KLAUS ERIK SCHAUSER

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

Compiling Dataflow into Threads

Efficient Compiler-Controlled Multithreading for Lenient Parallel Languages

July 2, 1991

RESEARCH PROJECT

Submitted to the Department of Electrical Engineering and
Computer Sciences, University of California, Berkeley,
in partial satisfaction of the requirements for the degree
of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee:	Prof. David E. Culler	Research Advisor
	7/3/91	Date
	Prof. D. A. Patterson	Second Reader
	7/9/91	Date

Compiling Dataflow into Threads

Efficient Compiler-Controlled Multithreading for Lenient Parallel Languages

Klaus Erik Schauser

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

July, 1991

This work was supported by National Science Foundation PYI Award (CCR-9058342) with matching funds from Motorola Inc. and the TRW Foundation. Support was also received from the International Computer Science Institute, and computational resources were provided, in part, under NSF Infrastructure Grant CDA-8722788.

Compiling Dataflow into Threads

Efficient Compiler-Controlled Multithreading for Lenient Parallel Languages

Klaus Erik Schauser

Abstract

Powerful non-strict parallel languages require fast dynamic scheduling. This thesis explores how the need for multithreaded execution can be addressed as a compilation problem, to achieve switching rates approaching what hardware mechanisms might provide. Compiler-controlled multithreading is examined through compilation of a lenient parallel language, ID90, for a threaded abstract machine, TAM. A key feature of TAM is that synchronization is explicit and occurs only at the start of a thread, so that a simple cost model can be applied. A scheduling hierarchy allows the compiler to schedule logically related threads closely together in time and to use registers across threads. Remote communication is via message sends and split-phase memory accesses. Messages and memory replies are received by compiler-generated message handlers which rapidly integrate these events with thread scheduling.

To compile ID90 for TAM, we employ a new parallel intermediate form, dual-graphs, with distinct control and data arcs. This provides a clean framework for partitioning the program into threads, scheduling threads, and managing registers under asynchronous execution. The compilation process is described and preliminary measurements of the effectiveness of the approach are discussed.

Previous to this work, execution of Id90 programs was limited to specialized architectures or dataflow graph interpreters. By compiling via TAM, we have achieved more than two orders of magnitude performance improvement over graph interpreters on conventional machines, making this Id90 implementation competitive with machines supporting dynamic instruction scheduling in hardware. Timing measurements show that our Id90 implementation on a standard RISC can achieve a performance close to Id90 on one processor of the recent dataflow machine Monsoon. It can be seen that the TAM partitioning presented in this thesis reduces the control overhead substantially and that more aggressive partitioning would yield modest additional benefit. There is, however, considerable room for improvement in scheduling and register management.

Acknowledgements

There is nothing more exciting than working in a small group of motivated people who want to revolutionize the world. 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, Professor John Wawrzynek, Thorsten von Eicken, Seth C. Goldstein, Mike Flaster, Meltin Bell, and Anurag Sah.

I am especially thankful to my advisor, Professor David Culler, for his support. He provided me with more ideas than I could implement, asked more questions than I could answer, and helped me write the thesis. Together we developed dual graphs and explored what really is behind dataflow.

I am grateful to Professor David Patterson, my second reader, for teaching me a quantitative approach to computer architecture. He carefully read my thesis and gave many helpful comments.

This work would not have been possible without the various implementations of our threaded abstract machine. I want to thank Thorsten von Eicken, our Unix wizard and my office mate, for producing a stable nCUBE backend, Seth C. Goldstein for his user friendly C backend, Mike Flaster and Ethan Bernstein for their fast native MIPS backend, as well as Anurag Sah for his shared memory Sequent implementation.

I would also like to thank all the 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.

Finally, I want to thank Krste Asanović for being a perfectionist. I learned my lesson: working with you takes twice as long.

My deepest appreciation goes to Martina, my wife, for her patience, love, and support, and to my little daughter, Natalie, for her impatience, love, and need for support. I know, the world is terrible when Daddy has to work.

Für Martina und Natalie

Contents

List of Figures

List of Tables

Chapter 1

Compilation Challenge

Multithreaded execution appears to be a key ingredient in general purpose parallel computing systems. Many researchers suggest that processors should support multiple instruction streams and switch very rapidly between them in response to remote memory reference latencies or synchronization[AI87, Smi90, HF88, ALKK90, ACC+90]. However, the proposed architectural solutions make thread scheduling invisible to the compiler, preventing it from applying optimizations that might reduce the cost of thread switching or improve scheduling based on analysis of the program. Inherently parallel languages, such as Id90[Nik90] and Multilisp[Hal85], require that small execution threads be scheduled dynamically, even if executed on a single processor[Tra88]. Traub's theoretical work demonstrates how to minimize thread switching for these languages on sequential machines. However, in compiling this class of languages for parallel machines, the goal is not simply to minimize the number of thread switches, but to minimize the total cost of synchronization while tolerating latency on remote references and making effective use of critical processor resources, such as registers and cache bandwidth. This thesis addresses this three-fold goal in compiling Id90 for execution on a threaded abstract machine, TAM, that exposes these costs to the compiler through explicit scheduling and storage hierarchies. TAM can be efficiently implemented on standard sequential and parallel machines,

Both the compilation of sequential languages for standard architectures[AU77] as well as the compilation of non-strict parallel languages for dataflow machines[Tra86] is well understood. The storage model of conventional machines is directly reflected in most sequential languages. The control structures provided by those languages can be easily mapped to standard processors which follow only one path of execution.

Non-strict parallel languages, such as Id90, allow functions to execute and possibly return results before all arguments have been provided. This requires dynamic scheduling. Non-strict languages can be implemented efficiently on dataflow and multithreaded architectures, because those machine provide hardware support for fast dynamic synchronization and scheduling of threads of computation. Execution of these languages on standard machines was long viewed as inefficient, and special architectural support was thought to be essential.

Our approach shows how multithreaded execution can be addressed as a compilation problem, rather than one that needs special architectural solutions. We have designed an abstract threaded machine, which we use as an intermediate form in the compilation of Id90 to standard machines (see Figure 1.1). TAM provides simple primitives to the compiler for scheduling, synchronization and fast message transmission. Although it would be possible

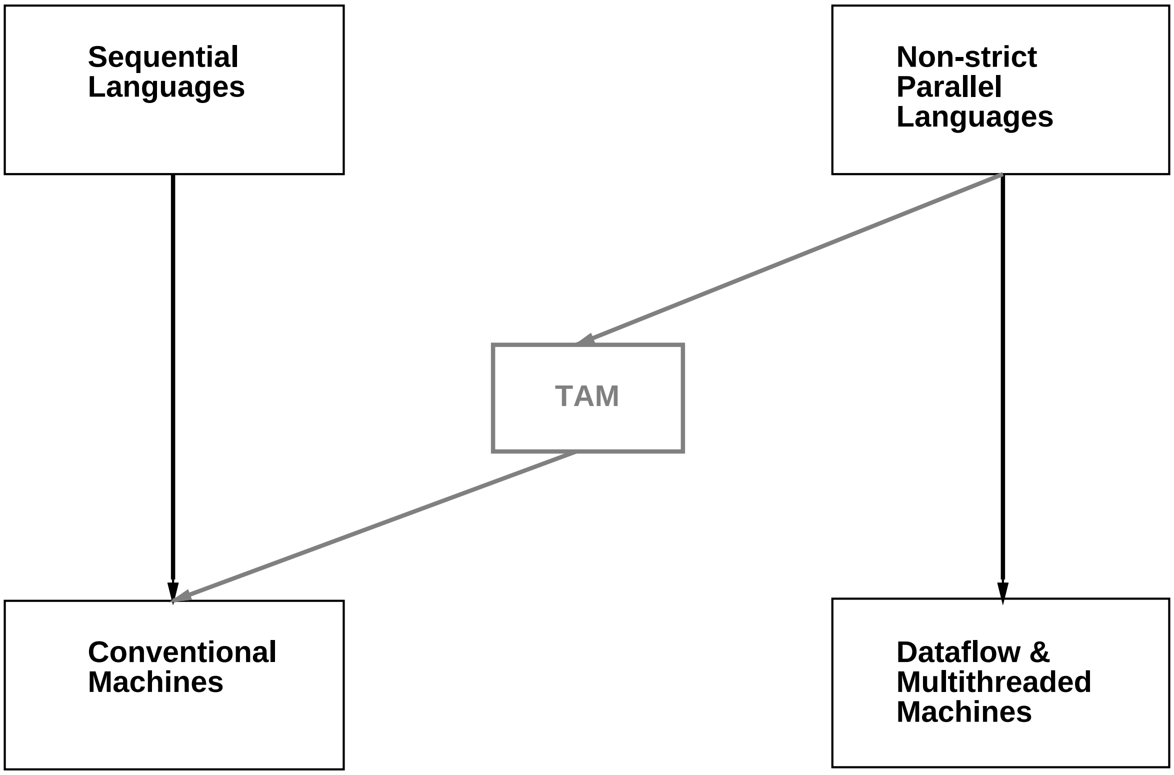

Figure 1.1: Compilation Paths.

to build TAM directly in hardware, it can be translated efficiently to standard parallel and sequential machines. The advantage of TAM over other multithreaded approaches is that it allows the compiler to exploit locality among threads and integrate thread scheduling, processor register assignment and message handling in code generation. Using TAM, compiling Id90 to standard machines, although still challenging, is doable and results in efficient code as our implementation demonstrates.

1.1 Overview

This thesis aims to demonstrate how to compile non-strict parallel languages for a multithreaded execution model. In this chapter we first analyze why non-strict parallel languages offer so much parallelism and discuss why they are difficult to implement on standard machines. We present the extended functional language Id90, which is taken as a starting point. We then discuss the main features of TAM, the threaded abstract machine into which we compile. In Chapter 2, we introduce a new intermediate form, called dual graphs, which provides a vehicle for integrated treatment of thread scheduling and register usage. In dual graphs, control and data dependences are separate, but stand on an equal footing. We show how dual graphs are produced for the basic constructs of the language. Chapter 3 then presents partitioning, the fundamental step in compiling a lenient language for a machine that executes instruction sequences. The task of the various partitioning algorithms is to identify portions of the graph that can safely be mapped into threads. The actual thread generation, which includes frame slot and register assignment, instruction scheduling and thread scheduling, is discussed in Chapter 4. Finally, preliminary results are presented in Chapter 5.

1.2 Language Issues

We are using the parallel functional language Id90[Nik90] as a starting point. Id90 is a non-strict but eager language. Traub uses the term *lenient* for this class of languages[Tra88]. A language is called *non-strict*, if it may be necessary to start computation of a function body before all (or even any) arguments have been provided. Conversely, it is called *strict* if all arguments can be evaluated before calling a function. Non-strictness gives the programmer much more expressive power, but at the expense of implementation and execution cost: non-strict languages require dynamic scheduling of computations, they cannot be completely scheduled statically, as examples given below will show. A language is called *lazy* if it starts computations only when it needs their result, *e.g.* a function would start evaluation of its arguments only by demand. The language is *eager* if all evaluations are started as soon as possible. It is the responsibility of the programmer to avoid divergent computations.

Functional languages are a good basis for a parallel computing, since they offer ample opportunities for implicit parallelism. In pure (side-effect free) functional languages, arguments to a function call can be evaluated in parallel. Non-strictness can substantially enhance the parallelism by allowing functions and arbitrary expressions (even conditionals) to execute and possibly return results before all arguments have been provided. This results in an overlap of computation between the function body and the argument evaluation. Non-strict, but eager, parallel functional languages, such as Id90, exhibit a large amount of parallelism on all levels[ACM88, Cul90, AE88]. Id90 would would exhibit less parallelism if the language were lazy, because then evaluation of arguments could only be started when functions need the corresponding value. This tends to lengthen substantially the critical path, *i.e.*, the time from the start of the computation to when the result is produced.

We begin with several examples in Id90 to indicate the subtlety of compiling such a language and the need for multithreading. These are not intended to be indicative of important applications, but serve to demonstrate the compilation principles. The function `lookup_array`, as shown in Figure 1.2, takes as arguments an array `A` of values and an ordered table `T` and returns an array of the table indexes corresponding to the values in `A`. There is very little parallelism in the `lookup` function, however, all the lookups can be performed in parallel. Each access to `T[m]` in the lookup may require a remote access or may even suspend, if the table `T` is still being produced by some other part of the program. Thus, we want to execute several lookups on each processor and be able to suspend and resume them cheaply upon remote or deferred access. `Flat` produces a list of the leaves of a binary tree using accumulation lists which are constructed with the build-in function `cons`. If `cons` and `flat` were strict, this code would exhibit no parallelism. Under lenient execution, the entire list is constructed in parallel[Nik91]. Simulations showed that with non-strictness the critical path on an example of a full binary tree of depth 10 constists of 250 time steps; with a maximum parallelism of 1776 and an average parallelism of 266 instructions (assuming the resources were available). If executed strictly, the critical path would grow to 26,650 time steps, with a maximum parallelism of 4 instructions. The contrived function `two_things` returns a pair containing the square of its first argument and the product of its two arguments. It can compute and return `x*x` before `y` has been evaluated, which enhances parallelism. In fact, it must be able to do so, since the first result could be used as the second

```
def lookup_array A T = {(al,ah) = bounds A;
                         (tl,th) = bounds T;
                        in {array (al,ah) of
                             [i] = (lookup tl th T A[i]) || i <- al to ah}};

def lookup l h T v = {while l < h do
                       m = div (l + h) 2;
                       next l, next h = if (v <= T[m]) then (l,m)
                                                        else (m+1,h)
                       finally l};

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

def two_things x y = (x*x, x*y);

def cube x = {a,b = two_things x,a;
              in b};
def strange x p = {a,b,c = if p then bb, x, aa else x, aa, bb;
                   aa = 3*a;
                   bb = 4*b;
                   in c};
```

Figure 1.2: Small Id90 programs

argument, as in the unusual function `cube`. The final example, due to Traub[Tra88], shows a cyclic dependence through a conditional that must be resolved dynamically. Because of the three mutually recursive bindings, no top to bottom static ordering of the statements yields a correct execution.

None of these examples present problems for a machine with dynamic instruction scheduling, such as Monsoon[PC90]. At the same time, none require dynamic scheduling throughout. Thus, it makes sense to investigate hybrid execution modes[Ian88a], where statically ordered *threads* are scheduled dynamically. Our TAM model takes this idea one step further by exposing the scheduling of threads to the compiler as well, so even the dynamic scheduling is done without hardware support. This allows register management to be closely tied to thread scheduling in order to minimize the overhead where dynamic scheduling is required. In the following, we use the `lookup` function to illustrate the compilation techniques.

1.3 TAM

To investigate compiler-controlled multithreading, a simple threaded abstract machine (TAM) has been developed. Synchronization, thread scheduling and storage management are explicit in the machine language and, thus, exposed to the compiler. TAM is presented elsewhere[CSS+91, vESC91]; in this section we describe the salient features of TAM as a compilation target. The primary design goal in TAM is to provide a means of exploiting locality, even under asynchronous execution, to minimize the overhead of multithreading.

TAM also provides a separation of concerns, since to execute programs compiled to TAM we expand the TAM instructions into machine code for a particular target architecture in a separate compilation step.

A TAM program is a collection of *code-blocks*, typically representing functions in the program text. Each code-block comprises several *threads* and *inlets* described below. Invoking a code-block involves allocating an *activation frame* to hold its local variables, depositing argument values into the (possibly remote) frame and enabling threads within the code-block for execution in the context of the frame. Since an activation does not suspend when it invokes a subordinate, the dynamic call structure is represented by a tree of activation frames, rather than a stack. Instructions in a thread may refer to slots in the current frame or processor registers. A frame is said to be *resident* when a processor is executing threads relative to the frame. Once a frame is made resident, it remains resident as long as it has enabled threads. A *quantum* is the set of threads executed during a single residency.

A thread is simply a sequence of instructions; it contains no jumps or suspension points; synchronization occurs only at the top of a thread. TAM control primitives initiate or terminate threads. *Fork* attempts to enable a thread in the current activation. The *stop* instruction terminates its thread and causes some other enabled thread to begin execution. Conditional execution is supported by a *cfork* operation, which forks one of two threads, based on a boolean operand. Merging of conditionally executed threads is implicit, since the arms of a conditional can both contain a fork to a common thread.

In essence, each thread begins with a multiway join. A thread may have associated with it a frame slot containing its *entry count*. The entry count is initialized prior to any fork of the thread with the number of control paths joining. Each time the thread is forked, its entry count is decremented. The thread is enabled when the entry count reaches zero. Threads with a single entry need no associated entry count slot.

This Fork-based control paradigm is somewhat unusual and merits discussion. Fork generalizes branch, since a branch can be simulated by a fork followed by a stop. When fork and stop are separated by a sequence of instructions, it serves as a generalized delayed branch. On a machine that executes multiple threads concurrently, fork generates parallel activity; otherwise, it builds up a queue of threads that can be run while waiting for long latency requests and synchronization events to complete. Architecturally, providing fork, rather than branch, is interesting because it allows instruction fetch and execution to be decoupled without branch prediction. From a compiling viewpoint, mixing forks and branches presents very tricky code-generation issues[Tra88], especially in handling "non-strict" conditionals where computations must started even if not all inputs to the conditional are available. It should be noted, however, that when mapping TAM to native code for existing machines fork and stop are removed whenever possible to produce a branch or fall-through.

TAM assumes that an activation will execute on a single processor; so work is distributed over processors at the activation level. Thus, passing arguments and results between frames represents (possible) interprocessor communication. The *send* operation delivers a sequence of data values to an *inlet* relative to the target frame. An inlet is a restricted thread that primarily serves to extract data from a message, deposit it into specific slots in the designated frame, and fork threads for the corresponding activation. Inlets are compiler generated message handlers that avoid message parsing. An inlet may interrupt a thread, but it does not disturb the current quantum. Threads enabled by executing the inlet will run when its frame becomes resident.

TAM provides a specialized form of send to support split-phase access to data structures.

The heap is assumed to be distributed over processors, so access to a data element may require interprocessor communication. In addition, accesses may be synchronizing, as with I-structures[ANP87] where a read of an empty element is deferred until the corresponding store takes place. The I-Read, I-Fetch, and I-Take operations generate a request for a particular heap location and the response is received by an inlet. Meanwhile, the processor may continue with other enabled threads or, if none remain, will start working on another activation frame.

Scheduling in TAM is primarily under compiler control and is tied closely to the storage hierarchy. The first level of scheduling is reflected by the static partitioning of instructions into threads and ordering within threads. Values defined and used within a thread can be retained in processor registers. The next level of scheduling is dynamic — a quantum. Threads enabled by *fork* or *cfork* operations execute within the same quantum as the fork. However, the order of execution of the threads is not determined. Values can be transmitted in registers between threads in a quantum. When no enabled threads remain, another activation with enabled threads must be made resident. This is also under compiler control. The scheduling queue is contained within the frames, and the last thread executed in a quantum, called the *leave thread*, includes code to locate the next activation and fork to a designated *enter thread* within that activation. Registers can be used to carry values between threads within a quantum. Empirical studies show that quanta are often large, crossing many points of possible suspension[CSS+91]. Thus, it is advantageous for the compiler to be able to keep values in registers between threads that it cannot prove will execute in a single quantum. The leave and enter threads can save and restore specific registers if the guess proves incorrect.

The task of compiling for TAM has two aspects. First, a program must be partitioned into valid threads. This aspect is constrained partly by the language and partly by the execution model. The language dictates which portions of the program can be scheduled statically and which require dynamic synchronization. An elegant theoretical framework for addressing the language requirements is provided by Traub's work[Tra88]. The execution model places further constraints on partitioning, since synchronization only occurs at the entry to a thread and conditional execution occurs only between threads.

The second aspect is management of processor and storage resources in the context of dynamic scheduling to gain maximum performance. This involves analysis of expected quantum boundaries, frame and register assignment under asynchronous thread scheduling, and generation of inlets.

1.4 Id90-to-TAM Compilation Stages

Figure 1.3 gives an overview of the structure of the Id90 to TAM compiler. Id90 programs are first translated into program graphs by an Id90 front-end from MIT[Tra86]. Program graphs are a hierarchical graphical intermediate form with only one kind of arc. Program graphs 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, so control flow is implicitly prescribed by the dynamic propagation of values. In our threaded execution model, control is explicit and the flow of data is implicit in the use of registers and frame slots. In order to bridge this gap, we introduce a new graphical intermediate form, *dual graphs*, in which control and data flow are both explicit.

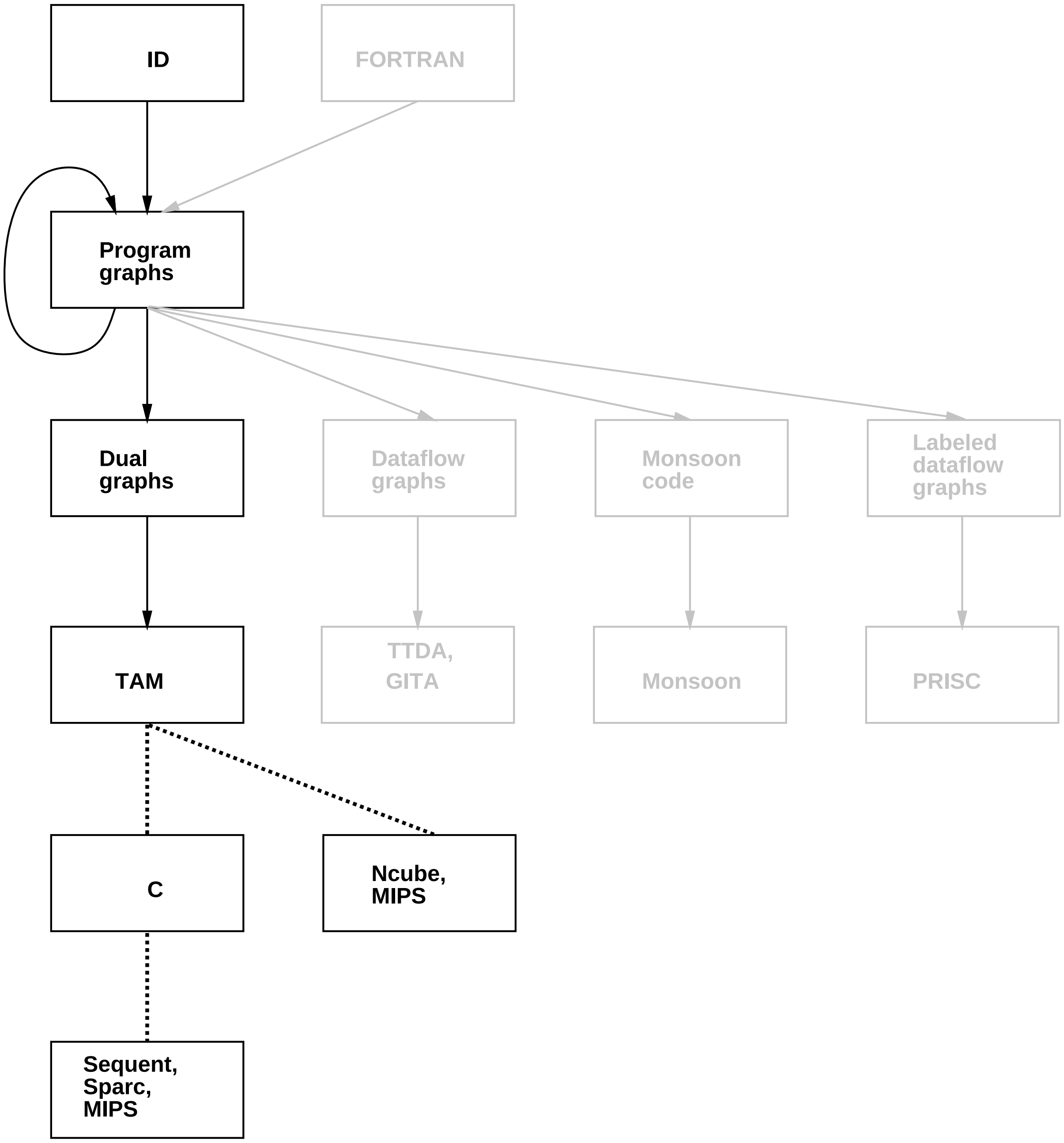

Figure 1.3: Overview of Id90 to TAM compiler. Shown in gray are other compilation approaches.

