LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Partitioning Non-strict Functional Languages for Multi-threaded Code Generation†

CSG Memo 378
September 19, 1995

Satyan R. Coorg

To appear in the Proceedings of Static Analysis Symposium'95, September 25–27, Glasgow, Scotland, UK.

The research described in this paper was funded in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-1310.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Partitioning Non-strict Functional Languages for Multi-threaded Code Generation†

Satyan R. Coorg*
MIT Laboratory for Computer Science
NE43-217, 545 Technology Square
Cambridge, MA 02139, USA

Abstract

In this paper, we present a new approach to *partitioning*, the problem of generating sequential threads for programs written in a non-strict functional language. The goal of partitioning is to generate threads as large as possible, while retaining the non-strict semantics of the program. We define partitioning as a program transformation and design algorithms for *basic block partitioning* and *inter-procedural partitioning*. The inter-procedural algorithm presented here is more powerful than the ones previously known and is based on abstract interpretation, enabling the algorithm to handle recursion in a straightforward manner. We prove the correctness of these algorithms in a denotational semantic framework.

Keywords: Partitioning, abstract interpretation, demand and tolerance sets, inter-procedural analysis, non-strict functional languages.

1 Introduction

Functional programming languages can be divided into two classes: *strict* and *non-strict*. In a non-strict language, functions may return values before all their arguments are available, and data structures may be defined before all their components are defined. Many modern functional languages are non-strict; examples include Haskell [13] and Id [17]. Such languages give greater expressive power to the programmer than a strict language. Some of the languages provide non-strictness because of its cleaner semantics. Others, like Id, also use non-strictness to generate parallelism in a program.

Compiling non-strict languages to conventional microprocessors has been the focus of many recent papers [26, 21, 20]. Some features in the design of commercial microprocessors make it difficult to execute non-strict programs directly. These processors are highly tuned to execute a sequence of instructions efficiently by providing features like a large register set and deep pipelining. Unfortunately, this also implies that there is a high penalty for dynamic scheduling of instructions. As serial execution is interrupted, registers have to be saved and restored, there may be bubbles in the instruction pipeline etc. Synchronization is also expensive in these processors since it has to be explicitly performed (in software) using instructions.

For programs in a non-strict language, it is not always possible at compile-time to specify an exact ordering of instructions. This is illustrated in the following example written in Id.

Example 1:

```
def f a b =
 {x = a + 1;
  y = b * 2
 in
   (x,y)};

{...                   {...
 (c,d) = f 2 c          (c,d) = f d 1
...}                   ...}
```

The function `f` takes two arguments `a`,`b` and returns two results `x`,`y`. The set of statements enclosed by {...} form a recursive *let* block definition. There is no syntactic ordering of the statements of the block. Also shown are calls to the function `f` in some other parts of the program (these are referred to as *contexts* of `f`). In both the contexts, a result of `f` is fed back into one of the function's arguments.

Consider the execution of the first context in a language with strict semantics. The function `f` is not called until both of its arguments are available. In this case, it is not going to be called until the variable `c` gets a value. As the value of `c` depends on the call itself, this leads to a *deadlock* in the program; the variables `c` and `d` never get defined.

In a non-strict execution, it is possible to call a function even though not all of its arguments are available. Also, it is possible to return some results even though not all of the function's results are available. So, the function `f` is called with the first argument 2 and the second argument $\perp$ (an *undefined* value). Then, the value of `x` can be computed to be 3. Now, the variable `c` gets defined to be the value 3, and the computation of `y` and `d` can proceed. Hence, the non-strict execution produces meaningful values for the variables `c` and `d`. Note that in this execution, the + instruction *must* have executed before the $*$ instruction – there is no other order that could produce the correct results for `f`.

Similarly, the second context produces a deadlock in a strict execution, but produces valid results in a non-strict execution. However, this context requires that the $*$ instruction execute before the +. Thus, depending upon the context, the instructions in `f` can execute in either order. We have no choice but to compile these programs such

†The research described in this paper was funded in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-1310.
*Email:satyan@lcs.mit.edu, Phone: 617-253-8858, Fax: 617-253-6652.

that the two instructions execute independently, when their data becomes available. Compiling `f` in this manner entails performing dynamic scheduling/synchronization at each instruction – leading to an inefficient execution on microprocessors.

Traub [25] made the significant observation that dependence analysis could enable efficient compilation of some functions. A simple example of such a function is given below.
Example 2:

```
def f a b =
 {x = a + b;
  y = a * b
 in
  (x,y)}
```

In this function, we can infer that the dependence behavior of both the $*$ and $+$ instruction is very similar: they both depend on the values of `a` and `b`. For this function, any context that feeds results back to arguments always deadlocks, even in a non-strict execution. Thus, it is *safe* to compile the function such that the two instruction execute together, in some order.

Thus, the goal *partitioning* [25] is to infer the possible dependencies in a program and use that information to *sequentialize* parts of a program. This is achieved by compiling each function into a set of *threads* while preserving the function's non-strict semantics. Each thread is a sequence of instructions and once a thread begins execution, it does not need any dynamic scheduling/synchronization during its execution. Threads of this form yield efficient implementation on conventional architectures using an abstract machine such as TAM [9]. They can also be implemented on parallel multi-threaded hardware (*e.g.*, *T [18]).

Combining instructions into threads enables more efficient use of registers and amortizes the overheard of dynamic scheduling/synchronization over many instructions. Partitioning attempts to make threads as large as possible, to enable efficient execution of non-strict programs on microprocessors.

The strategy used to implement non-strictness (that is, functions accepting *undefined* arguments and returning *partial* results) is called the *lenient* strategy [25]. This strategy gives flexibility to the programmer in using the recursive *let* block. It enables circular dependencies to be resolved at runtime, instead of being statically resolved. Non-strictness in data-structures also gives the programmer ability to create circular data-structures functionally [25].

Another strategy that provides non-strictness is the *lazy evaluation* strategy, where the evaluation of an expression is delayed until the execution cannot proceed without the value of that expression. Unlike lenient evaluation, lazy evaluation also gives the user the ability to manipulate infinite data structures [11]. In this paper, we are primarily interested in implementing a non-strict language with the lenient strategy (though we do remark on the applicability of the techniques developed here to implement lazy evaluation).

The contributions of this paper are:

- The definition of partitioning as a program transformation.
- Design of new basic block partitioning algorithms. Also, a precise characterization of the complexity of the partitioning problem.
- A new inter-procedural partitioning algorithm based on abstract interpretation.
- The proof of correctness of the partitioning algorithms using denotational semantics.

The rest of the paper is organized as follows. Section 2 defines a simple non-strict kernel language and its semantics. Section 3 formally defines the partitioning problem and the notion of correctness of a partitioning. Section 4 deals with the problem of partitioning basic blocks. Section 5 uses abstract interpretation techniques to extend the basic block partitioning algorithm to partition arbitrary blocks. Section 6 compares this paper with related work and section 7 concludes. In the appendix, we present proof sketches of some of the theorems stated in this paper.

2 Notation, Syntax and Semantics

We design our algorithms on a simple *kernel* language, SP-TAC (Simple P-TAC) language based on the intermediate language P-TAC (Parallel Three-Address Code) [2] that is used during the compilation of Id. SP-TAC is a simple first order functional language, with constants, variables, primitive operators, conditionals, user defined functions and recursive *let*-blocks. It is a restricted form of P-TAC in several ways: there are no higher order functions, structured data types (like lists), or side-effects. Though SP-TAC is a very restrictive language to write programs in, it is a non-strict language and it brings out the issues in compiling non-strictness. Also, techniques developed for partitioning SP-TAC are directly applicable to more expressive languages.

