Global Analysis for Partitioning Non-Strict Programs into Sequential Threads

Kenneth R. Traub
Motorola Cambridge Research Center[1]

David E. Culler
UC Berkeley Computer Science Division[2]

Klaus E. Schauser
UC Berkeley Computer Science Division[2]

1 Introduction

In this paper we present a new solution to the problem of compiling an eager, non-strict language into multiple sequential threads. The solution is described using an intermediate program form developed for the programming language Id [Nik90], a functional language extended with I-structures [ANP86] and M-structures [BNA91]. A similar intermediate form has also been suggested for imperative languages [BP89], as a way of exposing parallelism in those languages. With suitable restrictions, we believe our method is also appropriate for lazy, purely functional languages such as Haskell [HWe90].

Throughout this paper, a *thread* will mean a subset of the instructions comprising a procedure body, such that:

1. A compile-time instruction ordering can be determined for the thread which is valid for all contexts in which the containing procedure can be invoked.

2. Once the first instruction in a thread is executed, it is always possible to execute each of the remaining instructions, in the compile-time ordering, without pause, interruption, or execution of instructions from other threads. [Tra91b]

Threads of this form are required to implement a language on parallel multithreaded hardware (*e.g.*, Monsoon [PT91, PC90] and *T [NPA92]), and they yield efficient implementations on conventional architectures when combined with a suitable abstract machine such as TAM [CSS$^+$91].

Our algorithms are greedy, and the following comments from [SCvE91] apply:

> 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 ...

[1] Motorola Cambridge Research Center; One Kendall Square, Bldg. 200; Cambridge MA 02139; kt@mcrc.mot.com

[2] Computer Science Division–EECS; 571 Evans Hall; UC Berkeley; Berkeley CA 94720; culler@cs.berkeley.edu; schauser@cs.berkeley.edu

Obtaining such threads is not trivial, because the order of subexpression evaluation in non-strict programs is not specified syntactically, as it is in imperative languages. To determine an ordering, the compiler must respect all data dependences. For example, a fetch from an element of a non-strict array depends on the subexpressions that yield the array and offset (which are identifiable at compile time), and on the subexpression computing the component (which, in general, is not). Fundamentally, therefore, a compiler must sort out *certain* dependence (*i.e.*, known at compile time), and *potential* dependence [Tra91a].

This paper extends or improves upon prior work [Tra91a, Ian90, SCvE91, HDGS91] in the following ways:

More abstract framework. The intermediate form used here is an ordinary directed graph, where vertices are primitive operations and edges indicate flow of operands (and therefore certain dependence), with three additional annotations: an *inlet* annotation on vertices that may be endpoints for incoming potential dependence, an *outlet* annotation on vertices that may be endpoints for outgoing potential dependence, and squiggly edges to indicate certain, but indirect dependence. All relevant information about how operators behave and their inter-relationships are encoded into these notations, as is information exchanged between graphs during interprocedural analysis. This is in contrast to previous approaches which require special and *ad hoc* treatment of several kinds of nodes in the graph.

Congruence. While the concept of inlet and outlet is found elsewhere ([Ian90], [SCvE91], and [HDGS91]), our annotations allow for the expression of *congruence*, where, for example, two inlets depend on the same (but still unknown at compile time) set of outlets. Even more complicated patterns of overlap and partial overlap are also expressible. Congruence is exploited extensively by our interprocedural analysis.

Interprocedural analysis. If procedure f was previously compiled, information gained during its compilation may be used to form larger threads when compiling another procedure g which calls f, compared to compiling g in isolation. If g is known to be the only caller of f, then compilation information from g can likewise improve the compilation of f. This process may be iterated to arrive at mutually improved versions of both f and g. None of these interprocedural analyses have been considered in the prior work. It is also interesting to note that our interprocedural analysis is in some ways more powerful than conventional strictness analysis [BHA85, Myc80]. For example, we can conclude that the expressions computing the actual parameters to a two-argument function may be placed in the same thread, even though the function is not strict in either argument.

Conditionals. The interprocedural analysis as outlined above is extended to the case where a particular call in g is known to call one of several procedures $f_1, f_2, \ldots$, depending on context or input data. A special case of this paradigm is a conditional expression, where $f_1, f_2, \ldots$ are the arms of a conditional and g is the enclosing procedure.[3] Again, it is possible to improve both the caller g and the callees f_1, f_2, etc. Another generalization, though perhaps less useful in practice, is the specialization of f when it is known that its only callers are g_1, g_2, etc. Prior work has *ad hoc* treatment of conditionals, in some cases violating the semantics of the language (as in [Ian90]), and in other cases inherently limited in effectiveness (*e.g.*, [SCvE91]).

Preservation of control structure. The membership of operators in different control regions is preserved; *e.g.*, instructions belonging to the "then" arm of a conditional are kept separate from those in the "else" arm and from the containing expression. This is very desirable for later phases of compilation; *e.g.*, register allocation. Previous techniques lose this structure in the course of partitioning.

2 Dataflow Graphs

Programs to be partitioned are expressed in a *structured dataflow graph.* A structured dataflow graph consists of a collection of acyclic graphs describing *basic blocks*, and *interfaces* which describe how the blocks relate to one another. The term *basic block* is used here in the dataflow sense as defined in [Tra86]; 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 basic block.

2.1 Basic Blocks

Basic blocks are represented as acyclic directed graphs, with some additional annotations. The vertices are primitive operators, and straight edges connecting them indicate the flow of operands.[4] Figure 1 shows the repertoire of operators commonly used to compile Id [Tra86]. The *send* and *receive* operators communicate values between basic blocks, corresponding to procedure linkage and conditional expressions in the source language. (A real compiler would likely employ several varieties of *send* and *receive* to encode precise details of linkage conventions, which we ignore in this paper.) Two varieties of memory operators are shown. Ordinary *fetch* and *store* perform no run-time synchronization—the basic block representation is assumed to include enough edges to insure deterministic sequencing of their side-effects [BP89]. *I-fetch* and *I-store* [ANP86] are examples of synchronizing memory operations, where the response to an *I-fetch* is not received until an *I-store* to the same location takes place. In general, the compiler does not know which *I-store*s go with which *I-fetch*es. The opcode labels shown in Figure 1

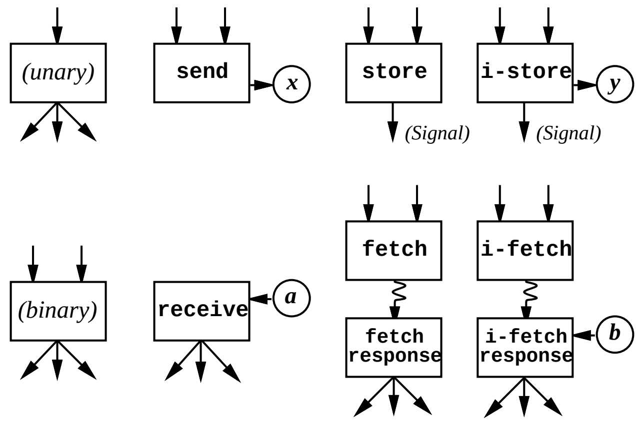


Figure 1: Dataflow Graph Base Language

are extraneous as far as the partitioning algorithms are concerned. On the other hand, they are needed by later phases of compilation, and so it is convenient to carry them along during partitioning.

