Separation Constraint Partitioning - A New Algorithm for Partitioning Non-strict Programs into Sequential Threads

Klaus E. Schauser
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
schauser@cs.ucsb.edu

David E. Culler, Seth C. Goldstein
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
{culler,sethg}@cs.berkeley.edu

Abstract

In this paper we present substantially improved thread partitioning algorithms for modern implicitly parallel languages. We present a new block partitioning algorithm, *separation constraint partitioning*, which is both more powerful and more flexible than previous algorithms. Our algorithm is guaranteed to derive maximal threads. We present a theoretical framework for proving the correctness of our partitioning approach, and we show how separation constraint partitioning makes interprocedural partitioning viable.

We have implemented the partitioning algorithms in an Id90 compiler for workstations and parallel machines. Using this experimental platform, we quantify the effectiveness of different partitioning schemes on whole applications.

1 Introduction

Modern implicitly parallel languages, such as the functional language Id90, allow the elegant formulation of a broad class of problems while exposing substantial parallelism. However, their non-strict semantics require fine-grain dynamic scheduling and synchronization, making an efficient implementation on conventional parallel machines challenging. In compiling these languages for commodity processors, the most important step is partitioning the program into sequential threads. This paper presents a new partitioning algorithm and experimentally quantifies its effectiveness.

Many of the issues that arise in implementing non-strict languages (dynamic scheduling, synchronization, and heap management) are present, independent of the source language, when producing code for parallel machines. Thus, the techniques developed in this paper are also applicable to parallel implementations of other languages. Moreover, dealing with non-strictness requires that these issues be faced even when compiling for sequential processors, because non-strict programs require logical parallel execution in order to make forward progress.

The language studied in this paper, Id90 [Nik90], is a *non-strict* functional language with *eager* evaluation. This combination, termed *lenient* evaluation [Tra91], exhibits more parallelism than lazy evaluation while retaining much of its expressive power.[1] To further increase parallelism, Id90 provides data structures that automatically synchronize between producer and consumers: I-structures and M-structures.

When executing a lenient program on a parallel machine, dynamic scheduling may be required for two reasons. First, the semantics of the language make it impossible in general to statically determine the order of operations. The order in which operations of a function execute may depend on the dynamic context in which the function is invoked (cf., Section 2.1), not just on the value of its arguments. Second, long latency inter-processor communication and accesses to synchronizing data structures require that the computation dependent on the messages be scheduled dynamically. Dynamic scheduling is expensive on commodity microprocessors, incurring a high cost for context switching. Therefore, these languages have been accompanied by the development of specialized computer architectures, e.g., graph reduction machines [PCSH87, Kie87], dataflow machines [ACI+83, GKW85, SYH+89, PC90], and multithreaded architectures [Jor83, NPA92]. Much research has been done in compiling lenient languages for dataflow architectures [ACI+83, Tra86, AN90, GKW85, Cul90]. As a clearer separation of language and architecture has been obtained, attention has shifted to compilation aspects of these languages for commodity processors [Tra91, SCvE91, Nik93].

The emphasis of the compilation work is to statically schedule groups of instructions into *sequential threads* and restrict dynamic scheduling to occur only between threads. A thread forms the basic unit of work: once scheduled it runs until completion. The task of identifying portions of the program that can be scheduled statically and ordered into threads is called *partitioning* [Tra91]. Partitioning the program into sequential threads requires substantial compiler analysis (dependence analysis) because, unlike in imperative languages, the evaluation order is not specified by the programmer. Care has to be taken to generate threads which obey all dynamic data dependencies and do not lead to deadlock. Partitioning decisions imply trade-offs between parallelism, synchronization cost, and sequential efficiency [SCvE91]. However, given the limits on thread size imposed by the language model, the use of split-phase accesses, and the control paradigm, our goal simply is to max-

To appear in the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95).

[1] Usually non-strictness is combined with lazy evaluation (e.g., in LML and Haskell). Under lazy evaluation an expression is only evaluated if it contributes to the final result. Lazy evaluation decreases the parallelism because the evaluation of an expression is only started after it is known to contribute to the result.

imize the length of the threads and minimize the number of thread switches.

1.1 Contributions

The main contribution of this paper is the development of a new thread partitioning algorithm that is substantially more powerful than any previously known [Tra91, HDGS91, SCvE91, TCS92]. A compiler for Id90 has been developed with back-ends for workstations and the CM-5 multiprocessor. It serves as an experimental platform for studying the effectiveness of the partitioning algorithms. The partitioning algorithms presented in [SCvE91] and [TCS92] are a starting point for this work and are extended in several ways. The paper:

- presents a new block partitioning algorithm, *separation constraint partitioning*, which is more powerful than iterated partitioning, the previously best known block partitioning,
- shows how separation constraint partitioning can be integrated successfully into the interprocedural partitioning algorithm to improve code at the call boundaries,
- outlines the theoretical framework and sketches the proof of correctness for our partitioning approach,[2]
- implements the partitioning algorithms, resulting in a running execution vehicle for Id90 on workstations and several parallel machines, and
- quantifies the effectiveness of the different partitioning schemes on whole applications.

In addition, although not documented here, we have extended interprocedural analysis to handle recursion and mutually dependent call sites [Sch94].

Section 2 formalizes the problem of thread partitioning and presents an example which shows that non-strict languages require dynamic scheduling. In Section 3 we present our new partitioning algorithm, separation constraint partitioning. Section 4 briefly discusses how separation constraint partitioning can be integrated with interprocedural partitioning. Section 5 presents experimental results which show the effectiveness of our partitioning algorithm. Finally, Section 6 contains the summary and conclusions. A short sketch of the proof of correctness for the partitioning algorithm appears in Appendix A.

1.2 Related Work

Partitioning is similar in spirit to compilation techniques for lazy functional languages [SNvGP91, Pey92]. Strictness analysis [Myc80, CPJ85] tries to determine which arguments can be evaluated before invoking the body of a function, thus avoiding the creation of expensive thunks for strict arguments. Partitioning goes a step further as it may derive that arguments can be evaluated together even if a function is not strict in them [Tra91]. Path analysis [BH87] detects the order in which arguments are evaluated, which may result in a cheaper representation of thunks and reduce the cost of forcing and updating them. Serial combinators by Hudak and Goldberg are one of the first attempts to improve the parallel execution of lazy functional programs by increasing their granularity [Gol88]. Their approach is to group several combinators into larger serial combinators.

Partitioning plays a crucial role for the parallel execution of strict functional languages, but unlike non-strict languages, the ordering of instructions can be determined statically. Thus, the difficulty is not what can be put into the same thread, but rather what should be placed into the same thread given communication and load balancing constraints [SH86, NRB93].

Most of the partitioning research for lenient languages was inspired by Traub's seminal theoretical work, which uses dependence analysis to characterize when instructions can be grouped into a thread [Tra91].[3] Traub showed that the problem of finding a partitioning with the minimum number of threads is NP-complete. Thus, all of the partitioning approaches rely on heuristics to group nodes into threads. Iannucci developed *dependence set partitioning*, which groups nodes that depend on the same set of inputs [Ian88]. *Demand set partitioning*, presented in [SCvE91] and [HDGS91] is analogous to dependence set partitioning, but it groups nodes which are demanded by the same set of outputs. *Iterated partitioning* combines the power of dependence and demand set partitioning by applying them iteratively [TCS92, HDGS91]. One of the algorithms is applied, then the reduced graph is formed, and the other algorithm is applied. This process is repeated until no further changes occur. Schauser *et al.* extended the two basic partitioning schemes with local "merge up" and "merge down" rules, thus achieving essentially the same degree of grouping as iterated partitioning [SCvE91]. Traub *et al.* [TCS92] extended iterated partitioning with interprocedural analysis to obtain larger threads. Recently, [Coo94] and [Sch94] independently developed extensions to the interprocedural algorithm to handle recursive functions. Separation constraint partitioning identifies all possible "merges" allowing the thread partitioning to be guided by high level heuristics, such as minimizing the cost of procedural boundaries.