2.1 Abstract Syntax of SP-TAC

c	$\in$	Constants
x, y	$\in$	Variables
p	$\in$	Primitive operators
f	$\in$	User defined functions
se	$\in$	Simple expressions
e	$\in$	Expressions
st	$\in$	Statements
b	$\in$	Blocks
pr	$\in$	Programs

$$
\begin{array}{lcl}
c & ::= & \textbf{true} \mid \textbf{false} \mid \textbf{1} \mid \textbf{2} \mid \ldots \\
p & ::= & + \mid - \mid \textbf{and} \mid \ldots \\
se & ::= & c \mid x \\
e & ::= & se \mid b \mid p(se_1, se_2, \ldots, se_n) \mid \\
 & & f(se_1, se_2, \ldots, se_n) \mid \textbf{if}\ se\ \textbf{then}\ b_1\ \textbf{else}\ b_2 \\
st & ::= & (y_1, y_2, \ldots, y_n) = e \\
b & ::= & \{\{st\}^*\ \textbf{in}\ (se_1, se_2, \ldots, se_n)\} \\
pr & ::= & \{f_i(x_1, x_2, \ldots, x_n) = b_i\}
\end{array}
$$

Figure 1: The Syntax of a Kernel Language

Figure 1 shows the abstract syntax of SP-TAC. We make the following assumptions about the language.

- There is no currying of user defined functions. This ensures that we are dealing with a first order language.
- All primitive operators are strict. That is, they require all their inputs to produce their output value.

- All local and bound variables are α-renamed so that they are defined uniquely in the program.

For notational convenience, examples in this paper are given in Id. It is straightforward to translate them into programs in the kernel language [3].

2.2 Representing SP-TAC using Graphs

In this paper, we also use a *graph* representation of a block of SP-TAC. This representation (called a *dataflow graph*) is similar to the one used in the Id compiler to perform various optimizations [23]. The language syntax shown in the previous section is essentially a textual representation of these dataflow graphs.

Definition 1 (Dataflow Graph) *Given a block b, the dataflow graph $G(b)$ corresponding to b is a directed graph. The nodes $N(b)$ in the graph consist of* input nodes *– corresponding to the free variables of the block,* output nodes *– corresponding to the values returned by the block, and* operator nodes *– corresponding to the various operators specified in the block. The edges $E(b)$ of this graph correspond to the flow of values in the block.*

In this paper, we treat the graph representation and the textual representation as equivalent. For example, when we refer to *nodes* in a block, we actually refer to the nodes in the dataflow graph corresponding to the block. Figure 2 shows the dataflow graph corresponding to the functions defined in examples 1 and 2 in section 1.

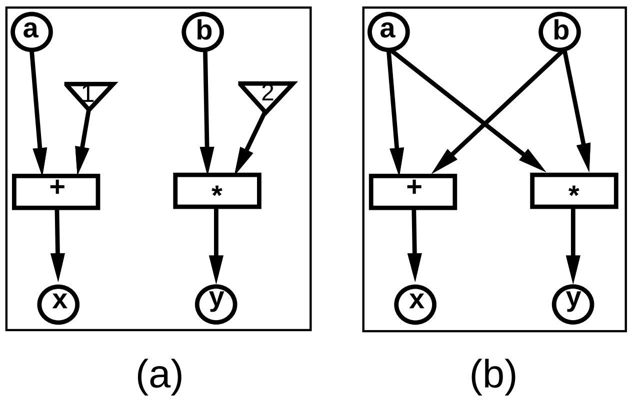

Figure 2: Figures (a) and (b) show examples of dataflow graphs. Input and output nodes are denoted by circles and constant nodes by triangles. Rectangular nodes correspond to the various operators in the function.

2.3 Semantics

Our framework is based on denotational semantics. We chose to do this because techniques for designing and proving analyses based on abstract interpretation are well developed in this framework and these techniques handle recursive function definitions in a natural manner. The semantics of SP-TAC itself is straightforward as it is a simple first order language. It is based on the semantics given in [4] and specifies a non-strict semantics of the language.

$$
\begin{aligned}
&\mathcal{E}_c[\![n]\!] &&= n, \text{integer } n\\
&\mathcal{E}_c[\![true]\!] &&= true\\
&\mathcal{E}_c[\![false]\!] &&= false\\[1ex]
&\mathcal{E}_k[\![+]\!] &&= \lambda(x,y).(Int?(x) \; and \; Int?(y))\\
& && \quad \rightarrow x+y, error\\[1ex]
&\mathcal{E}_s[\![c]\!]\, bve &&= \mathcal{E}_c[\![c]\!]\\
&\mathcal{E}_s[\![x]\!]\, bve &&= bve[\![x]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}[\![se]\!]\, bve\; fenv \;=\; \mathcal{E}_s[\![se]\!]\, bve\\
&\mathcal{E}[\![p(se_1, se_2, \ldots, se_n)]\!]\, bve\; fenv =\\
&\quad \mathcal{E}_k[\![p]\!]\,(\mathcal{E}_s[\![se_1]\!]\, bve, \ldots, \mathcal{E}_s[\![se_n]\!]\, bve)\\
&\mathcal{E}[\![f(se_1, se_2, \ldots, se_n)]\!]\, bve\; fenv =\\
&\quad fenv[\![f]\!]\,(\mathcal{E}_s[\![se_1]\!]\, bve, \ldots, \mathcal{E}_s[\![se_n]\!]\, bve)\\
&\mathcal{E}[\![\mathbf{if}\; se\; \mathbf{then}\; b_1 \mathbf{else}\; b_2]\!]\, bve\; fenv =\\
&\quad (\mathcal{E}_s[\![se]\!]\, bve) \rightarrow (\mathcal{E}_b[\![b_1]\!]\, bve\; fenv), (\mathcal{E}_b[\![b_2]\!]\, bve\; fenv)\\
&\mathcal{E}[\![b]\!]\, bve\; fenv \;=\; \mathcal{E}_b[\![b]\!]\, bve\; fenv
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}_b[\![\{(y_{i1}, \ldots, y_{im}) = e_i\}^* \; \mathbf{in}\; (se_1, \ldots, se_n)\}]\!]\, bve\; fenv =\\
&\quad \mathbf{letrec}\\
&\quad newbve = [y_{ij} \mapsto (\mathcal{E}[\![e_i]\!]\,(newbve.bve)\; fenv) \downarrow j]\\
&\quad \mathbf{in}\; (\mathcal{E}_s[\![se_1]\!]\, newbve, \ldots, \mathcal{E}_s[\![se_n]\!]\, newbve)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}_p[\![\{f_i(x_1, x_2, \ldots, x_n) = b_i\}^*]\!] = fenv, \mathbf{whererec}\\
&\quad fenv = [f_i \mapsto (\lambda(y_1, y_2, \ldots, y_n).\mathcal{E}_b[\![b_i]\!]\,[x_i \mapsto y_i]\; fenv)]
\end{aligned}
$$

Figure 3: Semantic Equations of SP-TAC.

Denotational semantics [22] is given by defining domains (complete partially ordered sets) and constructing a mapping from entities in the language to elements of these domains. For SP-TAC, we need to define the following domains. We have the standard domains Int and $Bool$ for integers and booleans. These domains contain an element $\perp$ that stands for the "undefined" value. To model multiple values, we use *product* domains [22]. We use D^k to denote a domain formed by the k-product $D \times D \times \ldots \times D$. The infix operator $\downarrow$ is used to select any component of a value in D^k.

In addition to these domains, we have domains of functions that map (multiple) values to (multiple) values. There are also domains corresponding to variable environments and function environments. These domains map variables to values and function names to functions respectively. The formal definition of the domains we use in our semantics are:

Int	Integers
$Bool$	Booleans
$Val = Int + Bool$	Values
$Fun = \bigcup_{j,k \geq 1} (Val^j \rightarrow Val^k)$	First order functions
$D = Val + \{error\}$	Denotable values
$D_T = \bigcup_{k \geq 1} Val^k + \{error\}$	Tuples of values
$Bve = V \rightarrow D$	Variable Environments
$Fenv = Fv \rightarrow Fun$	Function Environments

The various semantic functions are given below:

$\mathcal{E}_c : Con \rightarrow Val$	Constants
$\mathcal{E}_k : Pf \rightarrow Fun$	Primitive operators
$\mathcal{E}_s : SE \rightarrow Bve \rightarrow D$	Simple Exps
$\mathcal{E}_b : Bo \rightarrow Bve \rightarrow Fenv \rightarrow D_T$	Blocks
$\mathcal{E} : Exp \rightarrow Bve \rightarrow Fenv \rightarrow D_T$	Expressions
$\mathcal{E}_p : Prog \rightarrow Fenv$	Programs

Now, we describe semantic equations that define the behavior of these semantic functions. Here is some notation used in describing our semantic equations:

- If env and env' are two environments, we use $env'.env$ to denote the environment env'' such that $env''[\![y]\!] = env'[\![y]\!]$ if $y \in Domain(env')$, $env''[\![y]\!] = env[\![y]\!]$, otherwise. This is the way environments are linked in a statically scoped block structured language.
- We also make use of a case operator (denoted by $\rightarrow$). The meaning of $x \rightarrow y, z$ is that depending on whether x is $true$ or $false$ either y or z is the value of the entire case expression.