Dual graphs are generated by expanding dataflow program graph nodes into equivalent dual graphs. This is a local transformation and can be achieved by following expansion rules for the individual program graph nodes. Compilation to TAM then involves a series of transformations on the dual graph, including partitioning, lifetime analysis, scheduling and linearization, register and frame-slot allocation and fork insertion. Finally TAM code is produced. All these steps, together with dual graphs, will be described in more detail in the next chapters.

TAM is left intentionally abstract, so that it could be implemented on sequential machines, shared-memory machines as well as on message passing machines. TAM is concrete enough though that it can be translated with ease in a second translation step to actual machines. One translation path has chosen C as a portable "intermediate form" and is producing code for parallel machines like the Sequent Symetry[Sah91] and Motorola Delta, as well as for various standard sequential machines. Other backends translate TAM directly into machine code. Currently there exists backends for the MIPS processor and for the parallel Ncube/2.

The figure also shows in gray other compilation approaches that chose program graphs as their intermediate form. The original MIT compiler translated program graphs into dataflow graphs which then could 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 very interesting approaches compile FORTRAN programs into representations similar to program graphs[BP89, FOW87].

Chapter 2

Dual Graphs

Compilation of Id90 to TAM begins after generation of dataflow program graphs. The meaning of program graphs is given in terms of a dataflow firing rule, so control flow is implicitly prescribed by the dynamic propagation of values. In our threaded execution model, control is explicit and the flow of data is implicit in the use of registers and frame slots. In order to bridge this gap, we introduce a new graphical intermediate form, *dual graphs*, in which control and data flow are both explicit. The dual graph thus tries to capture both the contents of a conventional control flow graph as well as those of a pure dataflow graph. Dual graphs are similar in form to the data structures used in most optimizing compilers; the key differences are that they describe parallel control flow and are in static single assignment form[CFR+89].

Compilation to TAM involves a series of transformations on the dual graph, including partitioning, fork and join insertion, lifetime analysis, register and frame-slot allocation, scheduling and linearization.

In this chapter, we will first present dual graphs and then explain how program graphs can be expanded into dual graphs. The expansion is described by presenting expansion rules for the individual program graph nodes. This description is quite long and is intended only for the interested reader. Other readers can skip that section and continue directly with the final section which presents the dual graph expansion for our little `lookup` example.

2.1 Dual Graph Definition

A dual graph is a graphical intermediate form in which both control flow and dataflow are explicit. It is a directed graph with three types of arcs: data, control and dependence. Dependence arcs capture indirect control and data. Each node has some number of ordered inputs and outputs, that are also divided into the three classes: data, control and dependence. An arc must always be of the same kind as the input and the output it connects. There is no restriction on the number of arcs emanating from an output, but each input can have at most one arc feeding it.

The dual graph arcs, shown in Figure 2.1, have the following meanings.

- **Data Arcs:** A data arc $(u, o) -- \rightarrow (v, i)$ specifies that the value produced by output o of node u is used as operand i by node v. A node may have several data output ports; each port represents a name (*i.e.* a memory location) to which a value can be bound and read. There can be multiple consumers, but at most one producer

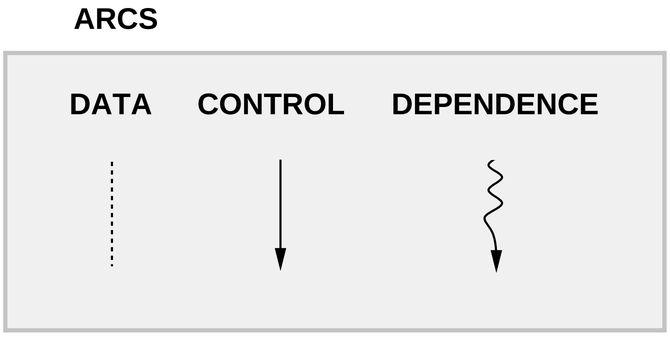

Figure 2.1: Dual Graph Arcs.

- **Control Arcs:** A control arc $u \to v$ specifies that instruction u will execute before instruction v, and instruction u has direct responsibility for scheduling v. A node may have one or more control output ports, each with a bundle of control arcs.

- **Dependence Arcs** (split-phase long-latency arcs): A dependence arc $u \leadsto v$ specifies that instruction u will execute before instruction v but that v will be scheduled as an indirect consequence of executing u. Being long latency arcs, they also indicate that the two nodes cannot be in the same thread.

Dual-graphs have well-defined operational semantics and can be executed directly. Control can be represented by tokens traveling along the control arcs. A node fires when control tokens are present on all its control inputs.[1] Upon firing, a node computes a result based on data values bound to its data inputs, binds the result to its data outputs, and propagates control tokens to its control outputs. In correct dual graphs, control will only appear on control inputs if corresponding data inputs have been produced. It is the task of the compiler to ensure this. As shown in Figure 2.2, there are eight types of node: simple, join, switch, label, merge, outlet, inlet and constant nodes.

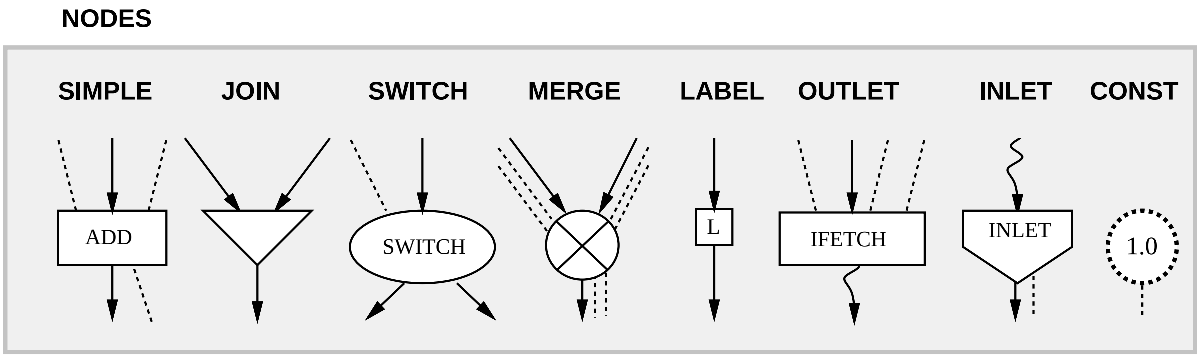

Figure 2.2: Dual Graph Nodes. (Arcs: dash = data, solid = control, curly = dependence)

- **Simple nodes:** All the arithmetic and logic operations are described by *simple nodes*. A simple node has a single control input and a data input for each operand. It has a single control output port (the *successors*) and typically a single data output port

[1] The merge nodes are the only exception to this rule

(the *result*). When control passes to a simple node, it reads the value bound to its data inputs, performs its operation, binds the result to the data outputs and produces a new token on the control output arcs.

- **Joins:** A *join* synchronizes control paths; it has multiple control inputs and a single control output port. This node will pass a control token to its control output once every control input has received a token.

- **Switches:** A *switch* has a control input and a data input called the *predicate*. A switch will pass control on to one of its control outputs depending on the value of the predicate.

- **Merges:** The *merge* has multiple matching input sets, each with a control input and zero or more data inputs. The output ports have the same topology. Merge complements the switch by steering control from any one of the control inputs to a single control output. Thus it has a special firing rule and is the only node which is not strict in all of its control inputs. Additionally, the merge unifies the corresponding data inputs to its data outputs, *i.e.* data is bound to the output name. In the final code generation this may require moves.

- **Labels:** A *label* has one control input port and one control output port. It is used to indicate that the adjacent nodes must be in distinct threads *i.e.*, it represents a *separation constraint.* (In generating dual graphs, a label is placed on each output of a switch, reflecting the fork-based control primitives in TAM.)

- **Outlet nodes:** Operations that send messages or initiate requests are described by *outlet* nodes. These have an effect external to their code block. An outlet has a single control input, a data input for each operand, and a dependence output connecting it to inlet nodes.

- **Inlets:** *Inlet* nodes receive messages, including arguments, values returned from a call, and responses to split-phase I-structure operations. An inlet may have a dependence input; it has a control output and zero or more data outputs. An inlet has no control inputs. It generates control and data as a result of some external message. An inlet represents a compiler generated message handler. It receives values corresponding to the data outputs and passes control to the operations connected to its control outputs. Usually its dependency input is connected to the dependency output of a node that sends the request (*e.g.* an IFETCH). The inlet then represents the answer coming back.

- **Constants:** Constant nodes represent constants. They have only a data output and neither a control input nor a control output, since their value is known at compile time.

Each node can have associated properties. Besides the opcode, the most important is whether a node is 'side-effecting' or not. Sends and I-Stores are examples of side-effecting nodes; the effects are not represented directly in the graph.

We now discuss in more detail what dependence arcs represent. They are used to associate a requester (*e.g.* an I-Fetch) with all of its local receivers (Inlets). This relationship between requester and receiver is shown in Figure 2.3, using the example of an I-fetch connected to an Inlet. When issuing the request, the I-fetch specifies what I-structure element to fetch, where the result should be placed and what subsequent computation should be started. This is achieved by naming the receiving inlet which is the message handler that knows where to store the data and what computation to start. The I-structure handler will send the data value back to the inlet. Therefore, dependence arcs represent both a special form of control and data arc. It represents the data send over the network and also specifies indirect control: the receiver will execute (but not necessarily immediately) as a consequence of sending of request. The partitioning stage of the compiler makes use of the control information in dependence arcs to eliminate redundant control arcs. Code generation needs the data information, the name of the receiving inlet is one of the requester's operands.

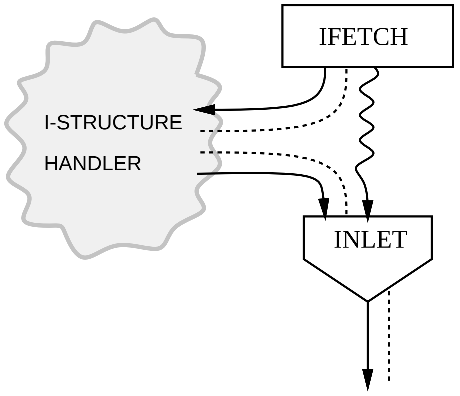

Figure 2.3: Dependence arcs connect requesters to receivers.

2.2 Generation of dual graphs

Dual graphs are generated by expanding dataflow program graph instructions into equivalent dual graphs. This is a local transformation and can be described by giving the expansion rules for the individual program graph nodes. Program graphs are a hierarchical graphical intermediate form and described in [Tra86]. The lowest level is represented by basic operations. Larger programs can be built up using program graph encapsulators, including conditionals, loops, function definition, and application. The meaning of program graphs is given in terms of a dataflow firing rule, so control flow is implicitly prescribed by the dynamic propagation of values.

Program graph arcs carry tokens which always represent both a data value and control. Sometimes, as in the case of signals and triggers, we are only interested in the control component. Often, the control information carried by a token is redundant, as the example in Figure 2.4 shows.

Here the control information between `b` and `d` is redundant since we know that `b` is also needed to compute `c`. With the expansion of program graphs into dual graphs, program graph arcs will be represented by two separate arcs: a data arc and a control arc. This allows us to to eliminate them separately if one of them is not necessary. In the previous example, we could have identified and eliminated the redundant control arc from `b` and only retained the corresponding data arc, without having changed the semantics of the original program. Note that, program graph arcs are never transformed into dependence arcs, since those only appear inside a program graph node.

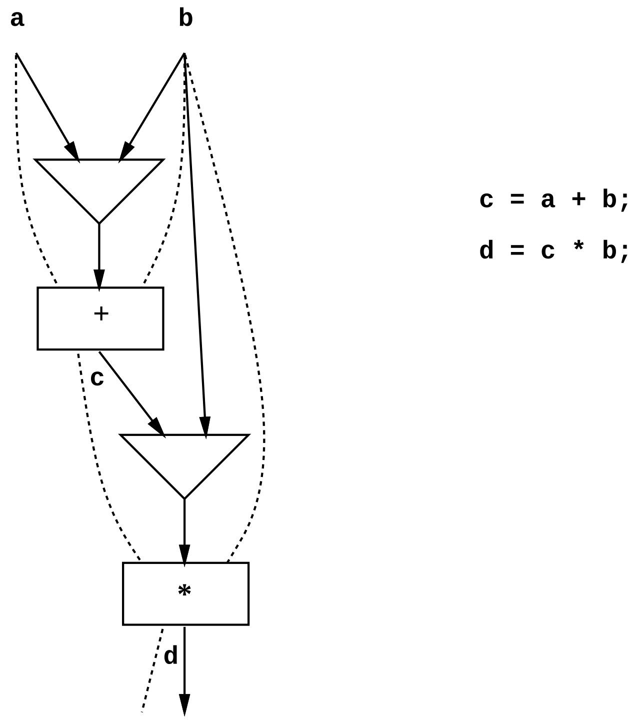

Figure 2.4: Example with redundant control arc.

After presenting the expansion of the basic program graph nodes, we will show the expansion of more complicated encapsulators: conditional and loop encapsulators, function definitions and applications. Finally, we will show how the program graph node expansions are combined to get a single dual graph.

The description of the expansion rules is quite long and can be skipped. The casual reader can continue with the final section which presents the dual graph expansion for our little `lookup` example.

2.2.1 Basic Operators

The expansion of basic unary and binary operators is straightforward as shown in Figures 2.5 and 2.6. The program graph instructions are represented by the grey area; the nodes and arcs inside are the dual graph equivalent. The inputs and outputs of the program graph nodes are shown with their names. Basic program graph nodes are strict in all their inputs. They fire when a token arrives on all inputs.

As can be seen in the case of the unary operation, an arc at the program graph level will usually be represented by two arcs at the dual graph level. In the case of the binary instruction, a control arc will be connected from each of the program graph inputs to a `join` node which will be connected to the control input port of the dual graph operator. This instruction will be executed once control has arrived on both control inputs, thus ensuring that the operands bound to the two data arcs are valid. Then the result is computed, bound to the data output port and control is passed to the control output port.

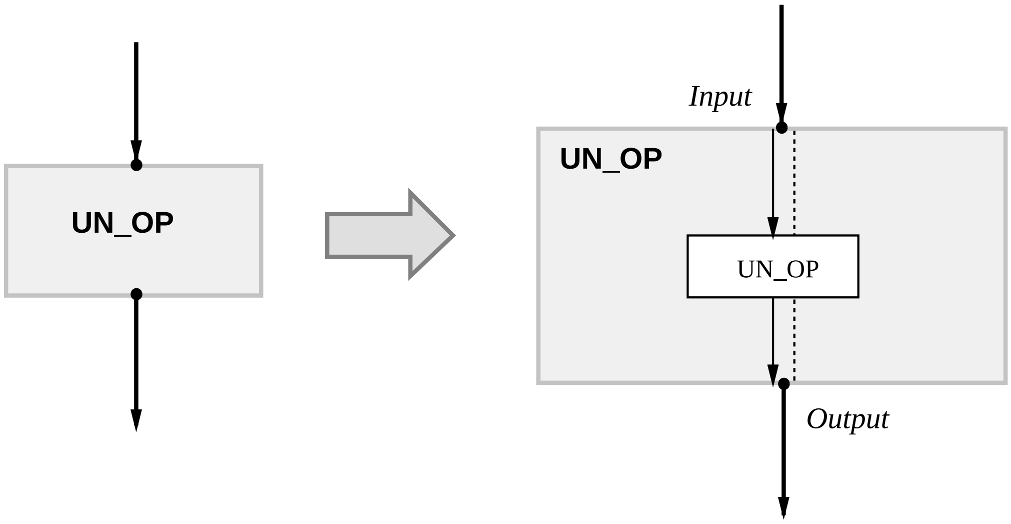

Figure 2.5: Dual graph expansion for unary instructions.

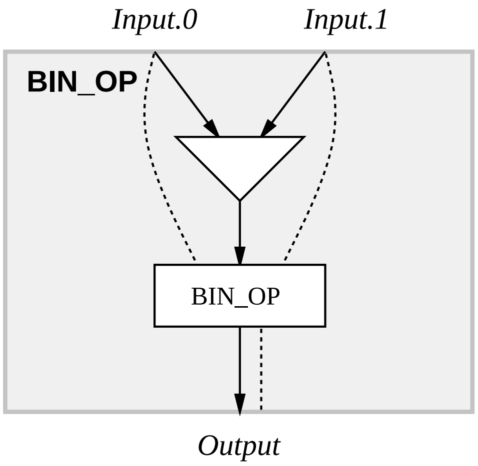

Figure 2.6: Dual graph expansion for binary instructions.

2.2.2 Literals

A literal program graph node, as can be seen in Figure 2.7 has only a single trigger input and an output for the value. When token arrives at the trigger input, the program graph node will produce a token containing the constant at its output. In the dual graph representation the input will be directly connected to the output by a control arc. The literal will be represented by a constant node which has only a data arc going to the program graph output.

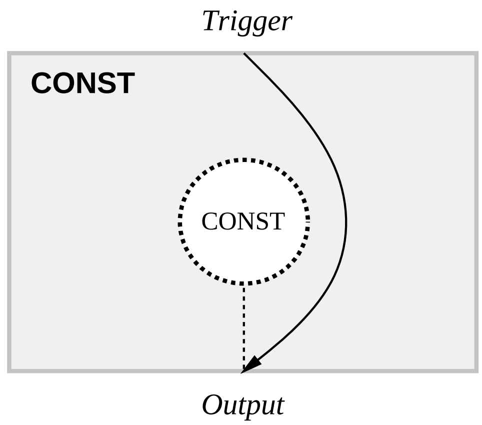

Figure 2.7: Dual graph expansion for literals.

2.2.3 Conditional

The translation for the conditional encapsulator is shown in Figure 2.8. Depending on the value of the token received at the predicate, the program graph conditional steers the tokens received at the data inputs either to the "then" side or to the "else" side. It also merges the results from the two sides into the output. The program graph conditional is only strict in the predicate, so it must be possible for one input to propagate through the conditional before other inputs have arrived. Therefore a `switch` node is needed for each input and a `merge` for each output. A `join` is placed in front of each `switch` to synchronize the corresponding input with the predicate. Since data and control flow are separated, the data will not be steered by the switch, as it is the case in the similar dataflow operation. The data arcs will go directly around the switch to the corresponding uses. Separating the control flow from the data flow, shows for all uses of values where they were produced. The task of the `merge` nodes is to pass control on from one of its inputs to the output of the conditional. It will also bind the values of the corresponding data inputs to the data outputs.

2.2.4 Loop

Figure 2.9 shows the expansion of a loop program graph encapsulator into a 1-bounded loop. The loop is a program graph schema for the while loop and encapsulates a predicate and a body. The predicate determines when the loop terminates, while the body will be executed for every iteration that the predicate evaluates to true. The loop has n circulating loop tokens. They enter at the beginning the loop inputs and go to the loop predicate.

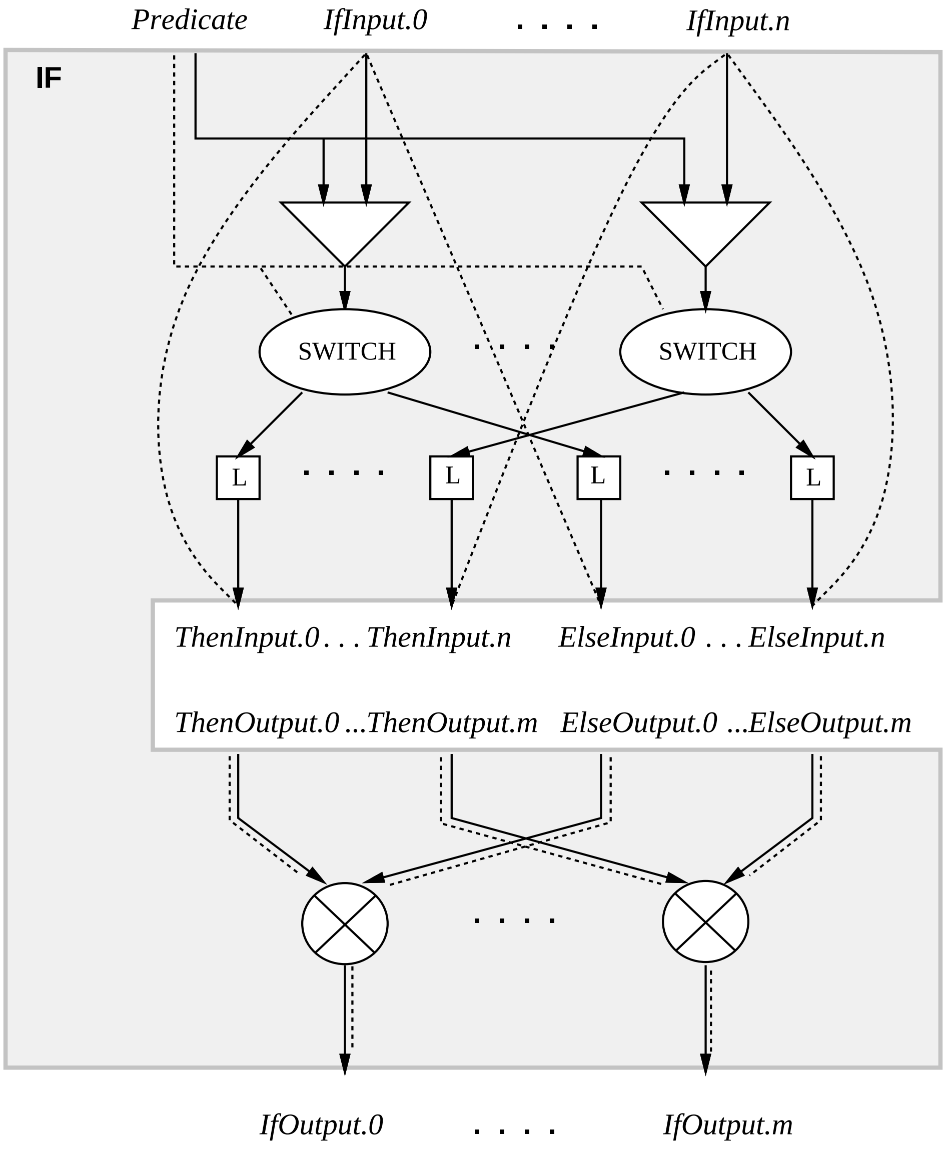

Figure 2.8: Dual graph expansion for conditionals.

