Semantic Foundations of Commutativity Analysis

Martin C. Rinard (martin@cs.ucsb.edu)
Pedro Diniz (pedro@cs.ucsb.edu)
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106

Abstract

This paper presents the semantic foundations of commutativity analysis, an analysis technique for automatically parallelizing programs written in a sequential, imperative programming language. Commutativity analysis views the computation as composed of operations on objects. It then analyzes the program at this granularity to discover when operations commute (i.e. generate the same result regardless of the order in which they execute). If all of the operations required to perform a given computation commute, the compiler can automatically generate parallel code.

This paper shows that the basic analysis and tranformations that form the foundation of the parallelization technique are sound. It also presents performance results from two automatically parallelized programs.

1 Introduction

Current parallelizing compilers preserve the semantics of the original serial program by preserving the data dependences[1, 12]. They analyze the program to identify independent pieces of computation (two pieces of computation are independent if neither writes a piece of memory that the other accesses), then generate code that executes independent pieces concurrently.

A significant limitation of this approach is the difficulty of performing dependence analysis that is precise enough to expose substantial amounts of concurrency. While researchers have been able to develop reasonably effective algorithms for loop nests that manipulate dense matrices using affine access functions[13], there has been little progress towards the successful automatic analysis of programs that manipulate pointer-based data structures. Researchers have attempted to build totally automatic systems [8], but the most promising approaches require the programmer to provide annotations that specify information about the global topology of the manipulated data structures[9].

A second, more fundamental limitation of data-dependence based approaches is an inability to parallelize computations that manipulate graphs. The aliases inherently present in these data structures preclude the static discovery of independent pieces of code, forcing the compiler to generate serial code. Finally, preserving the data dependences for programs that periodically update shared data structures can artificially limit the amount of exposed concurrency, since tasks must delay updates until they are sure that each update will not change the relative order of reads and writes to the shared data structure[3, 16].

This paper presents the semantic foundations of a new analysis technique called commutativity analysis[17, 19]. Instead of preserving the relative order of individual reads and writes to single words of memory, commutativity analysis views the computation as composed of operations on objects. It then analyzes the computation at this granularity to discover when pieces of the computation commute (i.e. generate the same result regardless of the order in which they execute). If all of the operations required to perform a given computation commute, the compiler can automatically generate parallel code. While the resulting parallel program may violate the data dependences of the original serial program, it is still guaranteed to generate the same result.

We expect commutativity analysis to eliminate many of the limitations of existing approaches. Compilers that use commutativity analysis will be able to automatically generate parallel code for many applications that periodically update shared data structures using commuting operations and/or manipulate recursive, pointer-based data structures such as lists, trees and graphs. Commutativity analysis allows compilers to automatically parallelize computations even though they have no information about the global topology of the manipulated data structure. It may therefore be especially useful for parallelizing computations that manipulate the persistent data in object-oriented databases. In this context the code that originally created the data structure may be unavailable, negating any approach (including data-dependence based approaches) that analyzes the code to verify or discover global properties of the data structure topology.

The rest of the paper is structured as follows. In Section 2 we present an example that illustrates how commutativity analysis can automatically parallelize graph traversals. In Section 3 we describe the basic approach and informally state the conditions that the compiler uses to recognize commuting operations. In Section 4 we provide a formal model of computation and define what it means for two operations to commute in this model. This section contains a key theorem establishing that if all of the operations in a computation commute, then all parallel executions are equivalent to the serial execution. In Section 6 we present the analysis algorithm that the compiler uses to determine if two operations commute. In Section 7 we present experimental results for two complete scientific applications parallelized by a prototype compiler that uses commutativity analysis as its basic analysis paradigm. We survey related work in Section 8 and conclude in Section 9.

