Compiler-Controlled Multithreading for Lenient Parallel Languages[1]

Klaus Erik Schauser
David E. Culler
Thorsten von Eicken

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

Abstract: Tolerance to communication latency and inexpensive synchronization are critical for general-purpose computing on large multiprocessors. Fast dynamic scheduling is required for powerful non-strict parallel languages. However, machines that support rapid switching between multiple execution threads remain a design challenge. This paper explores how 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 its effectiveness are discussed. Dynamic execution measurements are obtained via a second compilation step, which translates TAM into native code for existing machines with instrumentation incorporated. These measurements show that the cost of compiler-controlled multithreading is within a small factor of the cost of control flow in sequential languages.

1 Introduction

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 Id[Nik90] and Multilisp[Hal85], require that small execution threads be scheduled dynamically, even if executed on a uniprocessor[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. In this paper, we address 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.

The lenient parallel language Id90 is taken as a starting point for the study, using the MIT compiler to produce dataflow program graphs[Tra86]. A new intermediate form, dual graphs, is introduced to provide a

[1] A version of this report is to appear in the Proceedings of FPCA '91 Conference on Functional Programming Languages and Computer Architecture, Aug. 1991, Springer Verlag.

vehicle for integrated treatment of partitioning, 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 and how partitioning and thread generation are performed. Finally, code quality is evaluated on several benchmarks.

2 Language Issues

Several studies have demonstrated that by exposing parallelism at all levels ample parallelism is available on a broad class of programs[ACM88, Cul90, AE88, AHN88]. Exposing parallelism at all levels requires that functions or arbitrary expressions be able to execute and possibly return results before all operands are computed. Data structures must be able to be accessed or passed around while components are still being computed. In language terms, this means functions, expressions, and data structures are non-strict, but not lazy. Traub has termed this class of languages *lenient*.

We begin with a 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 issues. The function `lookup_array`, as shown below, takes an array `A` of values and an ordered table `T` and returns an array of the table indexes corresponding to the values in `A` computed by `lookup`. Although there is little parallelism in the `lookup` function, 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. Thus, we want to execute several lookups on each processor and be able to switch among them cheaply upon remote or deferred access. (The `lookup` function is used throughout the paper to illustrate the compilation process.) The function `flat` produces a list of the leaves of a binary tree using accumulation lists. If `cons` and `flat` are strict, this exhibits no parallelism. Under lenient execution, the list is constructed in parallel[Nik91]. 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` is available, which enhances parallelism. In fact, it must be able to do so, since the first result can be used as the second argument, as in the unusual function `cube`. The final example, due to Traub[Tra88], has three mutually recursive bindings, where the cyclic dependence through the conditional must be resolved dynamically.

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

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 models[Ian88, NA89], 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 that dynamic scheduling is done without hardware support. This means that register management can be closely tied to thread scheduling in order to minimize the overhead where dynamic scheduling is required.

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 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. A primary design goal in TAM is to provide a means of exploiting locality, even under asynchronous execution, to minimize the overhead of multithreading.

A TAM program is a collection of *code-blocks*, typically representing functions in the program text. Each code-block comprises several *threads* and *inlets*. 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 and to processor registers. A frame is said to be *resident* when a processor is executing threads relative to the frame. A resident frame continues executing 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. *Stop* terminates its thread and causes some other enabled thread to begin execution. A synchronizing thread has associated with it a frame slot containing its *entry count*. The entry count is explicitly set prior to any fork of the thread. Each time the thread is forked, its entry count is decremented; the thread is enabled when the entry count reaches zero. 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. Fork essentially puts a thread into a work pool that is serviced continuously, while multiple long latency requests and synchronization events are outstanding.

TAM assumes that an activation executes on a single processor; work is distributed over processors at the activation level. Thus, passing arguments and results between frames may involve 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 extracts data from a message, deposits it into specific slots in the designated frame, and forks threads for the corresponding activation. Inlets are compiler generated message handlers that quickly integrate message data into the computation. An inlet may interrupt a thread, but does not disturb the current quantum; threads enabled by the inlet will run when their 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 write takes place. I-structure operations generate a request for a particular heap location and the response is received by an inlet. Meanwhile, the processor continues with other enabled threads.

Scheduling in TAM is under compiler control and tied closely to the storage hierarchy. The first level of scheduling is static — grouping and ordering instructions into 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. Values can be transmitted in registers between threads that the compiler can prove will execute in the same quantum. When no enabled threads remain, another activation with enabled threads must be made resident. This also is 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. Empirically, quanta often cross many points of possible suspension[CSS+91]. Thus, it is advantageous to keep values in registers between threads that the compiler cannot prove will execute in a single quantum. The compiler can construct leave and enter threads that 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. These constraints simplify treatment of the language requirements. 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.

4 Dual Graphs

Compilation of Id90 to TAM begins after generation of dataflow program graphs[Tra86]. Program graphs are 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 TAM, 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 similar in form to data structures used in most optimizing compilers, but 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, described below.

A **dual graph** is a directed graph with three types of arcs: data, control and dependence.

- **Data arcs:** A data arc $(u, o) - -(v, i)$ specifies that the value produced by output o of node u is used as operand i by node v. A node may have zero or more data output ports; each with a bundle of data arcs. Each port represents a name (*i.e.*, a memory location) to which a value can be bound (one producer) and accessed (multiple consumers).

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

- **Dependence arcs (split-phase long-latency arcs):** A dependence arc, $u \rightsquigarrow v$ specifies that inlet instruction v will be scheduled as an indirect consequence of executing outlet instruction u.