The semantic equations are given in figure 3. For clarity of the equations, we have omitted the projections and injections between the various domains. Given a program pr, we use $\mathcal{E}^{pr}[\![b]\!]\, bve$ to denote $\mathcal{E}[\![b]\!]\, bve\, (\mathcal{E}_p[\![pr]\!])$.

3 Definition of Partitioning

In section 1, we informally defined the partitioning problem as compiling a non-strict program into sequential threads. In this section, we formalize that intuition, by defining partitioning as a transformation from a program in SP-TAC to another program in SP-TAC.

Definition 2 (Partitioning) *A partitioning of a block b is a set of partitions $Q = \{q_1, \ldots, q_k\}$. Each partition q_j is a set of primitive operator nodes $n_1, \ldots, n_r$ of b.*

Note that the execution of a thread is very similar to the execution of a primitive operator. The thread waits for all its inputs to become available, performs some computation and produces its outputs when all the computation in the thread has terminated. Thus, executing a partitioned block is similar to executing a block where the primitive nodes in a partition are merged into a single node. Such a graph is called the *reduced graph* corresponding to a partitioning. Figure 4 shows a dataflow graph and its reduced graph corresponding to a partitioning.

How can we capture the semantics of merging primitive operators into a single node? We use a new primitive operator (called *synchronization* or S operator) to achieve this. There is actually a *family* of such operators; each operator in the family differs in the number of operands it accepts.

Definition 3 (Synchronization Operator) *A synchronization operator S_k is a primitive operator with the following behavior. For each $k \geq 1$,*

$$\begin{aligned} S_k(v_1, \ldots, \perp, \ldots, v_k) &= (\perp, \ldots, \perp) \\ S_k(v_1, \ldots, v_k) &= (v_1, \ldots, v_k) \end{aligned}$$

Intuitively, the operator "waits" for all its operands to become values. Once this happens, the S operator "fires" and acts as an identity operator on each of its operands.

Using this operator, it is possible to express the behavior of executing a partitioned block. We achieve this by defining a *transformed block* that models the behavior of a partitioned block. The formal definition is given below.

Definition 4 (Partitioning Transformation) *Given a block b and a partitioning Q, the partitioned block b_Q is defined as follows. For each partition $q \in Q$ define:*

1. $I_q = \{n \mid (n \notin q) \wedge (m \in q) \wedge\ edge\,(n, m) \in b\}$
2. $O_q = \{m \mid (n \notin q) \wedge (m \in q) \wedge\ edge\,(m, n) \in b\}$
3. *Add two synchronization operators $S_{|I_q|}$ and $S_{|O_q|}$ with inputs I_q and O_q respectively. Connect outputs of $S_{|I_q|}$ to nodes in q that were connected to I_q, and outputs of $S_{|O_q|}$ to nodes not in q that were connected to O_q.*

The idea behind the transformation is to use two synchronization operators to model the execution of a thread. One "waits" for all the inputs of a thread to become available. The other makes sure that all the computation in a thread is fully performed before any other thread can use values computed by this thread. This transformation ensures that the set of primitive operators in a partition behave as though they are a single primitive operator. A compile-time ordering can now be generated for each partition by a topological sort of the operators in it. Our algorithms only produce the partitions for a block – they do not produce the actual threads. Figure 4 also shows a transformed dataflow graph using S operators.

3.1 Correctness of Partitioning

When can we say that a partitioning of a block is *correct*? Informally, we would like partitioning to behave as any other compiler optimization. That is, it should not change the "meaning" of the input program. As we are partitioning at the level of blocks, we say that a partitioning is correct if the meaning of the block (given by its denotational semantics) is preserved. Formally,

Definition 5 (Correctness of Partitioning) *A partitioning Q of a block b in a program pr is correct iff the partitioned block b_Q satisfies the following property. For all bve:*

$$\mathcal{E}^{pr}[\![b_Q]\!]\, bve = \mathcal{E}^{pr}[\![b]\!]\, bve$$

This definition guarantees that we can use the partitioned block anywhere in the program where the original block was used. However, this definition is not directly verifiable at compile-time, it requires knowledge of run-time values of the variables in a block. Later, we will show a simpler syntactic criterion that guarantees the semantic correctness of partitioning, and make use of the simpler criterion to design our partitioning algorithms.

How can the partitioning of a block be incorrect? By the above definition, a partitioning is incorrect if it returns values that are different from those returned by the original block. Can the partitioning change the behavior of the block such that it returns a different fully defined value (say 4) instead of the value 3? Intuitively, this is not possible, as partitioning introduces only additional synchronization constraints – it does not change the value semantics of the block. The only change that could happen is that a block, instead of computing a value (say 3) on some set of inputs now computes $\perp$ (*i.e.*, deadlocks). It is possible to formalize this into the following theorem.

Theorem 6 (Effect of Partitioning) *Let b be a block in a program pr and b_Q its partitioned version of a partitioning Q. Then, for all bve:*

$$\mathcal{E}^{pr}[\![b_Q]\!]\, bve \sqsubseteq \mathcal{E}^{pr}[\![b]\!]\, bve$$

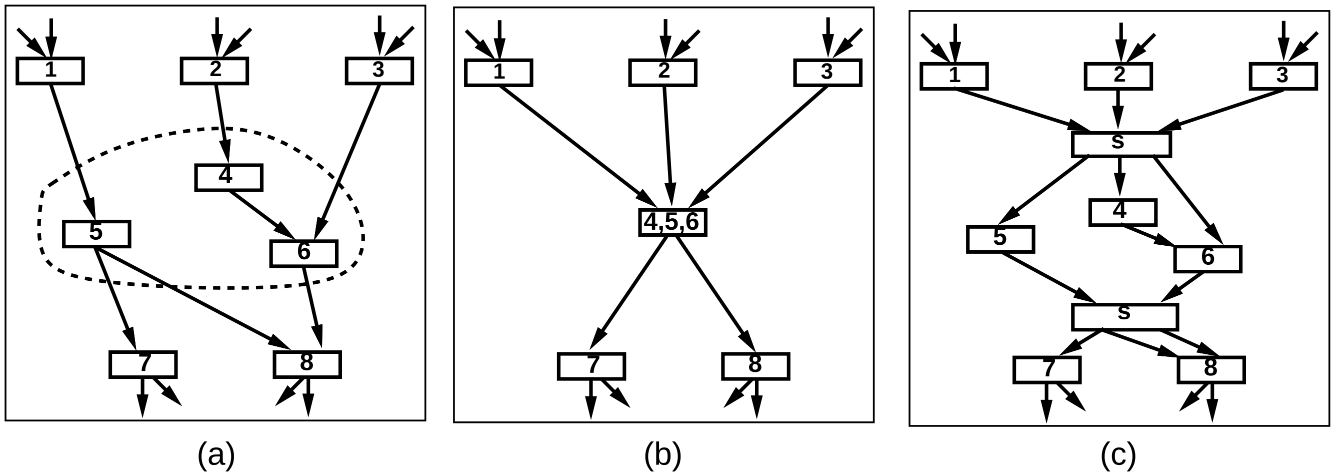

Figure 4: Figure (a) shows a fragment of a dataflow graph with a partition {4,5,6}. Figure (b) shows the reduced graph obtained by coalescing the nodes in a partition to a single node. Figure (c) shows the transformed graph corresponding to the partition. Two S operators have been introduced to model the behavior of a thread.

The proof of this theorem involves establishing a correspondence between the variables in b with the variables in b_Q and showing by fix-point induction that the value of a variable in b_Q is $\sqsubseteq$ the value of the corresponding variable in b.