2 An Example

In this section we present a simple example that shows how recognizing commuting operations can enable the automatic generation of parallel code. The `visit` method in Figure 1 serially traverses a graph. When the traversal completes, each node's `sum` instance variable contains the sum of its original value and the values of the `val` instance variables in all of the nodes that directly point to that node. The example is written in C++.

```
class graph {
  boolean mark;
  int val, sum;
  graph *left; graph *right;
};
graph::visit(int p) {
  graph *l = left; graph *r = right;
  int v = val;
  sum = sum + p;
  if (!mark) {
    mark = TRUE;
    if (l != NULL) l->visit(v);
    if (r != NULL) r->visit(v);
  }
}
```

Figure 1: Serial Graph Traversal

The traversal generates one invocation of the `visit` method for each edge in the graph. We call each method invocation an *operation*. The receiver of each `visit` operation is the node to traverse. Each `visit` operation takes as a parameter `p` the value of the instance variable `val` of the node that points to the receiver. The `visit` operation first adds `p` into the running sum stored in the receiver's `sum` instance variable. It then checks the receiver's `mark` instance variable to see if the traversal has already visited the receiver. If not, the operation marks the receiver, then recursively invokes the `visit` method for all of the nodes that the receiver points to.

The way to parallelize the traversal is to execute the two recursive `visit` operations concurrently. But this parallelization may violate the data dependences. The serial computation executes all of the accesses generated by the left traversal before all of the accesses generated by the right traversal. If the two traversals visit the same node, in the parallel execution the right traversal may visit the node before the left traversal, changing the order of reads and writes to that node. This violation of the data dependences may generate cascading changes in the overall execution of the computation. Because of the marking algorithm, a node only executes the recursive calls the first time it is visited. If the right traversal reaches a node before the left traversal, the parallel execution may also change the order in which the overall traversal is generated.

In fact, none of these changes affects the overall result of the computation. It is possible to automatically parallelize the computation even though the resulting parallel program may generate computations that differ substantially from the original serial computation. The key property that enables the parallelization is that the parallel computation generates the same set of `visit` operations as the serial computation and the generated `visit` operations can execute in any order without affecting the overall behavior of the traversal.

Given this commutativity information, the compiler can automatically generate the parallel `visit` method in Figure 2. The top level `visit` method first invokes the `parallel_visit` method, then invokes the `wait` construct, which blocks until all parallel tasks created by the current task or its descendant tasks finishes. The `parallel_visit` method executes the recursive calls concurrently using the `spawn` construct, which creates a new task for the execution of each invocation. The compiler also augments the `graph` data structure with a mutual exclusion lock `mutex`. This lock ensures that each invocation of `parallel_visit` executes atomically with respect to all other invocations with the same receiver. The `parallel_visit` method acquires the lock before accessing the receiver and releases the lock before invoking any methods. A straightforward application of lazy task creation techniques [14] can increase the granularity of the resulting parallel computation.

```
class graph {
  lock mutex;
  boolean mark;
  int val, sum;
  graph *left; graph *right;
};
graph::visit(int s) {
  this->parallel_visit(s);
  wait();
}
graph::parallel_visit(int p) {
  mutex.acquire();
  graph *l = left; graph *r = right;
  int v = val;
  sum = sum + p;
  if (!mark) {
    mark = TRUE;
    mutex.release();
    if (l != NULL)
      spawn(l->parallel_visit(v));
    if (r != NULL)
      spawn(r->parallel_visit(v));
  } else {
    mutex.release();
  }
}
```

Figure 2: Parallel Graph Traversal

3 The Basic Approach

Commutativity analysis is designed for programs written using a pure object-based paradigm. Such programs structure the computation as a sequence of operations on objects. Each operation consists of a receiver object, an operation name and several parameters. Each operation name identifies a method that defines the behavior of the operation; when the operation executes it invokes that method. Each object implements its state using a set of instance variables. When a method executes it can recursively invoke other operations and/or use primitive operators (such as addition and multiplication) to perform computations involving the parameters and the instance variables of the receiver.