A straight edge in the graph shows the flow of an operand, but more importantly, it indicates *certain* dependence. A certain dependence is an ordering constraint due to data dependence that exists for every invocation of the block, regardless of context. A squiggly edge is used to indicate an *indirect* certain dependence; that is, a certain dependence which is completed through one or more vertices in a different basic block. Indirect certain dependences commonly arise in two situations. First, there is an indirect certain dependence between a vertex that issues a *split-phase transaction* [AN87] and the vertex that receives the response (the dependence is completed through an implicit vertex that services the request). The *fetch* and *I-fetch* operators in Figure 1 are examples of split-phase transactions. Second, global analysis can discover indirect certain dependences between pairs of *send* and *receive* vertices that interface to other basic blocks. Vertices separated by an indirect dependence can never be placed in the same thread, owing to part 2 of the definition of a thread.

Also noted in the graph are *potential* dependences: ordering constraints due to data dependence that may differ depending on the context in which a procedure is invoked, or which are otherwise unknown at compile time. Rather than introducing edges into the graph, all vertices that may be endpoints of potential dependence are annotated with *inlet* and *outlet* annotations (the circles in Figure 1). For a given context in which the procedure is invoked, there will be dependence paths from some of the outlets to some of the inlets; these dependences must be detected through synchronization counters, presence bits, or other means, and will constrain the order of evaluation. Note that potential dependences are assumed to be arbitrary paths, possibly completed through nodes outside the basic block. For this reason, an inlet annotation on an *I-fetch*, for example, may depend on the outlet on a *send* as well as on the outlet on an *I-store*.

Inlet and outlet annotations are sets of names; if a vertex has an empty inlet (outlet) annotation, it is never an endpoint for incoming (outgoing) potential dependence. Often,

[3] The idea of a conditional as a generalization of procedure call is due to [AA89]. Treating conditionals in this way has the beneficial effect of breaking all cycles in the dataflow graph for a legal program.

[4] There may be edges that carry no data but serve only to insure correct sequencing of operators that cause side-effects. For partitioning purposes, these may be viewed as carrying dummy operands of size zero; they need special treatment only during later phases of code generation.

non-empty inlet and outlet sets are unique singletons, meaning that any inlet may depend on any outlet. Two vertices may be given the same set of inlet names, indicating *congruence*: if two vertices have the same set of inlet names, that means they both depend on the same set of outlets, even though that set is not known at compile time. Overlapping name sets imply partial congruence. For example, if three vertices have inlet annotations $\{a, b\}$, $\{b, c\}$, and $\{a, b, c\}$, respectively, then any vertex v which depends on the first two vertices depends on the same set of outlets as does the third vertex (assuming that v depends on no other vertices, nor has its own inlet annotation).

Apart from congruence, the compiler has no further information about which inlets might depend on which outlets, except that potential dependence is ruled out when contradicted by certain dependence. This, then, is the key to successful partitioning: the more the compiler knows about certain dependence, the larger the threads it can form. This is why indirect certain dependences (squiggly edges) are useful. Though they force the separation of vertices into separate threads, they help to rule out other, potential, dependences. The goal of global analysis is to transfer as much congruence and indirect certain dependence information between basic blocks as possible.

2.2 Interfaces

Interfaces describe how basic blocks interact with each other, and are used in global analysis to propagate information between blocks. Global analysis alternates between partitioning basic blocks in isolation and propagating information across interfaces. Each interface establishes a correspondence between *call sites* in one or more basic blocks and the *def sites* of one or more basic blocks.

A basic block is essentially an n-argument, m-result procedure. The arguments are received via n *receive* nodes, and the results are returned via m *send* nodes. Collectively, these *receive*s and *send*s comprise an n-argument, m-result def site.

Conversely, one basic block calls another through an n-argument, m-result call site, consisting of n *send* nodes and m *receive* nodes. While every basic block has exactly one def site, it may have any number of call sites (including zero).

The simplest interface relates a single call site to a single def site, resulting from a procedure call to a known procedure. The basic block containing the call site is the *caller*, and the basic block containing the def site is the *callee*. During the propagation phase of global analysis, information gained from partitioning the callee is used to change the annotations on the *send* and *receive* nodes in the caller's call site. These changes include better inlet and outlet annotations to indicate a greater degree of congruence, as well as squiggly arcs from *send*s to *receive*s to reflect certain dependence paths in the callee. These can lead to larger threads in the caller during the next round of basic block partitioning (and in no case will result in smaller threads). Likewise, information gained from partitioning the caller is used to change annotations in the callee's def site. The compiler is, of course, free to propagate information in only one direction across an interface, or not at all. For example, information would not be propagated from caller f to callee g if it was not desirable to produce a version of g specialized to work properly only when called from f.

A more complex variety of interface relates a single call site to multiple def sites. This case commonly arises from conditionals, where the multiple callees are the arms of the conditional and the caller is the enclosing expression. Here, information from the caller results in new inlet/outlet/squiggly annotations to be attached to each def site in the same way. In the other direction, the information from the callees are combined conservatively to yield a single set of new inlet/outlet/squiggly annotations for the caller. The combination, described in Sections 5 and 6, is done such that the caller will work properly regardless of which callee is selected at run time.

A dual variety of interface with multiple call sites and a single def site is also possible, with transposed propagation rules. This case would arise if all the callers of a single procedure are known statically, and it is desirable to specialize the callee for these callers (and vice versa). In theory one could have an interface with multiple call sites and multiple def sites, but the need never arises in practice, and in any event it can be expressed as the composition of a multiple-call-single-def interface with a single-call-multiple-def interface.

3 Basic Block Partitioning

This section gives the algorithm for partitioning a basic block in isolation. This is essentially a subroutine of the full global partitioning algorithm described in later sections. The goal of basic block partitioning is to group the vertices of a basic block into disjoint subsets that are trivially mapped into threads which meet the two requirements laid out in Section 1. A compile-time ordering of the operations can be obtained by any topological sort according to the certain dependence edges (*i.e.*, straight edges) wholly within the thread. The ordering chosen for one thread of a basic block does not affect or constrain the orderings that may be chosen for other threads. We will generally refer to such a subset of vertices simply as a thread.