In designing our algorithms in later sections, we assume that there is no cycle consisting of primitive operators in our input blocks. If such a cycle exists, it is easy to show that all the nodes in the cycle will have undefined values, and hence these nodes do not affect the correctness of a partitioning.

4 Basic Block Partitioning

In this section, we consider the problem of partitioning *basic blocks*, that is, blocks whose operator nodes are all primitive operators[1] – they do not contain conditionals and calls to user-defined functions. It is useful to study this restricted version of the problem, as the techniques developed to partition basic blocks can be extended to partition more general blocks. Note that we consider only acyclic basic blocks, as we have made the assumption that there are no cycles involving primitive nodes in our inputs.

4.1 Correctness of Basic Block Partitioning

In section 3, we have defined the general notion of correctness of a partitioning. For basic blocks, it is possible to simplify the correctness criterion to simple conditions on the structure of the partitioned basic block.

Note that partitioning does not distinguish one primitive operator (say $+$) from another (say $*$) – correctness of a partitioning depends on whether a value is computed at all, and not whether a different value is computed by a block. In some sense, we can consider all primitive operators to be "equivalent". As a basic block consists only of primitive operators, two basic blocks differ only in their *structure*, *i.e.*, the way the operators are connected together. A partitioning is correct if it preserves the structure of a basic block, as seen by a context. Clearly, it should not introduce any cycles in the partitioned block – that would destroy the acyclicity property and cause a deadlock in the block. Also, a context "looks at" only the inputs and outputs of a basic block – thus input-output connectivity should be maintained. It turns out that these conditions are sufficient to ensure the correctness of a partitioning of a basic block. These conditions are formalized in the theorem below (see the appendix for the proof).

[1] Nodes corresponding to constants and their edges can be deleted from a basic block – they do not affect partitioning.

Theorem 7 (Basic Block Correctness) *Let b be a basic block and b_Q its partitioned version. Q is a correct partitioning of b iff acyclicity and input-output connectivity are preserved. That is,*

1. *b_Q is acyclic.*
2. *Let i be an input of b and o an output of b. Then, i is connected to o in b iff it is connected to o in b_Q.*

Using this theorem, we can now design algorithms to partition basic blocks. It is sufficient to prove that our algorithms preserve the two conditions of the above theorem; that is enough to guarantee the correctness of our partitioning algorithms.

4.2 Demand and Tolerance Sets

In this section, we describe the properties of *demand* and *tolerance* sets that can be computed for each node in a basic block. These sets are useful in designing a partitioning algorithm as the correctness property can be formulated in terms of these sets. Moreover, they can be efficiently computed for any basic block.

Definition 8 (Demand Sets) *In a block b, the* demand *set of a node is the set of outputs that depend on that node. Demand sets can be computed by the following (backward) propagation algorithm on basic block.*

$$\begin{aligned} Demand(o) &= \{o\}, \textit{if } o \textit{ is an output node} \\ Demand(n) &= \bigcup_{(n,m)\in b} Demand(m), \textit{otherwise} \end{aligned}$$

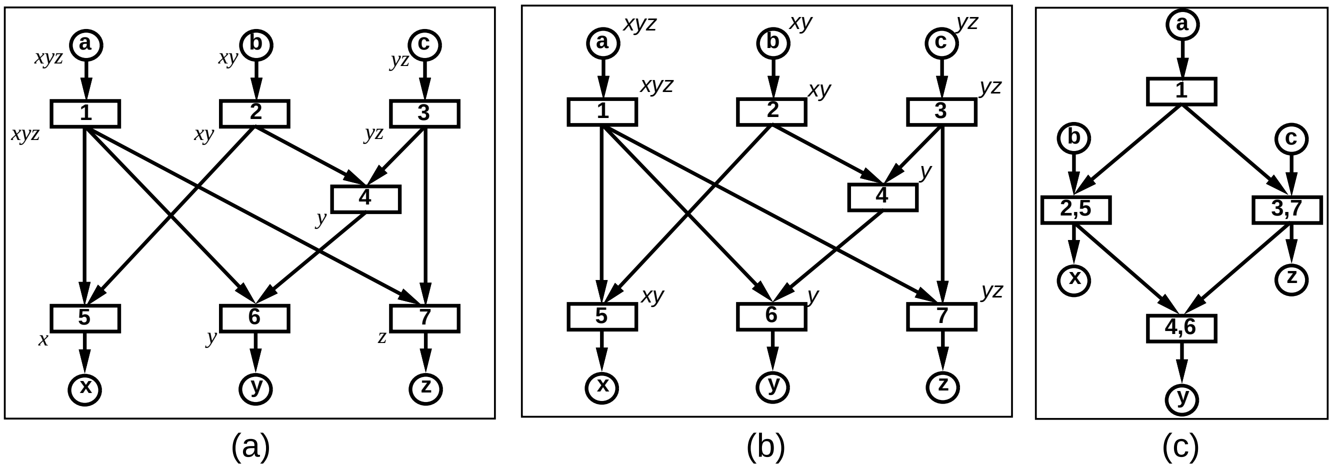

Figure 5: Figure (a) shows the demand sets of each node of the given basic block. Figure (b) shows its tolerance sets. Figure (c) shows the partitions and reduced graph obtained by both algorithm 1 and algorithm 2. The partitions are optimal for this example.

Demand sets for a basic block are show in figure 5.

Definition 9 (Tolerance Sets) *In a block b, an output o* tolerates *a node n iff edge (n, o) can be added to b without affecting input-output connectivity. The* tolerance set *of a node n is the set of outputs that tolerate n. Tolerance sets can be computed by the following (forward) propagation algorithm on basic block graphs.*

$$\begin{aligned} Tolerance(i) &= Demand(i), \textit{if } i \textit{ is an input node} \\ Tolerance(n) &= \bigcap_{(m,n)\in b} Tolerance(m), \textit{otherwise} \end{aligned}$$

Clearly, the tolerance set of an input node contains at least its demand set; adding edges from an input node to any node in its demand set will not affect input-output connectivity. Also, no other output can be in the tolerance set of an input – that would add a connection where none existed. Thus, the tolerance set of an input node is exactly its demand set. Similarly, it can be argued that the a node o tolerates n iff o tolerates all of n's parents. Thus, the tolerance set of a node is the intersection of the tolerance sets of its parents. Tolerance sets for a basic block are shown in figure 5.

Using the definitions of demand and tolerance sets, we can design a simple algorithm that computes the demand sets in a backward pass and then performs a forward pass to compute the tolerance sets. The algorithm runs in $O(N^2)$ time and space, where N is the number of nodes in the basic block.

4.3 Algorithms for Basic Block Partitioning

In this section, we design two algorithms for partitioning based on demand and tolerance sets. The basic idea behind the algorithms is to preserve input-output connectivity of the basic block, thus ensuring correctness.

What effect does merging two nodes (say n and m) have on the connectivity of the basic block? For input-output connectivity, this is equivalent to adding edges in the block from node m to each output connected to n and vice-versa. By definition of tolerance, this preserves the connectivity if each output connected to n tolerates node m, that is, $Demand(n) \subseteq Tolerance(m)$ and vice-versa. Algorithm 1 (shown in figure 6) uses this condition to identify nodes that can be merged and partitions the basic block.

Algorithm 1
Given a basic block b:

1. Compute demand and tolerance sets for each node of b.
2. Compute an (undirected) graph G_{merge} as follows:
   (a) The nodes of G_{merge} are the nodes of b.
   (b) An edge (n, m) is in G_{merge} iff $Demand(n) \subseteq Tolerance(m)$ and $Demand(m) \subseteq Tolerance(n)$.
3. Do
   (a) Let C be a maximal clique[2] of G_{merge}. Merge nodes in C into a single partition.
   (b) Form the reduced basic block. Recompute demand sets, tolerance sets, and G_{merge} for the new basic block.

   until the partitions do not change.

Figure 6: Algorithm 1 for Basic Block Partitioning

The idea in algorithm 1 is to construct an auxiliary (undirected) graph, with the nodes corresponding to the operators in the block. An edge between two nodes indicates that the two nodes can be merged (using the condition given above). A clique in the graph identifies a set of nodes such that a node in the clique can be merged with any other node – thus, yielding a partition of the block. This process is

[2] A *clique* of a graph is a set of nodes that are all adjacent to each other. A maximal clique is one to which no other node can be added without destroying the clique property.

repeated on the reduced basic block until no more partitions can be formed. It is clear that algorithm 1 preserves input-output connectivity. It also preserves acyclicity, see the proof sketch of theorem 13 in the appendix for details. Figure 5 also shows the partitions obtained by algorithm 1 on a basic block.