2 Block Partitioning

The partitioning algorithm produces a collection of threads. The instructions of each thread are statically scheduled, and all dynamic scheduling required by non-strictness or potentially long latency communication occurs between threads.

Definition 1 (Thread [TCS92]) A *thread* is a subset of the instructions of a procedure body, such that:

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

[2] The complete proof can be found in [Sch94].

[3] Traub's original framework allows threads to suspend; thus, his threads capture the sequential ordering which is required between instructions, but not the dynamic scheduling which may occur between the threads. Subsequent research in this area disallows thread suspension, which has the advantage of capturing the cost of switching between threads.

Our partitioning algorithms work on *structured dataflow graphs* [Tra86], the intermediate form used in the Id90 compiler. It is similar to intermediate representations found in other optimizing compilers. A structured dataflow graph consists of a collection of *blocks*,[4] one for each function and each arm of a conditional, and *interfaces* which describe how the blocks relate to one another [TCS92]. Each block is represented by an acyclic dataflow graph; roughly, it corresponds to a group of operators with the same control dependence [FOW87]. For example, all operators comprising the "then" arm of a conditional, excluding those in nested conditionals, are a block.

Definition 2 (Dataflow Graph) A *dataflow graph* is a directed acyclic graph of vertices and dependence edges, (V, E_s, E_q), where $E_s \subseteq V \times V$ are the *certain direct dependence* edges and $E_q \subseteq V \times V$ are the *certain indirect dependence* edges.

The vertices describe the instructions, including arithmetic and logic operators, creation and access of data structures, and the sending and receiving of arguments and results. The edges capture *certain* data dependencies, which are present in every context in which the procedure can be invoked. We distinguish two kind of certain dependencies: *direct* dependencies (represented in the examples by straight arcs—see Figure 2a) and *indirect* dependencies (represented by squiggly arcs—see Figure 2a). Indirect dependencies represent potentially long latency dependencies which may involve nodes of other blocks. For example, an indirect edge connects the request and response nodes for a split-phase synchronizing data structure access. Such an access may require a long time to complete due to network latency or synchronization delay, i.e., it may have to wait until an other computation completes and stores the referenced value. Definition 1 implies that nodes connected by an indirect dependence must reside in different threads.

In addition, we define a *potential indirect dependence* (PID) as one which may exist in some but not all invocations of the block. PIDs may go through nodes of other blocks. This concept of potential indirect dependencies is very important. A PID is a dependence which could exist in some legal execution of the program, where a legal execution is defined as one which does not lead to a deadlock in the absence of partitioning. The key observation is that the compiler does not have to consider PIDs which are contradicted by certain dependencies because such dependencies would lead to deadlock. Certain dependencies provide the mechanism to reduce the set of PIDs. We need to be conservative and overapproximate the PIDs. Due to the non-strict nature of the language the compiler initially assumes that for each function any argument may depend on any of its results. Through the process of analysis some PIDs are ruled out. The challenge is to represent PIDs as precisely and as efficiently as possible. We use *inlet* and *outlet* annotations to represent PIDs in the dataflow graph.

Definition 3 (Annotation) An *annotation* for a block is a 5 tuple $A = (\Sigma_i, \mathit{Inlet}, \Sigma_o, \mathit{Outlet}, \mathit{CID})$, where Σ_i is the inlet alphabet, $\mathit{Inlet} : V \rightarrow \mathit{Pow}(\Sigma_i)$ maps each node to a set of inlet names (the inlet annotation), Σ_o the outlet alphabet, $\mathit{Outlet} : V \rightarrow \mathit{Pow}(\Sigma_o)$ maps each node to a set of outlet names (the outlet annotation), and $\mathit{CID} \subseteq (V \times V)$ are the *certain indirect dependencies* ($\mathit{CID} = E_q$).

In the graphical representation, we attach incoming circles to nodes for inlet annotations and outgoing circles for outlet annotations. For example, in Figure 1 the inlet annotations are $\{a\}$ and $\{b\}$.

Nodes with outlet and inlet annotations form the endpoints of PIDs. A PID may travel from a node with an outlet annotation to a node with an inlet annotation. An inlet name represents a set of outlet nodes that this node certainly depends on. This set is not known at compile time, but every node which contains the same inlet name in its annotation depends on this same set of outlet nodes, although we can not identify which outlet nodes they are. Thus, the initial assumption of any inlet depending on any set of outlets (not contradicted by certain dependencies) is captured by giving each inlet a unique annotation. Likewise, an outlet name represents a set of inlet nodes that depend on this node. The process of assigning the same or partially overlapping inlet (or outlet) names to multiple nodes allows us express sharing of dependencies between nodes.

As mentioned above, the PIDs capture the potential dependencies from outlet nodes to inlet nodes that do not lead to deadlock at runtime. We assume that a PID exists unless it is contradicted by certain dependencies. More formally, we define a PID to exist from a node s to a node r if, there exists an $o \in \mathit{Outlet}(s)$ and $i \in \mathit{Inlet}(r)$ such that there does not exist a path over straight and squiggly edges from a node r' with $i \in \mathit{Inlet}(r')$ to a node s' with $o \in \mathit{Outlet}(s')$.

The task of a partitioning algorithm is to take as input a structured dataflow graph and to partition the vertices of each block into non-overlapping regions such that each subset can be mapped into a non-suspending sequential thread. Deriving the threads is done in two steps. First, the nodes of each block are partitioned into disjoint subsets. Then, the instructions of each subset are linearized (any topological ordering will do). The partitioning algorithms presented here only derive the subsets of vertices and leave the actual ordering of instructions within each thread to a later stage of the compiler. We shall refer to each subset of vertices simply as a thread.

A *correct partitioning* has no circular dependencies between threads, i.e., no static cycles within blocks and dynamic cycles across blocks. Without circular dependencies, it is possible to delay the scheduling of a thread until all of its predecessors have completed and then run the thread until completion. In addition, a correct partitioning must ensure that requests and responses to split-phase operations are in different threads.

2.1 Simple Examples

We now present a simple example to illustrate the concepts just introduced. Figure 1 shows the dataflow graph for the function f which is called by the procedures g and h.

```
def f u v = (u*u, v+v);
def g z = {s,t = (f z s) in t};    computes (z*z) + (z*z)
def h z = {s,t = (f t z) in s};    computes (z+z) * (z+z)
```

This example illustrates the need for dynamic scheduling even in the absence of conditionals. The function f takes two arguments, u and v, and returns two results, $u * u$ and $v + v$. Within f there is no dependence between the multiplication

[4] In previous work the term *basic block* was used [Tra86, TCS92]. Since this term has a different meaning in the compiler literature for imperative languages we use the term *block*.

and addition. Therefore, they can be scheduled in any order under traditional strict evaluation. This is not true for non-strict evaluation. The function g feeds the first result of the function f back in as the second argument, while the function h feeds the second result back in as the first argument. These two dependencies are PIDs. In the context of function g the multiplication must be executed before the addition, while in function h the opposite is true. Thus, the multiplication and the addition have to be scheduled independently, and it is impossible to put them together into a single non-suspensive thread.