A basic block algorithm similar to that presented here is independently developed by Hoch *et al.* [HDGS91], and our algorithm is also equivalent in power to the algorithm of Schauser *et al.* [SCvE91]. A significant difference is that prior algorithms had *ad hoc* treatment of split-phase transactions, while for us this falls out of the general method for handling indirect certain dependences (squiggly edges). Our algorithm consists of *dependence set partitioning* and *demand set partitioning*, each of which is combined with a *subpartitioning* algorithm to insure that all squiggly edges are inter-thread. These are then composed via an iterative procedure to obtain the final algorithm.

3.1 Dependence Set and Demand Set Partitioning

Dependence Set and Demand Set Partitioning seek to group together nodes of the basic block which may be executed as a single thread, regardless of what potential dependences may occur at run time. No distinction is made between straight edges and squiggly edges in the graph, and so these algorithms are combined with Subpartitioning which splits partitions that enclose squiggly edges.

Dependence Set partitioning forms partitions by grouping together all nodes that depend on the same set of inlets, and was originally discovered by Iannucci [Ian90].

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

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

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

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

1. Compute $Dep(v)$ for all nodes v. Because the graph is acyclic, this is easily done by a single traversal in topological order.

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

The second algorithm is the dual of dependence set partitioning: it groups together all nodes which are demanded by the same set of outlets. That is, if an outlet depends on one node in a group, it depends on all the other nodes in the group as well. This algorithm is called Demand Set partitioning, and was first reported in [SCvE91].[5]

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

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

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

The Demand Set Partitioning algorithm is analogous to Algorithm 1.

3.2 Subpartitioning

After partitioning with either of the previous section's algorithms, there may be two nodes in the same subset which are connected by a squiggly edge, violating Part 2 of the thread definition of Section 1. Subpartitioning is applied to divide subsets in which this occurs into smaller threads.

Threads may not be split into subpartitions arbitrarily, as doing so may introduce a cycle between threads (Figure 2). Such a cycle would also violate the second part of the thread definition. The following algorithm correctly subpartitions a thread, assigning a subpartition number to each node within a thread such that all nodes with the same number are part of the same subpartition. The number is simply the maximum distance in squiggly arcs from the roots of the thread.

Algorithm 2 (Subpartitioning (forward))
Given a thread:

[5] In [SCvE91], the algorithm was called "dominance set partitioning."

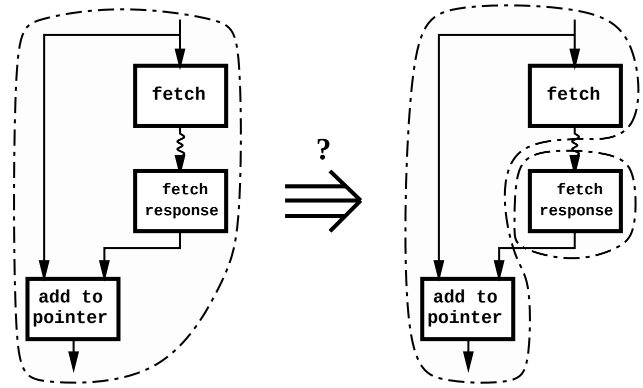


Figure 2: Incorrect Subpartitioning

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

$$Subpart(v) \leftarrow \max(0, \max_{u \in Pred_s(v)} Subpart(u), 1 + \max_{u \in Pred_q(v)} Subpart(u))$$

   where $Pred_s(v)$ and $Pred_q(v)$ are the immediate predecessors of v via straight edges and squiggly edges, respectively.

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

Clearly, this algorithm cannot introduce a cycle between subpartitions, as subpartition numbers are monotonically increasing along any path.

Forward subpartitioning can easily be accomplished during the same topological-order pass made by the dependence set partitioning algorithm. For each node v, $Subpart(v)$ is computed just after computing $Dep(v)$ according to the formula in Step 1a, above, except that $Pred_s(v)$ and $Pred_q(v)$ are taken to refer only to predecessors of v which have the same dependence set. Threads are then formed by finding maximal subsets of vertices with both the same $Dep(v)$ and the same $Subpart(v)$. Similarly, backward subpartitioning can be accomplished during demand set partitioning through a similar modification. Backward subpartitioning is the dual of Algorithm 2, where the traversal is in reverse topological order, and where $Succ_s$ and $Succ_q$ replace $Pred_s$ and $Pred_q$, respectively.

To show these algorithms are correct, we must show that the threads they discover satisfy both parts of the thread definition. Informally, one can argue correctness by showing that no static or dynamic dependence path between two nodes in a partition can be completed through a node outside the partition. No static path can exist, because nodes along such a path must have monotonically increasing dependence sets (or monotonically decreasing demand sets). No dynamic path can exist, because such a path would have to be completed through one of the inlets (outlets) in the thread's dependence (demand) set, implying a cycle. Finally, subpartitioning insures that the second part of the definition of a thread is satisfied.

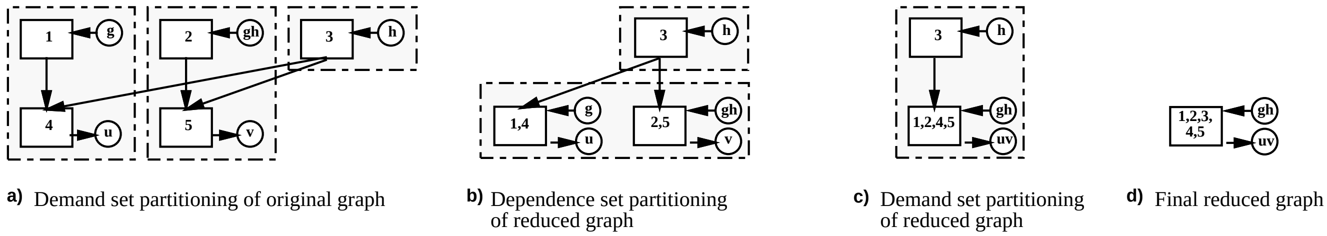


Figure 3: Example of Iterated Partitioning

3.3 Iterated Partitioning

A better partitioning algorithm can be obtained by the iterated application of dependence set and demand set partitioning. This is done by considering each thread computed by one algorithm as a node of a *reduced graph*, and applying the other algorithm to the reduced graph.

Given a basic block and a partitioning (a collection of disjoint subsets of vertices), the reduced graph is a basic block with: a vertex for every thread in the partitioning, an edge for every pair of threads with an inter-thread edge (squiggly edges taking priority), an inlet annotation on each vertex which is the union of inlet annotations on the original vertices comprising the thread, and likewise for the outlet annotations.

The iterated partitioning algorithm is a cycle of Dependence Set partitioning (with subpartitioning), form the reduced graph, Demand Set partitioning (with subpartitioning), form the reduced graph. This is repeated until the number of threads does not change.

