 we obtain that the annotation $A'(S)$ is also valid with respect to BB_i and Θ.

2b) We know that the property 2b) is correct before reannotation. Algorithm 8 makes only use of the annotations of the sites in $BB_k, 1 \leq k \leq l$, the dependence structure of those basic blocks BB_k, and their certain indirect dependence edges. Let Let $BB_i \in P$ such that $BB_i \notin Uses'(S)$. We need to show that repartitioning BB_i does not affect the validity of the new annotation $A'(S)$. By the definition of $Uses'(S)$ we obtain that $BB_i \neq BB_k$ and $BB_i \notin Uses(S')$ for all $S' \in BB_k, 1 \leq k \leq l$. Because property 2b) applies to all annotations of these sites $S' \in BB_k, 1 \leq k \leq l$, we know that the validity of these annotations does not change when repartitioning BB_i. We also know that $BB_i \neq BB_k, 1 \leq k \leq l$. We also know from Proposition 11 that all certain indirect dependence edges are still valid after repartitioning BB_i. Thus we can apply b1) and obtain that the validity of $A'(S)$ has not changed.

3) From the partitioning invariant we know that all of the certain indirect dependence edges $CID(S')$ for all $S' \in BB_k, 1 \leq k \leq l$ are approximated correctly. From Lemma 7 we obtain that the new certain indirect dependence edges $CID'(S)$ are also correctly approximated. □

Lemma 12 (Repartitioning maintains Invariant) If the current partitioning state satisfies the partitioning invariant and the annotation A for the basic block *BB* is valid with respect to *BB* and Θ, then repartitioning that basic block using a legal repartitioning and setting to the trivial annotation all nodes v in the program for which $BB \in Uses(v)$ results in a new partitioning state that also satisfies the partitioning invariant.

Proof: Let $(P, \Theta, A, Uses)$ be a partitioning state which satisfies the partitioning invariant. Let *BB* be a basic block such that all sites in *BB* have a valid annotation with respect to *BB* and Θ. Let Θ'_{BB} be a legal repartitioning of *BB* under the annotation $A(BB)$ and Θ. The new partitioning state is $(P, \Theta', A', Uses')$ where Θ' is the new resulting partitioning and $A'(S)$ the trivial annotation and $Uses(S) = \{\}$ if $BB \in Uses(S)$, otherwise $A'(S) = A(S)$ and $Uses'(S) = Uses(S)$. We need show that this new partitioning state satisfies the three properties of the partitioning invariant.

1) From Proposition 9 we know that the basic block annotation must be valid under Θ because all sites in *BB* have a valid annotation with respect to *BB* and Θ. Since Θ'_{BB} is a legal repartitioning of *BB* under the annotation $A(BB)$ and Θ we obtain from Lemma 8 that Θ'_{BB} is also a correct repartitioning. Since Θ' is a correct partitioning of the program Θ' must also be.

2a and 2b) Follows directly from property b2) of the partitioning invariant.

3) It follows from Proposition 11 that all certain indirect dependence edges are still correctly approximated. □

Theorem 1 (Correctness of Interprocedural Algorithm) The interprocedural partitioning algorithm is correct.

Proof: We need to show that the final partitioning of the program is correct. The correctness follows directly from the previous lemmas and their relationship as shown in Figure A.3 1-3. The partitioning algorithm starts with the trivial partitioning and trivial annotation which establishes the partitioning invariant as shown in Lemma 9. In subsequent steps, the algorithm either reannotates a site with a trivial annotation, propagates annotations across basic blocks, or repartitions some basic block. In Lemma 10, Lemma 11, and Lemma 12 we proved that each of the three operations maintains the partitioning invariant. Therefore the final partitioning state must also satisfy the partitioning invariant, and from its first property we obtain that the partitioning of the program is correct. □