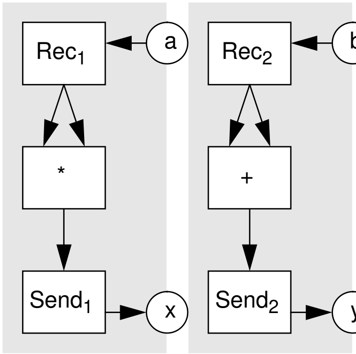

Figure 1: *Small example of a dataflow graph for the function* `f u v = (u*u, v+v);` *and its partitioning into two threads. The arcs represent direct dependencies while the inlet and outlet annotations represent sets of potential dependencies as explained in the text. The shaded regions represent the threads.*

The two PIDs are represented by inlet and outlet annotations. Without any interprocedural analysis to indicate otherwise, each node is given a unique singleton annotation, implying that we have to assume that each depends on (or influences) a different set of unknown nodes. In our example, the argument receive nodes are given the inlet annotations $\{a\}$ and $\{b\}$, while the send nodes have the outlet annotation $\{x\}$ and $\{y\}$. The names themselves are not important; the absence of sharing between the names is what is important. By our definition a potential dependence exists from the send node with outlet annotation $\{x\}$ back to the receive node with inlet annotation $\{b\}$ because there does not exist a certain dependence path contradicting this, i.e., from a node with b in its inlet annotation to a node with x in its outlet annotation. Likewise, there exists a PID from $Send_2$ to Rec_1. Functions g and h contain these PIDs. On the other hand, there cannot exist a PID from $Send_1$ back to the Rec_1 because this is contradicted by a certain dependence path. Thus, the inlet and outlet annotations correctly capture the two potential dependencies which may arise at run time. As a result, the left and right nodes must stay in separate threads, and the partitioning algorithm can at best obtain two threads, as indicated by the shaded regions.

We can improve the partitioning by using interprocedural analysis if we know that the function f is only called in the following context:

`def foo z = {s,t = (f z z) in s+t};` *computes* $(z * z) + (z + z)$

In this case, it is valid to give both receive nodes of the def site the same inlet annotation, say $\{a\}$, as both argument send nodes at the call site depend on the same argument of the function *foo*. Likewise, we can give the same outlet annotation, say $\{x\}$, to both send nodes of f. Now when partitioning f, the compiler can determine that there cannot exist a potential dependence from a result back to an argument, since under the new annotation there exists a certain dependence path from a receive node with the inlet name a to a send node with the outlet annotation x. Thus, the compiler can group all of the nodes in f into a single thread.

Dynamic scheduling may also arise when accessing synchronizing data structures. For example, assume that a function contains the following code which manipulates I-structures.

```
A[k] = A[m] * A[m];
A[l] = A[n] + A[n];
```

The corresponding dataflow graph is shown in Figure 2. This code fetches an element from $A[m]$, multiplies this with itself and stores the result into location $A[k]$. It also fetches from location $A[n]$, adds this element with itself and stores it into location $A[l]$. The declarative nature of the non-strict language does not specify the order in which these statements are executed. Actually, that order may depend on the context in which this code is executed.

If $k = n$, the store into location $A[k]$ defines the value which is fetched from $A[n]$. Therefore there exists a PID from the store to the fetch response, as indicated by the dashed line in Figure 2b. Thus, the multiplication has to be executed before the addition. If $l = m$ the operations would execute in the reverse order (see Figure 2c). Note that these dependencies are not directly present in the function, they are established through the synchronizing I-structure. These potential dependencies are captured by the annotations; an inlet annotation on a fetch node represents a dependence on some store.

I-structure accesses have to be represented by split-phase operations which separate the request from the response. There are two reasons why the request and the response may not execute together. First, a fetch may get deferred should it occur before the corresponding store. Second, execution on a parallel machine may result in a long communication latency if the accessed element resides on another processor. Both forms require dynamic scheduling. Thus the request and response have to reside in different threads. With split-phase accesses the processor can continue working after issuing the request, making it possible to hide the communication latency with computation that is not dependent on the requested data. The potentially long latency between the request and response is indicated by the squiggly edges in the dataflow graph, which represent certain indirect dependencies.

These examples illustrate that potential dependencies cannot be known at compile time. They can travel through arguments, results, internal call sites, and through I-structure accesses.

2.2 Limits of Iterated Partitioning

The previously best known block partitioning scheme, iterated partitioning [TCS92], is not powerful enough to always find maximal threads. A slightly revised version of the first example, shown in Figure 3, proves that separation constraint partitioning is strictly more powerful than iterated partitioning, which fails to find maximal threads.

This example consists of six nodes. Iterated partitioning forms two threads. The inlet/outlet annotations are not

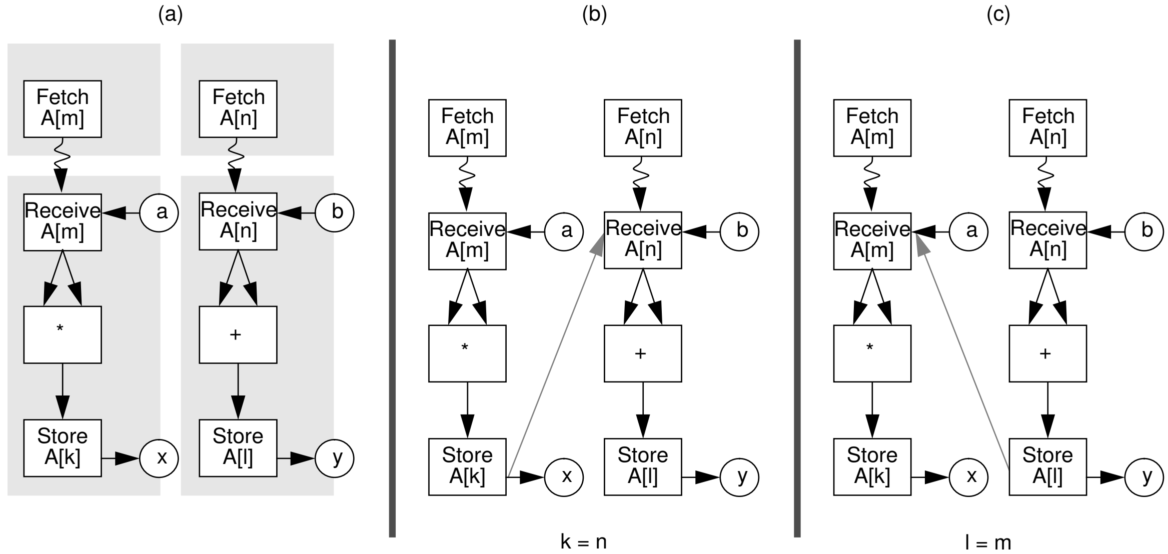

Figure 2: *Simple example of a dataflow graph with I-structures for the code* `A[k] = A[m] * A[m]; A[l] = A[n] + A[n];`*. The shaded regions show the four threads. Since a fetch of an I-structure element may defer, it cannot be placed into the same thread as the response. The threads cannot be grouped into a single thread because there may exist potential indirect dependencies which require dynamic scheduling. These* PID *edges are indicated by the dashed arcs in Part (b) for $k = n$ and Part (c) for $l = m$.*

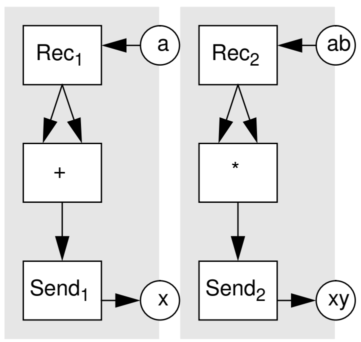

Figure 3: *Example where iterated partitioning fails to merge two threads.*

unique singleton sets, but instead reflect dependence sharing which could be the result of interprocedural analysis. The dependence set of the three left nodes is $\{a\}$, and their demand set is $\{x\}$. Iterated partitioning will group them all into a single thread. Likewise, the three right nodes are grouped into a single thread because their dependence set is $\{a, b\}$, and their demand set is $\{x, y\}$.

Iterated partitioning cannot merge the left and the right nodes since their dependence and demand sets are different. However, they can safely be merged for the following reasons. The dependence sets represent the set of (unknown) outlets a node depends on. The dependence set for the left nodes is a subset of that for the right nodes. Since the right nodes depend on a larger set of outlet nodes, they cannot influence any of the left nodes, and thus there cannot exist a PID from the right to the left nodes. The same argument in reverse holds for the demand sets. It is therefore possible to merge the two threads into one. Merging these threads requires a more powerful partitioning rule which is not based solely on equal dependence or demand sets. This observation is formalized by separation constraint partitioning.