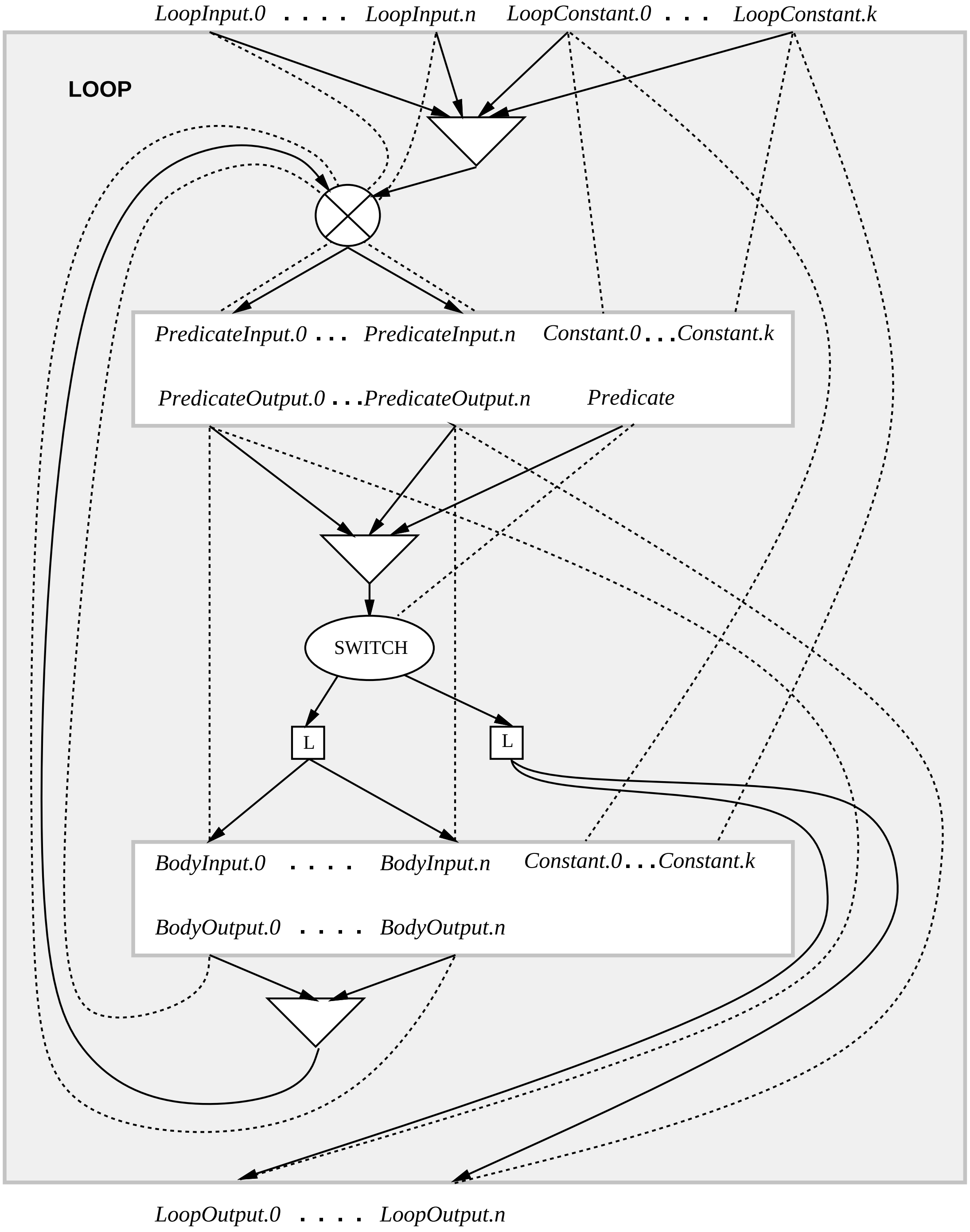

Figure 2.9: Dual graph expansion for the loop encapsulator.

Depending on the value of the predicate they will either be routed to the loop outputs (if the predicate evaluates to false), or to the loop body. Here the token for the next iteration will be computed. External variables used inside the predicate or the loop body, but which are not changed, are called loop constants. They enter through the loop constant inputs and go to the corresponding predicate and body inputs.

A loop is strict in all its inputs, *i.e.* an implementation is allowed to wait until all the loop inputs as well as all loop constants have arrived before starting it. A `join` synchronizes all these inputs. Control is then passed on to a `merge`, that will start the evaluation of the predicate. Data arcs connect the loop inputs to one side of the arcs, while the loop constant inputs are directly connected by data arcs to the predicate and body. The result of the predicate is connected to a `switch`, which decides whether to pass control on to the the loop body or whether to end the loop. All outputs of the loop body will be synchronized by another `join` which feeds the other control input of the `merge`. The `merge` also serves to unify the names of the inputs to the loop with the corresponding outputs of the loop body. This is being represented by the data arcs connected to each side.

2.2.5 I-Structure Operations

Figures 2.10 and 2.11 show the expansion for I-structure operations. I-fetch is used to fetch an element from a I-structure, while the I-store stores a value into an I-structure element. It might take a long time before the result from an I-fetch is available. The element might be fetched from some distant processor, or the fetch may be deferred waiting for the value to be stored. An I-fetch first needs to synchronize on both the structure pointer and the offset before initiating the request. The result of an `ifetch` returns into an `inlet`. Since the `ifetch` is a long latency operation, the `inlet` must be in a different thread. This is indicated by the dependence arc connecting these instructions.

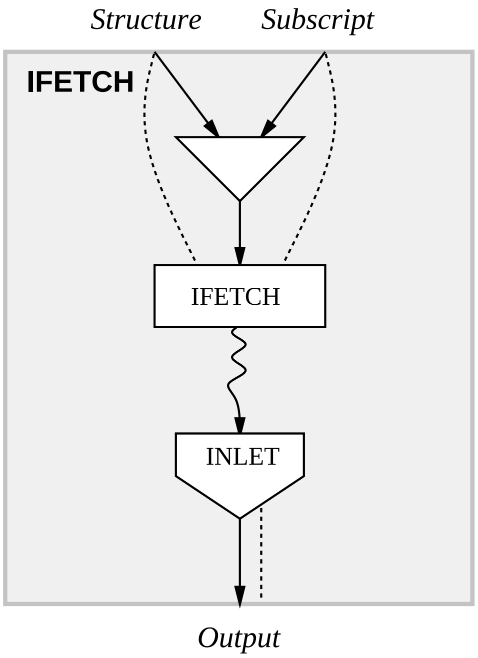

Figure 2.10: Dual graph expansion for I-structure fetch.

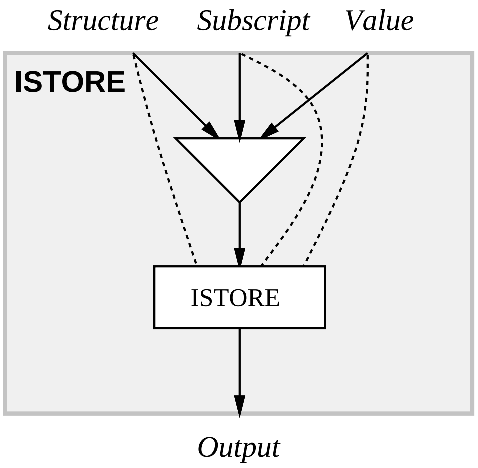

Figure 2.11: Dual graph expansion for I-structure store.

For the I-store operation, the `join` first synchronizes on the structure pointer, the offset, and the value to be stored. In our execution model, we do not expect the receiver to signal the store, therefore the control output of the `istore` is directly connected to the the signal output of the program graph node. For machines that allow messages to get out of order, it might be necessary to acknowledge the store. In this case the expansion of the I-store would look similar to the I-fetch; the `istore` would be connected by a dependence arc to an `inlet` which would receive the acknowledgement signal.

2.2.6 Signal tree

Our expansion to dual graphs starts with program graphs that have already been enhanced with triggers and signals. Triggers are necessary to start pieces of computation that do not receive some input. That is, for example, why program graph constants have a data input. Signals are used to detect when all computation in a region of code has terminated. Usually many individual signals will be connected to a signal tree. A signal tree serves to synchronize all inputs and produce a signal on the output. Since the program graph arcs feeding this instruction carry only control and no data information, our expansion will connect all inputs to a `join` using only control arcs (see Figure 2.12). The control output of the `join` is connected to the output of the signal tree.

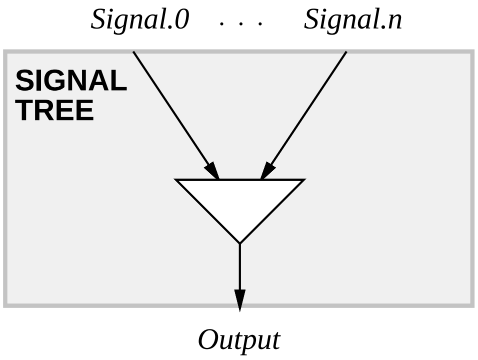

Figure 2.12: Dual graph expansion for signal trees.

2.2.7 Function Definition

Id90 allows functions to be non-strict in their arguments. Therefore it must be possible to receive each argument independently and start the computation dependent on the argument. A function may also need to be started without receiving any argument, therefore the program graph node has a special trigger output. Id90 provides higher order functions and allows the programmer to build closures by applying a function to some, but not all, of its arguments. A closure representation consists of the name of the procedure to be invoked, an integer indicating how many arguments must still be provided before the arity is satisfied, and a field containing all the arguments collected so far. The last field is also called the *chain*. In Id90 a function can return only a single result.

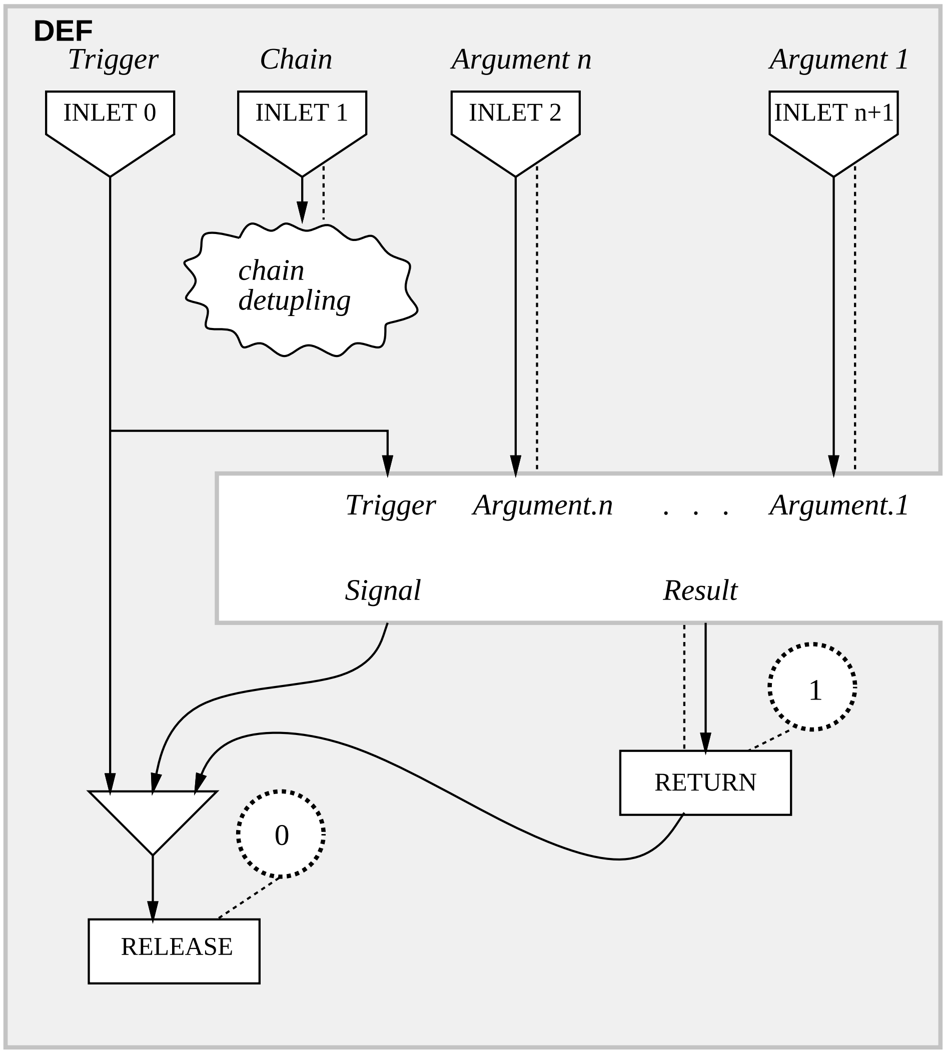

Figure 2.13: Dual graph expansion for function definitions.

The dual graph expansion for function definition program graph encapsulators is shown in Figure 2.13. For a function with n arguments, $n+2$ `inlet` nodes are needed for the dual graph representation; n are used for the arguments, the other two for the trigger and for the chain. A separate trigger `inlet` is needed because the function might have to be started

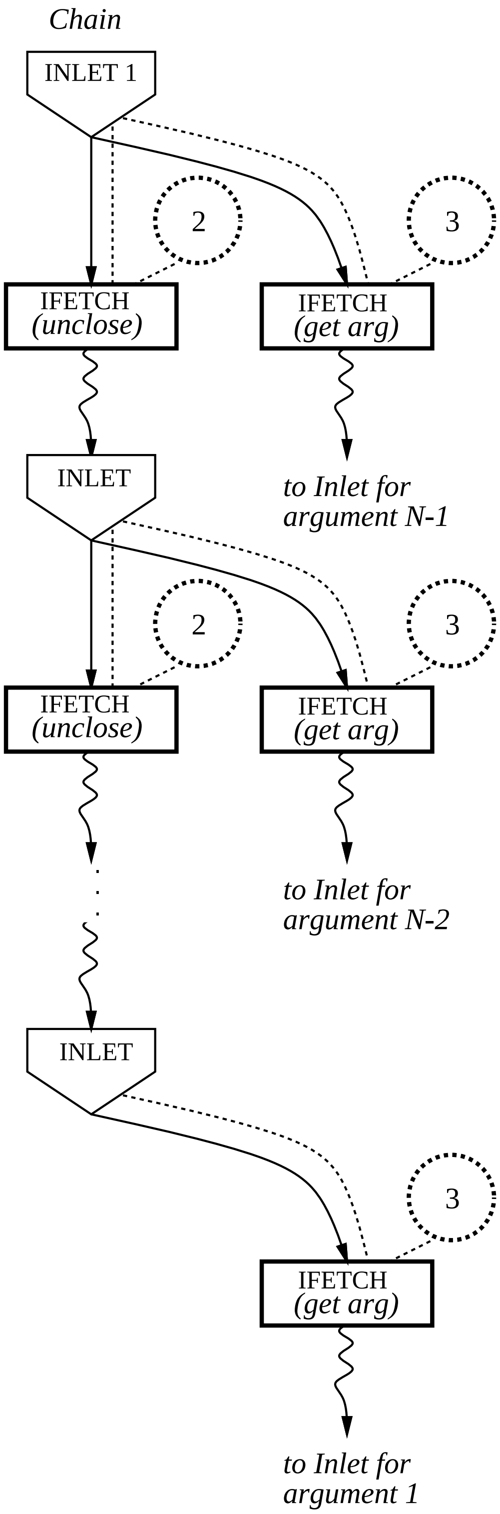

Figure 2.14: Dual graph expansion for chain detupling.

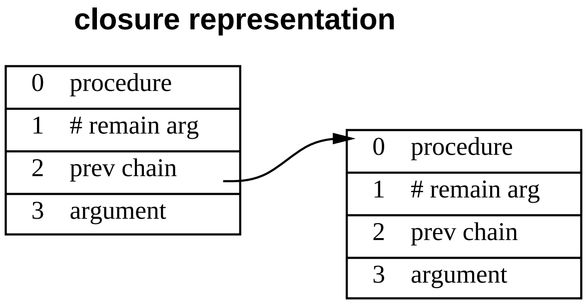

Figure 2.15: Closure Structure.

before any argument is provided. The chain that collects all arguments except the last will be sent to a special chain `inlet`, while the last argument will be directly sent to its `inlet`. There the chain will be "detupled" to get all the other arguments. Detupling the chain, which is shown in Figure 2.14, involves fetching both the argument and the next chain pointer until all arguments are available. Figure 2.15 shows our convention for building closures.

In our calling and argument passing convention the trigger for a function will be received by inlet 0, the chain by inlet 1, and the last argument by inlet 2. All the other arguments from $n-1$ down to the first will be received by inlet 3 through inlet $n+2$. So the argument i is received by inlet $n+2-i$. Each of the argument inlets defines a data value; it also has to start the computation dependent upon this value. Therefore both a data and a control arc will be connected to the corresponding argument port of the program graph.

The `return` instruction will send the result back to the caller. After synchronizing on the return value as well as on the signal, the function can release its frame and send a completion signal back to its caller.

2.2.8 Function Application

Figure 2.16 shows how a caller invokes a function. The inputs to the program graph node *apply* are the function (which could be any closure) as well as an argument. The outputs are the result and the signal. The apply first needs to test whether the arity of the function is already satisfied. Depending on the test it would either need to build up a new closure or, when the arity is satisfied, allocate a frame for the function. The *if* is a program graph conditional and will be expanded by the rule presented earlier. Allocating a frame, possibly on a different processor, is a split phase operation. The frame pointer will be received by an inlet and then the caller will send both the chain as well as the last argument to the corresponding inlets. The called function will return its result and the signal into the two inlets. Allocating the frame will automatically cause a trigger to be sent to the function trigger inlet. Since the return of the values are only known to be a result of the frame allocation, a dependence arc connects the frame allocation node to the result and signal inlet.

In the case where the compiler knows what function is to be invoked and the producers of its arguments a special program graph encapsulator called *direct apply* will be produced. As can be seen in Figure 2.17 the caller first needs to allocate a frame and then send all the arguments directly into the corresponding inlet of the function. The function may still be non-strict in these arguments, therefore it is necessary to separately synchronize each

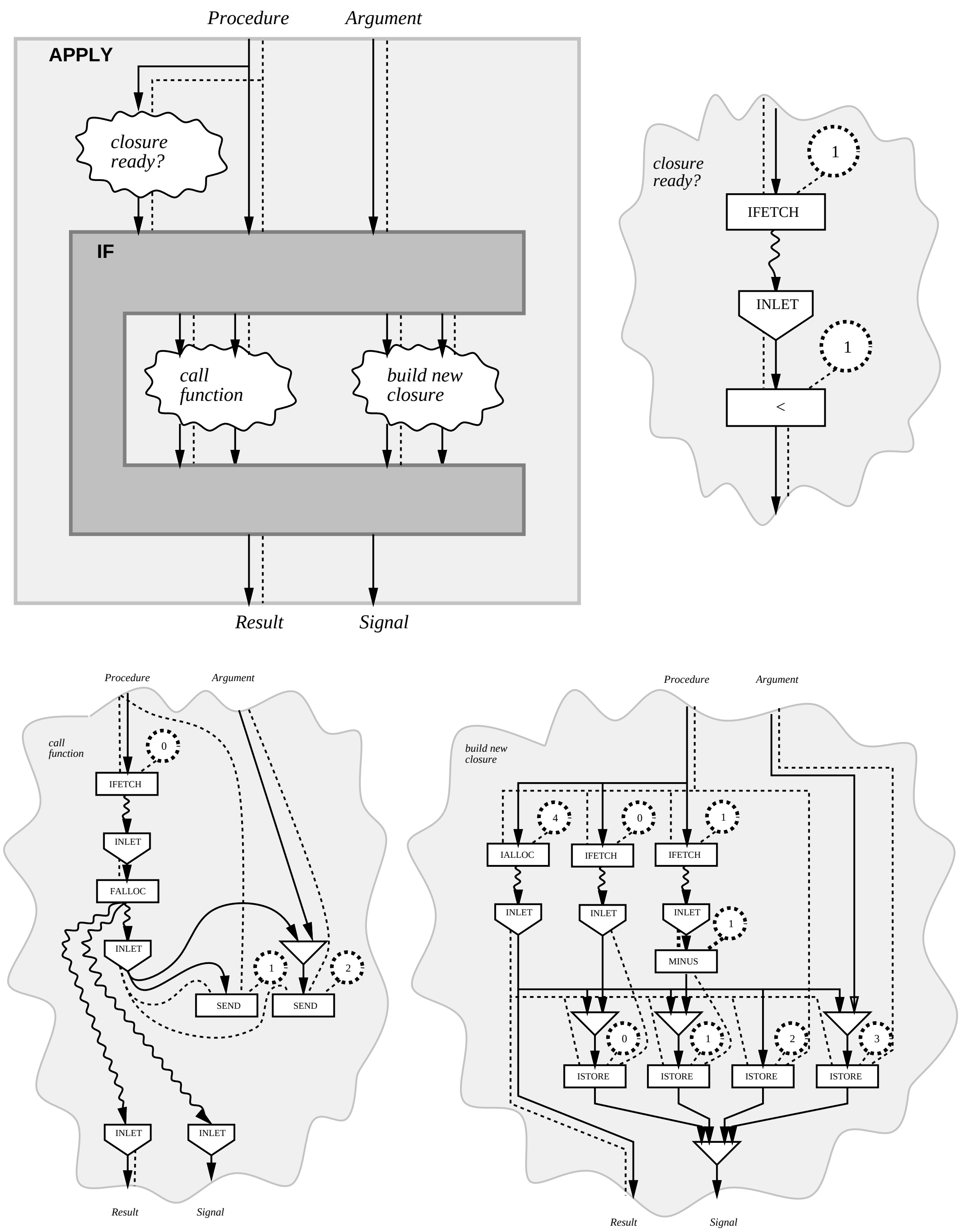

Figure 2.16: Dual graph expansion for function application.

argument with the reception of the frame pointer. As in the previous case, result and signal will be received by an inlet and a dependence arc connects the frame allocation node to these inlets.