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 (merge nodes are the only exception to this rule). 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 appear on control inputs only if corresponding data inputs have been produced. It is the task of the compiler to ensure this. As shown in Figure 1, there are eight types of nodes.

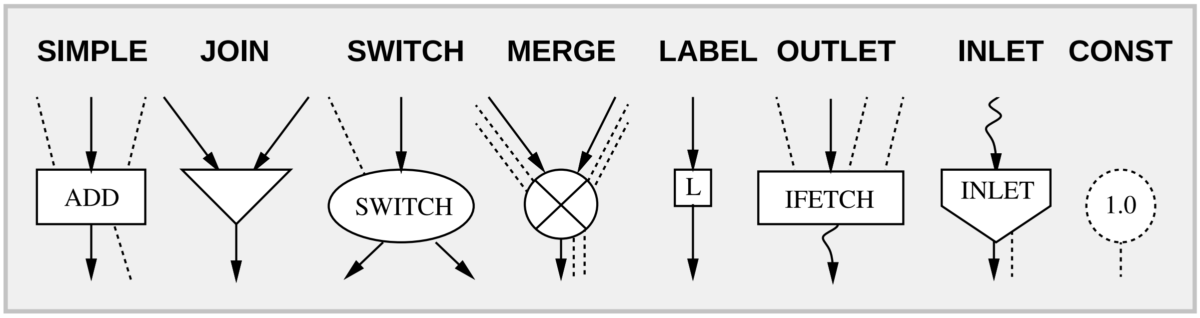

Figure 1: Dual Graph Nodes

- A **simple node** describes an arithmetic or logic operation. It has a single control input, a data input for each operand, a single control output port (the successors), and typically a single data output port (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 output control arcs.
- A **join** synchronizes control paths. It has multiple control inputs and a single control output port. Control passes to its control output once a token arrives on every control input.
- A **switch** conditionally steers control. It has a control input and a boolean data input; control passes on to one of its control outputs depending on the value of the data input.
- A **merge** complements the switch by steering control from one of many control inputs to a single control output. (It is the only node that is not strict in its control inputs.) It also unifies the data inputs associated with the active control input to the data outputs, *i.e.*, the input data is bound to the output name. A merge has multiple matching input sets, each with a control input and zero or more data inputs. The output ports have the same topology. In the final code generation for TAM, merge may expand into collections of data moves.
- A **label** indicates a separation constraint; the adjacent nodes must be in distinct threads. It has one control input port and one control output port. (In generating dual graphs, a label is placed on each output of a switch, reflecting the fork-based control primitives in TAM.)
- An **outlet** sends a message or initiates a request. 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 that receive responses.
- An **inlet** receives a message or split-phase response. It may have a dependence input and has one control output and zero or more data outputs. It will receive values corresponding to its data outputs and pass control to the operations connected to its control output. Usually its dependence input will be connected to the dependence output of a node that sends a split-phase request indicating that it will handle the response. Inlets that receive arguments are identified by convention.

- A **constant** node represents a manifest constant. It has a data output, but neither a control input nor a control output, since the value is known at compile time.

Dual graphs are generated by expanding dataflow program graph instructions. This is a local transformation described by expansion rules for individual program graph nodes[Sch91]. A program graph arc expands into a data arc and a control arc in the dual graph. In many cases, one or the other will prove unnecessary and be eliminated.

The program graph for the `lookup` example includes a function DEF node, enclosing a LOOP node, enclosing a IF node, as shown in Figure 2. The figure shows the corresponding dual graph representation, using a 1-bounded loop[Cul90]. The four arguments enter at the inlet nodes at the top of the graph. Since the function is strict in all its arguments, their control outputs are joined before the merge. The data arcs (shown dashed) 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 do 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, producing 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 at the bottom of the loop and directed to the loop merge.

The dual-graph for an Id program could be executed directly, but the number of dynamic synchronizations per useful operation would be 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 sequencing of instructions in a thread. Identifying portions of the dual graph that can be executed as a thread is called partitioning.

5 Partitioning

The key step in compiling a lenient language for a machine that executes instruction sequences is partitioning the program into statically schedulable entities[Tra88]. Fundamental limits on partitioning are imposed by dependence cycles that can only be resolved dynamically. In Id90 these arise due to conditionals, function calls, and access 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. Partitioning uses only the control and dependence arcs of the dual graph. Assignment of storage to output ports is deferred until after partitioning; the critical information is retained in the data arcs.

Definition 1 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.

Definition 2 (Safe Partition) We call a TAM partition *safe* if (i) no output of the partition needs be produced before all inputs to the body are available, (ii) when the inputs to the body are available, all nodes in the body are executed, and (iii) no arc connects a body node to an input node of the same partition.