Figure 3 shows the stages of applying the iterated partitioning algorithm to a small example. The example consists of five nodes, three *receive*s and two *send*s annotated with the inlet and outlet names shown in part (a) of the figure. Starting with demand set partitioning, three threads, as indicated by the shaded area, are identified (the three demand sets are $\{u\}$, $\{v\}$, and $\{u, v\}$). The reduced graph, shown in part (b), has a node for every thread and inlet/outlet annotations which are the union of the annotations of the original nodes comprising the threads in part (a). Two dependence sets $\{h\}$ and $\{g, h\}$ determine the new partitioning of the reduced graph. Forming the reduced graph again and applying demand set partitioning results then in a single thread, as shown in part (c) of the figure. Thus, the iterated partitioning algorithm was able to group all five nodes of the original graph into a single thread.

4 Global Analysis

The effectiveness of the partitioning algorithm presented above is limited by the inability to propagate dependence and demand information across control boundaries and procedure calls. Without global analysis, each *send* in a call site or def site is annotated with a unique outlet name and each *receive* is annotated with a unique inlet name. Our global analysis attempts to eliminate unnecessary outlets and inlets in a call site in favor of squiggly arcs, where certain, indirect dependences completed through the callee can be discovered. In other cases, it will give the same inlet (outlet) name to multiple *receive*s (*send*s). This occurs where it is discovered that the corresponding *send*s (*receive*s) in the callee have the same dependence set (or demand set). More generally, nodes may be given overlapping annotations describing the sharing between their dependence or demand sets. Eliminating or refining annotations in this way allows the repeated application of the partitioning procedure to produce larger threads.

The analysis works in the opposite direction as well; that is, it can use information obtained from a call site in a caller to eliminate or refine annotations in the corresponding def site in the callee. In this direction, squiggly arcs result when the caller feeds a result returned by the callee back into an argument of the callee. The possibility of such feedback is the chief reason why non-strict languages are more expressive than their strict counterparts. [Tra91a]

In this section, all interfaces are assumed to be single-call-single-def, *i.e.*, they relate a single call site to a single def site. Extensions for single-call-multiple-def and multiple-call-single-def interfaces will be presented in the subsequent section.

4.1 Global Partitioning

The global partitioning algorithm starts off by initializing all sites with the most general annotation. Then, iteratively one of two rules is applied: either a site annotation is refined by propagating information across an interface, or a basic block is partitioned. The algorithm terminates when neither additional propagation nor partitioning results in a change.

After a block is partitioned, the sites on the other sides of interfaces to that block may require updated annotations to reflect the block's new structure. To keep track of this, the algorithm associates with every call and def site a boolean, $Valid(s)$, indicating whether the annotations at that site are an accurate summary of the block on the other side of the interface. A propagation step makes the reannotated site valid, while a partitioning step invalidates all the sites to which information about the block had previously been propagated. A set of blocks, $Uses(s)$, is also associated with every site. If $Valid(s)$ is true, then $Uses(s)$ is the set of blocks whose structure was directly or indirectly propagated into s. $Uses$ helps determine which sites to invalidate after a partitioning step.

Algorithm 3 (Global Partitioning)
Given a structured dataflow graph $\mathcal{G}$:

1. Initially, annotate every receive with a unique singleton inlet and every send with a unique singleton outlet. Set $Valid(s) \leftarrow true$ and $Uses(s) \leftarrow \emptyset$ for all sites s.

2. Select a basic block B such that $Valid(s)$ is $true$ for all sites s in B, and apply either Step 3 or Step 4.

3. *(Propagate information across an interface)* Choose a site s in B. Let s' denote the corresponding site on the other side of the interface, and let B' be the basic block containing s'. Perform the following three steps:

   (a) Reannotate the *sends* and *receives* of site s' as a function of the site s, according to Algorithm 4, below.

   (b) Set $Valid(s') \leftarrow true$.

   (c) Set $Uses(s') \leftarrow \{B\} \cup \bigcup_{s_i \in S(B), s_i \neq s} Uses(s_i)$, where $S(B)$ denotes the set of sites that B contains. (This rule says that what site s' uses is the block containing s, plus any blocks which influenced the annotations at sites in B other than s itself. Annotations which may be present at s itself do not contribute to what is propagated to s', according to Algorithm 4.)

4. *(Partition the basic block)*

   (a) Partition B using any legal partitioning algorithm.

   (b) Set $Valid(s) \leftarrow false$ for all sites s in the program such that $B \in Uses(s)$.

5. Repeat from Step 2 until there is no change.

In some sense, Algorithm 3 is an algorithm schema: to obtain a complete algorithm, it must be combined with a strategy for making the choice in Step 2. The blocks and interfaces form a call graph, through which the algorithm must navigate. If the call graph is a tree, one strategy is to take sweeps from the leaves to the root, partitioning blocks and propagating to their parents, alternating with sweeps from the root to the leaves. If there are cycles in the call graph, the algorithm cannot propagate information around a complete cycle—if it does, partitioning any block in that cycle will permanently invalidate all site annotations on the cycle, preventing further progress. This is a fundamental limitation in our global analysis's ability to deal with recursion. A realistic strategy must content itself with choosing some spanning tree (or spanning DAG) of a cyclic call graph and limit propagation to those interfaces.

4.2 Propagation

The procedure for reannotating the site s' based on the corresponding site s is described now in detail. The interface relates nodes $Rcv_1, \ldots, Rcv_i, \ldots, Rcv_n$ in site s to nodes $Send'_1, \ldots, Send'_i, \ldots, Send'_n$ in site s'. If s is a def site and s' a call site, these nodes are communicating n arguments; if s is a call site and s' a def site, these nodes are communicating n results. The interface also relates nodes $Send_1, \ldots, Send_j, \ldots, Send_m$ in site s to nodes $Rcv'_1, \ldots, Rcv'_j, \ldots, Rcv'_m$ in site s'. If s is a def site and s' a call site, these nodes are communicating m results; if s is a call site and s' a def site, these nodes are communicating m arguments. The notation $\theta(Rcv_i)$ refers to the thread containing Rcv_i in the reduced graph.

Algorithm 4 (Propagation)

1. Let $R_i = Dem(\theta(Rcv_i))$, where this demand set is computed in the reduced graph of the basic block B, but where the *send* outlets of site s are given new unique names, $Outlet(Send_j) = \{\sigma_j\}$, $1 \leq j \leq m$, for the purpose of computing these demand sets.

2. Let $S_j = Dep(\theta(Send_j))$, where this dependence set is computed in the reduced graph of the basic block B, but where the *receive* inlets of site s are given new unique names, $Inlet(Rcv_i) = \{\rho_i\}$, $1 \leq i \leq n$, for the purpose of computing these dependence sets.

3. For each i, j such that $\sigma_j \in R_i$ (which is equivalent to $\rho_i \in S_j$; both are true if and only if there is a path from $\theta(Rcv_i)$ to $\theta(Send_j)$),

   (a) Add a squiggly arc $(Send'_i, Rcv'_j)$ in site s'.