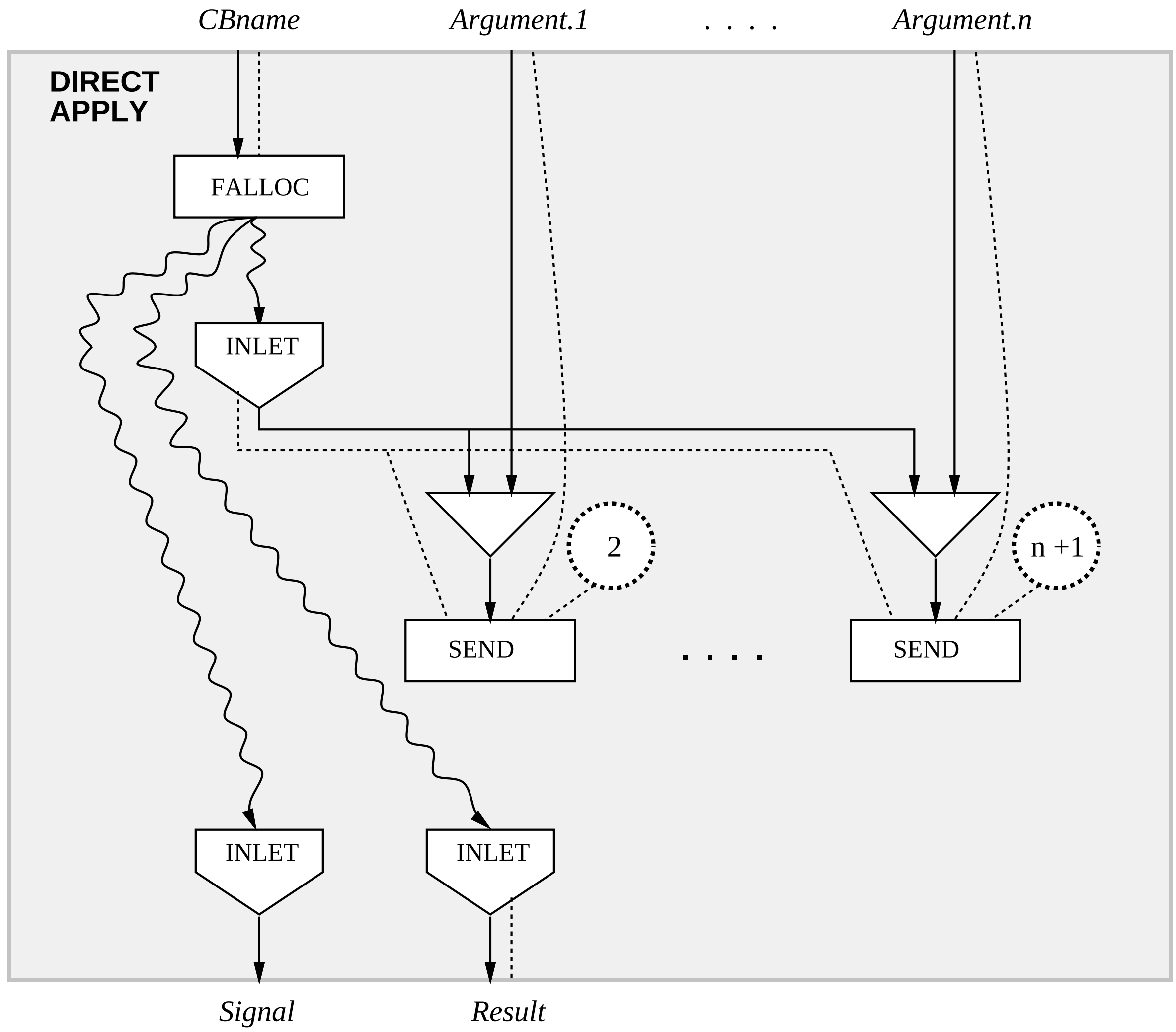

Figure 2.17: Dual graph expansion for direct apply.

With strictness analysis it might be possible to prove that some arguments are strict with respect to each other — they could be grouped together, making sending as well as receiving them cheaper. In general, this might require that we generate specialized function code blocks; we did not implement this optimization in our compiler.

2.3 Combining Dual Graph Expansions

Combining dual graph expansions, as in Figure 2.18, produces the complete dual graph equivalent to the original program graph. First, all program graph nodes are individually expanded into the dual graph equivalent, following the rules presented in the previous section. After this step, each program graph output port can have at most one control arc and one data arc feeding it. Now dual graph arcs emanating from program graph

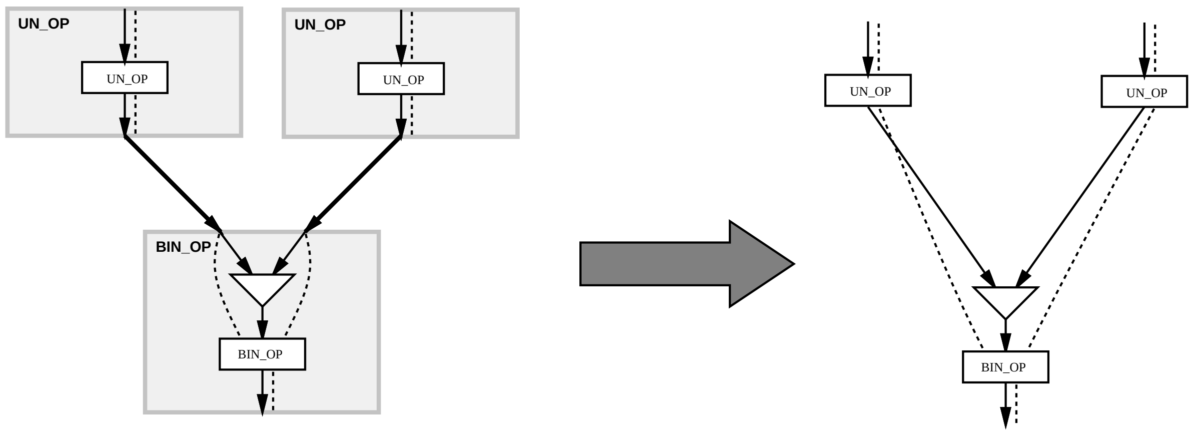

Figure 2.18: Combining dual graph expansions.

inputs are re-connected so that they directly emanate from the corresponding dual graph node (following backwards the corresponding program graph and dual graph arcs). All dual graph arcs still feeding program graph outputs are then eliminated, the previous re-connection step has made them unnecessary.

2.4 Example

The program graph for the `lookup` example from Figure 1.2 (page 9) includes a function DEF node, enclosing a LOOP node, enclosing an IF node, as shown in Figure 2.19. Figure 2.20 shows the corresponding dual graph representation, using a 1-bounded loop form, *i.e.* where only one iteration is active at once[Cul90]. The compiler actually produces a slightly more complicated dual graph; we have omitted the trigger inlet and its arcs. The four arguments enter at the inlet nodes at the top of the graph. The control outputs are joined before arriving at the merge. The function is strict in all its arguments. The data arcs for `l` and `h` connect to the merge nodes at the top of the loop. The other inlets are connected directly to their uses within the loop and the enclosed conditional, as are the data outputs of the loop merge. By separating the control and data arcs, the flow of information is not obscured by control constructs. In each iteration, control is directed to the loop body or exit based on the loop predicate. Within the body of the loop, the value of `m` is calculated and used in an I-fetch operation. The result of the I-fetch will eventually arrive at the inlet indicated by the dependence arc. This inlet feeds the conditional predicate, which controls three separate switches, one for each data value used in the conditional. The three switches cause the correct values to be routed to the merges to produce the next iteration values of `l` and `h`. The third merge has only control inputs and serves to indicate that all the switches have executed. Control is joined once again at the bottom of the loop and directed to the loop merge.

The dual-graph for an Id90 program could be executed directly, but the number of dynamic synchronizations per useful operation is very high. The compilation goal is to minimize this cost by employing the cheapest form of synchronization available in the synchronization hierarchy provided by TAM. The cheapest form of synchronization is the sequencing of instructions in a thread; here synchronization is implicit in the static ordering. Identifying portions of the dual graph that can be executed as a thread is called *partitioning*.

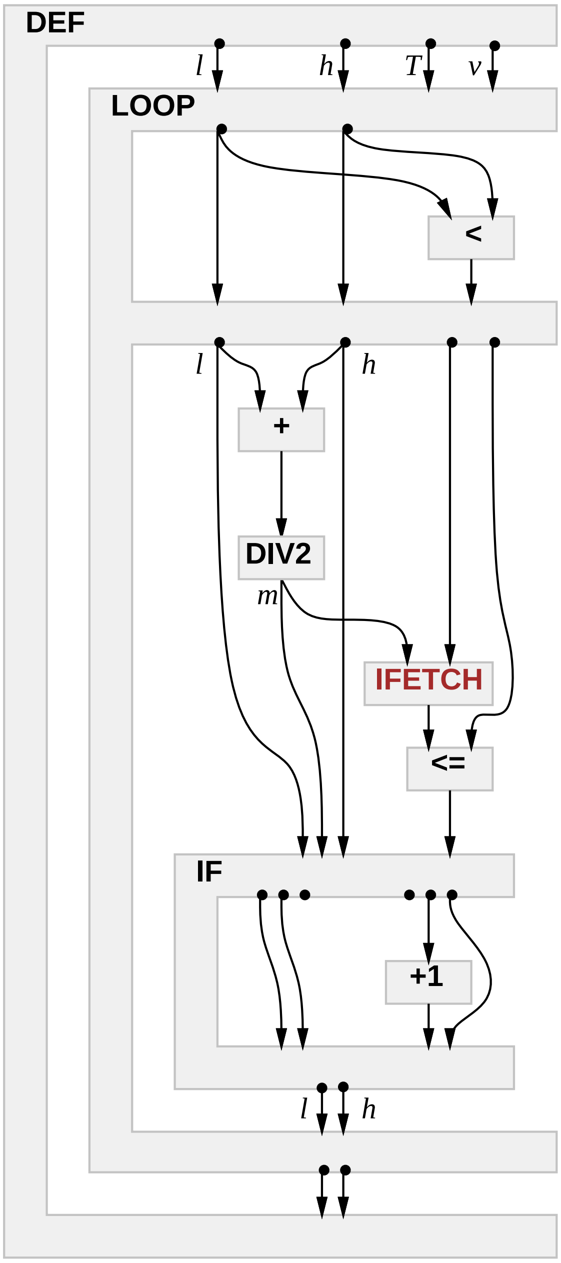

Figure 2.19: Program graph for lookup example.

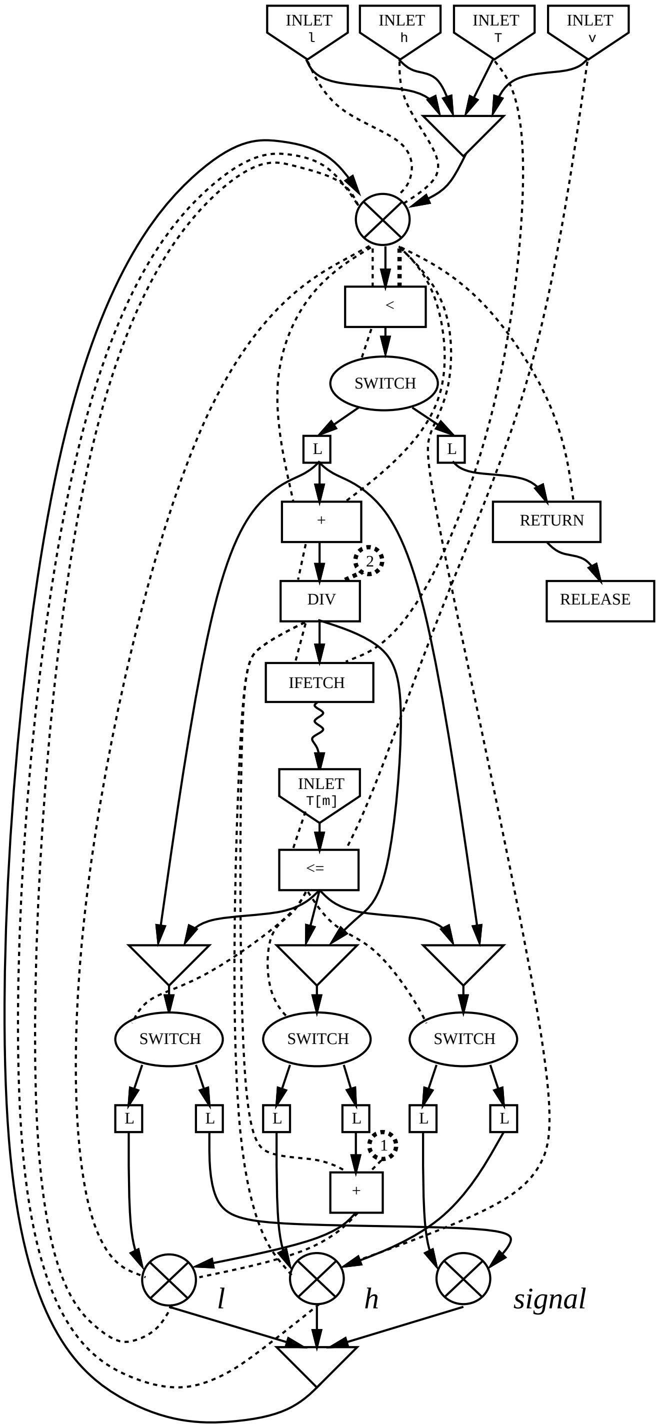

Figure 2.20: Dual graph for lookup example.

Chapter 3

Partitioning

The fundamental step in compiling a lenient language for a machine that executes instruction sequences is partitioning the program into statically schedulable entities[Tra88]. Limits on partitioning are imposed by dependence cycles that can only be resolved dynamically. In Id90 these arise due to conditionals, function calls, and accesses to I-structures. Partitioning for TAM involves identifying portions of the dual graph that can be executed as a TAM thread, *i.e.*, a partition must be linearizable with synchronization and control entry occurring only at the top. The number of entries to a thread must be statically determined. In the context of the dual graph representation, partitioning can be accomplished using only the control and dependence arcs, ignoring data arcs and the constant nodes which carry only data information. Assignment of storage to output ports is deferred until after partitioning, the critical information is retained in the data arcs.

Definition 1 (TAM Partition) A *TAM partition* is a subset of dual graph nodes and their incident control and dependence edges. In a valid partitioning, partitions are node-disjoint and cover the graph. A partition consists of an *input region* containing only inlet, merge, and label nodes and a *body* containing simple nodes, outlets, switches and joins. The *outputs* of a partition are its outlet nodes and all leaving control arcs. Control edges that connect two partitions belong to both partitions.

Figure 3.1 shows a partition with four nodes in the input region and six nodes in the body.

Definition 2 (Safe Partition) We call a TAM partition *safe* if

1. no output of the partition need be produced before all inputs to the body are available,
2. when the inputs to the body are available, all nodes in the body are executed, and
3. no arc connects a body node to an input node of the same partition.

The first property says that body of the partition can be treated as strict, *i.e.* it is safe to wait for all inputs to arrive before executing any instruction of the partition. The second says that there is no conditional execution within a partition; conditional execution occurs only between partitions. The third implies that a partition is acyclic and can be linearized in a manner consistent with the control arcs, since all cycles include a switch and a merge. Also, all dependence arcs must cross partitions. Finally, the entry count for any valid execution of the partition is constant. This implies the following.

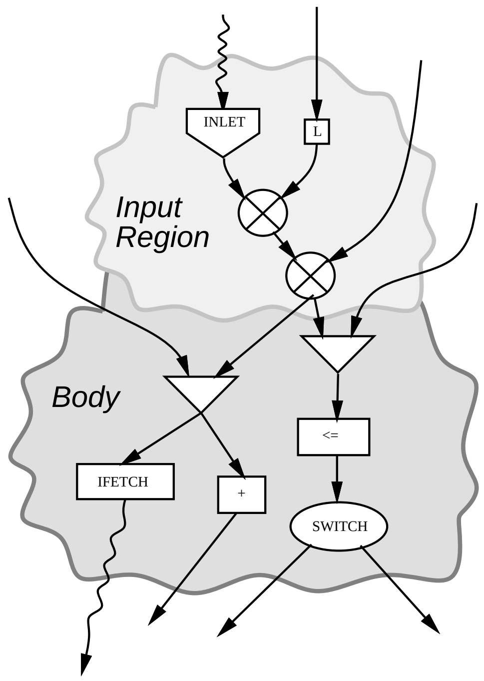

Figure 3.1: Partition example.

Lemma 1 *A safe partition can be mapped into a TAM thread.*

Figure 3.2 gives three examples with unsafe partitions. In the first example an I-fetch and the receiving inlet were placed in the same partition; this violates the third property of a safe partition. The second example has conditional execution within a single partition, thus violating the second property. The last example has a cyclic dependence and violates the first property of a safe partition. While the first two examples could have been partitioned correctly, it is impossible to do so for the third example. This last example would correspond to the code `a = a + b;` which would deadlock in Id90. Throughout the rest of this chapter, we will assume that we only need to find safe partitions if the programs can be partitioned safely in the first place.

Our partitioning algorithm starts by finding small safe partitions. We have four different ways of creating these basic partitions: simple partitioning, dataflow partitioning, dependence sets partitioning and dominance sets partitioning. These basic partitions are then iteratively merged into larger partitions by applying simple merge rules. The merge rules will ensure that the newly formed partitions are also safe. We first present a small example to illustrate the partitionings schemes, before introducing them in a formal way.

3.1 Partitioning a small example

The small program

```
c = a + b; d = a * b;
```

is used to illustrate the different partitioning schemes. Figure 3.3 shows the dual graph (only the control arcs are shown) for this piece of code together with the various partitioning possibilities.

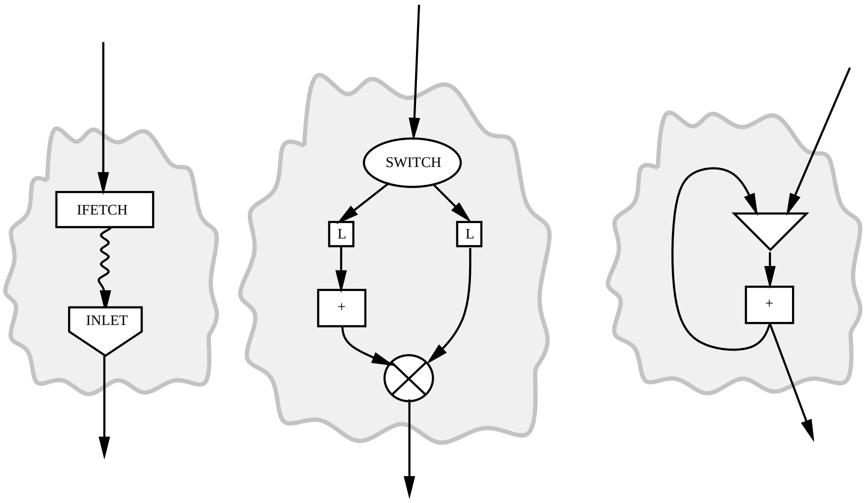

Figure 3.2: Unsafe partitions.

Simple partitioning is the most trivial form of partitioning, here each node is placed into its own partition, resulting in a total of six partitions. This directly yields a safe TAM partitioning, but its dynamic synchronization overhead is large.

Dataflow partitioning recognizes the fact that unary operations never need dynamic synchronization. In this scheme the inlet and sync nodes start a new partition. The add and the multiply nodes are placed into the partition of their control predecessor, we thus get four basic partitions.

Far more powerful is *dependence sets partitioning*, which finds safe partitions by grouping together nodes that depend on the same set of input nodes (inlets, merges and labels). A node u depends on an input node v if there exist a direct control path from v to u. For our example, dependence sets partitioning produces three basic partitions. We have annotated each partition with its dependence set. The four bottom nodes depend on exactly the same inputs, they are therefore placed into the same partition. By only placing nodes with the same dependence sets into the same partition, no node can produce a side-effect on another (because it would then also produce a side-effect on itself).

Dominance sets partitioning finds safe partitions by grouping together nodes which dominate the same set of output nodes. A node u dominates an output node v if there exist a direct control path from u to v. This partitioning scheme also ensures that there are no cyclic dependencies within a partition. Since all nodes of a basic partition dominate the same outputs, under no circumstances can one node have a side-effect on another. We have indicated the dominance sets in our example. With dominance sets partitioning, the top two nodes can go into the same partition since they dominate the same set of outputs: `c` and `d`.

After basic partitioning we can apply merge rules which ensure that the merged partition is also safe. A partition with a single control predecessor can be *merged up* if no separation constraints are violated. A partition can be *merged down*, if it feeds strictly a single successor partition. We will present the merge rules in more detail later.

In our example, starting with simple partitioning, we can merge up the add and multiply node into the partition of the corresponding control predecessor. This results in the same safe partitioning as with dataflow partitioning. At this stage no further merge rule can

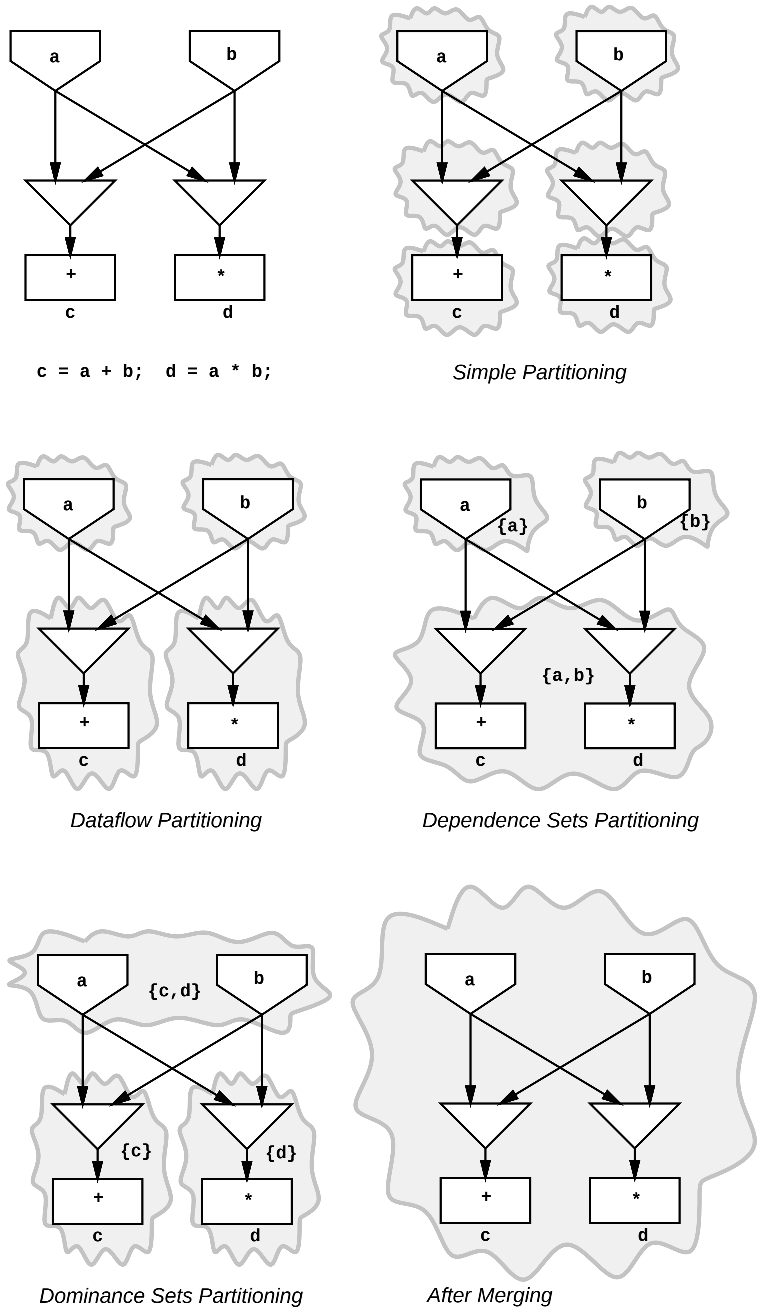

Figure 3.3: Partitioning `c = a + b; d = a * b;`

be applied. Starting with dependence sets partitioning, we can apply the "merge down" rule and finally obtain a single safe partition containing the six nodes. Similarly, starting with dominance sets partitioning, we can apply the "merge up" rule. This will also produce a single partition. As this example shows, the quality of partitioning after merging depends strongly on basic partitioning. The power of dependence sets and dominance sets partitioning lies in the fact that they can work across different fan-out or fan-in trees.