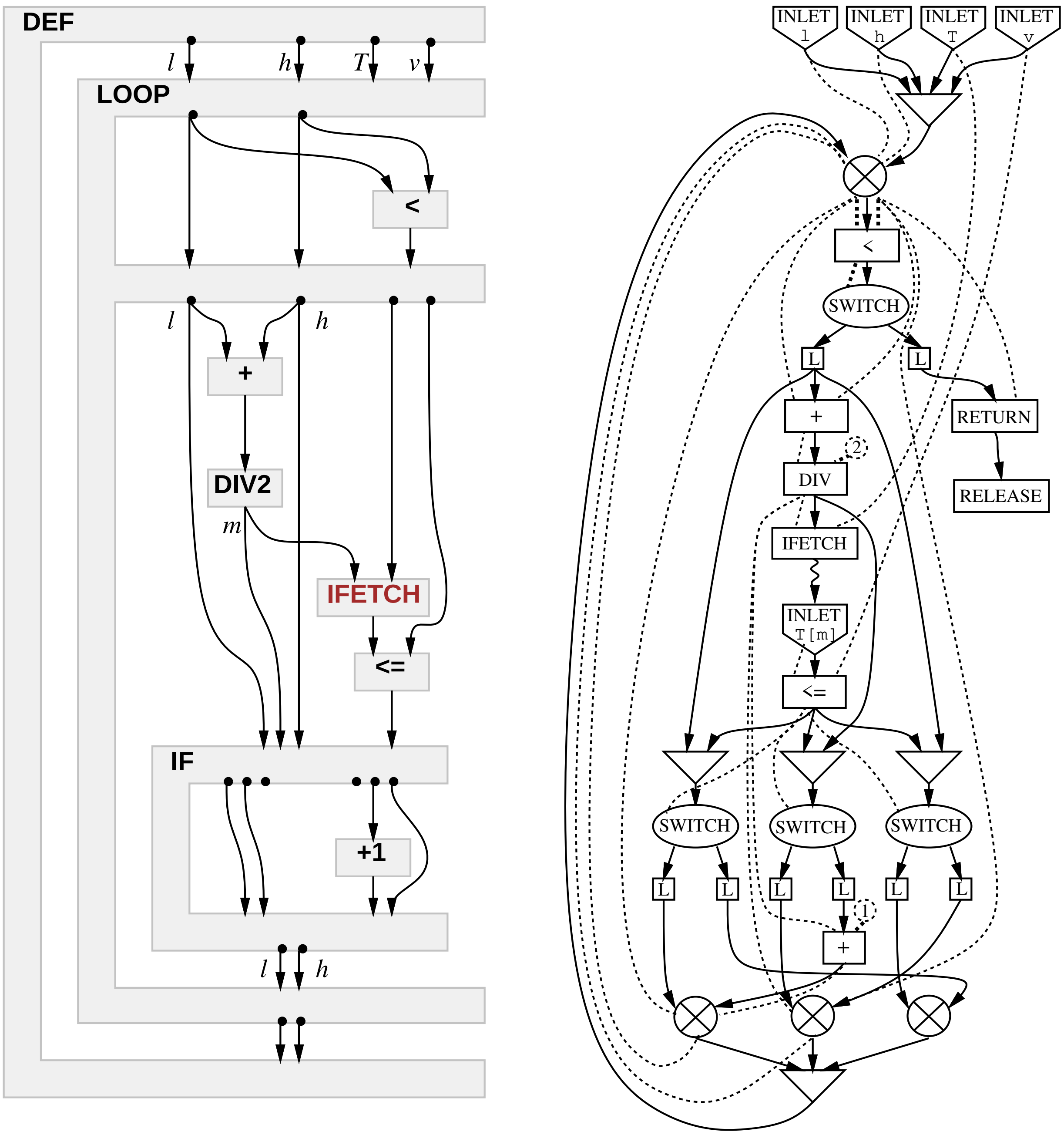

Figure 2: Program graph and dual graph for `lookup` example

The first property says that the body of the partition can be treated as strict. 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, since all cycles include a switch and a merge, and can be linearized by a topological sort on control arcs. Also, all dependence arcs must cross partitions. Finally, the entry count for any valid execution of the partition is constant. These properties imply the following lemma. (We will omit proofs throughout this paper. The interested reader is referred to [Sch91].)

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

Our partitioning algorithm first identifies small safe partitions. Then, these *basic partitions* are iteratively *merged* into larger safe partitions by applying simple merge rules, eliminating redundant control arcs, and combining switches and merges until the process converges. Below, we describe creation of basic partitions and discuss the merge and elimination rules.

5.1 Basic Partitioning

A simple method of basic partitioning observes that unary operations never need dynamic synchronization; thus, joins, inlets, merges and labels each start a new partition. Simple, switch and outlet nodes are placed into the partition of their control predecessor. We call this **dataflow partitioning** because each partition is like a dataflow actor, enabled by a binary match or arrival of a message. Note, that this form of partitioning puts fan-out trees into a partition.

Dependence sets partitioning is far more powerful. It finds safe partitions by grouping together nodes which depend on the same set of input nodes (inlets, merges and labels). This guarantees that there are no cyclic dependences within a partition. This is a variant of Iannucci's method of dependence sets [Ian88]. It groups overlapping fan-out trees into a partition. Dependence sets is powerful and practical to implement.

Definition 3 (Dependence Set) The *dependence set* for a dual graph node u is the set of input nodes i such that there exists a control path of length zero or more from i to u that does not go through any other input node.

To compute the dependence sets, assign each input node the dependence set containing only itself and for all other nodes assign the union of the dependence sets of the control predecessors. Our definition will not allow dependence to cross switches because every control output of a switch will be connected to a label.

One can propose many other partitioning schemes. For example, **dominance sets partitioning** finds safe partitions by grouping together nodes which dominate the same set of output nodes (outlet nodes and nodes that directly feed a control input of an merge or label).

Lemma 2 (Basic Partitioning) *Dataflow partitioning, dependence sets partitioning and dominance sets partitioning create only safe partitions.*

5.2 Merging partitions

After basic partitioning, partitions will be merged into larger safe partitions by iteratively applying two merge rules.

Merge up rule: Two partitions α and β can be merged into a larger partition γ if
(i) all input arcs to β come from α, (ii) β contains no inlet nodes, and (iii) no output arc from the body of α goes to an input node in β.