Algorithm 1 generates the largest possible threads if instead of merging maximal cliques, it merged *maximum* cliques. However, it is well known that finding the maximum clique in a general graph is NP-hard [10]. The best known algorithm for this problem takes exponential time in the size of the input graph. Do the graphs we construct (G_{merge}) have any special structure that can be exploited to give a more efficient algorithm? Unfortunately, the answer is no. It is possible to show that finding an optimal partitioning (either minimum number of partitions or largest size partitions) is NP-hard [8].

Assuming that we are content with maximal cliques, we can estimate the complexity of the algorithm as follows. The complexity is dominated by the loop in step 3, which merges partitions at each step. The number of such steps is $O(N)$. In each iteration, we compute the maximal clique and recompute G_{merge}. For finding a maximal clique, a simple greedy algorithm can be designed which runs in $O(N^2)$ time. Also, a clever recomputation of the new graph G_{merge} that makes use of its older version takes $O(N^2)$ time. Thus, the overall time complexity is $O(N^3)$.

It is possible to design a simpler algorithm for basic block partitioning. This algorithm detects only a subset of possible merges in the block and merges them into a single partition. Thus, it may not be able to detect all the partitions that could be formed. However, it does not require the construction of the auxiliary graph (G_{merge}) and can be implemented more efficiently. Algorithm 2 (shown in figure 7) is based on the idea that merging nodes with the same demand and tolerance sets produces correct partitions.

Algorithm 2
Given a basic block b:

1. Compute demand and tolerance sets for each node of b.
2. Do
   (a) Merge nodes with the same demand set into a single partition. Recompute tolerance sets from reduced graph.
   (b) Merge nodes with the same tolerance set into a single partition. Recompute demand sets from reduced graph.

   until the partitions do not change.

Figure 7: Algorithm 2 for Basic Block Partitioning

Clearly, merging nodes with the same demand set preserves input-output connectivity; merging nodes m and n has the effect of introducing edges from m to the demand set of n, and this will have no effect on input-output connectivity as every node in the demand set of n is already connected to m. Similarly, using the properties of tolerance sets, it can be argued that merging nodes with the same tolerance sets also preserves input-output connectivity.

The worst-case complexity of algorithm 2 is the same as that of algorithm 1 ($O(N^3)$). However, it has been observed in practice [20] that a similar algorithm converges very fast – the loop in step 2 executes only a few (< 4) iterations. Thus, it may have a practical advantage over algorithm 1. For the basic block in figure 5, algorithm 2 also generates the same partitions generated by algorithm 1. An implementation may use a mix of the two algorithms present in this section to derive large partitions while keeping the complexity of the algorithm low. One simple strategy is to run algorithm 2 first and then run algorithm 1 on the reduced graph.

5 Global Partitioning

In the previous section, we have seen how to partition basic blocks. Now, we address the problem of partitioning blocks that contain conditional expressions and calls to other user defined functions. We use the term *general* blocks to refer to such blocks and *global partitioning* to refer to partitioning such blocks. Although our discussion in this section is focussed primarily on dealing with calls to user defined functions, the same techniques can be used to handle conditionals by abstracting them into a separate function.

5.1 Converting General to Basic Blocks

Our strategy in performing global partitioning is to reduce a general block to a set of basic blocks by using dependence information of functions called in the general block. In this section, we present our approach using an example. We formalize these techniques in sections 5.2 and 5.3.

Consider the simple block shown in figure 8(a). The block has two inputs, two outputs and calls a function `f`. Suppose the function `f` is not known, but we would like to partition the block anyway. We can convert the block into a basic block by assuming "worst case" or "unknown" behavior of the function `f`. As shown in figure 8(b), we *disconnect* the node calling `f`. Then, we create new outputs (z, w) and a new input (c) and use these nodes instead of a call to the function f. Partitioning this basic block merges nodes {3,4} into a single partition.

Why does this process yield correct partitions? Note that our basic block algorithm is correct, that is, the behavior of the partitioned block is identical to the behavior of the original block under any context, regardless of the dependencies between the inputs and outputs of the block. In particular, the algorithm will generate safe partitions even when the dependencies between outputs z, w and input c correspond to the actual dependencies of function `f`.

Now, suppose that the function `f` being called in figure 8 is the one given below:

```
def f u v = u + v;
```

It is obvious that `f` acts exactly like the primitive operator $+$, so the block can be converted to the basic block in figure 8(c) (node 5 is shown shaded as it is not an actual primitive operator but a "dummy" one). We can then partition the basic block yielding the partitions {1,2} and {3,4}[3]. Thus, we have been able to generate better partitions than that was possible by disconnecting the call to `f`. It is also easy to observe that the same basic block is also sufficient to model the following function:

[3] In an ordinary basic block, all five nodes would be in a single partition. As node 5 is a dummy node, we cannot include it in any partition.

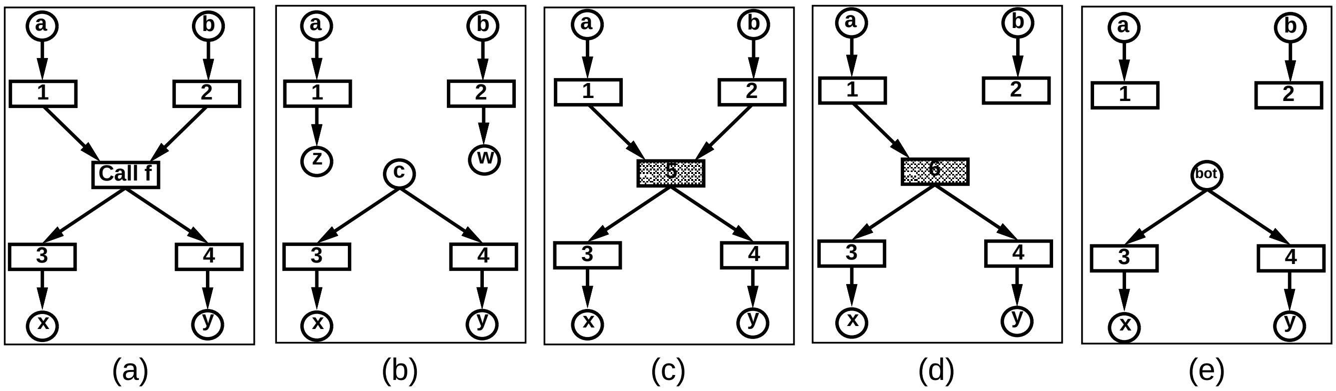

Figure 8: Figure (a) shows a simple block that calls a function `f`. Figure (b) shows the basic block obtained by disconnecting the call to `f`. Figure (c) corresponds to the path $[u, v]$ of `f`. Figure (d) corresponds to the path $[u]$ and figure (e) corresponds to the path $\perp_p$.

```
def f u v =
 {t1 = u * 2;
  t2 = v + 3;
  t3 = t1 - t2
 in
  t3};
```

In this case too, `f` acts like a primitive operator on its two arguments – the result depends on both the arguments, and the result is *always* available if both the arguments are supplied. We say that `f` has the *path* $[u, v]$ to denote this property.

It may not be possible to specify a single path for every function. A simple example of such a function is given below:

```
def f u v = if u then v else 3;
```

Here, we do not know (at compile-time) whether `f` depends on its second argument – that depends on the value of `u`. We do know that it can only be one of two possibilities: the result depends on both the arguments if the *then* branch is selected or just the first argument if the *else* branch is selected. We allow both these possibilities to be represented: `f` is assigned a *set* of paths $\{[u, v], [u]\}$. In this case, we generate two basic blocks from the block calling `f`: one for each path of `f`. For our example, in addition to the basic block in figure 8(c), we generate another basic block; the one shown in figure 8(d).