3.2 Basic Partitioning

We now present the basic partitioning schemes and partition merging formally.

3.2.1 Simple Partitioning

Simple partitioning is the most trivial form of partitioning, here each node is placed into its own partition. This directly yields a safe TAM partitioning, but dynamic synchronization overhead is large.

3.2.2 Dataflow Partitioning

Dataflow partitioning recognizes the fact that unary operations never need dynamic synchronization. In this scheme joins, inlets, merges and labels start a new partition. Simple, switch and outlet nodes are placed into the partition of their control predecessor. Dataflow partitioning also yields safe TAM partitions.

3.2.3 Dependence Sets Partitioning

Far more powerful is *dependence sets partitioning*, which is based on a variant of Iannucci's method of dependence sets [Ian88a].

Definition 3 (Dependence Set) The *dependence set* for a dual graph node u is the set of input nodes i (inlets, merges and labels) on which it depends, *i.e.* there exists a control path of length zero or greater from i to u that does not go through any other input node.

The algorithm for computing the dependence sets first assigns each input node the dependence set containing only itself; for all other nodes the dependence set is the union of the dependence sets of the control predecessors. Our definition will not allow dependence to cross switches, since every control output of a switch is connected to a label, thus indicating that they must be in a different partition. Stronger notions of dependence could be employed that would capture dependence across conditionals.

Having computed the dependence sets for all nodes, we can then find safe partitions by grouping together nodes that depend on the same set of input nodes. This guarantees that there are no cyclic dependences within a partition. For example, all nodes that depend only on a particular inlet will be grouped with the inlet, and similarly for merges and labels.

Algorithm 1 (Basic Partitioning using Dependence Sets)

- *Compute the dependence sets for all nodes*
- *Put all nodes with the same dependence set into the same basic partition.*

Lemma 2 *All basic partitions produced by dependence sets partitioning are safe TAM partitions.*

Proof: We have to show that each basic partition satisfies the three properties of a safe partition. Assume that an output of the partition needs to be produced before all inputs are available. All nodes in the the partition depend on all of the inputs (since they all have the same dependence set). This implies that the node that produces this output also depends on the input. Thus the node depends on itself and no safe partition can be found. We disallowed this case, the basic partition thus satisfies the first property of safe partitions. It also satisfies the second property since all arms of conditional execution are marked with a label. Dependence cannot propagate across inlets, labels, and merges. Also, since we have only structured dual graphs where a circular dependence must go through a merge and a switch we satisfy the third property. □

3.2.4 Dominance Sets Partitioning

Analogous to dependence sets partitioning, we can find safe partitions by grouping together nodes which dominate the same set of output nodes.

Definition 4 (Dominance Set) The *dominance set* for a dual graph node u is the set of output nodes o (outlet nodes and nodes directly feeding a merge or label) it dominates, *i.e.* there exists a control path of length zero or greater from u to o that does not go through any input node except itself.

The algorithm for computing the dominance sets first assigns each node that is a direct control predecessor to a merge or label node the dominance set containing only itself. For all other nodes the dominance set is the union of the dominance sets of the control successors plus itself if it is an outlet node.

Grouping all nodes that dominate the same set of output nodes into the same partition also ensures that there are no cyclic dependencies within a partition. Since all nodes of a basic partition dominate the same outputs, under no circumstances can one node have a side-effect on another.

Algorithm 2 (Basic Partitioning using Dominance Sets)

- *Compute the dominance sets for all nodes*
- *Put all nodes with the same dominance set into the same basic partition.*

Lemma 3 *All basic partitions produced by dominance sets partitioning are safe TAM partitions.*

Proof: Assume again that an output of the partition needs to be produced before all inputs are available. All nodes in the partition dominate the same set of outputs. This implies that the node that need this input also dominates it, which was disallowed. The basic partition thus satisfies the first property of safe partitions. It also satisfies the second property since all arms of conditional execution are marked with a label. All partitions satisfy the third property due to the fact that dominance sets cannot propagate across merge and labels and we only have structured dual graphs. □

Other possible basic partitioning schemes could identify regions of the graph that depend solely on a single node (or dominate a single node.) These tree like regions will also create safe partitions. Dependence sets partitioning (or dominance sets partitioning) will create much large basic partitions, since they allow nodes that depend on (dominate) several nodes to be grouped. Downward and upward trees will always be found by partition merging discussed next.

3.3 Merging partitions

After basic partitioning, partitions will be merged into larger, but still safe, partitions by iteratively applying one of the following two merge rules.

3.3.1 Merging up

"Merge up" rule: Two partitions α and β can be merged into a larger partition γ if

- all input arcs to β come from α,
- β contains no inlet instructions, and
- no output arc from the body of α goes to an input node of β.

Figure 3.4 shows this case graphically. The arcs connecting α to β indicate that it is necessary for α to execute before β. The first two points of the merge up rule imply that this is also sufficient. The two partitions cannot be merged if β has an arc coming from some other partition besides α or if it contains an inlet. In both cases it could be that α or some output from α will cause a side-effect on this input or inlet. The last point of this merge rule ensures that no separation constraint is violated.

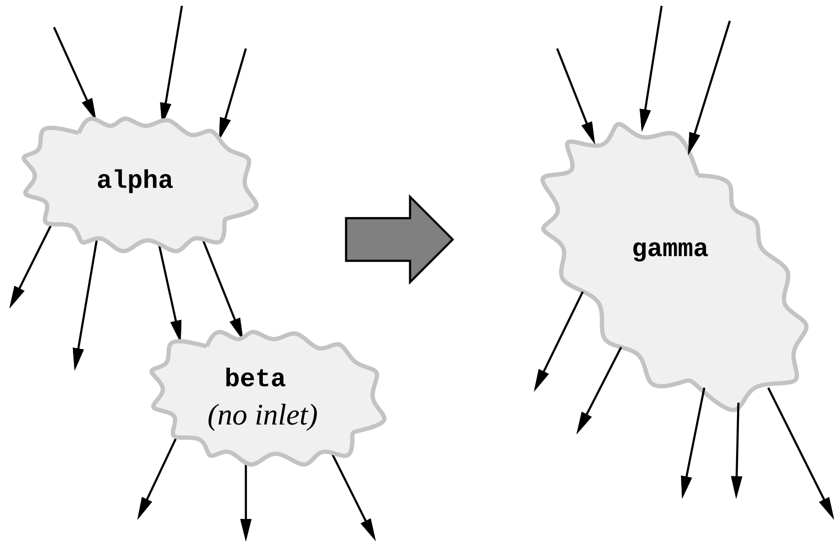

Figure 3.4: Merge Up: A partition with a single control predecessor can be merged up if no "separation constraints" are violated.

The astute reader will notice that this rule cannot be applied after basic partitioning, since then all nodes in β would have the same dependence set as the nodes in α and the two partitions would already have been merged. Opportunities for this rule arise as a result of merging down, discussed below.

Lemma 4 ("Merge Up") *If α and β are safe partitions then the "merge up" rule will produce a safe partition.*

Proof: Assume that both α and β are safe partitions and that the merge rule conditions apply. All inputs from β come from α, it is therefore valid for the merged partition to produce no output until all inputs to α are available. β will always be executed if α is. Otherwise β must be one side of a conditional, but then α would have contained a switch with an output to a label in β and the "merge up" would not be applicable. $\square$

3.3.2 Merging down

"Merge down" rule: Two partitions α and β can be merged into a bigger partition γ if

- all output arcs from α go to β,
- α contains no outlet instructions, and
- no output arc from the body of α goes to an input node of β.

This situation is shown in figure 3.5. The first two points ensure that α has no side-effect on an input of β. The last point ensures that no separation constraint is violated.

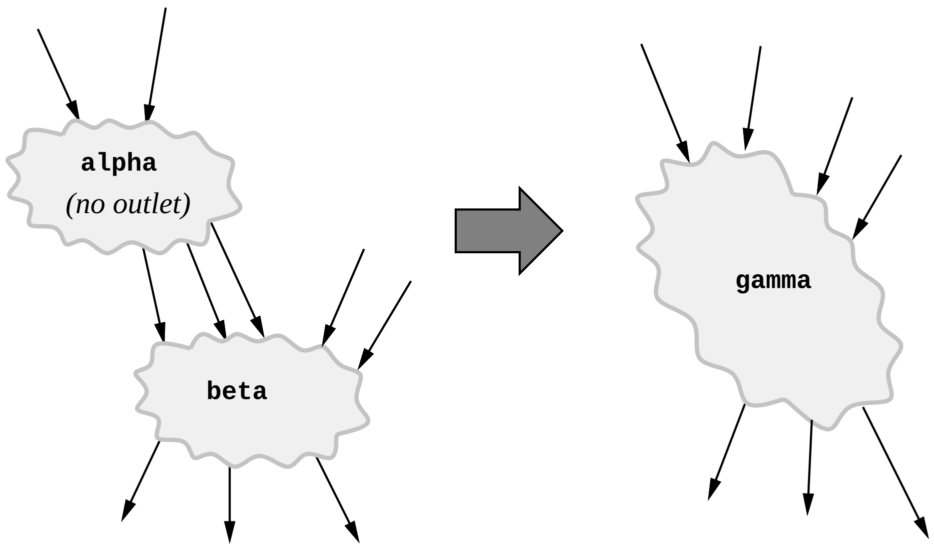

Figure 3.5: Merge Down: The results of a partition feed strictly into a successor, so it can be merged into the successor.

Lemma 5 ("Merge Down") *If α and β are safe partitions then the "merge down" rule will also produce a safe partition.*

Proof: Assume α and β are safe partitions and that the merge down rule can be applied. Since α contains no outlets and no output of the partition β must be produced before all inputs to the body of β are available, α cannot produce an output that influences any of the inputs to β other than those from α. The last condition ensures that no separation constraint will be violated in the merged partition $\Box$

As stated, the two merge rules allow two partitions to be merged even if an output arc from an input node in partition α goes to an input node in partition β. The three possible configurations where this could occur are shown in Figure 3.6. The first configuration arises with an I-fetch in a conditional, the second case when one side of a conditional does not have to perform any computation, and the last case with nested conditionals. In all three cases the merge must be at the beginning of a thread. But because the inlet, label and merge also indicate the start of a new thread, and because they denote no real computations, we can merge the two partitions.

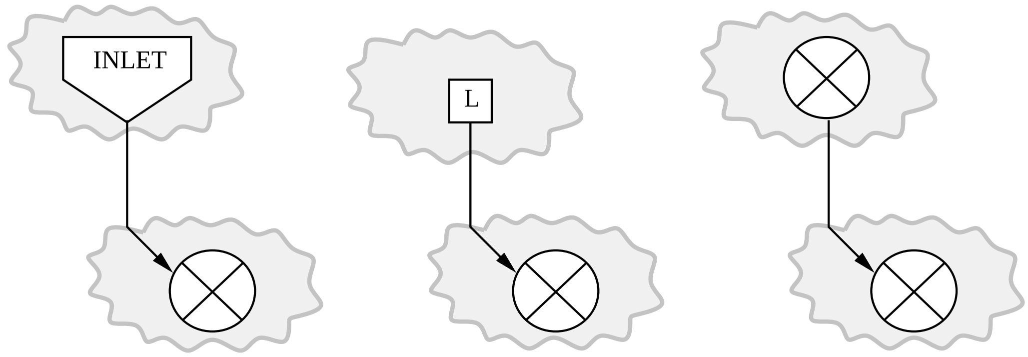

Figure 3.6: Configurations where merging could occur.

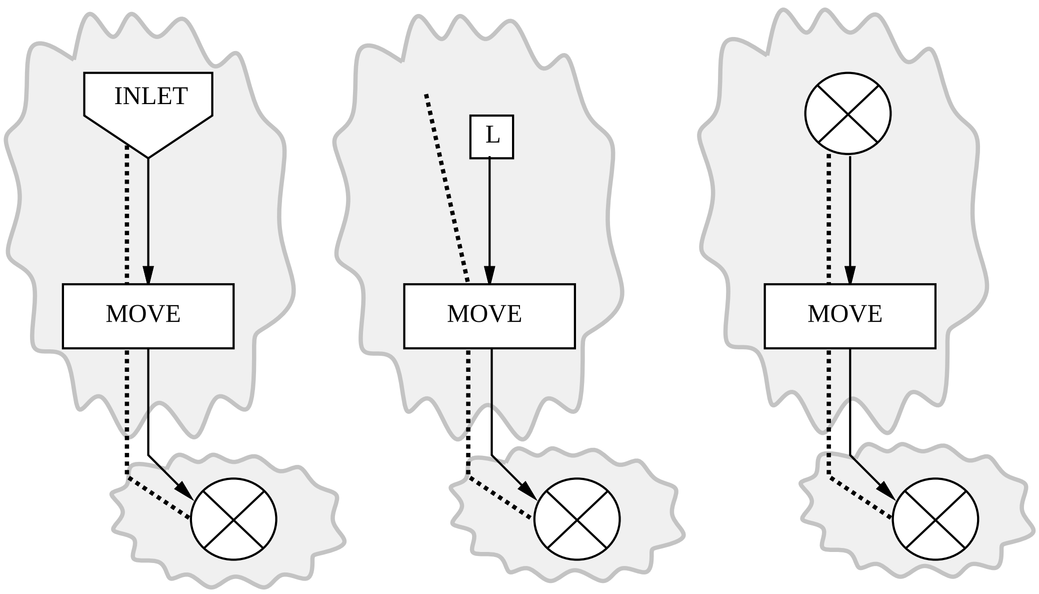

Figure 3.7: Inserting a move forces to split the partition.

Care is needed at a later stage, in frame slot assignment, when mapping partitions into TAM threads. The merge does not only serve to combine control, but also as a name unifier. If the frame slot assignment is unable to statically assign a data input of the merge the same name as the corresponding data output, a move instruction needs to be inserted as shown in Figure 3.7. The move should only be executed on the side of the conditional where it was inserted, thus it can not be placed into a partition together with the merge. The consequence is that if the frame slot assignment has to insert a move it also needs to separate the partition into two. This does not affect the correctness of partitioning; the final partitioning will still be safe.

We can now summarize our partitioning algorithm:

Algorithm 3 (Partitioning based on dependence sets)

- *Compute the dependence sets for all nodes*
- *Put all nodes with the same dependence set into the same basic partition.*
- *Iteratively merge partitions into larger partitions by applying one of the two merge rules.*

The results of the previous lemmas directly imply the following correctness theorem of our partitioning algorithm:

Theorem 1 (Partitioning Algorithm) *The partitioning algorithm will produce only safe partitions.*

Traub showed that optimal partitioning is NP-complete. Our algorithm is merely a heuristic, since starting with the basic partition, it will iteratively merge two partitions as long as a merge rule can be applied. If mutually exclusive merge rules are applicable at some point, one is picked arbitrarily. Partitioning decisions imply trade-offs between parallelism, synchronization cost, and sequential efficiency. However, given the limits on thread size imposed by the language model, the use of split-phase accesses, and the control paradigm, we simply attempt to make partitions as large as possible and try to minimize the synchronization cost. The number of control tokens required to enable a partition is the entry count for the corresponding thread. Each entry will incur the cost of decrementing the entry count, but only the last will enable the thread.

Lemma 6 *Applying one of the partitioning merge rules will never increase the "synchronization cost".*

Proof: The synchronization cost for the merged partition is proportional to the number of control arcs that enter the body from other partitions plus the number of nodes from the input region that have control arcs going to the body. This number can never be higher than the sum of the synchronization costs of the two unmerged partitions □

3.4 Small examples

The small examples given in Figures 3.8 and 3.9 show that dependence sets and dominance sets partitioning may find different basic partitions, and thus the merging will also produce different results. In the first example dependence sets partitioning and merging will produce

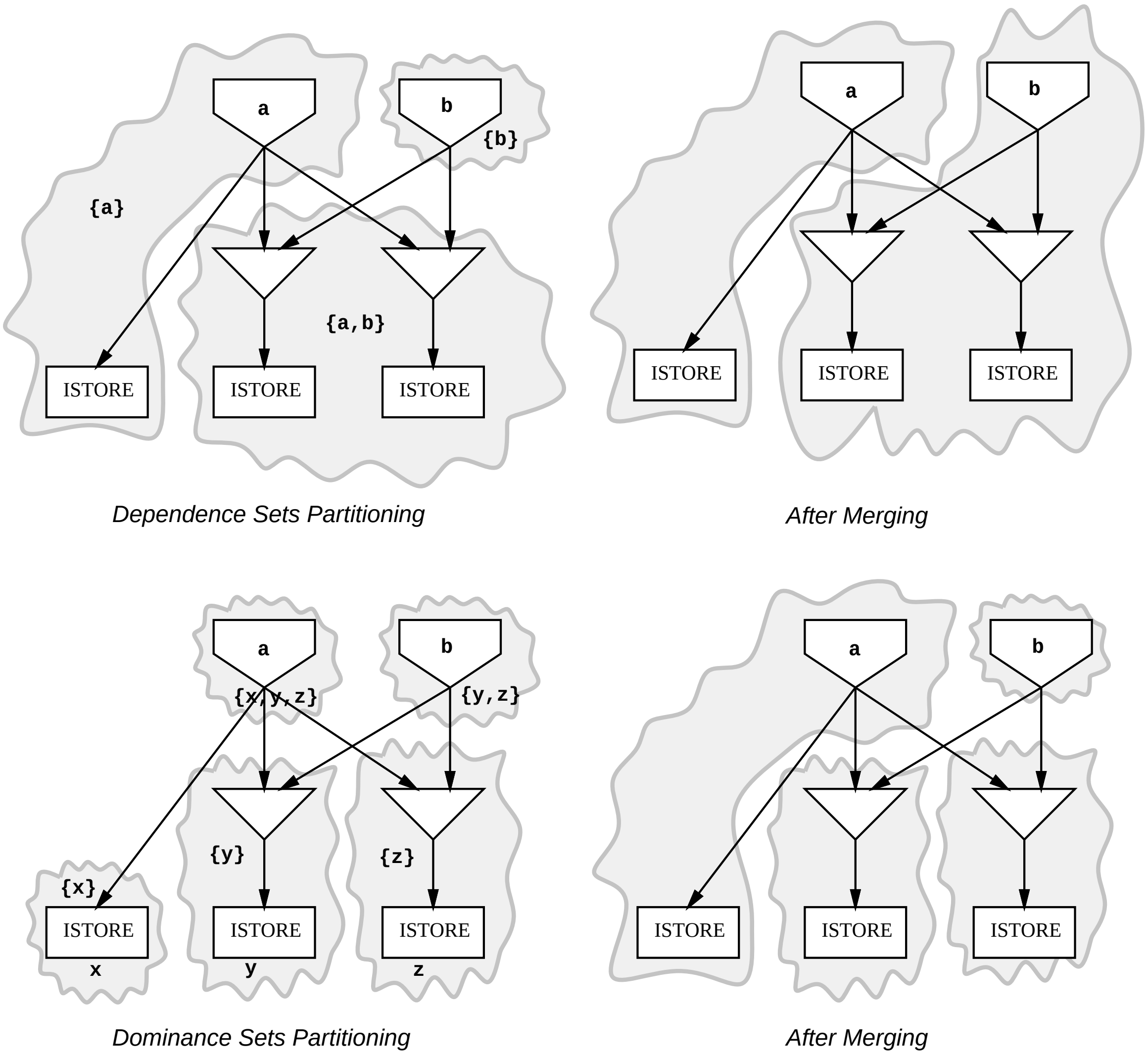

Figure 3.8: Example where dependence sets partitioning works better than dominance sets partitioning.

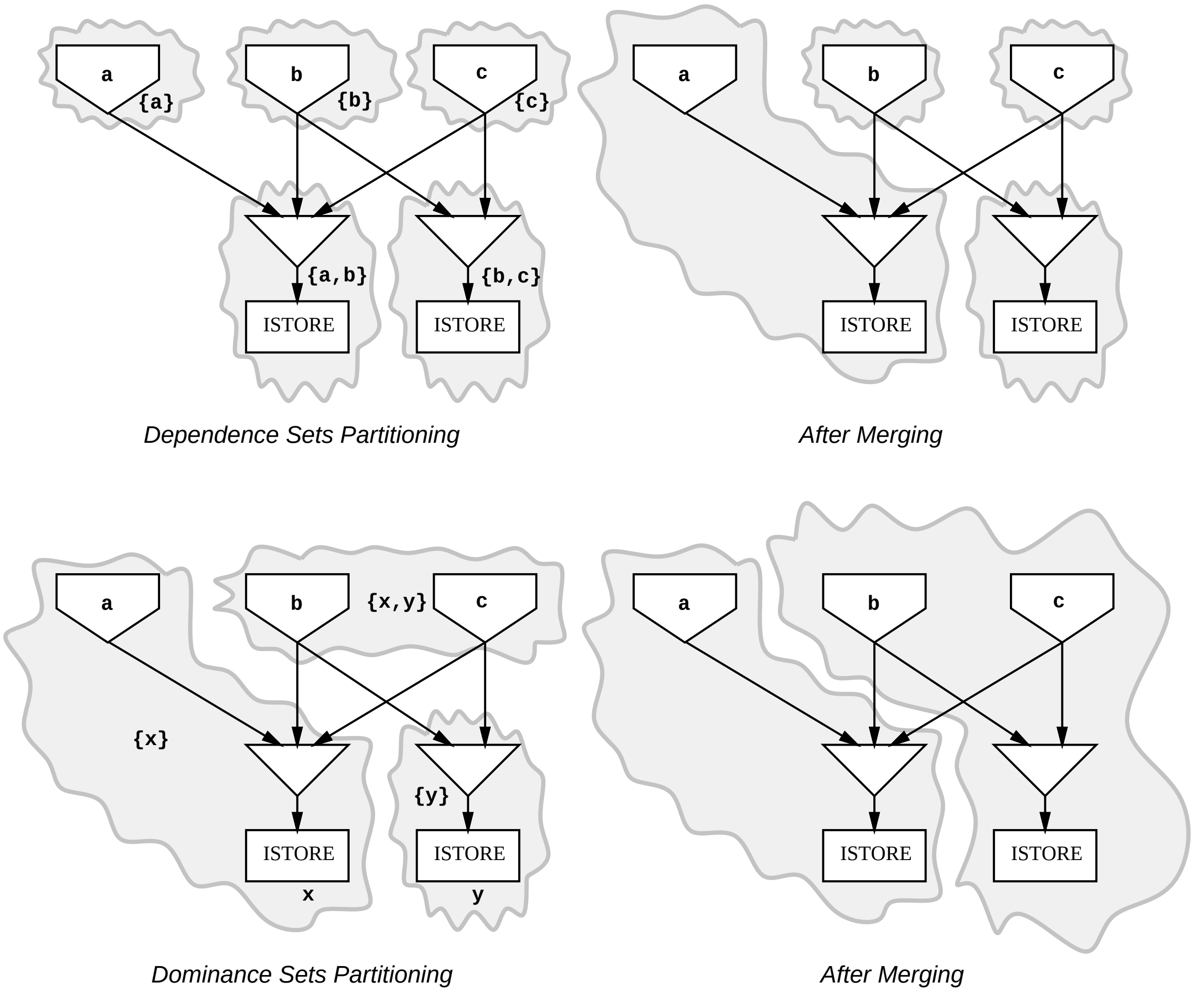

Figure 3.9: Example where dominance sets partitioning works better than dependence sets partitioning.

only two final partitions, while dominance sets partitioning with merging ends with four partitions. In the second example, the outcome is reversed.