Figure 3 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 last point

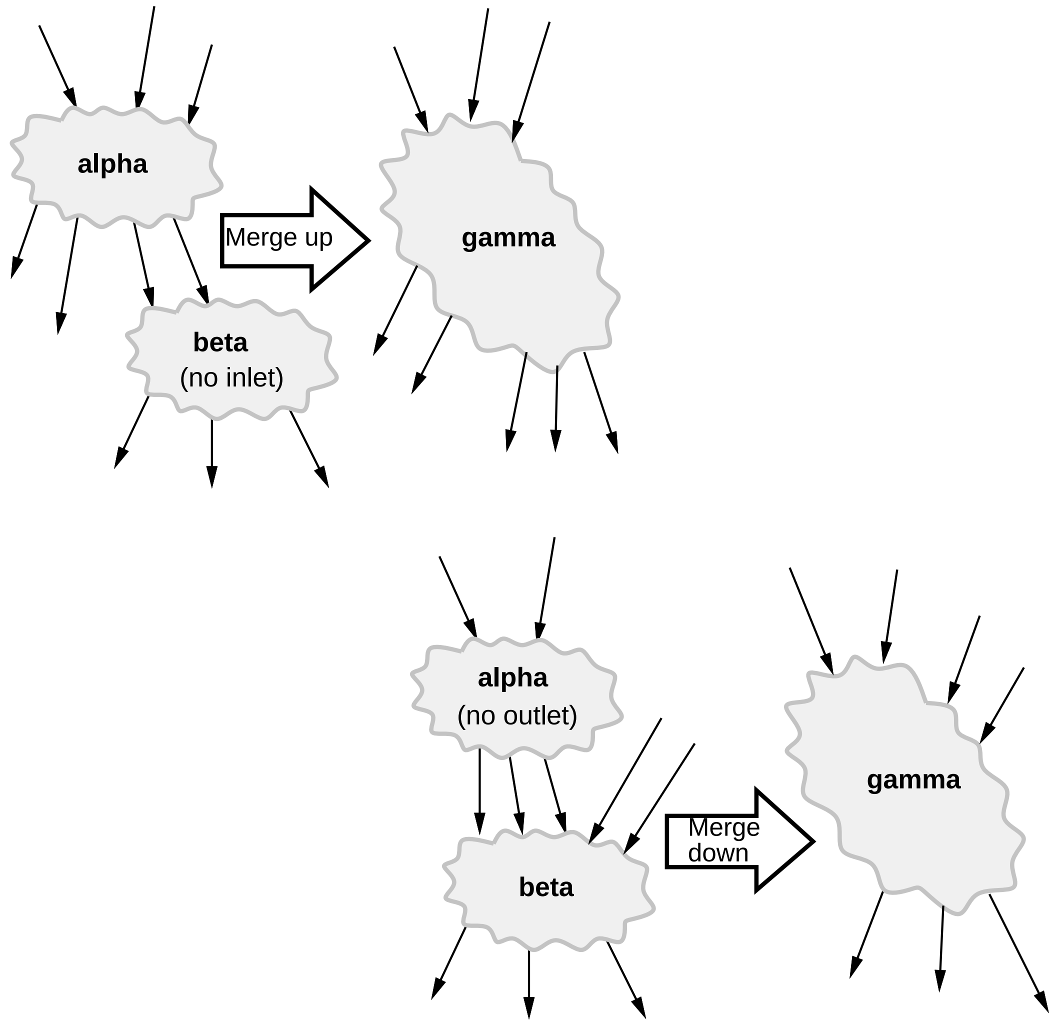

Figure 3: Merge rules: A partition with a single control predecessor can be *merged up* if no "separation constraints" are violated. A partition can be *merged down* into a successor, if the results of the partition feed strictly into the successor.

allows body nodes in α to be connected with body nodes in β. However, they cannot be connected to input nodes so that no separation constraint is violated. The astute reader will notice that this rule cannot be applied after basic partitioning. Opportunities for this rule arise as a result of "merging down".

Merge down rule: Two partitions α and β can be merged into a larger partition γ if
(i) all output arcs from α go to β, (ii) α contains no outlet nodes, and (iii) no output arc from the body of α goes to an input node of β.

The first two points of the rule ensure that α has no side-effect on an input of β. The last point guarantees that no separation constraint is violated.

Lemma 3 (Merge) *If α and β are safe partitions meeting the conditions of the "merge up" rule, then the merged partition is safe, and similarly for the "merge down" rule.*

The synchronization cost for a partition is proportional to the number of control arcs that enter the body from other partitions plus the number of nodes in the input region that have control arcs going to the body. For a merged partition, this number can never be greater than the sum of the synchronization costs of the two unmerged partitions. Thus we have the following.

Lemma 4 *Applying the partitioning merge rules will never increase the synchronization cost.*

The quality of partitioning after merging depends strongly on basic partitioning. Dependence sets partitioning always produces better partitions than dataflow 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. It is possible to find examples where dependence sets partitioning is superior to dominance sets partitioning, and vice versa[Sch91]. A better choice may be to combine the two forms of basic partitioning.

5.3 Redundant Arc Elimination

The goal of redundant arc elimination is to reduce synchronization cost. Eliminating a control arc between two partitions avoids a fork and decreases the entry count of the target partition. A control arc from partition u to v is redundant if there exist 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. Eliminating the redundant arc is a simple transformation on the control portion of the dual graph; the data arcs are unchanged. However, identifying candidates for elimination can be expensive, so limitations are placed on the search.

Redundant control arcs within a partition can be ignored, since the partition will eventually be linearized into a sequential thread. Elimination of arcs between partitions improves the quality of partitioning, since it may enable additional merges. Thus, after each partition merge the incident arcs to the new partition should be checked for redundancy.

5.4 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 reduce the control transfer overhead where the full power of lenient conditionals is not required. Switch combining is a simple transformation on the control graph, as shown in the top part of Figure 4.