3 Separation Constraint Partitioning

Separation constraint partitioning can, with respect to any annotation, precisely determine for any two nodes whether they can be merged or not. The rule is simple: two nodes of a block cannot be merged (i.e., they must reside in different threads) if there exists either a certain indirect dependence (CID) or a potential indirect dependence (PID) between them. The reason is that both forms of indirect dependencies may require dynamic scheduling.

Given this separation rule, we can easily devise an effective partitioning algorithm. Starting with the unpartitioned dataflow graph, we find two nodes without a separation constraint, merge them, form the reduced graph, and repeat this process until no further nodes can be merged. Although this method is more powerful and elegant than the previous partitioning algorithms, unfortunately it is computationally more expensive. As discussed below, this problem can be alleviated by only running it on a subset of the graph.

Separation constraint partitioning has four advantages. First, it is guaranteed to derive maximal (but not necessarily optimal) threads. After it has finished, every pair of threads has a separation constraint between them, and therefore it is impossible to merge further. Second, it deals in a unified way with the partitioning constraints introduced by certain and potential indirect dependencies, and therefore does not require subpartitioning (as do dependence and demand set partitioning [TCS92]). Third, the algorithm can be combined with heuristics that attempt to merge the nodes in an order which minimizes communication, dynamic scheduling, and synchronization overhead. Finally, it can also be naturally integrated into the interprocedural partitioning algorithm.

3.1 The Algorithm

The most complicated aspects of the algorithm are the initial computation of the *separation constraints* and their update when two nodes are merged. Separation constraints arise from CIDs, which connect send nodes to receive nodes, and PIDs, derived from the annotations for the block. We say that any two nodes that are connected through a PID or a CID have an indirect dependence and cannot be merged. Deriving the CIDs is easy, as they are directly represented in the graph. The challenge is to efficiently determine the PIDs, which the compiler does not know and has to approximate safely.

Algorithm 1 (Separation constraint partitioning) *Given a dataflow graph with inlet/outlet annotations:*

1. Compute the reflexive, transitive closure of the successor relation $Succ^*$ over $E_s \cup CID$.
2. Compute the set of potential indirect dependence edges, i.e., those edges from outlets to inlets which are not contradicted by certain dependencies.
$$PID = \{(s,r) | \exists i, o : i \in Inlet(r), o \in Outlet(s), \neg\exists (r', s') \in Succ^* : i \in Inlet(r'), o \in Outlet(s')\}$$
3. Combining PID and CID, compute the set of nodes with an indirect dependence between them.
$$ID = \{(u,v) | \exists (s,r) \in PID \cup CID : (u,s) \in Succ^*, (r,v) \in Succ^*\}$$
4. Find two nodes u, v without an indirect dependence between them, i.e., for which $(u,v) \notin ID$ and $(v,u) \notin ID$. Merge u, v into a single thread, and update the representation.
   (a) Derive the new set of nodes, use v as representative for the two merged nodes and discard u.
   $V_{new} = V - \{u\}$
   (b) Compute the new reflexive transitive closure.
   $$\begin{aligned} Succ^*_{new} = &\{(p,s) | (p,s) \in Succ^*,\ p \neq u, s \neq u\} \\ \cup &\{(p,s) | (p,u) \in Succ^*, (v,s) \in Succ^*,\ p \neq u, s \neq u\} \\ \cup &\{(p,s) | (p,v) \in Succ^*, (u,s) \in Succ^*,\ p \neq u, s \neq u\} \end{aligned}$$
   (c) Compute the new set of indirect dependencies.
   $$\begin{aligned} ID_{new} = &\{(p,s) | (p,s) \in ID,\ p \neq u, s \neq u\} \\ \cup &\{(p,s) | (p,u) \in ID, (v,s) \in Succ^*,\ p \neq u, s \neq u\} \\ \cup &\{(p,s) | (p,v) \in ID, (u,s) \in Succ^*,\ p \neq u, s \neq u\} \\ \cup &\{(p,s) | (p,u) \in Succ^*, (v,s) \in ID,\ p \neq u, s \neq u\} \\ \cup &\{(p,s) | (p,v) \in Succ^*, (u,s) \in ID,\ p \neq u, s \neq u\} \end{aligned}$$
   (d) Set $V = V_{new}$, $Succ^* = Succ^*_{new}$, and $ID = ID_{new}$.
5. Repeat from Step 4 until no more nodes can be merged.

Observe that existing separation constraints never disappear. Merging two nodes can only introduce new constraints. Thus every pair of nodes has to be tested at most once for merging. After merging two nodes, the transitive closure and the indirect dependencies are updated. Furthermore, the new ID can be computed from the old ID and $Succ^*$.

We apply this algorithm to the example in Figure 1. Following the rule in Step 2, we derive that there exists a PID from the left send to the right receive and from the right send to the left receive. $(Send_1, Rec_2) \in PID$ exists because $b \in Inlet(Rec_2)$ and $x \in Outlet(Send_1)$, and there is no path from a node with b in its inlet set to a node with x in its outlet set to contradict this. A similar argument can be made for the other PID. As a result there exists a separation constraint from any of the left nodes to any of the right nodes, and the left nodes cannot be merged with the right nodes, as observed earlier.

Now we apply this algorithm to the example in Figure 3. Following the rule in Step 2, we derive that there are no PIDs because they are all contradicted by certain dependencies: for every inlet/outlet name pair there exists a path from a node with the inlet name in its inlet set to a node with the outlet name in its outlet set. Therefore, $PID = \emptyset$. Since $CID = \emptyset$ we ascertain that $ID = \emptyset$. Thus there are no separation constraints and any two nodes can be merged. Separation constraint partitioning will, as expected, end with a single thread.

3.2 Merge Order Heuristics

The algorithm as presented so far does not specify the order in which pairs of nodes are visited and tested for merging. This flexibility is an important advantage, as it permits the algorithm to be combined with a heuristic that visits the nodes in an order that minimizes communication, dynamic scheduling, and synchronization overhead. All three operations are expensive on commodity processors. We decided to address communication first, since on most parallel machines communication has the highest overhead. Our heuristic is first to try merging nodes belonging to the same function call boundary (which reduces communication), then nodes at conditional boundaries (which reduces dynamic scheduling), and finally the remaining interior nodes of the block.

After interprocedural analysis (explained in Section 4), the annotations for a block may have been refined and the block can be repartitioned. Repeatedly repartitioning using iterated partitioning is very expensive. However by using separation constraint partitioning, we can perform the interprocedural analysis on a restricted graph—consisting of the nodes at def and call site boundaries and their connectivity—and then partition the interior nodes after the annotations have been completely refined. Extracting this restricted graph from the original program is fairly simple and involves only computing the transitive closure of each block's dataflow graph. The saving is enormous: for our benchmark programs the graph sizes are reduced by a factor of 10 to 20—the largest block was reduced from 619 nodes to 40 nodes. Running separation constraint partitioning on the restricted graph is very fast, making interprocedural partitioning viable. Finally, after obtaining the best possible partitioning at the function call boundaries, we partition the interior of blocks. Our approach is to run separation constraint partitioning only on a subset of the nodes of the block (the most critical nodes) and for the rest of the block use iterated partitioning, which in practice runs faster.

3.3 Complexity

To compute the complexity of the above algorithm we assume that ID and $Succ^*$ are represented by an adjacency matrix. Assume that the problem size n is the maximum over the number of edges, number of inlet names, and number of outlet names. Since the dataflow graph is acyclic, initially computing the transitive closure is $\mathcal{O}(n^2)$. Determining the *PID* edges in Step 2 is $\mathcal{O}(n^3)$. Computing *ID* is $\mathcal{O}(n^2)$. Testing whether two nodes can be merged takes only constant time, since *ID* is represented as a matrix. Since merging never eliminates separation constraints, at most $\mathcal{O}(n^2)$ pair of nodes have to be tested, thus this part of Step 4 is $\mathcal{O}(n^2)$. Merging occurs at most $\mathcal{O}(n)$ times, and the complexity of Steps 4*(a)*–*(d)* is $\mathcal{O}(n^2)$. Overall, the total complexity of the algorithm is $\mathcal{O}(n^3)$. In practice the running time is too long for large blocks.