It is possible to combine both forms of basic partitioning. Care must be taken though, as the example Figure 3.10 shows. Here both forms of partitioning will produce the optimal result, two partitions. Both approaches have correctly identified that the I-store may have a side-effect on the inlet `c`. Therefore it is not permissible to put these two nodes together into a partition. It is only possible to combine both forms of basic partitioning where they do not contradict one another, *i.e.* if one partition completely overlaps partitions from the other basic partitioning, then the larger partition can be taken. The combined result will still produce a safe partitioning. In this example it is possible to start with dependence sets partitioning, and then take from dominance sets partitioning the information that inlet `a` and `b` can be placed together in a single partition.

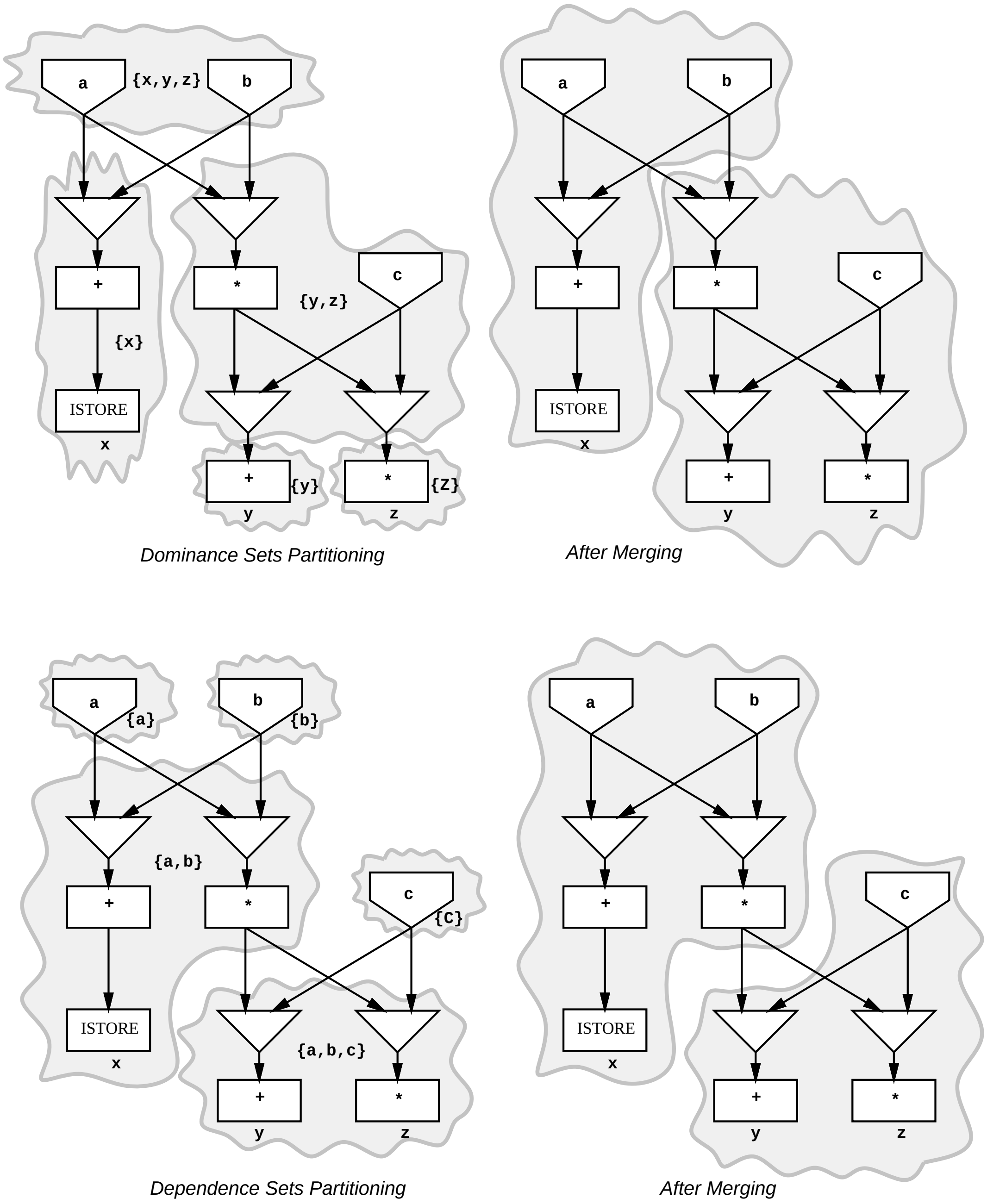

Figure 3.10: Dependence sets and dominance sets partitioning.

3.5 Redundant Arc Elimination

The goal of redundant arc elimination is to reduce synchronization cost. In this step we eliminate redundant control arcs that go between partitions, and thereby decrease the entry count of the target partition. Perhaps more importantly, redundant arc elimination may enable additional partition merges. Thus, after each partition merge the incident arcs to the new partition are checked for redundancy.

One of the benefits of dual graphs is that they allow the separate elimination of unnecessary control and data arcs. Partitions, which at this point are still in a partial order, are linearized at a later stage to create sequential threads. With linearization, many control arcs become redundant. We do not care about redundant control arcs within a partition. Only the elimination of redundant arcs crossing partitions can improve the quality of partitioning.

Definition 5 (Redundant Arc) A control arc from partition u to v is *redundant* if there exists another unconditional control path from u to v.

A trivial case of this is where multiple arcs cross from one partition to the body of another. Redundant control arcs can be eliminated, since they carry no new synchronization information. This transformation is shown in Figure 3.11.

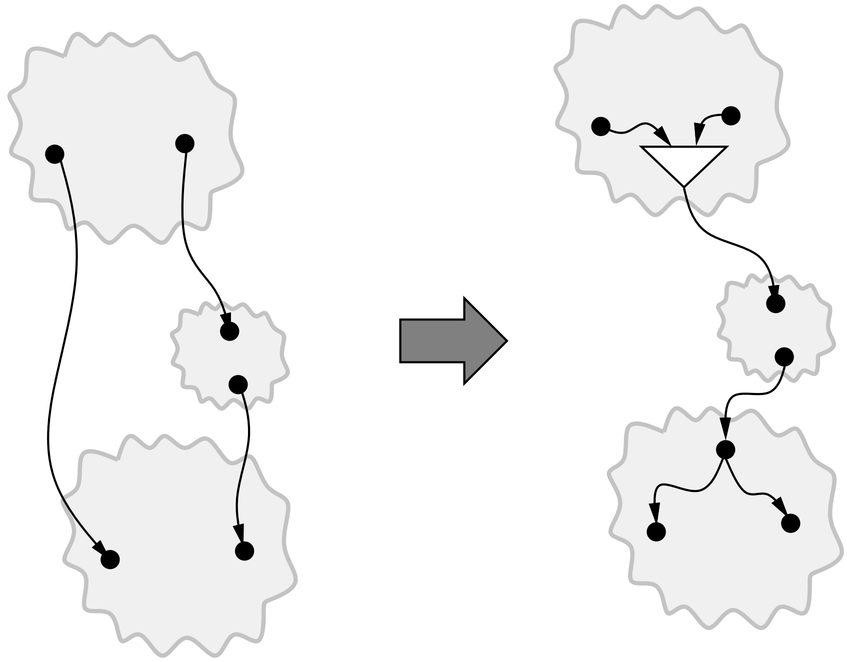

Figure 3.11: Redundant Arc Elimination Rule.

3.6 Switch and merge combining

Two switches that are in the same partition and are steered by the same predicate can be merged into a single switch. This optimization attempts to minimize the control transfer overhead where the full power of lenient conditionals is not required. In essence, it is a more complex form of redundant arc elimination. As shown in the top part of Figure 3.12, the control inputs to the switch are joined and the switches and labels are merged into a single switch with a label for each of the combined control outputs. Observe that the data arcs are unchanged by the transformation.

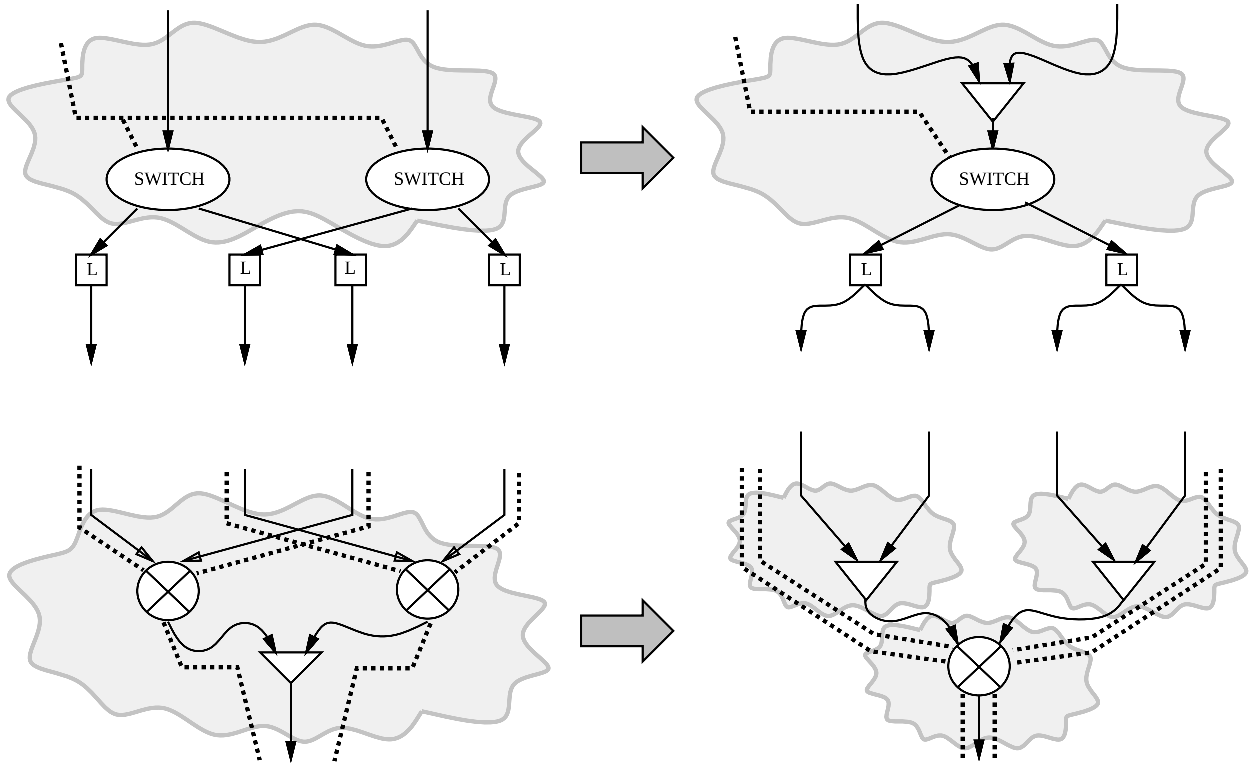

Figure 3.12: Switch and Merge Combining.

Merges that are in the same partition and are derived from the same conditional, *i.e.*, are determined by the same predicate, can be combined into a single merge that steers the union of the data arcs, as shown in the bottom part of Figure 3.12. This optimization serves primarily to reduce synchronization costs by enabling further merging of partitions within the arms of the conditional.

3.7 Partitioning the lookup example

To illustrate the partitioning process, we consider the `lookup` example from Figure 2.20. Grouping nodes with the same dependence set, we get twenty basic partitions. By iteratively merging partitions using the two rules, we end with twelve partitions as can be seen in Figure 3.13. The lookup example contains three redundant arcs, connecting to the `joins` above the three `switches`. One comes from the control output of the `div`, the two other from the loop body `label`. These are redundant since the dependence arc between the `ifetch` and the corresponding `inlet` guarantees that the partition with the `ifetch` will always be executed before the partition with the three switches. The three switches can be combined into a single switch, replacing the three labels on each side with a new label. Similarly, the three merges can be combined, yielding the dual graph shown in Figure 3.14.

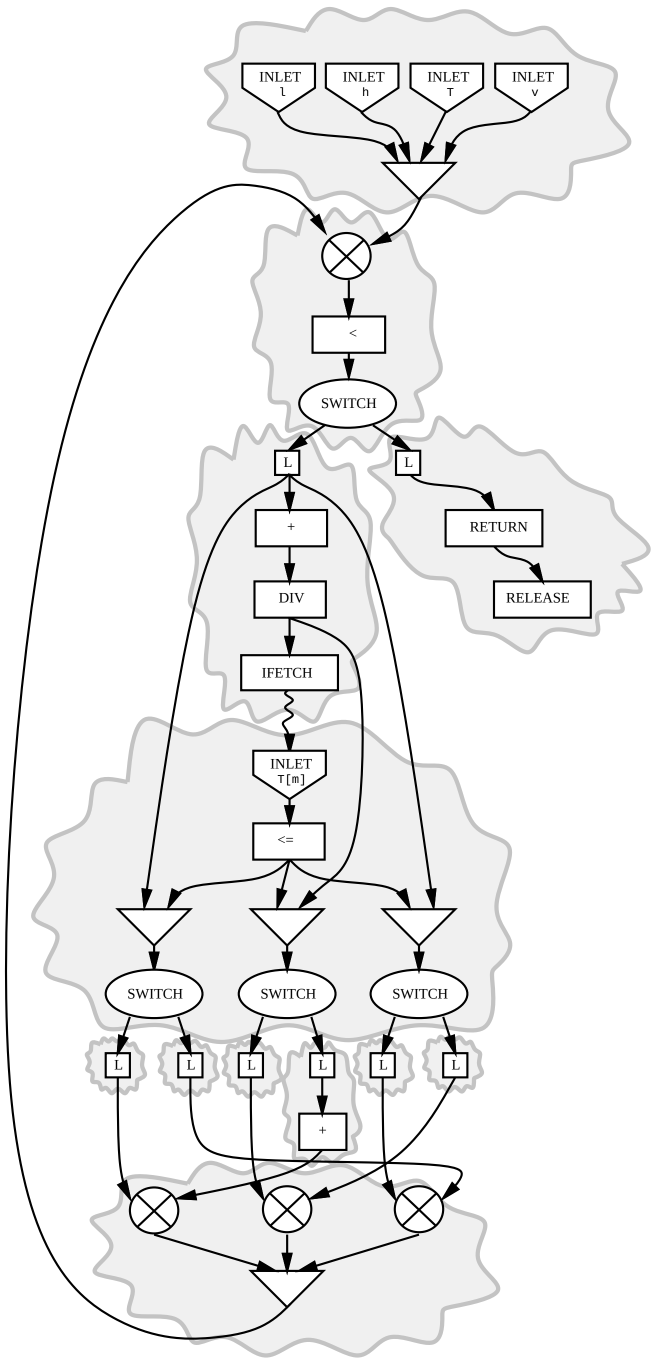

Figure 3.13: Control Graph for lookup after merging.

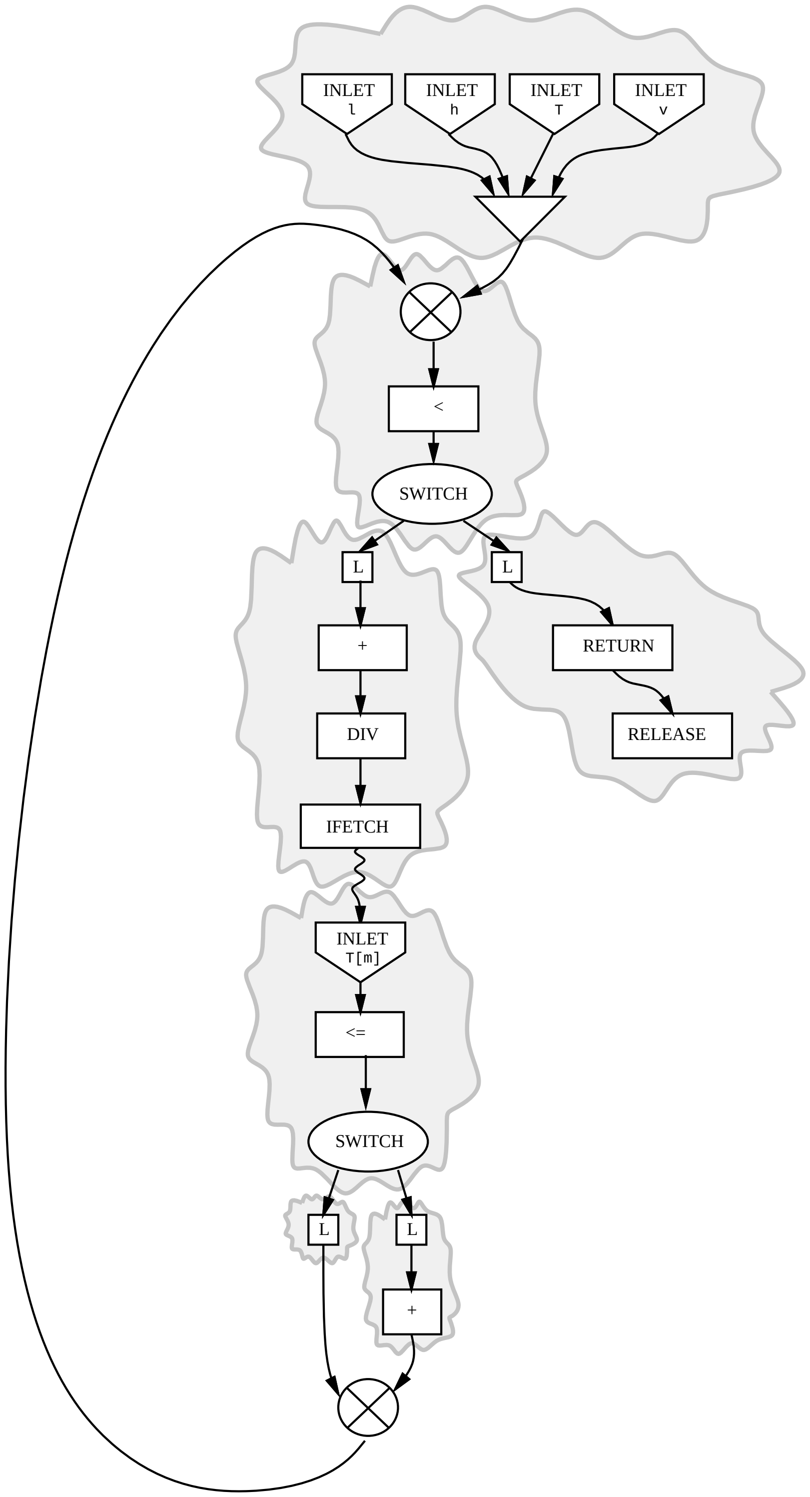

Figure 3.14: Final Dual Graph for lookup after merging, redundant arc elimination and combining.

Chapter 4

Thread Generation

To produce TAM code, the dual graph partitions must be ordered, each must be linearized to form a thread, and data outputs must be replaced by specific registers and frame slots, so that subsequent operations can use them. This is done by the following steps:

1. Lifetime and quantum analysis,
2. Data type determination,
3. Instruction scheduling,
4. Frame slot and register assignment and coloring,
5. Move insertion (name unifying at merges),
6. Entry counts determination,
7. Fork insertion, and
8. Thread ordering

There can be substantial interaction between instruction scheduling, frame slot and register assignment as well as the move insertion step. We will describe these steps in more detail in the subsequent sections.

4.1 Lifetime analysis

The compiler first determines 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 TAM variables can easily be determined from the data arcs of the dual graph. If all targets are in the same partition as the source node, the lifetime is limited to a single thread and the value can safely be placed in a register.

However, the TAM scheduling paradigm requires that when 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. The register assignment rule, using the quantum information, can be restated as follows: *A data value can safely be placed in a register if none of its corresponding data arcs cross a quantum boundary.*

Dynamically, quantum 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 IFETCH 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 not currently exploited by the compiler. This area of speculative register assignment is a subject for future research.

4.2 Determining Types

TL0, the language implementing TAM defines many different types, including integers, floating point numbers, characters, booleans, frame pointers, I-structure pointers, synchronization variables, threads, and inlets. The types were deliberately kept abstract to make this intermediate form as portable and machine independent as possible. The name-spaces for registers and frame slots are thus kept separated for each data type. The task of the compiler is to try to determine for each value in the dual graph the most specialized data type possible, since this will usually correspond to the smallest size usable.

To achieve this, each data input and output port has an associated type slot which is set when the dual graph is generated. For most operations, the type of their inputs and outputs is known statically. For example, we know that for binary integer operations both inputs, as well as the output must be an integer. The type *general* will be used whenever the compiler has no knowledge about the object. This type specifies that the container size must be at least as large as the largest of any size of the other types. If the output type is a *general*, the compiler will be able to deduce a more specialized type in these two following cases:

- If the output is used as a type other than general by another instruction, and this instruction is always executed if the instruction belonging to the data output is executed, then the output can be assigned the same type.
- If all inputs that use the output have the same type, then the output can be assigned the same type.

The algorithm tries to narrow down the type of all outputs by following these rules. Special care needs to be taken at merge nodes. After the type specialization step just described, the types will be propagated up merge nodes in the following manner: If a data output of a merge is of a type different from *general* then all corresponding inputs must be of the same type and this type can be propagated up. Then the first type specialization step is performed again. Now types can be propagated down through merges. If all data inputs to a merge are of the same specialized type, then this type can also be assigned to the corresponding output. Finally we use the following rule to check whether the dual graph (and thus the corresponding program) is correctly typed, *i.e.* to find out if all outputs are being used in a consistent way: *For any data arc in the dual graph both the data output and the data input must be either of the same type or one of them must be of type general.*

4.3 Instruction Scheduling

At this point, the partitions within the dual graph are still a partial order and must be linearized. Linearization influences the lifetime of values and thus has a strong influence on the total register and frame slot requirements. Instruction scheduling might be done differently for different machines, depending on pipeline structure and register availability. For a superscalar machine, for example, it might desirable to put many independent instructions close to each other so that they can be scheduled together. We currently use a heuristic that attempts to minimize the overlap of the lifetimes of values. Combined with graph coloring for register assignment, this tends to minimize the storage used.

4.4 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 distinct data outputs 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 value, and an edge between two nodes if their lifetimes overlap. We construct one interference graph for each combination of storage class (register or frame slot) and data type, since their name-spaces are kept separated in TL0. Coloring these graphs 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 Id90 to TAM the uncertainty arises because of dynamic scheduling. The language is single assignment. TAM does not specify the order in which concurrently enabled threads of a quantum will be scheduled. For standard machines, it is convenient to implement fork by pushing a thread address onto the stack. Hardware supporting TAM directly is likely to use a FIFO thread queue to improve the performance of instruction prefetch. The compiler thus has to make worst case assumptions when computing the lifetime of variables. If threads are interleaved in the datapath, lifetime disjointness is even more reduced.