Merges that are in the same partition and 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 4. This optimization serves primarily to reduce synchronization costs by enabling further merging of partitions within the arms of the conditional.

5.5 Partitioning algorithm

We can now summarize our partitioning algorithm for TAM:

Algorithm 1 (Dependence sets partitioning with merging)

- *Compute the dependence sets for all nodes,*
- *Put all nodes with the same dependence set into the same basic partition,*
- *iteratively apply the merge rules (and optionally redundant arc elimination, switch combining, and merge combining) until no rule applies.*

The previous lemmas imply the following correctness theorem.

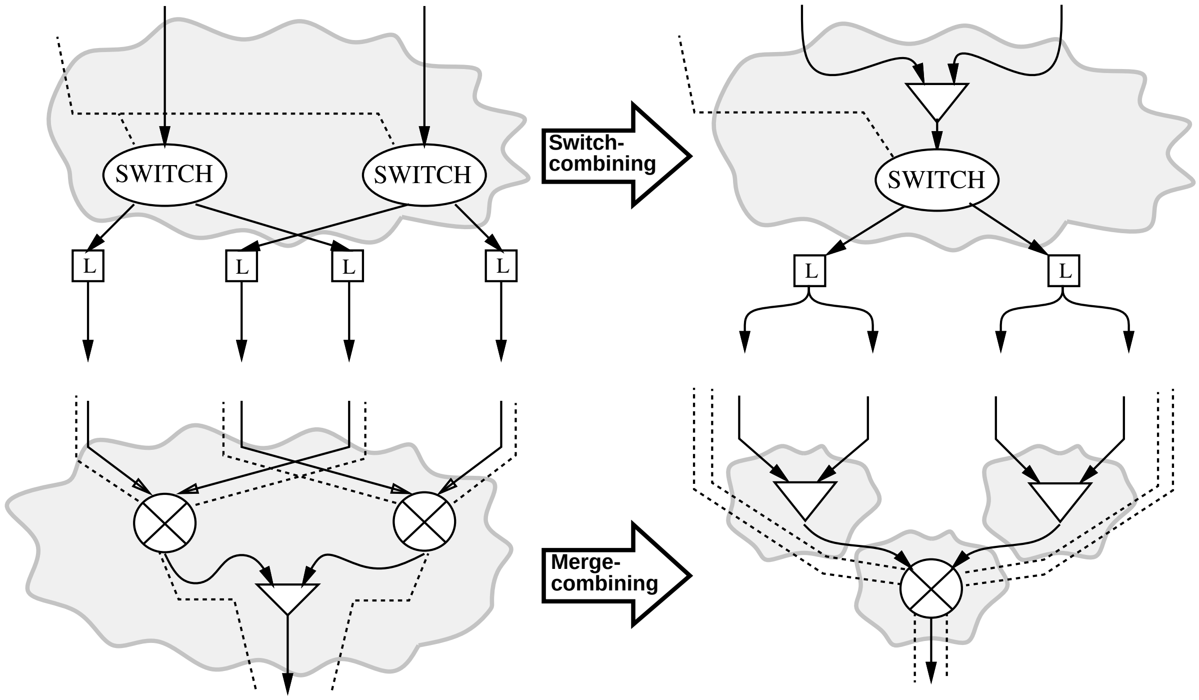

Figure 4: Switch and Merge Combining

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

Traub showed that optimal partitioning is NP-complete. Our algorithm is a heuristic, since starting with basic partitions, it will iteratively merge 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 minimize the synchronization cost. In Section 7 we compare partitioning according to the algorithm above against dependence sets basic partitioning without merging and simple dataflow partitioning.

5.6 Partitioning the lookup example

To illustrate the partitioning process, we consider the `lookup` example from Figure 2. Grouping nodes with the same dependence set, we get 20 basic partitions. Iteratively merging partitions using the two merge rules, yields twelve partitions as shown in the left part of Figure 5. The example contains three redundant arcs connecting to the `joins` above the three `switches`. One comes from the control output of the `div`, the other two 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 final partitioned dual graph shown in the right part of Figure 5.