Consider the following recursive function defined below. What are the paths for this function?

```
def f u v =  if (u == 0) then v else f (u-1) v;
```

If this function terminates, we can argue that the result depends on both `u` and `v` – thus the function has the path $[u, v]$. There is also the possibility that the function does not terminate. We use the path $\perp_p$ to denote that possibility. Thus, the paths for the function `f` are $\{[u, v], \perp_p\}$. Using this definition of f in the example, we generate the graph in figure 8(e) instead of figure 8(d). The *bot* node in the graph denotes that the node is undefined.

To summarize, in this section we have seen how one can disconnect function-calls or use the paths of the function being called to generate basic blocks. In the next two sections, we formalize these ideas and design a global partitioning algorithm using them.

5.2 Path Semantics

$$\mathcal{A}_k[\![+]\!] = \lambda s.\{x \cup y | (x, y) \in s\}$$

$$\mathcal{A}_s[\![c]\!]\, abve = \{[]\}$$
$$\mathcal{A}_s[\![x]\!]\, abve = abve[\![x]\!]$$

$$\mathcal{A}[\![se]\!]\ abve\ aenv = \mathcal{A}_s[\![se]\!]\, abve$$
$$\mathcal{A}[\![b]\!]\, abve\ aenv = \mathcal{A}_b[\![b]\!]\, abve\ aenv$$
$$\mathcal{A}[\![p(se_1, se_2, \ldots, se_n)]\!]\, abve\ aenv = \mathcal{A}_k[\![p]\!]\ (\mathcal{A}_s[\![se_1]\!]\ abve \times \cdots \times \mathcal{A}_s[\![se_n]\!]\ abve)$$
$$\mathcal{A}[\![f(se_1, se_2, \ldots, se_n)]\!]\, abve\ aenv = aenv[\![f]\!]\ (\mathcal{A}_s[\![se_1]\!]\ abve \times \cdots \times \mathcal{A}_s[\![se_n]\!]\ abve)$$
$$\mathcal{A}[\![\textbf{if}\ se\ \textbf{then}\ b_1\ \textbf{else}\ b_2]\!]\, abve\ aenv = \{(x \cup y_1, \ldots, x \cup y_n), (x \cup z_1, \ldots, x \cup z_n)\}, \textbf{where}\ x \in \mathcal{A}_s[\![se]\!]\, abve\ \textbf{and}\ (y_1, \ldots, y_n) \in \mathcal{A}_b[\![b_1]\!]\, abve\ aenv\ \textbf{and}\ (z_1, \ldots, z_n) \in \mathcal{A}_b[\![b_2]\!]\, abve\ aenv$$

$$\mathcal{A}_b[\![\{\{(y_{i1}, \ldots, y_{im}) = e_i\}^*\ \textbf{in}\ (se_1, \ldots, se_n)\}]\!]\, abve\ aenv = \textbf{letrec}\ newabve = [y_{ij} \mapsto (\mathcal{A}[\![e_i]\!]\ (newabve.abve)\ aenv) \downarrow j]\ \textbf{in}\ (\mathcal{A}_s[\![se_1]\!]\ newbve, \ldots, \mathcal{A}_s[\![se_n]\!]\ newbve)$$

$$\mathcal{A}_p[\![\{f_i(x_1, x_2, \ldots, x_n) = b_i\}]\!] = aenv, \textbf{whererec}\ aenv = [f_i \mapsto (\lambda s.\{\mathcal{A}_b[\![b_i]\!]\ [x_i \mapsto \{y_i\}]aenv | (y_1, \ldots, y_n) \in s\})]$$

Figure 9: Path Semantics of SP-TAC

How do we compute paths for every function in an input program? We follow the well known technique of abstract interpretation [1] as used by [4]. Paths of a function are derived by using a *non-standard* semantics of SP-TAC (as opposed to the *standard* semantics presented in section 2).

Definition 10 (Paths) *A* path *is either a set of variables or $\perp_p$. In the Path domain $\perp_p$ is the least element. The* paths *for a function f with N arguments and returning M results is a set*[4] *of M-tuples of paths. An individual path*

[4] Actually, it is an element of the Egli-Milner Powerdomain of the Paths domain. Though the powerdomain construction is important in proving properties of our path semantics, we do not go into the details here.

$[u_1, \ldots, u_k]$ *denotes that if the function f takes that path, the value corresponding to the path depends on all* $u_1, \ldots, u_k$ *and is always defined if* $u_1, \ldots, u_k$ *are defined.*

These are similar to the *paths* introduced in [5], except that the paths in [5] also considered different orderings of the arguments. That is, the path $[u, v]$ was considered different from the path $[v, u]$.

Here are the domains used in our non-standard semantics.

Path	Domain of Paths
PS	Sets of paths
PTS	Sets of path-tuples
Pfun	Abstract Functions
Aenv	Function environment
Abve	Variable Environment

The semantic functions are:

$\mathcal{A}_s : SE \rightarrow Abve \rightarrow PS$	Simple Exps
$\mathcal{A}_k : Pf \rightarrow Pfun$	Primitive Operators
$\mathcal{A}_b : Bo \rightarrow Abve \rightarrow Aenv \rightarrow PTS$	Blocks
$\mathcal{A} : Exp \rightarrow Abve \rightarrow Aenv \rightarrow PTS$	Expressions
$\mathcal{A}_p : Pr \rightarrow Aenv$	Programs

We use $p_1 \cup p_2$ to denote the ordinary set union operation when both p_1 and p_2 are sets. If one of p_1 or p_2 is $\perp_p$, then $p_1 \cup p_2$ is also $\perp_p$.

Consider the various semantic equations given in figure 9.

1. The semantic function $\mathcal{A}_s$ (for constants and variables) is straightforward. In case of a constant, it returns the empty path, reflecting the fact that a constant does not have to depend on any variable for its value. For variables, it returns the paths of the variable present in the environment.

2. For primitive functions, the paths returned reflect the strict behavior of primitive functions. For the + operator, a union of the variables of the first operand and the second operand is returned modeling the fact that + requires both operands to produce a value.

3. The semantic function for *if* returns two sets, one corresponding to the variables needed when the *then* branch is taken and the other corresponding to when the *else* branch is taken.

4. The semantic function $\mathcal{A}$ for expressions is very similar to the standard semantic function $\mathcal{E}$. It recurses appropriately when the expression is either a primitive function or a user-defined function.

5. The semantic function for blocks and functions are similar to their standard semantics counterparts. They use recursion in the environments to model recursion in the blocks and functions.

It is possible to consider the semantic equations given above as an algorithm and compute paths for every function using these equations. The algorithm uses fix-point iteration to compute paths of recursive functions. Intuitively, this assumes that the paths of a function are $\perp_p$ to begin with and computes the paths of the function using the above equations. This process is repeated with the new paths computed for the function until the paths of every function remain unchanged. The computation of paths for the recursive function `fact` is shown below:

```
def fact n =
   if (n == 0) then 1 else n * fact (n - 1);
```

Expression	$Paths_0$	$Paths_1$
`n == 0`	$[n]$	$[n]$
`1`	$[]$	$[]$
`n - 1`	$[n]$	$[n]$
`(fact (n-1))`	$\perp_p$	$[n], \perp_p$
`(fact n)`	$[n], \perp_p$	$[n], \perp_p$

There are two issues that need to be addressed here:

- (Correctness) We should show that there is a strong relation between the standard semantics defined in section 2 and the path semantics give here. The proof of this is quite long, the reader is referred to [4, 8] for details. Basically, the safety of path semantics guarantees that a function always takes one of the paths computed by the path semantics.

- (Computability) The algorithm computing paths should terminate on every possible input program. Informally, this is guaranteed by the fact that a function can have only a finite number of paths ($(2^N + 1)^M$ for an N-argument, M-result function) and paths of a function in an iteration "increases" with every iteration.

The complexity of the algorithm is $O(2^{MN})$ for a function with N-arguments and M-results. Though this is exponential, it is still practical for ordinary programs if the number of arguments of a function is bounded by some small constant. As it subsumes strictness analysis [5], it is not possible to give a more efficient algorithm (strictness analysis requires exponential time [14]).

5.3 Global Partitioning using Paths

In this section, we design a global partitioning algorithm using the path analysis described in the previous section. Our algorithm is based on the intuition presented in section 5, that is, convert general to basic blocks.