4.5 Move insertion: Name unifying at merges

A merge node has various control input ports, but only a single control output port. Associated with each of these control ports is some number of data ports. The merge will receive control on one of its control inputs. The task of the merge is then to read the values bound the associated data inputs, bind them to the data outputs, and pass control on to the control output.

We want to insure by this compilation step that the merges will have nothing to do at execution time; they will just be pseudo-operations. We achieve this by assigning all data inputs the same name (color) as their corresponding data outputs. The compiler has to insure that for each control input, data port i will be assigned the same frame slot or register as the data port i for the control output. The reason is that with conditionals, both sides should place the data values into the right location where it will later be used by the output of the conditional. The same problem also occurs with loops. Here the compiler has to place the loop variables at the bottom of the loop into the same location as they came into the loop, which should also be the same place they are being used at the outputs of the loop. Basically, this is a constraint imposed on the register coloring to use the same color for each of the data ports. Whenever this cannot be achieved, move instructions are inserted. When it is necessary to insert a move, the move will be placed into the same partition as the corresponding control predecessor of the merge. If this predecessor is in the same partition as the merge (*i.e.* the merge is directly fed by a label or merge), a new partition must be created as was discussed in the partitioning chapter. Currently we try to minimize the number of move instructions that need to be inserted. Therefore unifying names at merges will be done before the actual coloring of register and frame slots discussed in the previous section.

Identifying whether a move instruction needs to be inserted can be very tricky as the following small examples show.

```
def max x y = if (x > y) then x else y;
```

The dual graph for the function `max` is shown in Figure 4.1. (It has been slightly simplified; the trigger inlet, for example, has been left out.) For this function, partitioning with merging will produce exactly two partitions. Name unification cannot assign the data inputs of the merge to the same locations as the data output, since both `x` and `y` have an overlapping lifetime. In this case, a move instruction was inserted on `y`'s side. This allows us to assign the data inputs to the merge the same color as its data output (they will all use the same location as variable `x`). Inserting the move instruction has the consequence that the partition containing the merge must be split into two. The move will only be executed on the "else" side of the conditional, while nothing needs to be done on the "then" side.

The following loop example shows some additional difficulties that move insertion encounters:

```
def loop_ex x y =
  {while (x < y) do
  (next x, next y) = (x*y,x+y)
   finally y};
```

The corresponding dual graph is given in Figure 4.2. In general, if a move needs to be inserted at a loop, the algorithm tries first to insert it at the entry of the loop instead of in the loop body, because usually the loop body will be executed more than once. In this example, however, a move needs to be placed in the loop body. The problem is that the add and multiply instruction need both the old values of x and y of the previous iteration, while defining them for the next iteration. There is no way to avoid inserting a move instruction if the add and the multiply are executed sequentially.

Name unification at merges in this form is a problem unknown to compilers for conventional imperative languages because they do not allow the formulation of parallel assignments (an exception being parallel `let` and `do` statements in Lisp). In conventional languages it is up to the programmer to introduce the temporaries needed. It is this temporary variable that the Id90 compiler has to create by inserting a move. First it needs to place the add and multiply instructions into a total order since then it only needs to

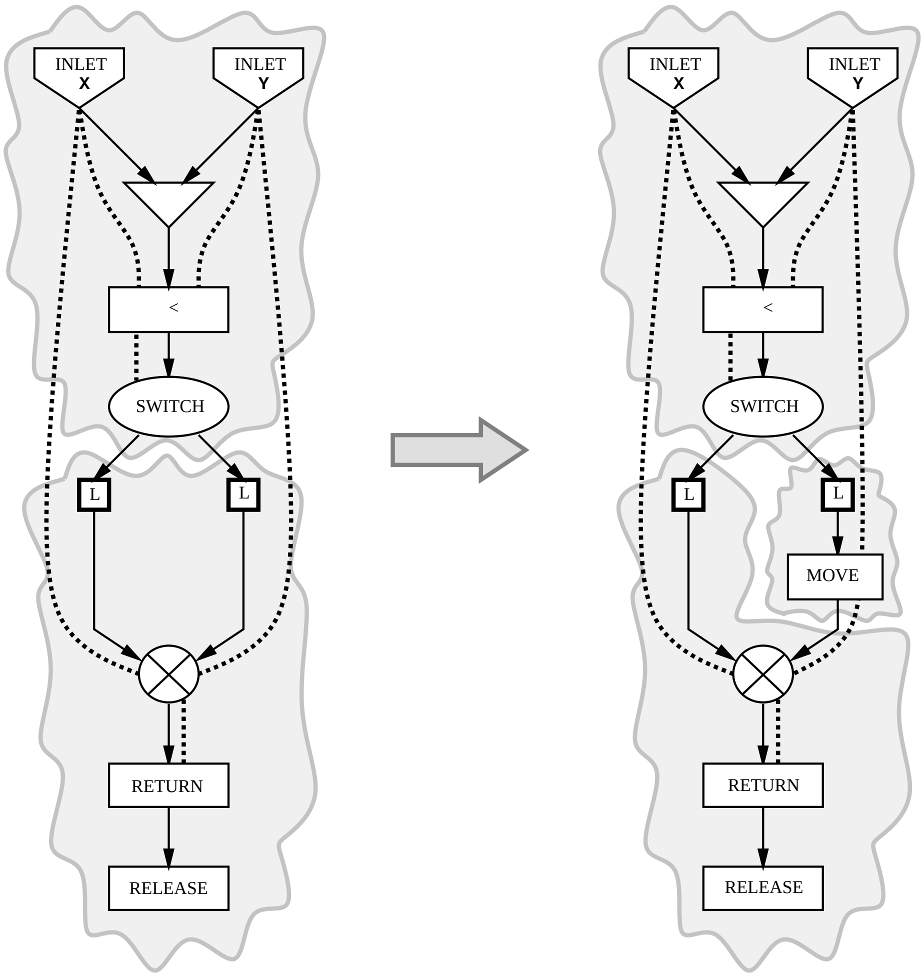

Figure 4.1: Partitioned dual graph before and after move insertion for function `max`.

insert one move for the instruction that executes first. This move will transport the value from the temporary location where the first instruction placed it to the location the next iteration needs it. This example shows the interplay between instruction scheduling and move insertion. Also note, that in this case we did not have to split the partition, since the control successor for the merge (the join) lies in a different partition. The move is placed into the partition of the join. Unifying the names for data inputs and outputs at merges will be done before frame slot and register assignments for all the other instructions.

4.6 Entry counts

Synchronizing threads in TAM have an associated frame slot where the entry counter is maintained. This must be explicitly established by the compiler. The input region of a partition consists only of inlets, merges and labels. Control arcs entering the input region can thus only go to labels or merges, since inlets can only be fed by dependence arcs. Control to a merge can only arrive on one side. Joins contribute to the entry count of containing partitions. Some of the control inputs to a join, however, will originate from within the

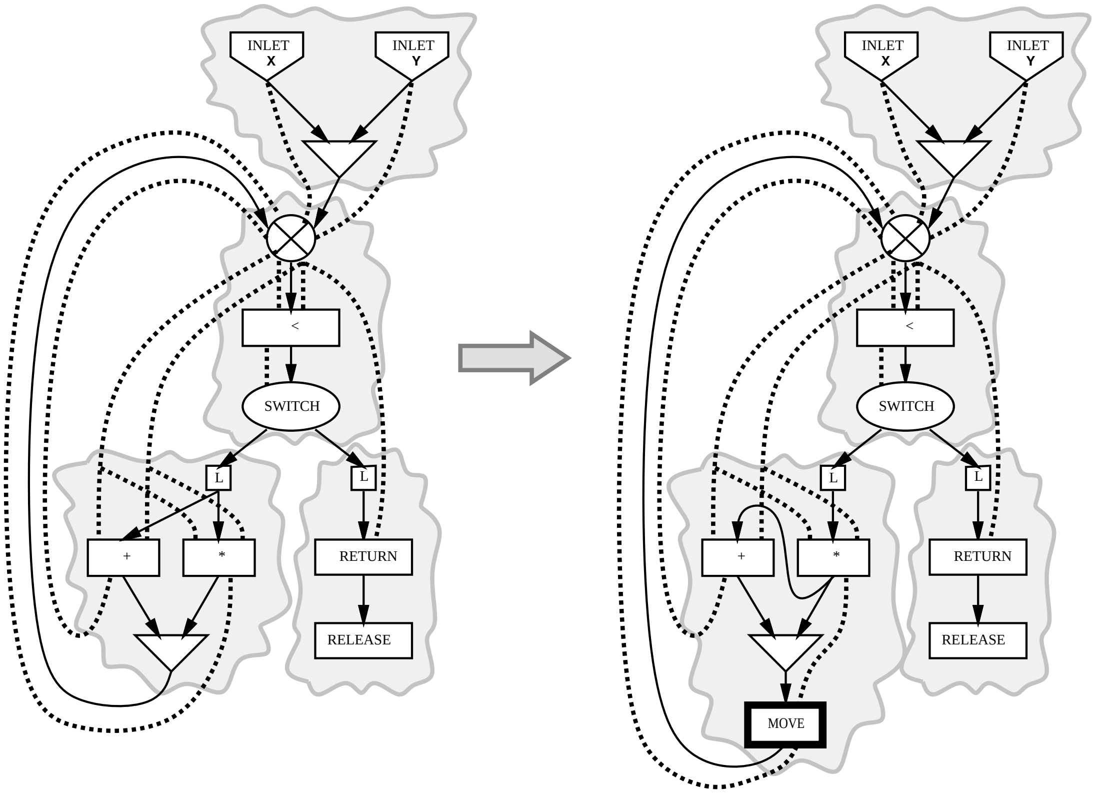

Figure 4.2: Partitioned dual graph before and after move insertion for function `loop_ex`.

body of the partition and, thus, do not require explicit synchronization. For each partition, the entry count of the corresponding thread is equal to the number of control arcs that enter the body from other partitions plus the number of nodes from the input region that have control arcs going to the body.

The initial entry counts are established by a designated initialization inlet. For threads appearing within a loop, the entry count is re-established by the thread. One way to determine all threads that appear in a loop is to find all strongly connected components of the dual graph. This approach has the disadvantage, though, that it will also identify strongly connected components created by circular dependencies from conditionals, as in the example `strange` from Figure 1.2. Since we are starting with a hierarchical program graph structure, where the loops are explicit, the easiest way is to mark at this stage all nodes that are inside the loop. For threads appearing within a loop, a move instruction is inserted that re-initializes its entry count.

4.7 Fork insertion

A partitioned dual graph contains switches where conditional forks are required, but does not contain forks. These are only determined after partitioning is complete. Where a control arc crosses from one partition to another, other than from a switch, a *fork* to the target partition is inserted.

4.8 Thread ordering

TAM provides a control transfer mechanism which forks threads for later execution. This could be exploited to fetch the next thread while completing the current one. Even so, by placing threads contiguously, a fork and a stop can often be replaced by a simple fall-through. Since existing processors cannot exploit fork-based control, the translator from TAM to target machine code moves the last fork or switch in a thread to the very bottom and replaces it by a branch. If the target is the next thread, this becomes a fall-through. Thread ordering is determined by a depth-first search starting from root partitions triggered only by inlets. An unnumbered adjacent partition 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. Figure 4.3 shows how thread numbers are assigned for the lookup example. Also given in Figure 4.4 is the TL0 code produced after all the steps discussed in this chapter. Note that threads 1 to 4 are placed contiguously. At machine code level the switch from thread 1 to threads 2 and 5, as well as the switch from thread 3 to 4 and 6 will be replaced by a single conditional branch, while the fork from thread 2 to 3 will turn into a fall-through.

The primary goal of this chapter was to describe the details of thread generation. Much of the actual implementation is still in preliminary form.

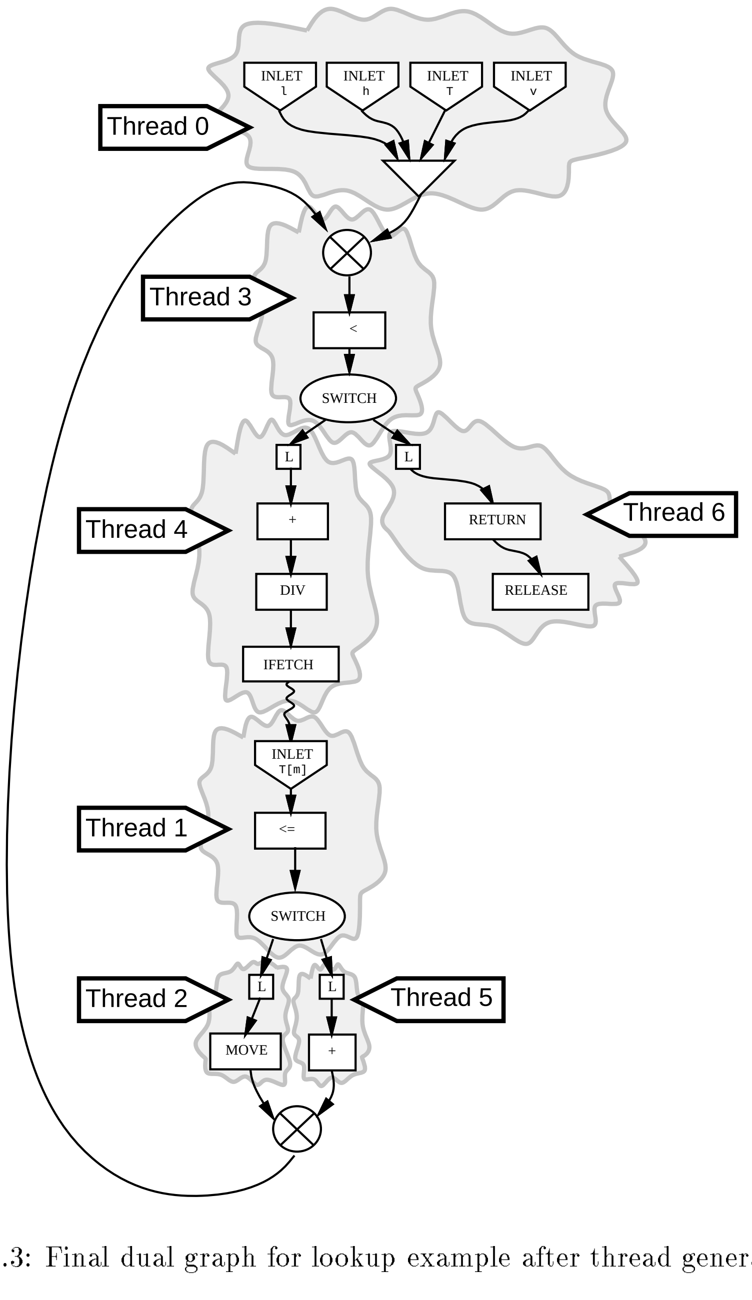

Figure 4.3: Final dual graph for lookup example after thread generation.

```
INLET 0                            % Inlet for trigger
   MOVE slot_sync0.s = 4.s         % initialize synchronization variables
   FORK THR0
   STOP
INLET 1                            % Inlet for first argument
   RECEIVE slot_T.ps               % receive pointer to I-structure T
   FORK THR0
   STOP
INLET 2                            % Inlet for second argument
   RECEIVE slot_l.i                % receive lower bound l
   FORK THR0
   STOP
INLET 3                            % Inlet for second argument
   RECEIVE slot_h.i                % receive upper bound h
   FORK THR0
   STOP
INLET 4                            % Inlet for second argument
   RECEIVE slot_v.f                % receive value v
   FORK THR0
   STOP
INLET 5                            % Inlet for I-fetch
   RECEIVE slot_T[m].f             % receive I-structure element T[m]
   FORK THR1
   STOP

THREAD 0
   SYNC slot_sync0.s              % Synchronize on arrival of all arguments
   FORK THR3
   STOP
THREAD 1                          % Check if T[m] < v
   LE reg_B.b = slot_T[m].f slot_v.f
   SWITCH reg_B.b THR2 THR5
   STOP
THREAD 2
   MOVE slot_h.i = slot_m.i       % change higher bound h = m
   FORK THR3
   STOP
THREAD 3                          % Check if l < h
   LE reg_B.b = slot_l.i slot_h.i
   SWITCH reg_b.b THR4 THR6
   STOP
THREAD 4                          % compute middle index m and fetch element
   ADD reg_tmp.i = slot_l.i slot_h.i
   DIV slot_m.i = reg_tmp.i 2.i
   IFETCH INLET5 <- slot_T.ps slot_m.i
   STOP
THREAD 5                          % change lower bound l = m + 1
   ADD slot_l.i = slot_m.i + 1.i
   FORK THR3
   STOP
THREAD 6                          % return result, release frame
   RETURN slot_l.i
   RELEASE
   STOP
```

Figure 4.4: TL0 code for lookup example

Chapter 5

Code Quality

This chapter presents preliminary data on the quality of TAM code produced under our compilation paradigm. Previous to this work, execution of Id90 programs was limited to specialized architectures or dataflow graph interpreters. By compiling via TAM, we have achieved more than two orders of magnitude performance improvement over graph interpreters on conventional machines, making this Id90 implementation competitive with machines supporting dynamic instruction scheduling in hardware[PC90, SYH+89, GH90, Ian88b]. Timing measurements show that our Id90 implementation on a standard RISC can achieve a performance close to Id90 on the recent dataflow machine Monsoon. Measurements of execution speed of small programs written in Id90 as well as the languages C and Lisp indicate that efficiency of Id90 falls between C and Lisp.

By constraining how dual-graphs are partitioned, we can generate TAM code that closely models other target architectures. It can be seen that the TAM partitioning described in the previous chapter reduces the control overhead substantially and that more aggressive partitioning would yield modest additional benefit. There is, however, considerable room for improvement in scheduling and register management.

5.1 Benchmarks

We use ten benchmark programs, as shown in Table 5.1, ranging up to 1,100 source code lines. *Lookup* is the small example program discussed above. The input is an array and a table of 10,000 elements. *AS* is an array selection sort, where the key is a function passed to the sort routine. The input is an array of 500 numbers. *QS* is a simple quick-sort using accumulation lists. The input is a list of 1,000 random numbers. *MMT* is a simple matrix operation test; two double precision identity matrices are created, multiplied, and subtracted from a third. The matrix size is 100×100. *Wavefront* computes a sequence of matrices, using a variant of successive over-relaxation. Each element of the new matrix is computed by combining the three new values to the north and west with value of corresponding element of the old matrix. Thirty iterations are run on matrices of size 100×100. *DTW* implements a dynamic time warp algorithm used in discrete word speech recognition[Sah91]. The size of the test template and number of cepstral coefficients is 100. *Speech* is used to determine cepstral coefficients for speech processing. We take 10240 speech samples and compute 30 cepstral coefficients. *Paraffins*[AHN88] enumerates the distinct isomers of paraffins of size up to 14. *Gamteb* is a Monte Carlo neutron transport code[BCS+89]. It is highly recursive with many conditionals. *Simple* is a hydrodynamics and heat conduction code widely used as an application benchmark, rewritten in Id90[CHR78, AE88]. One iteration is run on 50×50 matrices.

Program	Code Size (Lines)	Short Description	Input Size
lookup	18	Lookup example	10,000
as	70	Array sort	500
qs	55	Quick sort on lists	1,000
mmt	59	Matrix multiply test	100
wavefront	111	SOR on matrices	30 100 100
dtw	103	Dynamic time warp	100 100 100
speech	172	Speech processing	10240 30
paraffins	185	Enumerate isomers of paraffins	14
gamteb	649	Monte Carlo neutron transport	64
simple	1105	Hydrodynamics and heat conduction	1 2 50 2 50

Table 5.1: Benchmark programs

Our current compiler performs only a limited form of redundant arc elimination and does no switch or merge combining. Registers are used only for thread local values. TAM code can be expanded to run on MIPS, nCUBE, or (via C) several other platforms. The expansion can insert code to gather TAM-level statistics at run time. In this chapter we present only dynamic statistics.

5.2 Timings

To get a feeling of how efficient the language Id90 is, we re-wrote three small Id90 programs in C and Lisp and compared their execution speed. Table 5.2 shows the running time in seconds for the programs *AS*, *QS* and *MMT* under C, Id90/TAM and LISP. The programs were all run a MIPS R3000. The table gives two columns for C, using MIPS C compiler version 2.11 with different levels of optimizations. Id90 programs were first compiled to TAM and then expanded to native MIPS code. Lisp programs were compiled using Allegro CL 3.1.0 (speed 3, safety 0).

Program	Input	C (-O3)	C (-O2)	Id90 (TAM)	LISP
as	1500	2.1	4.4	7.8	55
qs	5000	0.6	0.6	0.7	2.4
mmt	200	15.2	20.7	31.0	441

Table 5.2: Run-time in seconds

For these programs execution speed using TAM is within a factor of 1 to 4 of C. Lisp always results in a slower execution speed than going via TAM. There are several reasons we expect execution of Id90 programs using TAM to be slower than equivalent C programs. TAM has to support dynamic scheduling, while the C code can be statically scheduled. TAM specifies a parallel execution model, the resulting activation tree is more expensive to manage than the activation stack provided by C. I-structure accesses in Id90 are synchronizing, a TAM implementation must associate tag-bits with each I-structure element and test their state. This is more costly than heap accesses in C which are just memory load/stores.

One approach to understanding the code quality is to compare it against alternative implementations of the same language. Difficulties in this approach arise because some of the machines supporting dynamic scheduling in hardware have not yet been built and others have not implemented Id90. Additionally, we have to take into account constantly changing hardware parameters, such as cycle time. We compared execution speed of Id90 programs on the recent dataflow machine Monsoon with that of TAM on the MIPS R3000, the run-times are shown in Table 5.3. The second column shows the inputs: *MM* multiplies two 500×500 matrices, *Wavefront* runs 144 iterations on 500×500 matrices, *Paraffins* enumerates the all the distinct isomers of paraffins up to size 19, *Gamteb* simulates 40,000 particles, and *Simple* runs 1 iteration on a problem size of 100×100. Column three gives the run-time for Monsoon, as reported in [Hic91]. The fourth columns uses the backend that directly translates TAM into native MIPS code, while the last column gives run-times for the backend that first expands TAM into C code and then compiles it for the MIPS. On Monsoon as well as in the native code TAM backend for MIPS, floating point numbers are represented as double precision numbers. The TAM backend translating to C represents them in single precision.

Program	Input	Monsoon	TAM (Mips R3000)	TAM (C MIPS)
mm	500	250	644	684
wavefront	500 144	300	239	227
paraffins	19	31	19	11
simple	1 1 100 1 100	19	40	28

Table 5.3: Run-time in seconds on Monsoon and TAM (on Mips R3000)

Monsoon performs well on the matrix multiply program. It runs 2.6 times faster. The code for Monsoon was highly optimized, with the innermost loop unfolded 10 times by the compiler. This optimization was not performed for TAM. Monsoon also performs well on *Simple*, it is twice as fast as executing the TAM program directly on the MIPS. *Simple* on Monsoon is only 1.5 times faster than the TAM/C implementation. TAM runs substantially better though, on *Wavefront* and *paraffins.* Here it is about 2.8 times faster. For some of the programs, the C version is faster than native code since it only uses single instead of double precision floating point numbers. This improves cache and memory performance. These measurements show that using stock hardware can result in an execution speeds comparable to machines that directly support dynamic synchronization in hardware. One has to take into account that the two machines are vastly different. The Monsoon is an experimental prototype, while the MIPS is a mature product. 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 did not see the benefits resulting out of fast inter-processor message handling, since we studied only a uniprocessor implementation. 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 MIPS is an M2000 with 128MB of memory running RISC/os 4.50. It consists of a 25 MHZ

R3000 processor chip, a MIPS R3010 floating point chip (peak rate of 3 MFLOPS) and a MIPS R3200 cpu board with 64KB data cache and 64KB instruction cache.