Commutativity analysis is designed to work with *separable* methods. A method is separable if its execution can be decomposed into an object section and an invocation section. The object section performs all accesses to the receiver. The invocation section invokes other operations and does not access the receiver. It is of course possible for local variables to carry values computed in the object section into the invocation section, and both sections can access the parameters. Separability imposes no expressibility limitations — although the current compiler does not do so, it is possible for a compiler to automatically convert any operation into a collection of separable operations via the introduction of auxiliary operations.

The foundation of commutativity analysis is the following conditions, which the compiler can use to test if two operations A and B commute.

1. (**Instance Variables**) The new value of each instance variable of the receiver objects of A and B under the execution order A;B must be the same as the new value under the execution order B;A.

2. (**Invoked Operations**) The multiset of operations directly invoked by either A or B under the execution order A followed by B must be the same as the multiset of operations directly invoked by either A or B under the execution order B followed by A.[1]

Note that these conditions do not deal with the entire recursively invoked computation that each operation generates — they only deal with the object and invocation sections of the two operations. These conditions are nevertheless sufficient because the compiler tests that all pairs of operations in a given computation satisfy these conditions before it parallelizes the computation.

The instance variables condition ensures that if a parallel computation executes the same multiset of operations as the serial computation, the values of the instance variables will be the same at the end of the parallel computation as at the end of the serial computation. The basic reasoning is that for each object, a parallel computation will execute the object sections of the operations on that object in some arbitrary order. The instance variables condition ensures that all orders yield the same final result.

The invoked operations condition provides the foundation for the application of the instance variables condition: it ensures that all parallel computations execute the same multiset of operations as the serial computation.

Both commutativity testing conditions are trivially satisfied if the two operations have different receivers — the executions of the two methods are independent because they access disjoint pieces of data. We therefore focus on the case when the two operations have the same receiver.

It is possible to determine if each of the receiver's instance variables has the same new value in both execution orders by analyzing the invoked methods to extract two

[1] Two operations are the same if they execute the same method and have the same parameters.

symbolic expressions. One of the symbolic expressions denotes the new value of the instance variable under the execution order A;B. The other denotes the new value under the execution order B;A. Given these two expressions, a compiler may be able to use algebraic reasoning to discover that they denote the same value. The compiler uses a similar approach to determine if A and B together invoke the same multiset of operations in both execution orders.

We illustrate these concepts by applying them to the graph traversal example in Figure 1. We assume two invocations `r->visit(p1)` and `r->visit(p2)` of the `visit` method. `r->visit(p1)` has parameter `p1`, `r->visit(p2)` has parameter `p2` and both operations have the same receiver `r`.

We first consider the instance variable `sum`. Table 1 contains the two expressions denoting the new values of `sum` under the two execution orders. In these expressions `sum` represents the old value of the `sum` instance variable before either method executes. Given these two expressions, the compiler can use the fact that + is both commutative and associative to discover that the two expressions denote the same value.[2]

We next consider the new value of the `mark` instance variable. Because its new value depends on the flow of control through the method, the expressions representing its new values contain conditionals. A conditional of the form $\mathtt{if}(\mathtt{ex}, \mathtt{ex}_1, \mathtt{ex}_2)$ denotes the expression $\mathtt{ex}_1$ if `ex` is nonzero and $\mathtt{ex}_2$ if `ex` is zero. Table 2 contains the new values of the `mark` instance variable, which are identical under both execution orders.

Finally, because the `visit` method writes none of the other instance variables, their new values are the same as their values before the execution of the two operations. The two operations `r->visit(p1)` and `r->visit(p2)` therefore satisfy the first commutativity testing condition.

We next consider the multiset of invoked operations. Because an operation may be invoked along one control flow path but not another, the symbolic operation expressions used to denote invoked operations may contain conditionals. A conditional symbolic operation expression of the form `if(ex,o)` denotes an operation `o` that is invoked only if `ex` is nonzero. Table 3 contains the symbolic operation expressions that denote the operations that `r->visit(p1)` and `r->visit(p2)` directly invoke under the two execution orders. The compiler checks that the two operations meet the second commutativity testing condition by comparing the union of the symbolic operation expressions under the execution order `r->visit(p1);r->visit(p2)` with the union of the symbolic operation expressions under the execution order `r->visit(p2);r->visit(p1)`. In this example `r->visit(p1)` and `r->visit(p2)` together invoke the same set of operations in both execution orders, and the two operations satisfy the second commutativity testing condition.