For iterated partitioning the worst case complexity is also $\mathcal{O}(n^3)$, since the complexity of dependence and demand set partitioning is $\mathcal{O}(n^2)$. However experimental data indicate that in practice iterated partitioning requires only a small number of iterations to find the final solution. Two cycles (i.e., four partitioning steps) were sufficient for partitioning the blocks of the set of Id90 programs we used for the experimental results section. On the other hand, it is possible to construct examples which require an arbitrary number of iterations (see [Sch94] for details).

3.4 Correctness

Proving correctness of separation constraint partitioning is much harder than dependence and demand set partitioning, which are quite intuitive. The appendix contains a short discussion of the correctness proof.

There are two key aspects to this proof. First, we show that the algorithm correctly updates the set of indirect dependencies *ID* throughout the execution of the program everytime two nodes u, v are merged. This implies that certain and potential indirect dependencies are correctly taken care of. Second, we prove that when the algorithm terminates, all partitions are convex, i.e., there do not exist any static cycles from a thread back to itself. This may not be the case at intermediate steps of the algorithm. Thus separation constraint partitioning is quite different from iterated partitioning. There the partitioning is correct after every step and we could choose to stop at any time if so desired. Separation constraint partitioning, on the other hand, has to run until termination.

4 Interprocedural Partitioning

The block partitioning algorithm presented so far is limited in its ability to derive threads because without global analysis it must assume that every send in a block may potentially feed back to any receive unless contradicted by certain dependencies. This is captured by the singleton inlet and outlet annotations given initially to send and receive nodes. Global analysis can determine that some of these potential dependencies cannot arise and thereby improve the partitioning [TCS92]. For example, the information gained while partitioning a procedure can be used to improve the inlet and outlet annotations of its call sites. These refined annotations may share names, reflecting the sharing among dependence and demand sets present in the procedure. In addition, squiggly edges from argument send nodes to result receive nodes can be introduced if the procedure has the corresponding paths from the argument receives to result sends. Both the refined annotations and the squiggly edges help to better approximate the PIDs and thereby improve subsequent partitioning.

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

Conditionals are handled similarly to function calls. A conditional with two arms can be viewed as a function call, where, depending on the result of the predicate, one of two blocks are called [AA89]. This representation simplifies the partitioning process, as we can use the same unified mechanism to deal with function calls and conditionals. When the analysis is applied to function calls it allows us to reduce communication; when applied to conditionals it allows us to reduce control flow overhead.

4.1 Interprocedural Partitioning Example

We will not present the formal interprocedural partitioning algorithm here as it already has been presented in [TCS92]. An extended version which can deal with recursive function can be found in [Sch94]. However, we discuss a small example to help illustrate it.

The example shown in Figure 4 consists of two blocks, a caller and callee. The left part of the figure shows the dataflow graph for the caller, the function g, while the right part shows the dataflow graph for the callee, the function f. Both procedures receive two arguments and return two results. The procedure g contains a call site of the procedure f, as indicated by the interior dashed rectangle, the two argument send nodes (AS1 and AS2), and result receive nodes (RR1 and RR2). The corresponding def site of the procedure f consists of the two argument receive nodes (AR1 and AR2) and two result send nodes (RS1 and RS2).

As shown in Part (a) of the figure, the algorithm starts by initially giving all receive and send nodes a unique singleton inlet or outlet annotation. As shown by the shaded regions in Part (b), partitioning the caller results in four threads, while partitioning the callee results in two threads. This is the best partitioning possible under the trivial annotation. The top four nodes of the caller cannot be placed into a single thread because the partitioning algorithm has to assume that a PID may exist from the node with the outlet annotation $\{u\}$ back to the node with the inlet annotation $\{b\}$. Analogous arguments can be made for why the other threads have to stay separate.

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

a) Initial Annotation

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

b) Initial Partitioning

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

c) Reannotation of Callee

Annotation propagation from the caller to the callee results in the new inlet and outlet annotations.

d) Partitioning of Callee

With the new annotation separation constraint partitioning can obtain a single thread.

e) Reannotation of Caller

Annotation propagation from the callee to the caller introduces four squiggly edges and eliminates the annotations.

f) Partitioning of Caller

Partitioning now obtains two threads. Further reannotation and partitioning does not improve this.

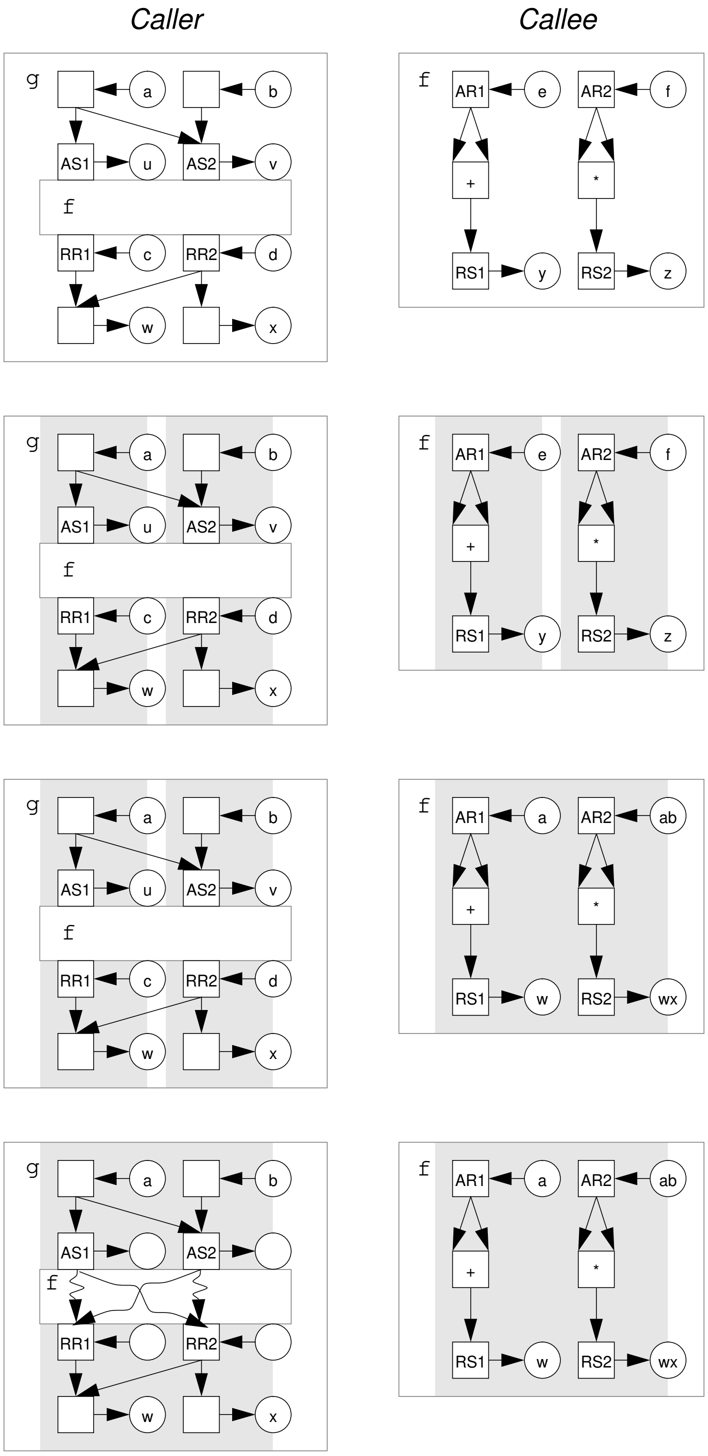

Figure 4: *Example of interprocedural partitioning with annotation propagation.*

notation $\{a, b\}$. Likewise, the new outlet annotations given to the result send nodes reflect the demand set of the corresponding result receive nodes, which are $\{w\}$ and $\{w, x\}$ respectively. The new annotations correspond precisely to the situation shown in Figure 3. Using separation constraint partitioning, we can group all nodes of the callee into a single thread, as indicated by the shaded region in Part (d) of the figure.