5.3 TAM vs Dataflow

To better understand the quality of TAM partitioning, we can constrain the compiler to produce code in the spirit of recent dataflow or hybrid architectures. These machines all provide a notion of execution thread and a specific synchronization mechanism. An instruction on these machines maps into multiple TAM instructions, providing a consistent cost metric for control, scheduling and message passing. We compare and produce code for the following forms of partitioning.

- **DF:** Threads produced by dataflow partitioning without merging reflects the limited thread capability of most dataflow machines. Hardware provides two-way synchronization as token matching on binary operations; unary operations do not require matching, so they can be scheduled into the pipeline following the instruction on which they depend.
- **DE:** Threads produced using dependence sets partitioning without merging correspond closely to Scheduling Quanta in Iannucci's hybrid architecture[Ian88b]. Iannucci integrates thread generation and register assignment to a limited extent; registers are assumed to vanish at every possible suspension point or control transfer. This style of register usage is incorporated in recent dataflow machines, including Monsoon[PT91], Epsilon[GH90] and EM-4[SYH+89], allowing partitioning similar to the hybrid model.
- **DE_ME:** For TAM we use our best partitioning: dependence sets partitioning with merging.

The dynamic measurements presented in this and subsequent sections were all collected on one node of a multiprocessor nCUBE/2. The distribution of instructions for the the ten benchmark programs is shown in Figure 5.1.a. For each program three columns are shown: dependence sets partitioning with merging (DE_ME), dependence sets partitioning without merging (DE), and dataflow partitioning without merging (DF). The final three columns give the arithmetic mean over all programs. The bar graphs show the distribution of instructions into classes: ALU, data moves, split-phase operations, instructions in inlets, control overhead, and moves needed to initialize or reset entry counts. For each program, the distributions are normalized with respect to DE_ME to better illustrate the relative costs. (Where possible numbers indicate size of containing box).

The number of ALU, data move, split-phase, and inlet instructions is independent of the type of partitioning. Under DE_ME, we have on average 20% ALU, 6% data moves, 10% split-phase, and 24% inlet instructions. An inlet will usually execute three instructions: one that receives the corresponding data value and stores it into the appropriate frame, a FORK which puts the corresponding thread into the remote continuation vector and enables the frame, and finally a STOP. The fraction of time spent in inlets may differ from instruction frequency, depending on how quickly the implementation can start the message handler, receive the message, and enable the corresponding thread.

The number of control instructions and entry count moves vary substantially under the different partitioning schemes. On average, 31% of the instructions are used for control

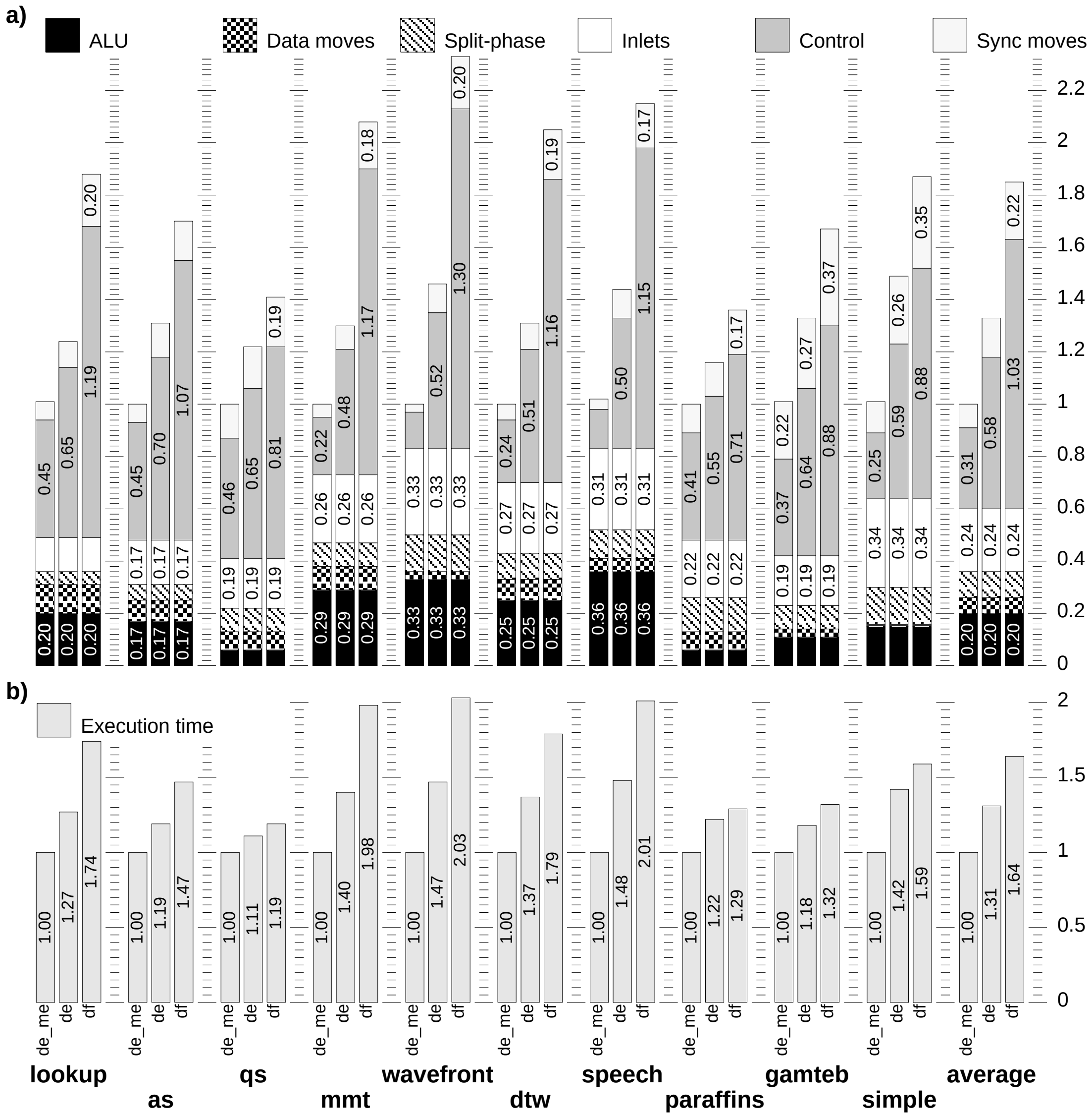

Figure 5.1: (a) Instruction distributions, (b) Relative execution times.

under DE_ME, less than twice the typical fraction of control operations in sequential languages. Without merging nearly twice as many control instructions are needed, and three times as many with only dataflow partitioning. Entry count moves follow the same trend.

Without merging, 1.33 times as many instructions are executed as with DE_ME. Dataflow partitioning yields about 1.85 times as many instructions executed. Comparing these partitioning styles highlights the effects of improved compilation. It is not meant to be a performance comparison between the three classes of machine, as merging could be employed to improve code quality for hybrid or dataflow machines to some extent, and actual performance depends on cycle time, specifics of operand fetch, instruction issue, etc. The qualitative difference is that the control portion is reduced with better partitioning. This gain derives from two sources. By increasing the thread size, a greater fraction of explicit control and synchronization operations are made implicit through instruction ordering. Secondly, separating control and data flow allows redundant synchronization between partitions to be identified and eliminated.

It is also possible for a restricted model to avoid the need to re-initialize synchronization variables, which could slightly improve the the total number of instruction executed with dataflow partitioning. For example, Monsoon and P-RISC[NA89] provide hardware support for two-way synchronization. They associate individual bits with each synchronizing operation or thread, and toggle the bit on each synchronization access. The first operand will set this bit, the second reset it. This scheme has the advantage that the value of the synchronization variable is automatically reset after synchronization has succeeded. Thus, any moves to re-initialize the synchronization variables could be omited. The moves could also be avoided by TAM, if we provide a more complicated synchronization primitive. The current synchronization operation decrements a counter and tests against zero. A more sophisticated synchronization operation would increment the counter, take the modulo with the entry count and test against zero.

The strong relationship between Figures 5.1.a and 5.1.b confirms that the reduction in control overhead translates into execution efficiency and that the TAM instruction as a cost unit is reasonable. With a more thorough analysis it might be possible to develop a performance model that predicts execution time given detailed TAM-level instruction counts.

5.4 Thread Characteristics

Average thread lengths are shown in Figure 5.2.a. Using DE_ME the average thread length is slightly over 5 instructions. While this is not large, it should be noted that control primitives in TAM fork threads, so thread length is expected to be close to typical branch distances. Also, global accesses are split-phase, so they initiate threads. Another reason for small thread sizes is that TAM is not a LOAD/STORE machine as it allows operations to directly specify operands in frame memory as well as in registers. Finally, we have not yet implemented the switch and merge combining discussed in the partitioning section. As a result, conditionals limit partition merging on some of the examples. This has not affected the two programs wavefront and speech, where partition merging had a big effect on thread size. Here the average thread size is 7.8 and 8.0 instructions. These large sizes arise because many requests were issued in one thread and all the responses received by another. Using only dependence sets partitioning without merging reduces the overall thread length to 3.5 instructions. This drops to 2.9 instructions using dataflow partitioning. For this kind of partitioning, the thread size does not vary very much among the different programs; it is always very low.

Improved partitioning not only changes the size of threads, it changes their structure. The stacked bars in Figure 5.2.b show the breakdown of threads into synchronizing and non-synchronizing (*i.e.*, entry count is one) relative to DE_ME. Without merging, roughly twice as many threads are executed and four times using only dataflow partitioning. The number of threads executed does not grow inversely to thread size, since the overall instruction count is reduced by better partitioning. With the exception of *Gamteb*, two-thirds of the threads are non-synchronizing, regardless of partitioning. Better partitioning reduces the number of both kinds of threads, although the average entry count for synchronizing threads increases. For DE_ME the average entry count varies between 3 and 8. Forks from a thread to a non-synchronizing thread are essentially jumps, however, non-synchronizing threads can also be forked by inlets in handling an incoming message or response.

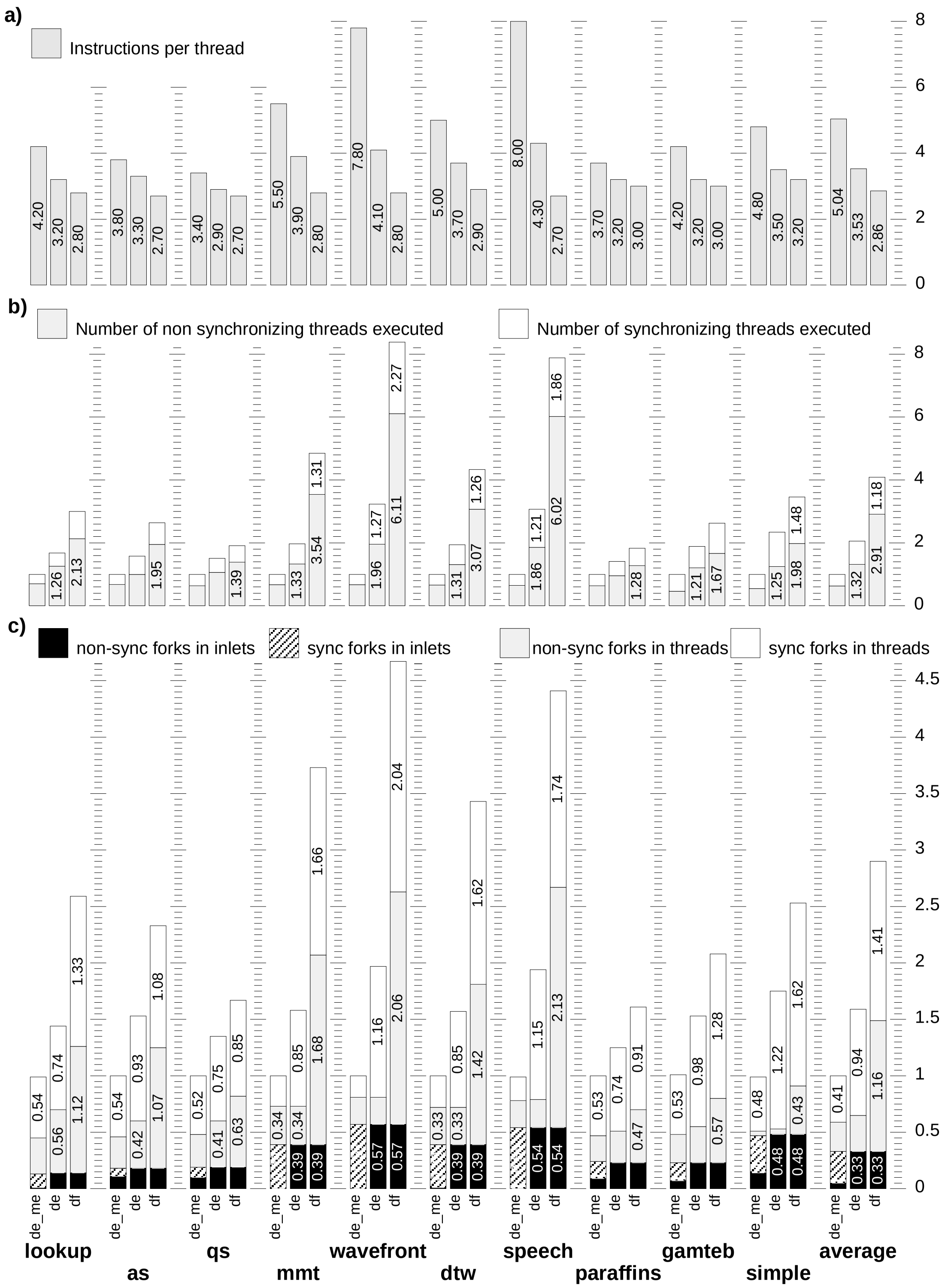

Figure 5.2: (a) Thread sizes, (b) Relative thread counts, (c) Control instruction distributions.

The effects of more sophisticated partitioning are more apparent in examining the fork operations. Figure 5.2.c shows the number of forks to synchronizing and non-synchronizing threads occurring in threads and inlets. The number of forks occurring in inlets is independent of partitioning, whereas the number of forks occurring in threads is reduced. However, as partitions are merged, inlet forks shift dramatically from non-synchronizing to synchronizing. Synchronizing messages in inlets means that frames are not activated until several operands have accumulated.

The data presented above was obtained without switch and merge combining. Implementing these will reduce the control portion further under TAM. Combining inlets, *i.e.*, sending multiple arguments as a single larger message, can be applied when the arguments feed the same partition; this will reduce the number of inlet instructions. Global strictness analysis could be applied to attempt to reduce these two components further, but the lower bound on control is the branch frequency and on inlets it is the frequency of split-phase operations. Keeping in mind that the target is parallel execution, not complete sequentialization, it appears that DE_ME partitioning is approaching the "knee of the curve."

5.5 Dynamic Scheduling

The second aspect of TAM compilation is management of processor and storage resources in the context of dynamic scheduling. Dual graphs were developed specifically to attack this problem, although our current compiler uses only registers in threads. Table 5.4 shows the dynamic scheduling behavior of programs under TAM using DE_ME.

Program	Activations	Quanta	Threads	Instructions	QPA	TPQ	IPQ	IPT
lookup	10002	20003	1105342	4683207	2.0	55.3	234.1	4.2
as	503	1005	2386788	9046137	2.0	2374.9	9001.1	3.8
qs	6004	14007	408945	1393260	2.3	29.2	99.5	3.4
mmt	10506	21013	3285714	18161654	2.0	156.4	864.3	5.5
wavefront	3196	6485	966555	7512370	2.0	149.0	1158.4	7.8
dtw	30402	61400	3593989	18131827	2.0	58.5	295.3	5.0
speech	4362	11156	1342402	10802677	2.6	120.3	968.3	8.0
paraffins	2841	5750	203068	758808	2.0	35.3	132.0	3.7
gamteb	13081	34837	661931	2792257	2.7	19.0	80.2	4.2
simple	58674	182097	1560974	7529351	3.1	8.6	41.3	4.8

Table 5.4: Dynamic Scheduling Behaviour

We assume zero latency, so I-Fetches return immediately, unless deferred. The table shows the number of code-block activations, quanta, threads and TAM instructions executed. Also shown are ratios of these. The number of quanta per activation (QPA) is small, usually between 2 and 3. Thus, the cost of swapping to another code-block activation (roughly 10 instructions) is paid infrequently. Although thread sizes (IPT) are small, quanta generally contain many threads (TPQ), so quantum based register allocation stands to make much better use of registers. Where the compiler fails to discover that two computations can be put into the same thread, often the two computations will occur in the same quantum. The cost of synchronizing them is not large — it requires that the entry count be decremented and tested. This dynamic scheduling behavior — although collected on a sequential machine — indicates the value of TAM's scheduling hierarchy.

In practice the full power of non-strict languages which requires dynamic scheduling of small threads is seldom used. The compiler has to be conservative, but TAM scheduling paradigm exploits the typical case by executing as many threads as possible for a frame in a single quantum. If a code-block is called in a strict manner and all the arguments arrive close together in time, all of the threads for the activation may execute within a single quantum. Only if it runs out of useful work after making split-phase requests or calls to other code-blocks will it execute in multiple quanta.

Our results show that better partitioning schemes would achieve only modest additional benefit. Our current compiler does not exploit the scheduling hierarchy provided by TAM. The data presented in this section indicates, however, that there is considerable room for improvement in scheduling and register management.

Chapter 6

Conclusion and future work

In this thesis we have shown that it is practical to implement Id90, a functional non-strict language, on conventional sequential machines without hardware support for fast dynamic scheduling. Functional languages provide a good basis for a parallel computing, since they can offer ample implicit parallelism. Non-strictness can substantially enhance the parallelism by allowing functions and arbitrary expressions to execute and possibly return results before all arguments have been provided. Unfortunately, non-strict languages require dynamic scheduling. This makes efficient implementation on conventional machines difficult. We demonstrate how such a language, the parallel language Id90, can be compiled into threads for the threaded abstract machine TAM. TAM allows the compiler to exploit locality at several levels, and can efficiently be implemented on conventional machines.

Id90 programs are first translated into program graphs, a hierarchical graphical intermediate form that facilitates powerful high level optimizations. The meaning of program graphs is given in terms of a dataflow firing rule, so control flow is implicitly prescribed by the dynamic propagation of values. In our threaded execution model, control is explicit and the flow of data is implicit in the use of registers and frame slots. In order to bridge this gap, we introduce a new graphical intermediate form, dual graphs, in which control and data flow are both explicit. Dual graphs are generated by expanding dataflow program graph nodes into equivalent dual graphs. Compilation to TAM then involves a series of transformations on the dual graph, including partitioning, lifetime analysis, scheduling and linearization, register and frame-slot allocation, and fork insertion. Our approach has produced the first efficient implementation of Id90 on conventional machines.

By compiling harder, we achieve execution times that are comparable to those of the recent dataflow machine Monsoon. This indicates that hardware support for dynamic scheduling is less valuable than previously thought. Our results show that there is no reason to build a uniprocessor dataflow machine to support non-strict languages, since its performance is within a factor of 2 of a standard processor. Also, run-time measurements of small programs re-written in C and Lisp show that the C programs execute a factor 1 to 4 faster than the equivalent Id90 programs; the Lisp programs always execute slower. Reasonably sophisticated partitioning is required for Id90; we describe a partitioning algorithm that is practical to employ on real programs and results in control overhead within a small factor of the typical cost of control flow in sequential languages. Measurements under several partitioning strategies show that arithmetic, data movement, heap access, and message handling costs are invariant with respect to the partitioning strategy. However, partitioning has substantial impact on control overhead; not using merging with dependence sets partitioning doubles control overhead. The overhead triples when only dataflow partitioning is used. This results in 1.8 times more instruction being executed. Studying the effects on threads, we find that the better the partitioning the fewer threads (but also the fewer total instructions) are executed. The dynamic scheduling behavior shows the value of TAM's scheduling hierarchy. On the average each activation can be executed in a little more than two quanta. Although the number of instructions per thread is relatively small, the number of threads per quanta is quite high. Our implementation on conventional state-of-the-art processors provides a baseline against which novel multithreaded machines can be judged.

While these results are encouraging, there is considerable room for improvement. Partitioning of conditionals will improve significantly when switch and merge combining are implemented. Redundant arc elimination is currently quite primitive. On a larger scale, more extensive analysis, such as strictness analysis and propagation of dependence through conditionals and function calls, can be used in partitioning. Furthermore, much of the power of TAM has not yet been exercised, including the management of processor registers across threads. Using these techniques, we expect to roughly double the performance.

Our study indicates the great potential in TAM's ability to carry registers across threads, even if the compiler cannot statically prove that they will be executed in the same quantum. This form of speculative register assignment will be implemented in the near future, together with an execution vehicle for TAM on a large parallel distributed memory machine. This will allow us to study the execution behavior of parallel programs on machines consisting of thousands of processors.

Bibliography

[ACC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proc. of the 1990 Int. Conf. on Supercomputing*, pages 1–6, Amsterdam, 1990.

[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. Revised October, 1984.

[ACM88] Arvind, D. E. Culler, and G. K. Maa. Assessing the Benefits of Fine-Grain Parallelism in Dataflow Programs. *The Int. Journal of Supercomputer Applications*, 2(3), November 1988.

[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.

[ALKK90] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proc. of the 17th Ann. Int. Symp. on Comp. Arch.*, pages 104–114, Seattle, Washington, May 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.).

[AU77] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley Pub. Co., Reading, Mass., 1977.

[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.

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

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Proc. of the 16th Annual ACM Symp. on Principles of Progr. Lang.*, pages 25–35, Los Angeles, January 1989.

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

[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).

[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).

[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.

[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.

[Hal85] R. H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HF88] R. H. Halstead, Jr. and T. Fujita. MASA: a Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proc. of the 15th Int. Symp. on Comp. Arch.*, pages 443–451, Hawaii, May 1988.

[Hic91] J. Hicks. Report on Running Id Applications on Monsoon. Technical Report FLA memo, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, March 1991.

[Ian88a] 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).

[Ian88b] R. A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proc. 15th Int. Symp. on Comp. Arch.*, pages 131–140, Hawaii, May 1988.

[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.

[Nik90] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo, to appear, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, 1990.

[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. Also: CSG Memo 313, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA.

[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.

[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.

[Sah91] A. Sah. Parallel Language Support for Shared memory multiprocessors. Master's thesis, Computer Science Div., University of California at Berkeley, May 1991.

[Smi90] B. Smith. Keynote Address. Proc. of the 17th Annual Int. Symp. on Comp. Arch., May 1990.

[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.

[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).

[vESC91] T. von Eicken, K. E. Schauser, and D. E. Culler. TL0: An Implementation of the TAM Threaded Abstract Machine, Version 2.1. Technical Report, Computer Science Div., University of California at Berkeley, 1991.