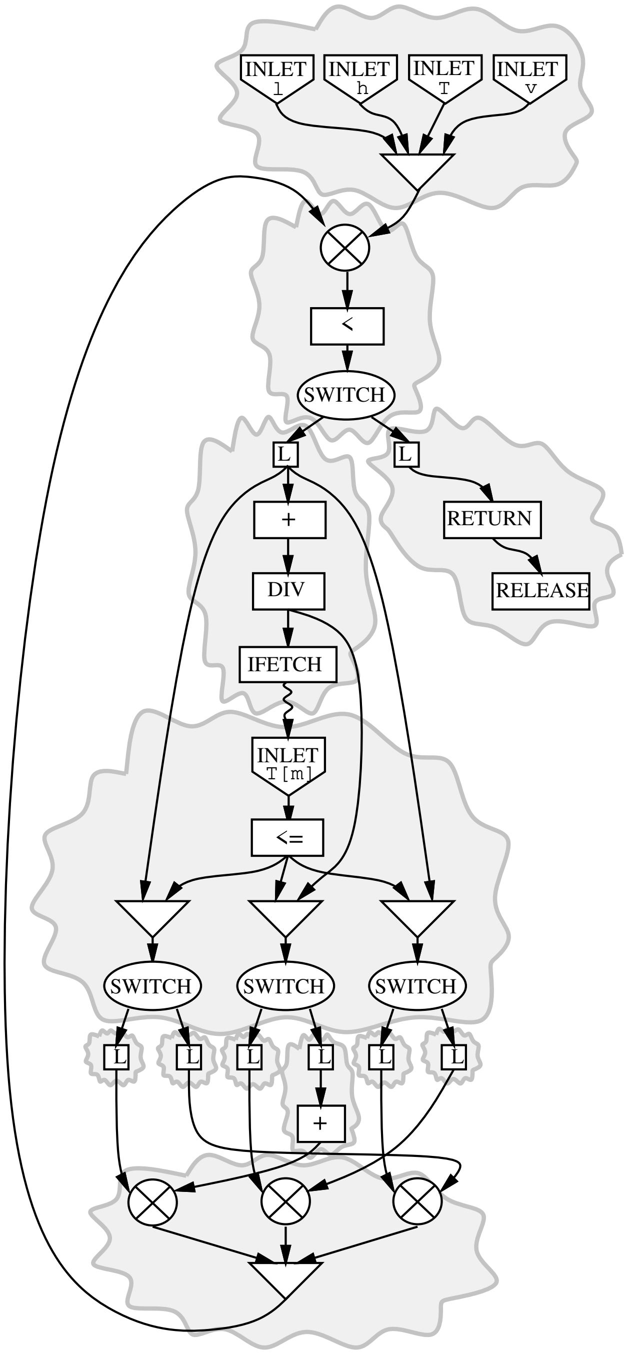

Figure 5: Dual Graph for `lookup` after dependence sets partitioning and merging. Right side shows graph after redundant arc elimination, switch and merge combining, move-insertion and thread ordering

6 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) instruction scheduling, (3) frame slot and register assignment, (4) move insertion, (5) entry count determination, (6) fork insertion, (7) thread ordering, and, finally, assembly.

Lifetime analysis: Determining whether a value can be stored in a register or needs to be placed in a frame slot involves a simple lifetime analysis using the data arcs of the dual graph. If all targets of a data output are in the same partition as the source node, the lifetime of the value is limited to a single thread. It can safely be placed in a register. More generally, values can be carried in registers across threads, as long as the threads are guaranteed to execute in the same quantum.

Instruction scheduling: The dual graph partitions are a partial order and must be linearized. Linearization influences the lifetime of values and thus has an effect on the register and frame slot usage. Furthermore, optimal instruction scheduling depends on pipeline structure and register availability. Although dual graphs are well suited for this kind of optimization, our current heuristic merely attempts to minimize the overlap of the lifetimes of values.

Frame slot and register assignment: In order to reduce frame size and the number of registers required, storage is reused for distinct data outputs that have disjoint lifetimes. An appropriate interference graph is constructed for each storage class using the data arcs in the linearized dual graph. It contains a node for each value and an edge between two nodes if their lifetimes overlap. Coloring this graph so that all vertices connected by an edge have different colors gives a valid register and frame slot assignment. Whereas in sequential languages the uncertainty in interference arises because of multiple assignments under unpredictable control paths, in compiling ID to TAM the uncertainty arises because of dynamic scheduling.

Move insertion: A merge node has a single control output port and various control input ports, each with some number of data ports. The compiler must 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. Basically, this constrains register coloring to use the same color for each of the data ports. Whenever this cannot be achieved a MOVE instruction is inserted in the partition providing input to the merge. If this predecessor is in the same partition as the merge, *e.g.*, if the merge is directly fed by a label or merge, a new partition must be created. Currently, we try to minimize the number of move insertions by unifying names at merges before coloring for register and frame slots.

Entry counts: Synchronizing threads in TAM have an associated frame slot where the entry counter is maintained. 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.

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.

Thread ordering: The TAM control transfer primitives fork 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 current 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. Our current thread ordering scheme tries to maximize the number of fall-throughs.

Figure 5 shows how thread numbers are assigned for the lookup example. 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.

7 Results

This section presents preliminary data on the quality of TAM code produced under our compilation paradigm. Previous to this work, execution of Id 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 Id implementation competitive with machines supporting dynamic instruction scheduling in hardware[PC90, SYH$^+$89, GH90, Ian88]. By constraining how dual-graphs are partitioned, we can generate TAM code that closely models these other target architectures. It can be seen that the TAM partitioning described in this paper 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.

7.1 Benchmarks

Ten benchmark programs ranging from 50 to 1,100 lines are used. *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 Id[CHR78, AE88]. One iteration is run on 50×50 matrices.

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 section we present only dynamic statistics, which were collected on one node of a multiprocessor nCUBE/2.

7.2 TAM vs Dataflow

To better understand the quality of TAM partitioning, we may 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.

- For TAM we use our best partitioning: dependence sets partitioning with merging.
- Threads produced using dependence sets partitioning without merging correspond closely to Scheduling Quanta in Iannucci's hybrid architecture[Ian88]. 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.

- Threads produced by dataflow partitioning without merging reflect 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.

The distribution of instructions, execution time, and instructions per thread are shown in Figure 6. For each benchmark 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 in Figure 6.a 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 enables the corresponding thread and 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 varies substantially under the different partitioning schemes. On average, 31% of the instructions are used for control under DE_ME, less than twice the 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 about 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 machines, 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.

The strong relationship between Figures 6.a and 6.b confirms that the reduction in control overhead translates into execution efficiency and that the TAM instruction as a cost unit is reasonable. For *AS*, *QS* and *MMT* running the same programs written in C is between 2 and 3 times as fast. *Simple* on the recent dataflow machine Monsoon is twice as fast as executing the TAM program on a standard RISC, a MIPS R3000.