   (b) Set $R_i \leftarrow R_i - \{\sigma_j\}$.

   (c) Set $S_j \leftarrow S_j - \{\rho_i\}$.

4. Set $Outlet(Send'_i) \leftarrow R_i$ in the site s' for all $1 \leq i \leq n$ (alpha-renamed if necessary to avoid congruence with other sites in B').

5. Set $Inlet(Rcv'_j) \leftarrow S_j$ in the site s' for all $1 \leq j \leq m$ (alpha-renamed if necessary to avoid congruence with other sites in B').

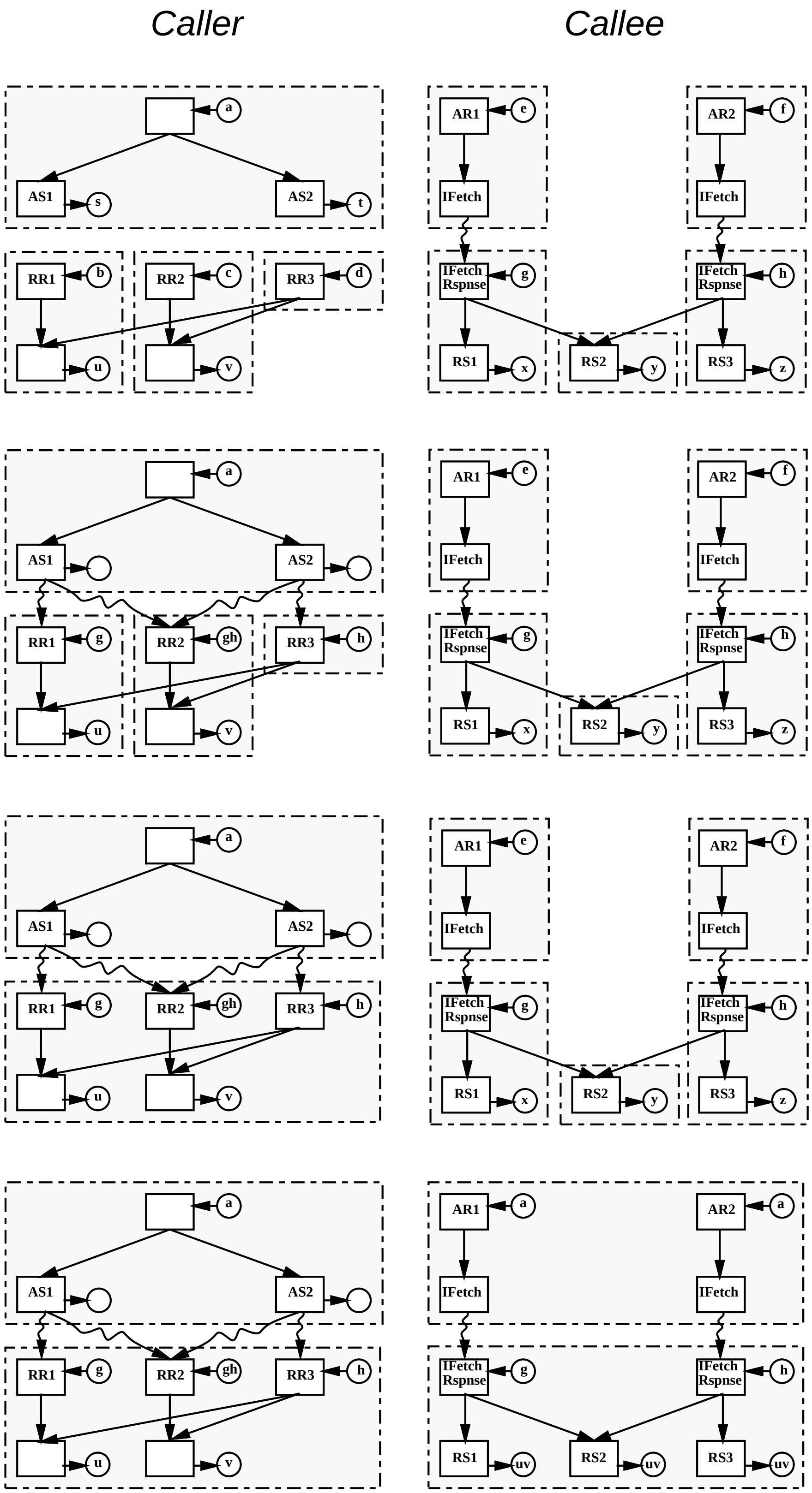
4.3 Example

The behavior of this global partitioning algorithm is illustrated in Figure 4. The top of the figure shows a caller with a 2-argument, 3-result call site (left) and corresponding callee, which contains two internal inlets due to I-fetch operations. After Step 1 of Algorithm 3, all site annotations are valid, and their *Uses* sets are empty. Therefore it is possible to partition each basic block in isolation before any inter-block propagation (top row of figure).

Now, consider def-to-call propagation. The callee's argument receive inlets are temporarily renamed ρ_1 and ρ_2 and the result send outlets renamed σ_1, σ_2, and σ_3, respectively. This gives $R_1 = \{\sigma_1, \sigma_2\}$, $R_2 = \{\sigma_2, \sigma_3\}$, $S_1 = \{\rho_1, g\}$, $S_2 = \{\rho_1, \rho_2, g, h\}$, and $S_3 = \{\rho_2, h\}$. The second row of the figure shows the resulting new annotations in the caller.

Next, the caller is partitioned. First, dependence set partitioning results in three partitions. Forming the reduced graph and then applying demand set partitioning gives one thread, which subpartitioning then splits in two, as shown in the third row of the figure. Note that the bottom part of the caller has exactly the structure of Figure 3a, and so the intermediate steps in iterated partitioning of the caller are similar to what is illustrated in Figure 3.

Finally, call-to-def propagation is used to specialize the callee for this caller. Here $S_1 = S_2 = \{a\}$, and $R_1 = R_2 = R_3 = \{u, v\}$. This introduces no squiggly edges in the callee, but makes all *receive* inlets the same ($\{a\}$) and all *send* outlets the same ($\{u, v\}$). Partitioning the newly annotated callee, we arrive at two threads (fourth row of figure). Another pass of propagation across the interface yields a small change to the caller annotation, but no improvement is possible in the partitioning.



Initial Annotation:

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

Initial Partitioning:

For the caller, dependence set partitioning results in six partitions. Forming the reduced graph and then applying demand set gives four partitions: {s,t}, {u}, {v}, and {u,v}.

For the callee, five dependence sets determine the partitioning: {e}, {f}, {e,g}, {f,h}, and {e,f,g,h}.