A compiler can use the commutativity testing conditions described above to determine if it can legally generate parallel code for a given method. The compiler first computes a conservative approximation to the set of methods invoked as a result of invoking the given method. It then applies the commutativity testing conditions described above to all pairs of potentially invoked methods that may have the same receiver. If all of the pairs commute, the compiler can legally generate parallel code.

[2]We ignore here potential anomalies caused by the finite representation of numbers. A compiler switch that disables the exploitation of commutativity and associativity for operators such as + will allow the programmer to prevent the compiler from performing transformations that may change the order in which the parallel program combines the summands.

4 Formal Treatment

We next provide a formal treatment of how commutativity enables parallel execution. We first present a formal model of computation for separable operations. We define parallel and serial execution in this model, and define what it means for two operations to commute. The key theorem establishes that if all of the operations in the parallel executions commute, then all parallel executions are equivalent to the serial execution.

4.1 The Model of Computation

We next describe a formal model of computation for separable programs. We assume a set of objects $r, o \in O$, a set of constants $\mathtt{c} \in \mathtt{C} \subseteq O$, a set of instance variable names $\mathtt{v} \in \mathtt{IV}$ and a set of operation names $\mathtt{op} \in \mathtt{OP}$. The operational semantics uses several functions. We model instance variable values with functions $i : I = \mathtt{IV} \to O$. We say that an object r is in state i if the value of each instance variable `v` of r is $i(\mathtt{v})$. We model object memories with functions $m : M = O \to I$. We also have applied operations $a \in A = O \times \mathtt{OP} \times O$, and write an element a in the form $r\text{->}\mathtt{op}(o)$. Each applied operation has a receiver r, an operation name `op` and a parameter o. To simplify the presentation we assume that each operation takes only one parameter, but the framework generalizes in a straightforward way to include operations with multiple parameters.

We next describe how we model computation. The execution of each invoked operation first updates the receiver, then invokes several other operations. We model the effect on the receiver with a receiver effect function $R : A \times I \to I$. $R(r\text{->}\mathtt{op}(o), i)$ is the new state of r when operation $r\text{->}\mathtt{op}(o)$ executes with the receiver r in

Execution Order	New Value of `sum`
`r->visit(p1);r->visit(p2)`	`(sum + p1) + p2`
`r->visit(p2);r->visit(p1)`	`(sum + p2) + p1`

Table 1: New Values of `sum` Under Different Execution Orders

Execution Order	New Value of `mark`
`r->visit(p1);r->visit(p2)`	`if(!if(!mark,TRUE,mark),TRUE,if(!mark,TRUE,mark))`
`r->visit(p2);r->visit(p1)`	`if(!if(!mark,TRUE,mark),TRUE,if(!mark,TRUE,mark))`

Table 2: New Values of `mark` Under Different Execution Orders

Execution Order	Symbolic Operations Invoked By `r->visit(p1)` and `r->visit(p2)`
`r->visit(p1);r->visit(p2)`	`if(!mark,if(left!=NULL,left->visit(val))),`
	`if(!mark,if(right!=NULL,right->visit(val)))`
	`if(!if(!mark,TRUE,mark),`
	`    if(left!=NULL,left->visit(val))),`
	`if(!if(!mark,TRUE,mark),`
	`    if(right!=NULL,right->visit(val)))`
`r->visit(p2);r->visit(p1)`	`if(!mark,if(left!=NULL,left->visit(val))),`
	`if(!mark,if(right!=NULL,right->visit(val)))`
	`if(!if(!mark,TRUE,mark),`
	`    if(left!=NULL,left->visit(val))),`
	`if(!if(!mark,TRUE,mark),`
	`    if(right!=NULL,right->visit(val)))`

Table 3: Symbolic Operations Invoked by `r->visit(p1)` and `r->visit(p2)` Under Different Execution Orders

state i; it models the effect of the operation's object section on the receiver.