Next we propagate annotations from the callee to the caller. This time we can introduce four squiggly edges at the call site, one from every argument send node to every result receive node, since the corresponding certain dependence paths are present in the callee now that it consists of a single thread. These squiggly edges capture all of the dependencies which can arise at this call site. Therefore, we can give the argument send and result receive nodes at the call site empty inlet and outlet annotations, as shown in Part (e) of the figure. Applying separation constraint partitioning, the two top threads in the caller can be merged into a single thread, as shown in Part (f) of the figure. Likewise, the bottom two threads can be grouped into a single thread. Because the top and the bottom threads are connected by squiggly edges, they have to remain separate. Thus, partitioning the caller results in two threads, the best partitioning that can be obtained for this example. Further reannotation and partitioning does not improve this. Note that the resulting threads are the same as in a strict sequential program.

5 Experimental Results

In this section we evaluate our partitioning scheme in the context of the Berkeley Id90 compiler. Using various metrics we show how separation constraint partitioning combined with interprocedural analysis approach the efficiency of an oracular "strict partitioner."

5.1 Methodology

The Berkeley Id90 compiler uses a front-end developed at MIT [Tra86], which produces structured dataflow graphs for the partitioning algorithms presented here. The partitioned graphs are used to generate code for TAM, a threaded abstract machine [CGSvE93]. The TAM code is then translated to the target machine. Our translation path uses C as a portable "intermediate form" and is producing code for the CM-5, as well as for various standard sequential machines [Gol94]. We used this implementation for statistics collection and measurements. All of the programs are compiled for parallel execution. As they run, lots parallelism is exposed. However in order to factor out a broad family of issues unrelated to partitioning, such as load balancing and locality, we present data here from runs on a single processor. See [CGSvE93, SGS+93] for data and discussion on running these programs on parallel machines.

We use six benchmark programs, shown in Table 1, ranging up to 1,100 source code lines. It should be noted that the code was taken as is, compiled for TAM, and executed on standard workstations or the CM-5 without any modifications. The programs range from very fine grained (e.g., Quicksort) to medium grained (e.g., MMT).

5.2 Evaluation

To measure the effectiveness of partitioning we compare four different partitioning schemes: *dataflow partitioning (DF)*, *iterated partitioning (IT)*, *separation constraint partitioning with interprocedural analysis (IN)*, and *strict partitioning (ST)*. Dataflow partitioning and strict partitioning represent the two extremes of the spectrum. Dataflow partitioning puts unary nodes into the thread of their predecessor, reflecting the limited thread capabilities supported by many dataflow machines. Strict partitioning ignores possible non-strictness and compiles function calls and conditionals strictly, thus representing the best possible interprocedural partitioning algorithm. Although it is not the case for our six benchmark programs, strict partitioning produces an incorrect partitioning for programs which require non-strictness. Iterated and interprocedural partitioning represent the two real partitioning schemes. With iterated partitioning every block is partitioned in isolation. Separation constraint partitioning with interprocedural analysis applies the techniques discussed in this paper—the interprocedural analysis uses separation constraint partitioning to first group nodes at def and call site boundaries, after which interior nodes are merged using iterated partitioning.

Figure 5 shows the dynamic TAM instruction distribution for the benchmark programs under the four partitioning schemes, each normalized to dataflow partitioning. Since the cost for each TAM instruction differs, this figure does not necessarily reflect execution time which is presented later. Instructions are classified into one of four categories: ALU operations, heap accesses, communication, and control operations. The programs toward the left of the figure exhibit very fine-grain parallelism and are control intensive. The moderate blocking (4x4) and regular structure of MMT shows a significant contrast. As expected, improved partitioning substantially reduces the number of control operations. For most programs, iterated partitioning reduces the number of control operations by more than a factor of 2. For Simple and MMT the reduction is much larger.[5] Interprocedural partitioning further reduces the control operations for the more finely grained programs, while for the coarse grained programs the improvement is insignificant. Interprocedural and strict partitioning also decrease the number of instructions related to communication, as the grouping of arguments and results reduces the number of messages. This effect is particularly important since communication operations are more than ten times as expensive as any other.

In order to see the effectiveness of separation constraint partitioning combined with interprocedural analysis we look at how boundary nodes are grouped into threads. In the code generation to TAM, passing of arguments and results for a function invocation requires send instructions. Similarly, the implementation of conditionals is based on switches, which, depending on the result of the predicate, steer the control to one of two successor threads. One distinguishing feature about partitioning across blocks is that it may group nodes at block boundaries. For example, multiple send nodes residing in the same thread can be grouped into a single send node if the corresponding receive nodes also reside in a single thread. A similar optimization also occurs at boundaries of conditionals. Here multiple switch operations can be replaced by a single switch.

[5] Just as important as the decrease of the number of control operations is the fact that they also become simpler. For example, forks to synchronizing thread often turn into forks to unsynchronizing threads.

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

Table 1: *Benchmark programs and their inputs. The programs are described in [CGSvE93].*

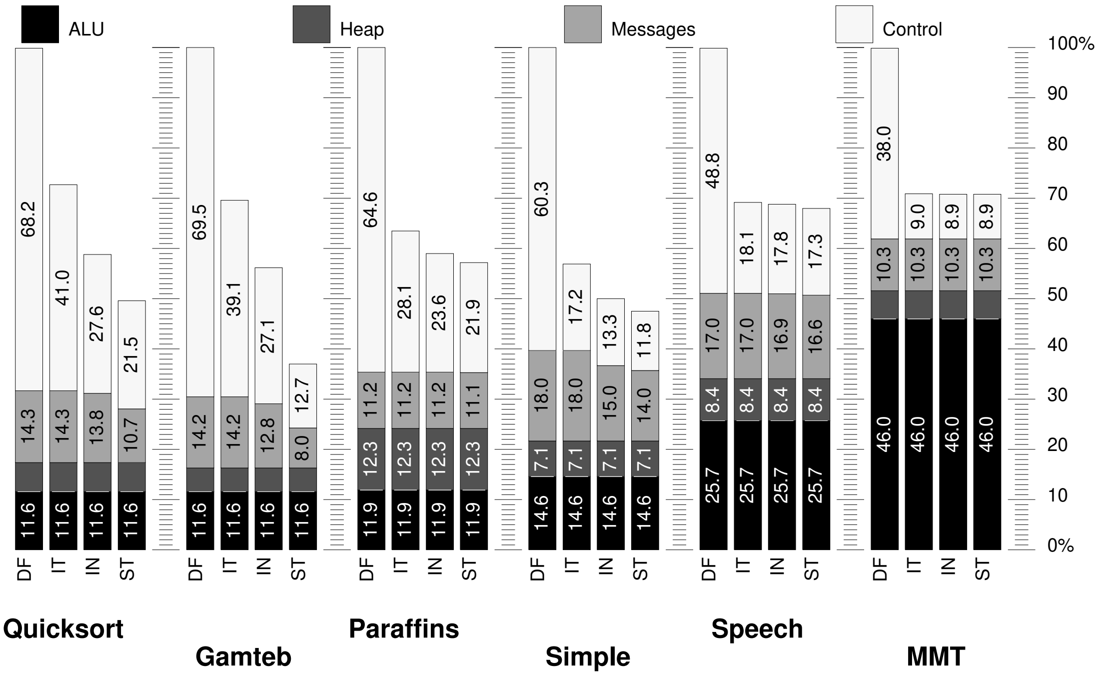

Figure 5: *Dynamic TAM instruction distribution for the benchmark programs under various partitioning schemes (normalized to dataflow partitioning). ALU includes integer and floating-point arithmetic,* messages *includes instructions executed to handle messages,* heap *includes global I-structure and M-structure accesses, and* control *represents all control-flow instructions including moves to initialize synchronization counters.*