Reannotation:

Propagation from the callee to the caller:
Indirect certain dependence from AS1 to RR1 and RR2, as well as from AS2 to RR2 and RR3 is represented by the four squiggly arcs in the caller.
Outlet annotations on AS1 and AS2 can be eliminated.
Inlet annotation on RR1, RR2, and RR3 reflect the inlets internal to the callee.

Partitioning Caller:

Dependence set partitioning of the reduced graph from the second row results in three partitions: {a}, {a,h}, and {a,g,h}.

Forming the reduced graph and then applying demand set partitioning will give one partition. Subpartitioning results in two threads, as shown.

Reannotation:

Propagation from the caller to the callee: Dependence sets for arg-sends and demand sets for results-rcvs are reflected in the callee annotations.

Partitioning Callee:

For the callee, demand set partitioning places all nodes into a single partition. Subpartitioning results in two threads.

Figure 4: Example of Global Analysis

5 Congruence Syndromes

We digress for a moment to answer the following question: what is the information carried in inlet (outlet) annotations by the propagation step of Algorithm 3? Clearly, the names themselves are not relevant. For example, in Figure 4 the annotations $\{g\}, \{g, h\}, \{h\}$ were carried from the callee to the caller, but the partitioning would have been the same if the caller had received the annotations $\{j\}, \{j, k\}, \{k\}$ instead. It is not the names appearing in the inlet (outlet) annotations that matter, only which nodes end up with the same dependence (demand) sets—in this example, whether the bottom two nodes end up with the same dependence set. Thus, the annotations $\{g, j\}, \{g, j, h\}, \{h\}$ would also do as well, even though this is not just a α-renaming of $\{g\}, \{g, h\}, \{h\}$. On the other hand, the annotations $\{g\}, \{j\}, \{h\}$ do *not* have the same effect, as the bottom two nodes in Figure 4 end up with different dependence sets: $\{a, g, h\}$,$\{a, j\}$.

The relevant information carried by a set of annotations is its sharing properties, which are captured by its *congruence syndrome*, defined below.

Definition 3 *The* congruence syndrome *of a collection of sets of names $C = S_1, \ldots, S_n$ is a $2^n \times 2^n$ matrix defined as follows:*

$Syn(C) : Pow(\{1, \ldots, n\}) \times Pow(\{1, \ldots, n\}) \rightarrow \{0, 1\}$ where

$$Syn(C)(I_1, I_2) = 1 \text{ iff } \bigcup_{i \in I_1} S_i = \bigcup_{j \in I_2} S_j$$

The reader may verify that the congruence syndrome of $\{g\}, \{g, h\}, \{h\}$ is the same as the congruence syndrome of $\{g, j\}, \{g, j, h\}, \{h\}$, but different from the congruence syndrome of $\{g\}, \{j\}, \{h\}$. In the reannotation procedure, Step 5, a bank of n nodes Rcv'_j were reannotated with new inlet sets S_j. In fact, the nodes could be reannotated with any collection of inlet sets S'_j, as long as $Syn(\{S'_1, \ldots, S'_n\}) = Syn(\{S_1, \ldots, S_n\})$, and as long as all names in S'_i are disjoint from other inlets in the caller (to prevent unintended congruence). An analogous generalization applies to Step 4. Note that an α-renaming of a collection always yields a collection with the same congruence syndrome.

We stress that the algorithms we propose do not require computing any congruence syndromes; we have only used congruence syndromes to explain the underlying mathematics.

5.1 Partial Order on Congruence Syndromes

We define a partial order on congruence syndromes as follows:

$$syn_1 \sqsubseteq syn_2 \text{ iff } syn_1(I_1, I_2) \leq syn_2(I_1, I_2) \; \forall I_1, I_2$$

The syndromes of all finite collections of a given size n form a finite lattice under this ordering; for $n = 2$, for example, there are seven possible syndromes.

If $Syn(C^1) \sqsubseteq Syn(C^2)$, then C^1 is a conservative approximation of the annotations in C^2. That is, if labeling a bank of inlets or outlets with C^1 allows two vertices to be placed in the same partition, then they can also be placed in the same partition if the inlets or outlets were labeled with C^2, though C^2 may also allow groupings not allowed by C^1. Therefore, in the earlier example it is safe (though not optimal) to reannotate the outlets with any collection R'_i where $Syn(\{R'_1, \ldots, R'_n\}) \sqsubseteq Syn(\{R_1, \ldots, R_n\})$. This observation shows exactly what sort of combination needs to be done at a multiple-def or multiple-call interface: the new annotations must have a congruence syndrome which is the greatest lower bound (glb) of the congruence syndromes from each of the defs or calls.

Fortunately, it is possible to construct a collection C^{12} such that $Syn(C^{12}) = \text{glb}(Syn(C^1), Syn(C^2))$ without actually computing any congruence syndromes. First, α-rename C^1 and C^2 so that they have disjoint sets of names (*i.e.*, no name in a member of C^1 appears in any member of C^2, and vice versa). Then

$$C_i^{12} = C_i^1 \cup C_i^2$$

We leave it as an exercise for the reader to verify that the congruence syndrome of C^{12} as defined above is indeed the glb of the congruence syndromes of C^1 and C^2. It is still open whether there is an efficient algorithm to compute such a collection *of minimal size*, *i.e.*, with as few names as possible. This would help to keep the dependence and demand sets small during partitioning.

6 Conditionals

Global analysis for a single-call-multiple-def interface is described here. In Algorithm 3, Step 3 is slightly modified. When propagating from the single call to the multiple defs, Steps 3a, 3b, and 3c are applied to each of the multiple defs. When propagating from the multiple defs to the single call, the condition in Step 2 must be true of all of the blocks $B_1, B_2, \ldots$ containing the defs, and the right hand side of the equation in Step 3c is taken as the union over the blocks $B_1, B_2, \ldots$. The propagation rule in Step 3a (*i.e.*, Algorithm 4) needs to combine the information from the def sites, and is given below. The def sites are indexed by k.

1. Let $R_i^k = Dem(\theta(Rcv_i^k))$, where this demand set is computed in the reduced graph of basic block B_k, but where the result *send* outlets are given new unique names, $Outlet(Send_j^k) = \{\sigma_j^k\}$, $1 \leq j \leq m$, for the purpose of computing these demand sets.

2. Let $S_j^k = Dep(\theta(Send_j^k))$, where this dependence set is computed in the reduced graph of B_k, but where the argument *receive* inlets are given new unique names, $Inlet(Rcv_i^k) = \{\rho_i^k\}$, $1 \leq i \leq n$, for the purpose of computing these dependence sets.

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

   (a) Add a squiggly arc $(Send_i, Rcv_j)$ in the caller.

   (b) Set $R_i^k \leftarrow R_i^k - \{\sigma_j^k\}$ for all k.

   (c) Set $S_j^k \leftarrow S_j^k - \{\rho_i^k\}$ for all k.