Algorithm 3 (shown in figure 10) partitions a general block by propagating information in the paths, one function at a time[5]. The idea is to generate the basic blocks corresponding to each path of the function. By the safety of path semantics, we are guaranteed that the behavior of the general block is identical to one of the basic blocks. If we generate a partitioning that is correct for each of the basic blocks, we can be assured that the partitioning is correct for the general block too.

This is achieved by using any basic block partitioning algorithm (algorithm 1 or algorithm 2); the difference is the way demand and tolerance sets are computed for each node in a general block. Instead of computing them directly, they are computed using the demand and tolerance sets of the basic blocks $b_1, \ldots, b_P$.

Computing demand and tolerance sets of the basic blocks $b_1, \ldots, b_P$ is done as in section 4 – with two minor changes. First, nodes that are connected to *bot* may be deleted – they never compute any value. Second, we rename the output labels in each of the basic blocks $b_1, \ldots, b_P$ to ensure that the dependence properties of one basic block are not

[5] It is possible to propagate information for more than one function at a time. However, this leads to the generation of a large number of basic blocks, increasing the complexity of the algorithm.

"mixed up" with that of another basic block. Then, the union of demand (tolerance) sets of a node n in $b_1, \ldots, b_P$ captures the dependence properties of all the basic blocks: if any relation between two sets is true in b, it will also be true in $b_1, \ldots, b_P$. In essence, algorithm 3 uses the demand and tolerance sets to simultaneously simulate a basic block algorithm on each $b_1, \ldots, b_P$, generating partitions safe for all of them.

Algorithm 3
Given a block b:

1. Select a node in b that calls some function (say f). Disconnect all other nodes that are not primitive operators. Disconnecting a node involves introducing a new output node corresponding to each of the function's arguments and a new input node corresponding to each of the function's results.
2. Let f be an M-result function and have P paths. For each $1 \leq j \leq P$ construct block b_j as follows:
   (a) Let the j^{th} path be $(p_{j1}, p_{j2}, \ldots, p_{jM})$.
   (b) For each $1 \leq k \leq M$ do
      i. Let O be the nodes of b connected to the k^{th} output of f.
      ii. If $p_{jk} = \bot_p$, connect each node in O to a new input node called *bot*.
      iii. Otherwise, introduce a new "dummy" node (op) with inputs connected to the arguments of f in p_{jk}. Connect the output op with each node in O.
3. For $1 \leq j \leq P$, delete all nodes in b_j that are (transitively) connected to a *bot* node.
4. Rename all input and output nodes so that there is no conflict of names between blocks.
5. Apply any basic block partitioning[6] algorithm with the demand and tolerance sets of nodes in b computed using the demand and tolerance sets of nodes in $b_1, \ldots, b_P$ as follows:
$$Demand_b(n) = \bigcup_{1 \leq j \leq P} Demand_{b_j}(n)$$
$$Tolerance_b(n) = \bigcup_{1 \leq j \leq P} Tolerance_{b_j}(n)$$
6. Go to step 1 if changes to the partitioning of b occur.

Figure 10: Algorithm 3 that Partitions General Blocks

The complexity of the algorithm is $O(PN^3)$ where P is the largest number of paths of a function in the program.

6 Comparison with Related Work

Strictness analysis [16, 6, 7] has similar goals to that of partitioning. It tries to determine which arguments of a function are strict, so that they may be evaluated directly – instead of building a closure/graph for them, which is significantly more expensive. Partitioning (as presented in this paper) differs from this technique in several ways: the algorithms given here do not seek to preserve laziness, the benefit of partitioning is more explicit – it results in the creation of larger threads.

Partitioning techniques could be useful in compiling lazy functional languages too. The difference is that the choice for partitioning algorithms is restricted: only demand set partitioning preserves laziness [24]. In some cases, demand set partitioning can generate better code than traditional strictness analysis, by making use of the fact that two arguments are always used "together" in a function, even though the function may not be strict in them [26].

Path analysis [5, 4] identifies the order in which expressions in a function are evaluated. It has applications in reusing storage, optimizing representation of thunks [4]. Our contribution is to recognize that a restricted definition of paths is very useful in capturing dependence information of functions, and this information can be effectively used in a global partitioning algorithm.

Much of the research in partitioning is based on the seminal work in [25]. It defined the partitioning problem, its relation to dependence analysis and provided some insight into the partitioning problem by proving a related problem NP-hard. Our work provides the first insight into the complexity of the problem – it shows that finding an optimal partitioning (minimum number of partitions or largest possible partitions) is NP-hard even for basic blocks.

Developing heuristics to generate safe partitions is the focus of many papers [12, 21, 20]. These include *dependence partitioning*[15] based on the notion of dependence sets. Dependence sets are the natural "dual" of demand sets, where each node is annotated with the set of *inputs* that it depends on. In dependence set partitioning, nodes with the same dependence set can be merged together. Tolerance sets defined in this paper subsume the notion of demand sets: *i.e.*, it is always possible to merge two nodes using their tolerance sets, if it is possible to do so using dependence sets, but not vice-versa. Figure 5 provides an example of this. The dependence sets of nodes 2 and 5 are $\{a, b\}$ and $\{b\}$ respectively, but it is still safe to merge them as they have the same tolerance sets.

Demand sets itself were introduced in [21, 12], and the iterated partitioning algorithm (similar to algorithm 2) was introduced in [26, 12]. Most of these papers did not do any global analysis for partitioning, they used the disconnecting operation used in algorithm 3 to handle conditionals and calls to user defined functions. Also, these algorithms did not produce *maximal* partitions, *i.e.*, even after these algorithms terminated, it was possible that there are two partitions that could be merged safely.

An inter-procedural algorithm for partitioning was first designed in [26]. This algorithm is based on the idea that the demand/dependence sets themselves contain some dependence information, and propagating these sets between the definition of a function to its call-site yields better partitions. A major limitation of the inter-procedural algorithm given there is its inability to handle recursive functions. Our algorithm has no such limitations, as the paths of recursive functions are well-defined and our inter-procedural algorithm is based *only* on a function's paths. For example, in figure 8, our inter-procedural algorithm succeeds in merging nodes 1 and 2 when `f` is the recursive function given in

[6] A small change is needed. We have to make sure the "dummy" operators introduced do not fall into any partition.

Section 5.1. Their algorithm has to make a worst case assumption about the recursive call to `f`, and is unable to merge these two nodes.

Interestingly, our algorithm is more powerful even in the non-recursive case.
Example 3:

```
def f p a b c =
 {x = a + b;
  y = b * c;
  z = if p then x else y
 in
   z};
```

In this example, the algorithm of [26] fails to merge the input nodes corresponding to `p` and `b`, even though the result is strict in both of them. This is because the strictness information is obscured by the presence of non-strict arguments `a` and `c`. Our inter-prodedural algorithm detects that `p` and `b` can be merged in both the basic blocks generated by the two branches of the conditional, thus merging `p` and `b`.

Independent of this work, the partitioning algorithms in [26] have been extended in [20, 19] to generate maximal partitions for basic blocks and to handle recursive functions. Algorithm 1 for partitioning basic blocks is equivalent to the one presented in [19], though it is a different formulation and its complexity is slightly better. Algorithm 2, however, is a new algorithm that subsumes the algorithm in [26], but still retains its structure. The inter-procedural algorithm given in [19] differs from our inter-procedural algorithm in the following ways.

- It still uses dependence and demand sets to propagate global information, and thus retains some of the limitations of [26] (*e.g.*, example 3).
- The extension that detects strictness information is an approximation to the abstract interpretation algorithm used here.

However, the complexity of our inter-procedural algorithm is higher than that of the algorithm in [19].

7 Conclusions and Extensions

In this paper, we have presented new algorithms for partitioning non-strict functional languages. We have presented two new algorithms for basic block partitioning and characterized the complexity of the problem. We have extended the basic block algorithms to propagate information globally. Our inter-procedural algorithm is based on abstract interpretation and handles recursive functions in a natural manner. We have also defined partitioning as a program transformation enabling us to prove the correctness of our algorithms using denotational semantics.