7.3 Thread Characteristics

Average thread lengths are shown in Figure 6.c. 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. Finally, the instruction count would increase if arithmetic instructions could only access registers. The larger thread sizes in wavefront and speech arise because many requests are issued in

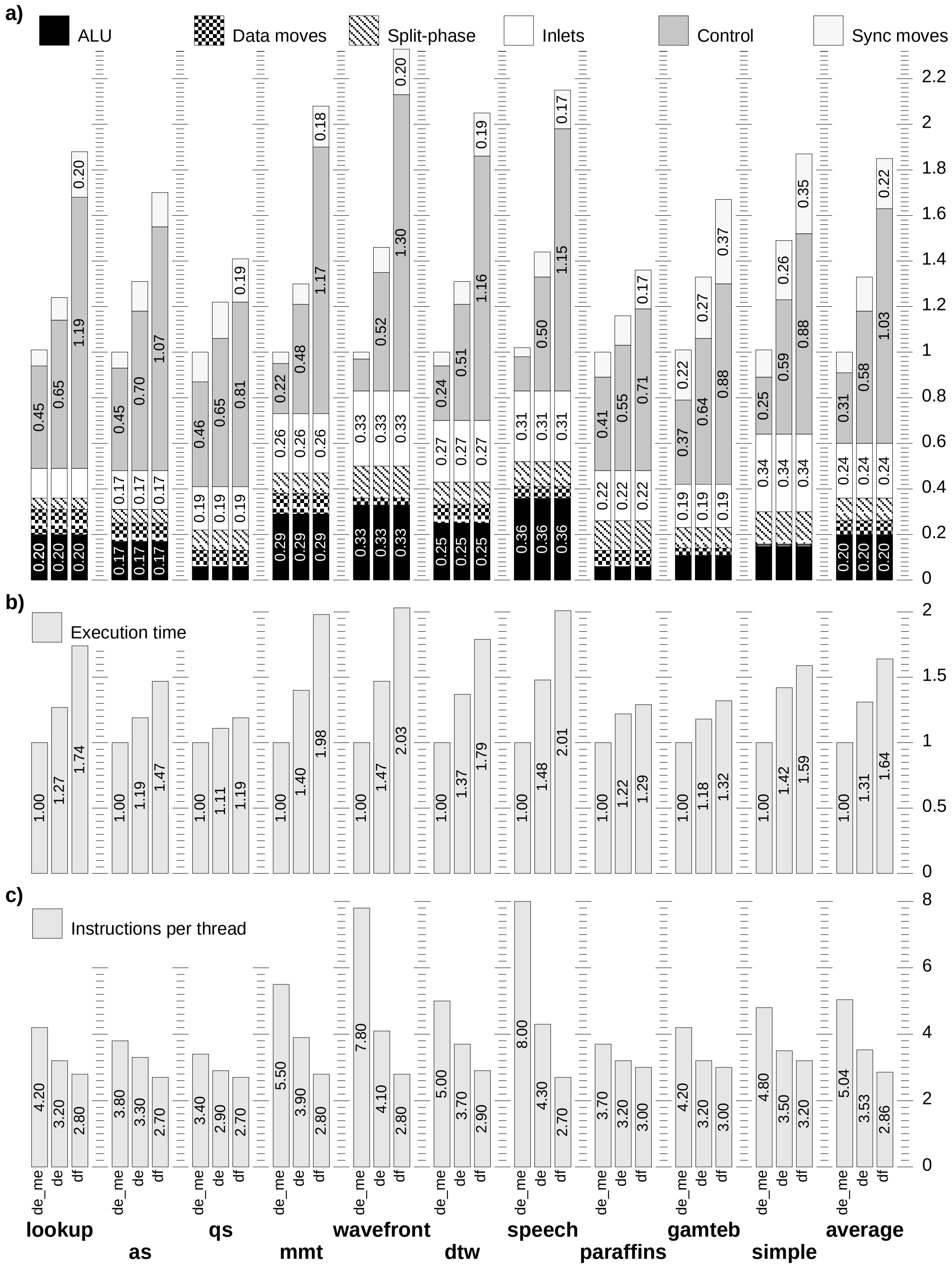

Figure 6: (a) Instruction distributions, (b) Relative execution times, (c) Thread sizes.

one thread and all the responses are received by another. Smaller threads result from conditionals, since no combining is performed.