	Sends per Call					
Partitioning	Quicksort	Gamteb	Paraffins	Simple	Speech	MMT
DF	4.7	8.4	3.1	7.9	8.3	8.2
IT	4.7	8.4	3.1	7.9	8.3	8.2
IN	4.2	6.6	2.9	2.7	5.1	5.2
ST	2.0	2.0	2.0	2.0	2.0	2.0

	Switches per Conditional					
Partitioning	Quicksort	Gamteb	Paraffins	Simple	Speech	MMT
DF	2.1	3.9	1.5	1.8	1.0	1.0
IT	2.1	3.9	1.5	1.8	1.0	1.0
IN	1.1	2.0	1.0	1.2	1.0	1.0
ST	1.0	1.0	1.0	1.0	1.0	1.0

Table 2: *Dynamic ratio of sends per call and the ratio of switches per conditional for the benchmark programs under various partitioning schemes.*

Table 2 shows the number of send instructions per function call and the number of switches per conditional across the different partitioning schemes. Recall that dataflow partitioning and iterated partitioning alone do not work across block boundaries, and thus do not group sends at function calls or switch nodes at conditional boundaries. A function call requires at least two sends: one for sending in the arguments and another for returning the results. This is precisely the case under strict partitioning. Similarly, every conditional requires at least one switch.

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

Table 3: *Dynamic run-time in seconds on a SparcStation 10 for the benchmark programs under various partitioning schemes.*

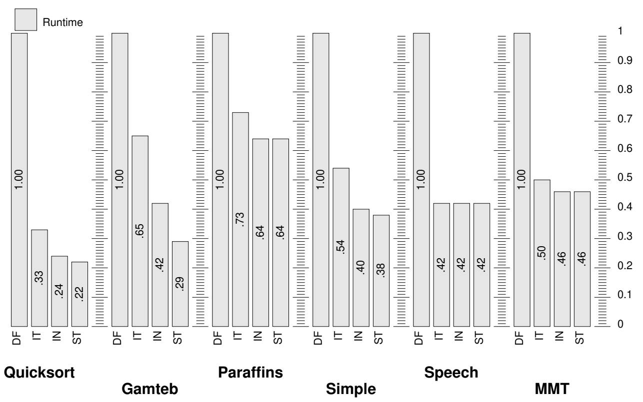

Figure 6: *Dynamic run-time on a SparcStation 10 for the benchmark programs under various partitioning schemes (normalized to dataflow partitioning).*

Finally, we see how the reduction in the number of control instructions leads to better performance. Table 3 shows the actual runtime measurements for the benchmark programs on a SparcStation 10 under the various partitioning schemes. Figure 6 shows the same timings graphically, normalized to dataflow partitioning. With the exception of Gamteb, interprocedural partitioning is very close to optimal as represented by strict partitioning.

6 Conclusions

The main contribution of this paper is the development and implementation of substantially improved thread partitioning algorithms. We present a new block partitioning algorithm, *separation constraint partitioning*, which is more powerful and also more flexible than iterated partitioning, the previously best known algorithm. Separation constraint partitioning uses a simple rule to determine whether or not any two nodes can be merged. Our algorithm is guaranteed to derive maximal threads. We present a theoretical framework for proving the correctness of our partitioning approach. In addition, we show how separation constraint partitioning makes interprocedural partitioning viable.

We have implemented the partitioning algorithms, resulting in a running execution vehicle for Id90 on workstations and parallel machines. Using this experimental platform, we were able to quantify the effectiveness of different partitioning schemes on whole applications. Experimental data show that sophisticated partitioning substantially reduces the control overhead compared to a partitioning reflecting the simple thread capabilities of dataflow machines. Separation constraint partitioning combined with interprocedural analysis obtains nearly the same efficiency as our overly optimistic strict partitioning scheme. Thus we are able to verify that the thread sizes are not limited by our partitioning algorithms, but rather by long latency operations and conventional control flow.

This work opens several fruitful avenues for future research. (i) Partitioning algorithms can be applied to other languages, especially strict and lazy functional programming languages. For example, in the case of lazy languages a partitioner may group the evaluation of several arguments if the compiler can show that they will always be evaluated together. This can even happen if the function is not strict in the grouped arguments [TCS92]. (ii) The flexibility offered by separation constraint partitioning should be explored more fully. Currently we only exploit this flexibility in the context of interprocedural partitioning, where we group boundary nodes before any other nodes. One can develop merge order heuristics for the interior nodes which minimize synchronization and dynamic scheduling. For example, merging two connected nodes results in less synchronization than merging two unrelated nodes. (iii) Using separation partitioning we have reduced the control overhead such that a large fraction of the remaining inefficiency is due to accesses to synchronizing I-structures. Better data structure analysis would reduce this overhead and further enhance the ability of interprocedural analysis to derive larger threads.

Acknowledgments

We are grateful to the anonymous referees for their valuable comments. We would also like to thank Martin Rinard, Chris Scheiman, Deborah Weisser, and Pedro Dinez for their comments. Computational support at Berkeley was provided by the NSF Infrastructure Grant number CDA-8722788. Klaus Erik Schauser received research support from the Department of Computer Science at UCSB. David Culler is supported by an NSF Presidential Faculty Fellowship CCR-9253705 and LLNL Grant UCB-ERL-92/172. Seth Copen Goldstein is supported by an AT&T Graduate Fellowship.

References

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

[ACI+83] Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali, and R. E. Thomas. The Tagged Token

Dataflow Architecture. Technical report, MIT Lab for Comp. Sci., August 1983.

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

[BH87] A. Bloss and P. Hudak. Path Semantics. In *Mathematical Foundations of Programming Language Semantics (LNCS 298)*. Springer-Verlag, April 1987.

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

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

[CPJ85] C. Clack and S. L. Peyton-Jones. Strictness Analysis - A Practical Approach. In *Proc. Functional Programming Languages and Computer Architecture*, Sept. 1985. Springer-Verlag LNCS 201.

[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.

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

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

[Gol94] S. C. Goldstein. Implementation of a Threaded Abstract Machine on Sequential and Multiprocessors. Master's thesis, Computer Science Division — EECS, U.C. Berkeley, 1994. (UCB/CSD 94-818).

[HDGS91] J. E. Hoch, D. M. Davenport, V. G. Grafe, and K. M. Steele. Compile-time Partitioning of a Non-strict Language into Sequential Threads. In *Proc. Symp. on Parallel and Distributed Processing*, Dec. 1991.

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

[Jor83] H. F. Jordan. Performance Measurement on HEP — A Pipelined MIMD Computer. In *Proc. of the 10th Annual Int. Symp. on Comp. Arch.*, Stockholm, Sweden, June 1983.

[Kie87] R. B. Kieburtz. A RISC architecture for symbolic computation. In *Proc. of 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[Myc80] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *International Symposium on Programming (LNCS 83)*. Springer-Verlag, April 1980.

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

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

[NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proc. 19th. Annual Intl. Symp. on Computer Architecture*, May 1992.

[NRB93] W. A. Najjar, L. Roh, and W. A. P. Böhm. Initial Performance of a Bottom-Up Clustering Algorithm for Dataflow Graphs. In *Proc. IFIP Conf. on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*. North-Holland, January 1993.

[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.

[PCSH87] S. L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP – A High Performance Architecture for Parallel Graph Reduction. In *Proc. Intl Conf. on Functional Programming and Comp. Arch.*, Sept. 1987.