4. Set $Outlet(Send_i) \leftarrow \bigcup_k R_i^k$ in the caller for all $1 \leq i \leq n$ (alpha-renamed if necessary to avoid congruence with other call sites in the caller).

5. Set $Inlet(Rcv_j) \leftarrow \bigcup_k S_j^k$ in the caller for all $1 \leq j \leq m$ (alpha-renamed if necessary to avoid congruence with other call sites in the caller).

As discussed in the previous section, the unioning operation in the last two steps computes a collection whose congruence syndrome is the glb of the R^k (or S^k) collections' syndromes, assuming the collections are pairwise disjoint beforehand (alpha-renaming can be used to make sure this is so).

Global analysis for a multiple-call-single-def interface is just like what is described above, except that the roles of caller and callee are exchanged.

6.1 Example

Global analysis across a single-call-multiple-def is illustrated in Figure 5, which is an example of an if-then-else expression. The top row of the figure shows the caller (that is, the expression containing the if-then-else expression), the "then" side, and the "else" side. The initial annotation and partitioning in isolation is also depicted in the top row.

The second row shows the call site annotations after propagation from the two conditional sites. Following the first three steps of the propagation rule results in three squigglies to be inserted, because both conditional sites have a certain dependence from AR1 to RS1, from AR2 to RS2, and from AR3 to RS3. Just before Step 4 of the propagation rule, we have $R_1^1 = \{\}$, $R_2^1 = \{\}$, $R_3^1 = \{\sigma_1^1, \sigma_2^1\}$, $R_1^2 = \{\sigma_2^2, \sigma_3^2\}$, $R_2^2 = \{\}$, and $R_3^2 = \{\}$; and $S_1^1 = \{\rho_3^1\}$, $S_2^1 = \{\rho_3^1\}$, $S_3^1 = \{\}$, $S_1^2 = \{\}$, $S_2^2 = \{\rho_1^2\}$, and $S_3^2 = \{\rho_1^2\}$. Taking the pair-wise unions in Steps 4 and 5 yields the new annotations shown in the second row of the figure. (To make the figure easier to read, we have alpha-renamed σ_1^1, σ_2^1, σ_2^2, and σ_3^2 to u, v, y, and z, respectively, and ρ_3^1 and ρ_1^2 to h and i, respectively.) Partitioning can now group all nodes at the bottom of the caller into a single thread.

In the third row, we propagate from the caller to the "then" and "else" sites by applying Step 3 of Algorithm 3 to each side. Since all new outlet annotations are equal ($\{s, t\}$), demand set partitioning will place all nodes of the "then" site and all nodes of the "else" site into a single thread.

Reannotating the caller for the second time and applying demand set partitioning groups together all nodes at the top of the caller into a single thread, as shown in the fourth row of the figure. The final partitioning consists of four threads: the top and bottom of the caller, the "then" side, and the "else" side.

7 Summary

We have presented a new approach to partitioning non-strict programs into collections of sequential threads, extending previous work in the area in several ways. The framework is simpler and less *ad hoc*. Iterated basic block partitioning is defined using two simple constructs: dependence sets and demand sets. A key contribution is the use inlet and outlet annotations to provide a concise summary of the effects of dependences external to each basic block. In particular, they permit the expression of congruence among dependences, which can be described mathematically as the congruence syndrome of an annotation set. This provides a coherent framework for interprocedural analysis and analysis of conditionals, since the common structure of the conditional arms can be described by simple operations on the sets of annotations, which corresponds to finding a safe approximation to the two syndromes. Tracking dependences and sharing across procedures and through conditionals results in larger threads with less dynamic scheduling. It also identifies a larger number of redundant synchronizations, thereby reducing the cost of dynamic scheduling.

We do not yet have empirical measurements of the quality of partitioning using global analysis relative to the simpler algorithm in [SCvE91], but the improvement appears to be substantial. The most significant shortcoming of the earlier work was the weak handling of conditionals, which resulted in a large number of very small threads. In the present framework, partitioning is applied across the conditional and within both arms before the expansion of the conditional into primitive control transfer operations. In many cases, the need to steer multiple threads through the conditional is eliminated.

Partitioning as described here has similar aims to the use of strictness analysis to avoid building closures in implementations of lazy functional languages. [BHY88, Joh86] In fact, one could apply the partitioning approach directly to lazy languages [Tra89], the key difference being in what partitioning algorithms are permitted. Lazy evaluation imposes the additional restriction that no subexpression may be computed until it is known to contribute to the final answer. Dependence set partitioning is therefore ruled out, because in general it will form partitions which compute several outgoing values, not all of which may be required for the final answer even if some of them are. Demand set partitioning, on the other hand, corresponds exactly to what is required by lazy evaluation, as it seeks to group together subexpressions which contribute to the same set of outputs. Demand set partitioning combined with our global analysis may yield larger threads than conventional strictness-based approaches: for example, given a function

```
def f x y = cons (x+y) nil;
```

and a call `f` E_1 E_2, our analysis would allow the expressions E_1 and E_2 to be put in the same thread, even though `f` is not strict in either argument. On the other hand, our global analysis is weaker than strictness analysis in its ability to handle recursion. One area of future work is a better treatment of recursion in our global analysis.

There are several additional directions for future work. The partitioning and global analysis algorithms have been presented here in their most straightforward form. There remains a lot of room for clever data structures and algorithmic techniques to improve the time and space complexity. Related to this is the open question of how best to approximate the glb of the congruence syndromes in conditionals so to minimize the size of the annotations. Considerable work remains to develop effective strategies for partitioning and propagation throughout the collection of basic blocks. Finally, we have assumed that "bigger is better" in forming threads, but ultimately we expect there to be some tradeoffs between thread size, parallelism, communication, and synchronization overhead.