Several straightforward extensions follow from the techniques presented in this paper. We can extend our algorithms to incorporate higher-order functions and data-structures that are not present in our kernel language. The basic technique is to be conservative when dealing with blocks that contain higher-order functions or data-structures. Though path analysis extends naturally to higher-order functions, it is computationally more efficient to make approximations when dealing with them [4]. On the partitioning side, one can disconnect nodes that call perform calls to unknown functions and then partition the block using algorithm 3. A similar approach can be used to deal with data-structures. However, it is interesting to extend the algorithms to deal with data-structures like lists and arrays making use of their dependence information. Research done in analyzing strictness properties of lists [27] and Fortran style array subscript analysis would probably be of relevance here.

A nontrivial extension of the partitioning algorithm is to incorporate *context* information during partitioning. The idea behind this is simple: in many cases, one can deduce that a function is always being used in a strict manner and this information can be used to generate better partitions for the the function. An extension to the partitioning algorithm that achieves this is presented in [8].

We plan to implement these algorithms in a compiler for Id. This would shed light on the quantitative benefits of the various parts of our partitioning algorithms.

8 Acknowledgments

Many thanks to Shail Aditya, Arvind, and Boon Ang for their helpful comments and suggestions.

References

[1] S. Abramsky and C. L. (eds) Hankin. *Abstract Interpretation of Declarative Languages.* Ellis-Horwood, 1987.

[2] Z. Ariola and Arvind. P-TAC: A Parallel Intermediate Language. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, London, UK*, pages 230–242. ACM, September 1989.

[3] Z. Ariola and Arvind. Compilation of Id. In *Proc. of the Fourth Workshop on Languages and Compilers for Parallel Computing (LNCS 589).* Springer-Verlag, August 1991.

[4] A. Bloss. *Path Analysis and the Optimization of Non-strict Functional Languages.* PhD thesis, Dept. of Computer Science, Yale University, May 1989.

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

[6] J. Burn, C. Hankin, and S. Abramsky. Strictness Analysis for Higher-order Functions. *Science of Computer Programming*, 9, 1986.

[7] C. Clack and S. L. Peyton-Jones. Strictness Analysis – A Practical Approach. In *Proceedings of ACM Conference of Functional Programming Languages and Computer Architecture.* ACM, September 1985.

[8] S. R. Coorg. Partitioning Non-strict Languages for Multi-threaded Code Generation. Masters Thesis, EECS, MIT, May 1994.

[9] 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(4):347–370, July 1993.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, San Francisco, CA, 1979.

[11] P. Henderson. *Functional Programming: Applications and Implementation.* Prentice Hall, Englewood Cliffs, NJ, 1980.

[12] J. E. Hoch, D. M. Davenprot, 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.

[13] 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, April 1990.

[14] P. Hudak and J. Young. Higher-order Strictness Analysis of the Untyped Lambda-Calculus. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, pages 97–109. ACM, Jan 1986.

[15] R. A. Ianucci. *A Dataflow/von Neumann Hybrid Architecture.* PhD thesis, Dept. of EECS, MIT, May 1988. (Also MIT LCS TR-418).

[16] 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.

[17] R. S. Nikhil. Id Language Reference Manual Version 90.1. Technical Report CSG Memo 284-2, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, July 15 1991.

[18] 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.

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

[20] K. E. Schauser, D. Culler, and S. C. Goldstein. Separation Constraint Partitioning – A New Algorithm for Partitioning Non-strict Programs into Sequential Threads. In *Proc. ACM Conference on Principles of Programming Languages (POPL)*, January 1995.

[21] K. E. Schauser, D. Culler, and von Eicken T. Compiler-controlled Multithreading for Lenient Parallel Languages. In *Proc. Conf. on Functional Programming Languages and Computer Architecture*, August 1991.

[22] D. A. Schmidt. *Denotational Semantics – A Methodology for Language Development.* Allyn and Bacon, Boston, MA, 1986.

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

[24] K. R. Traub. Compilation as Partitioning: A New Approach to Compile Non-strict Functional Languages. In *Proc. of the ACM Conf. on Functional Programming and Computer Architecture*, pages 75–88. ACM, 1989.

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

[26] 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.

[27] P. Wadler. Strictness Analysis on Non-Flat Domains. In *[1]*.

A Proof Sketches of the Theorems

In this section, we present proof sketches of some of the main theorems presented in the paper. Refer to [8] for full versions of these.

Proof: (Sketch of theorem 7) We only prove the sufficiency of these conditions here. Assume (for contradiction) that b_Q is not correct. That is, there is an environment bve such that $\mathcal{E}^{pr}[\![b]\!]\, bve \neq \mathcal{E}^{pr}[\![b_Q]\!]\, bve$. By the property of partitioning, this can only be true if one of the outputs (say o) has a value in the first case, and is $\perp$ in the second. As o has a value, each of inputs I of b that are connected to o should be defined in bve. As o is $\perp$ in the second case and b_Q is acyclic, one of the inputs I_Q of b_Q must be $\perp$ in bve. Thus, $I \neq I_Q$, a contradiction to the second condition. □

The proof of correctness of algorithms 1 and 2 make use of the following properties of demand and tolerance sets of a basic block.

Lemma 11 (Demand-Tolerance Relation) *For a node n, $Demand(n) \subseteq Tolerance(n)$.*

Lemma 12 (Demand and Tolerance Monotonicity) *If node n is transitively connected (from n) to node m in a basic block, then $Demand(m) \subseteq Demand(n)$ and $Tolerance(m) \subseteq Tolerance(n)$.*

Theorem 13 (Correctness of Algorithm 1) *Algorithm 1 correctly partitions a basic block.*

Proof: (Sketch) We use the basic block correctness criterion to prove that each step of the algorithm (where a clique is identified and merged) is correct. The correctness of the entire algorithm follows by an easy induction.

1. First, we prove that the connectivity condition is satisfied. For simplicity, consider the merging of just two nodes n and m such that (n, m) is an edge of G_{merge}. For input-output connectivity, this is equivalent to adding an edge from m to each of the outputs connected to n (*i.e.*, $Demand(n)$) and vice-versa. By the definition of tolerance sets, this does not affect input-output connectivity if $Demand(n) \subseteq Tolerance(m)$, which is true.
2. Now, we prove that the graph corresponding to the reduced basic block is acyclic. Assume (for contradiction) that a cycle is caused by merging the nodes in the clique. Without loss of generality, there are two nodes n and m in the clique and a node n' not in the clique, such that n is connected to n' and n' is connected to m in b. Due to monotonicity of demand and tolerance sets, for any other node m', if both (n, m') and (m, m') are in G_{merge}, then so

is (n', m'). Thus, n' could also be included in the clique and hence, the clique is not maximal, a contradiction.

□

Theorem 14 (Correctness of Algorithm 2) *Algorithm 2 correctly partitions a basic block.*

Proof: (Sketch) Correctness of Algorithm 2 follows from the fact that nodes with the same demand (or tolerance) set form a clique of the graph G_{merge} in algorithm 1. Consider two nodes n and m such that $Demand(n) = Demand(m)$. By lemma 11, $Demand(m) \subseteq Tolerance(m)$. Thus, $Demand(n) \subseteq Tolerance(m)$ and vice-versa. Thus, the edge (n, m) is present in G_{merge}. A similar argument proves that merging nodes with the same tolerance set is also safe. Acyclicity of the reduced graph follows from the monotonicity of demand and tolerance sets. □

Theorem 15 (Correctness of Algorithm 3) *Algorithm 3 correctly partitions any block.*

Proof: (Sketch) Let b be the input block and f the function being called. For simplicity, assume that the node calling f is only non-primitive node in b (*i.e.*, there are no nodes to be disconnected). First, note that the partitions generated for b are also correct partitions for $b_1, \ldots, b_P$. This is ensured as we are taking the unions of the demand and tolerance sets of each $b_j, 1 \leq j \leq P$. Next, it is possible to give precise definitions to the primitive operators introduced in $b_1, \ldots, b_P$ such that for any environment bve, $\mathcal{E}^{pr}[\![b]\!]\, bve = \mathcal{E}^{pr}[\![b_j]\!]\, bve$, for some $1 \leq j \leq P$ (this follows from the safety of path semantics).

Thus, if a partitioning is incorrect, it will be incorrect for one b_j too. This contradicts that fact that we are using a correct partitioning algorithm on b_j, which is a basic block. □