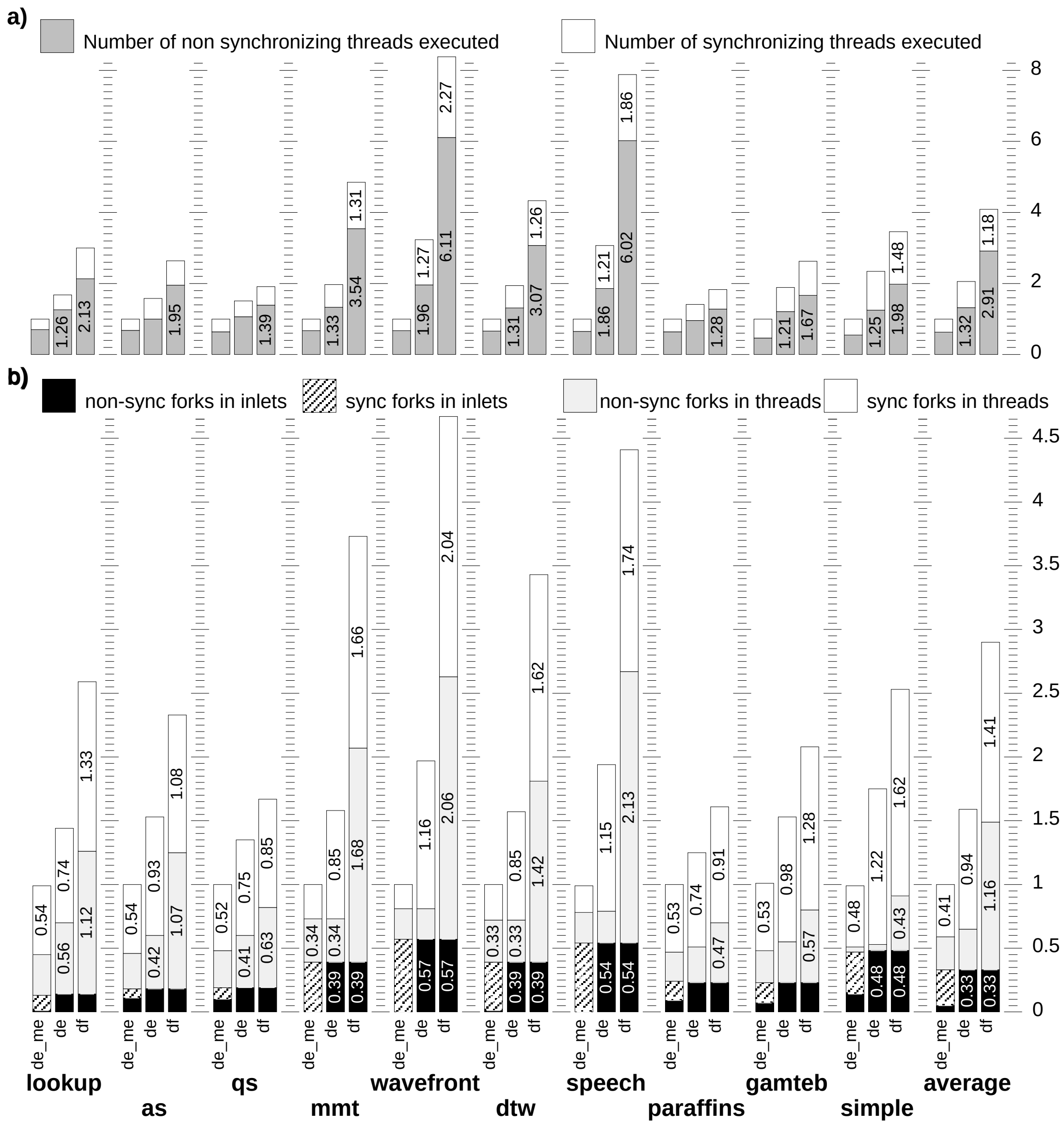

Figure 7: (a) Relative thread counts, (b) Control instruction distributions

Improved partitioning not only changes the size of threads, it changes their structure. The stacked bars in Figure 7.a 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 strategy. 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.

The effects of more sophisticated partitioning are more apparent in examining the fork operations. Figure 7.b 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 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."

7.4 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 1 shows the dynamic scheduling behavior of programs under TAM using DE_ME. We assume zero latency, so I-Fetches return immediately, unless deferred. The table shows the number of code-block activations, quanta, threads and 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.

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 1: Dynamic Scheduling Behaviour

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.

8 Conclusions

In this paper we demonstrate how a lenient parallel language, Id90, can be compiled for conventional processors using compiler-controlled multithreading. Our approach involves several large translation steps: Id90 to dataflow program graphs, program graphs to dual graphs, dual graphs to TAM threads, TAM to native machine code. Each of these intermediate forms plays a crucial role. Dataflow program graphs facilitate powerful, high-level transformations. They are compact and easy to manipulate, because they are hierarchical, employ a single kind of dependence arc, and assume scheduling is accomplished "as needed." Dual graphs facilitate the synthesis of control operations and management of storage. They make control and data flow explicit and, by retaining both in graphical form, allow transformations to be applied to one without prematurely constraining the other. In particular, unnecessary data movement and redundant control can be eliminated independently. TAM provides a means of describing a mixture of static and dynamic scheduling, makes scheduling costs apparent, exposes message handling, and emphasizes locality amongst dynamically scheduled entities. This approach has produced the first efficient implementation of Id90 on conventional machines.

We have shown that it is practical to implement lenient languages without fast dynamic scheduling in hardware. Reasonably sophisticated partitioning is required; we describe a partitioning algorithm that is practical to employ on real programs and results in control overhead within a small factor of the 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. Our implementation on conventional state-of-the-art processors provides a baseline against which novel multithreaded machines can be judged. Surprisingly, the compilation techniques developed for conventional machines can improve the performance of these novel architectures as well[PT91].

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. TAM has been implemented on uniprocessors and shared memory multiprocessors. We are currently implementing TAM on a 1,024 node nCUBE/2 using very fine grain message passing. The results presented in this paper, combined with these TAM implementations, suggest that latency tolerance and cheap synchronization can be achieved with sophisticated compilation in lieu of extensive hardware support.

Acknowledgements

We would like to thank the contributors to TAM at UC Berkeley, including John Wawrzynek, Anurag Sah, Seth Goldstein, Mike Flaster, Meltin Bell, and Bertrand Irissou. We are grateful also to the Computation Structures Group led by Arvind at MIT for providing Id90, the compiler front-end, and many fruitful interactions. TAM builds upon Greg Papadopoulos' Monsoon and Nikhil's PRISC. Ken Traub's dissertation inspired much of the work presented here, "to understand it we had to build it", and we have benefited from his insights.

This work was supported by National Science Foundation PYI Award (CCR-9058342) with matching funds from Motorola Inc. and the TRW Foundation. Thorsten von Eicken is supported by J. Wawrzynek's PYI Award (MIP-8958568) and the Semiconductor Research Corporation. Computational resources were provided, in part, under NSF Infrastructure Grant CDA-8722788.

References

[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.

[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.).

[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.

[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).

[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.

[Ian88] 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.

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

[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.