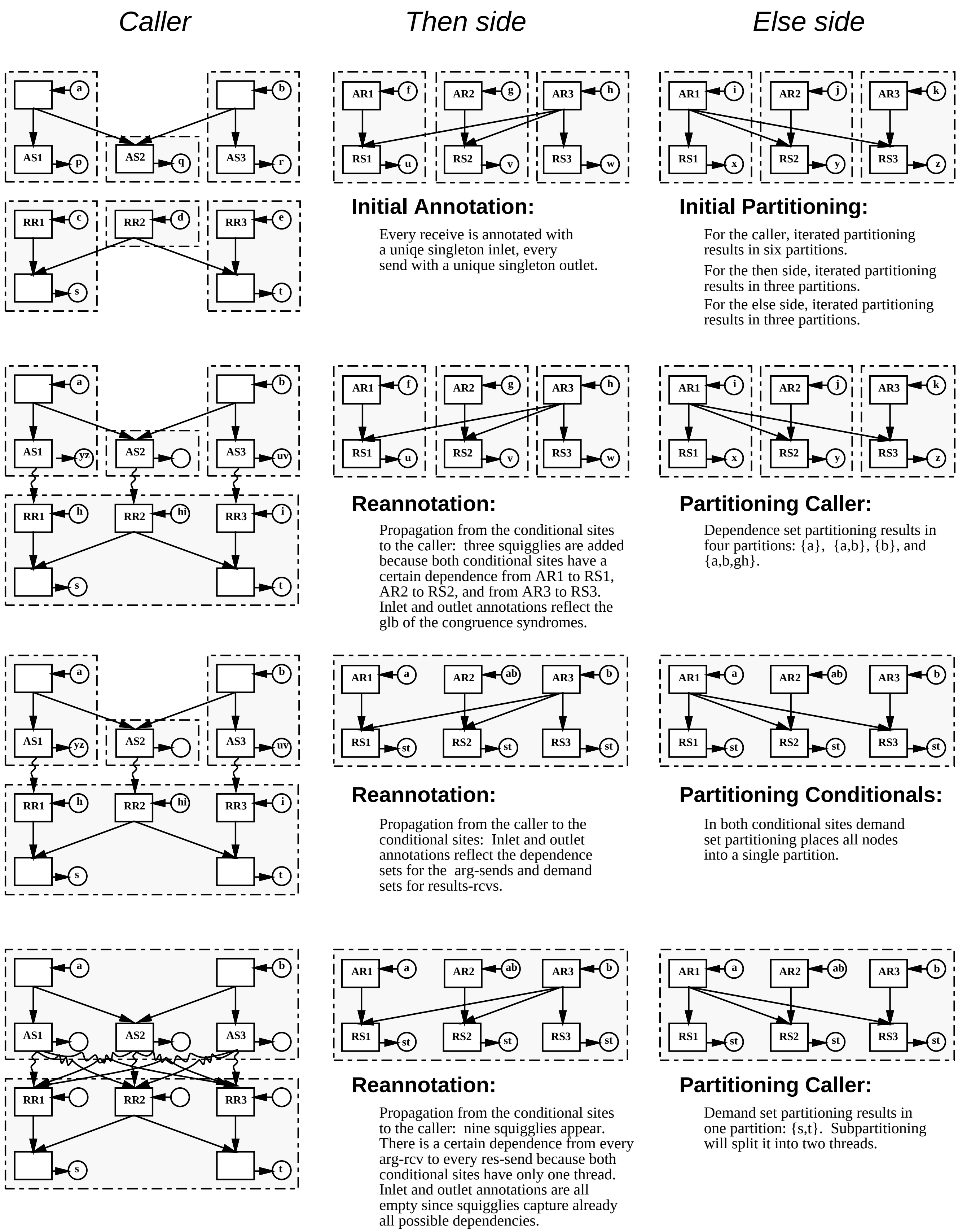


Figure 5: Example: Global Analysis of a Conditional

Acknowlegements

We thank Thorsten von Eicken, Seth Copen Goldstein, Venkat Natarajan, R. S. Nikhil, and Ellen Spertus for providing many useful comments on earlier drafts of the paper.

David Culler received support from a National Science Foundation PYI Award (CCR-9058342) with matching funds from Motorola Inc. and the TRW Foundation. Klaus Erik Schauser is supported by an IBM Graduate Fellowship.

References

[AA89] Z. Ariola and Arvind. P-TAC: A parallel intermediate language. In *FPCA '89*, pages 230–242. ACM, Sep 1989.

[AN87] Arvind and R. S. Nikhil. Executing a program on the Massachusetts Institute of Technology tagged-token dataflow architecture. In *PARLE: Parallel Architectures and Languages Europe Volume II*, volume 259 of *LNCS*, pages 1–29. Springer-Verlag, Jun 1987.

[ANP86] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. In *Graph Reduction*, volume 279 of *LNCS*, pages 336–369. Springer-Verlag, Oct 1986.

[BHA85] G. L. Burn, C. L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *Programs as Data Objects*, volume 217 of *LNCS*, pages 42–62. Springer-Verlag, Oct 1985.

[BHY88] A. Bloss, P. Hudak, and J. Young. Code optimizations for lazy evaluation. *Lisp and Symb. Comp.*, 1(2):147–164, Sep 1988.

[BNA91] P. S. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *FPCA '91*, volume 523 of *LNCS*, pages 538–568. Springer-Verlag, Aug 1991.

[BP89] M. Beck and K. Pingali. From control flow to dataflow. Technical Report TR 89-1050, Cornell U. Dept. of Comp. Sci., Ithaca NY, Oct 1989.

[CSS+91] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallellism with minimal hardware support: A compiler-controlled threaded abstract machine. In *4th ASPLOS*, pages 164–175. ACM, Apr 1991.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319–349, Jul 1987.

[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. 3rd Symp. on Par. and Dist. Processing*. IEEE, Dec 1991.

[HWe90] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University Department of Computer Science, New Haven CT, Apr 1990.

[Ian90] R. A. Iannucci. *Parallel Machines: Parallel Machine Languages*. Kluwer Academic Publishers, Boston, 1990.

[Joh86] T. Johnsson. Target code generation from G-machine code. In *Graph Reduction*, volume 279 of *LNCS*, pages 119–159. Springer-Verlag, Oct 1986.

[Myc80] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *International Symposium on Programming*, volume 83 of *LNCS*, pages 269–281. Springer-Verlag, Apr 1980.

[Nik90] R. S. Nikhil. Id version 90.0 reference manual. CSG Memo 284-1, MIT Lab. for Comp. Sci., Cambridge MA, Sep 1990.

[NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proc. 19th Ann. Int. Symp. on Comp. Arch.* IEEE, May 1992. (To appear).

[PC90] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token store architecture. In *Proc. 17th Ann. Int. Symp. on Comp. Arch.*, pages 82–91. IEEE, 1990.

[PT91] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proc. 18th Ann. Int. Symp. on Comp. Arch.*, pages 342–351. IEEE, May 1991.

[SCvE91] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-controlled multithreading for lenient parallel languages. In *FPCA '91*, volume 523 of *LNCS*, pages 50–72. Springer-Verlag, Aug 1991.

[Tra86] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical Report TR-370, MIT Lab. for Comp. Sci., Cambridge MA, Aug 1986.

[Tra89] K. R. Traub. Compilation as partitioning: A new approach to compiling non-strict functional languages. In *FPCA '89*, pages 75–88. ACM, Sep 1989.

[Tra91a] K. R. Traub. *Implementation of Non-Strict Functional Programming Languages*. Pitman Publishing, London, 1991. Also published by MIT Press, Cambridge MA.

[Tra91b] K. R. Traub. Multi-thread code generation for dataflow architectures from non-strict programs. In *FPCA '91*, volume 523 of *LNCS*, pages 73–101. Springer-Verlag, Aug 1991.