[Pey92] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-Machine. *J. Functional Programming*, April 1992.

[Sch94] K. E. Schauser. *Compiling Lenient Languages for Parallel Asynchronous Execution*. PhD thesis, Computer Science Div., University of California at Berkeley, May 1994.

[SCvE91] K. E. Schauser, D. Culler, and T. von Eicken. Compiler-controlled Multithreading for Lenient Parallel Languages. In *Proc. Conf. on Functional Programming Languages and Comp. Arch.*, Aug. 1991.

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

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

[SNvGP91] S. Smetsers, E. Nöcker, J. van Groningen, and R. Plasmeijer. Generating Efficient Code for Lazy Functional Languages. In *Proc. Functional Programming Languages and Comp. Arch.*, Aug. 1991.

[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.*, June 1989.

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

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

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

A Separation Constraint Partitioning Correctness

Here we briefly sketch the correctness proof for separation constraint partitioning. For convenience we introduce a definition of a legal partitioning under a particular inlet and outlet annotation, which represents the potential indirect dependencies. We show that a legal partitioning can be mapped into a non-suspensive thread satisfying Definition 1, and that separation constraint partitioning is a legal partitioning. The proof that the initial singleton annotation and the annotations derived by interprocedural analysis are actually valid can be found in [Sch94].

Definition 4 (Potential Indirect Dependence Set) Let $A = (\Sigma_i, Inlet, \Sigma_o, Outlet, CID)$ be the annotation given to a block. For a set $D \subseteq \Sigma_o \times \Sigma_i$ the set of edges

$$P_D = \{(s,r) | \exists o \in Outlet(s), \exists i \in Inlet(r) \text{ such that } (o,i) \in D\}$$

is called a *potential indirect dependence set* if the graph $(V, E_s \cup CID \cup P_D)$ is acyclic. We will use the notation $PID_D = P_D$ for such a potential indirect dependence set.

The dependencies of a P_D which lead to a cycle in the above graph cannot appear at run-time since they would lead to a deadlock of the program. Thus it is sufficient for the partitioning algorithm to consider only the potential indirect dependence sets, PID_D.

Definition 5 (Partitioning) A *partitioning* of a block $(V, E_s \cup E_q)$ is a set Θ of non-overlapping subsets of the nodes V which cover it completely. For a partitioning Θ and a node $v \in V$, $\Theta(v)$ denotes the partition that contains v. The reduced graph $(V, E_s \cup E_q) \downarrow \Theta$ contains a node for every partition of the block and the edges that cross these partitions. If both a straight and squiggly edge cross the same two partitions, the squiggly edge is given preference.

A correct partitioning has to avoid introducing circular dependencies between threads, including both static cycles within blocks and dynamic cycles across blocks. In addition, a correct partitioning must ensure that requests and responses to split-phase operations are mapped to different threads.

Definition 6 (Legal Partitioning) A partitioning Θ of a block is *legal* under an annotation A if for all PID_D the following three conditions are satisfied

1. the reduced graph $(V, E_s \cup CID \cup PID_D) \downarrow \Theta$ is acyclic,
2. $\Theta(s) \neq \Theta(r)$ for all $(s,r) \in PID_D$, and
3. $\Theta(s) \neq \Theta(r)$ for all $(s,r) \in CID$.

Lemma 1 (Correctness) A legal partitioning can be mapped to threads obeying Definition 1.

> **Proof sketch:** We need to show that a sequential ordering can be found for each thread and that the thread does not suspend. The threads do not have to suspend because the reduced graph is acyclic and all indirect dependencies (certain and potential) cross threads. This also implies that the ordering for any thread can be determined independently of the ordering of other threads. Since the dataflow graph of a block is acyclic, any topological ordering will do. □

Before proving the correctness of separation constraint partitioning we prove three auxiliary propositions.

Proposition 1 (PID transitivity) If there exists a PID_D such that $(s_1,r_1) \in PID_D$, $(s_2,r_2) \in PID_D$, and there exists a path from r_1 to s_2 in $(V, E \cup CID \cup PID_D)$ then there also exists a $PID_{D'} \supseteq PID_D$ such that $(s_1,r_2) \in PID_{D'}$.

> **Proof sketch:** Let PID_D be the potential indirect dependence set. Let $o_1 \in Outlet(r_1)$, $i_1 \in Inlet(r_1)$, $o_2 \in Outlet(r_2)$, $i_2 \in Inlet(r_2)$, be the annotations for the two PID edges, i.e., $(o_1,i_1),(o_2,i_2) \in D$. For $D' = D \cup \{(o_1,i_2)\}$ we can show that $PID_{D'}$ is also a potential indirect dependence set conforming to Definition 4. □

The next proposition may seem surprising at first glance but is due to our use of inlet/outlet annotations to represent potential indirect dependence edges: we assume that potential indirect dependence edges are present unless contradicted by certain dependencies.

Proposition 2 (PID arcs) If there exists a PID_D such that $(s_1,r_1) \in PID_D$ and $(s_2,r_2) \in PID_D$ then there also exists a $PID_{D'} \supseteq PID_D$ such that either $(s_1,r_2) \in PID_{D'}$ or $(s_2,r_1) \in PID_{D'}$.

> **Proof sketch:** Trying to contradict both $(s_1,r_2) \in PID_{D'}$ and $(s_2,r_1) \in PID_{D'}$ will also contradict one of $(s_1,r_1) \in PID_D$ or $(s_2,r_2) \in PID_D$. □

Now we show that if partitioning a dataflow graph results in a reduced graph with a cycle, then we can always find a cycle involving at most one PID edge.

Proposition 3 (Single PID Arc) If there exists a PID_D such that $(s,r) \in PID_D$ and there exists a path from $\Theta(r)$ to $\Theta(s)$ in $(V, E_s \cup CID \cup PID_D) \downarrow \Theta$, then there exists a $PID_{D'}$ and $(s',r') \in PID_{D'}$ such that there exists a path from $\Theta'(r')$ to $\Theta'(s')$ in $(V, E_s \cup CID) \downarrow \Theta$.

> **Proof sketch:** We can show this by induction over the number of PID edges. Assume that (s_1,r_1) and (s_2,r_2) are two neighboring PID edges in the cycle. Then, from Proposition 2 we determine that there exists $PID_{D'}$ such that $(s_1,r_2) \in PID_{D'}$ or $(s_2,r_1) \in PID_{D'}$. In the first case, Propositions 1 and the induction hypothesis yields the result. In the second, we immediately obtain a cycle with only one PID edge. □

There are two key aspects to the main correctness proof of separation constraint partitioning. First, we need to show that the algorithm correctly updates the set of indirect dependencies ID throughout the execution of the program. This implies that certain and potential indirect dependencies are correctly taken care of. Second, we prove that when the algorithm terminates, all partitions are convex, i.e., there do not exist any static cycles from a thread back to itself. This may not be the case at intermediate steps of the algorithm.

Theorem 1 (Separation Constraint Partitioning) Separation constraint partitioning is a legal partitioning.

> **Proof sketch:** We need to show that the three properties of legal partitioning are satisfied.
>
> 1) We first show that while the algorithm runs, there do not exist any cycles which involves certain indirect dependence edges or a single potential indirect dependence edges. This follows from the construction of ID and how ID is being updated at the merge step. Proposition 3 then implies that there are no cycles involving any PID edges. Next we show that when the algorithm finishes all partitions are convex, i.e., do not have cycles involving only straight edges. This is the case because nodes along such a cycle would not have any separation constraints between them and therefore could still be merged.
>
> 2) and 3) directly follows from the way ID is initially created, and that separation constraints never disappear. □

This proof not only shows that separation constraint partitioning is correct, but it also shows that at the end there is a separation constraint between any pair of partitions. This implies that no further merging is possible, even when other partitioning schemes are used. Therefore separation constraint partitioning finds a maximal solution.