We model the invoked operations using sequences $s \in S = seq(A)$ of the form $a_1 \circ \cdots \circ a_k$. ϵ is the empty sequence, and $\epsilon \circ s = s \circ \epsilon = s$. We use an invoked operation function $N : A \times I \rightarrow seq(A)$ to model the sequence of operations invoked as a result of executing an operation. $N(r\text{->}\mathrm{op}(o), i)$ is the sequence of operations directly invoked when $r\text{->}\mathrm{op}(o)$ executes with the receiver r in state i; it models the multiset of operations invoked in the invocation section of the operation.

The serial operational semantics of the program uses a transition function $\rightarrow$ on serial states $\langle m, s\rangle \in M \times seq(A)$, where m is the current object memory and s is the sequence of operations left to invoke. $\rightarrow$ models the execution of the next operation in s. As Definition 1 shows, $\rightarrow$ updates the memory to reflect the new state of the receiver of the executed operation. It also removes the executed operation from the current sequence of operations left to invoke and prepends the multiset of operations that the executed operation invokes. The operations therefore execute in the standard depth-first execution order. Strictly speaking, $\rightarrow$ depends on R and N, but we do not represent this dependence explicitly as the correct R and N are always obvious from context.

Definition 1 *$\rightarrow$ is the function on $M \times seq(A)$ defined by:*

$$\frac{m' = m[r \mapsto R(r\text{->}\mathrm{op}(o), m(r))], \quad s' = N(r\text{->}\mathrm{op}(o), m(r)) \circ s}{\langle m, r\text{->}\mathrm{op}(o) \circ s\rangle \rightarrow \langle m', s'\rangle}$$

The states $\langle m, p\rangle$ in the parallel operational semantics are similar to the states in the serial operational semantics, but have a multiset p, rather than a sequence s, of operations left to invoke. Such multisets $p \in mst(A)$ are of the form $\{a_1\} \uplus \cdots \uplus \{a_k\}$. We use the sequence to multiset function $s2p : seq(A) \rightarrow mst(A)$ defined by $s2p(a_1 \circ \cdots \circ a_k) = \{a_1\} \uplus \cdots \uplus \{a_k\}$ to map sequences of applied operations into the corresponding multisets.

We model parallel execution by generating all possible interleavings of the execution of the applied operations. In the parallel execution any of the current operations may execute next, with each operation taking the current state to a potentially different state than the other operations. The parallel operational semantics therefore models execution using a transition relation $\Rightarrow$ rather than a transition function. As the following definition illustrates, $\Rightarrow$ relates a state to all of the states reachable via the execution of any of the current multiset of operations left to invoke. The definition of $\Rightarrow$ completes the semantic framework required to precisely state when two operations commute.

Definition 2 *$\Rightarrow$ is the smallest relation* [3] *on $M \times mst(A)$ satisfying the following condition:*

$$\frac{\langle m, r\text{->}\mathrm{op}(o)\rangle \rightarrow \langle m', s'\rangle, p' = s2p(s') \uplus p}{\langle m, \{r\text{->}\mathrm{op}(o)\} \uplus p\rangle \Rightarrow \langle m', p'\rangle}$$

For two operations to commute, they must leave their receivers in identical states and invoke the same multiset of operations regardless of the order in which they execute.

Definition 3 *$r_1\text{->}\mathrm{op}_1(o_1)$ and $r_2\text{->}\mathrm{op}_2(o_2)$ commute in m if $\langle m, \{r_1\text{->}\mathrm{op}_1(o_1)\}\rangle \Rightarrow \langle m_1, p_1\rangle$ and $\langle m, \{r_2\text{->}\mathrm{op}_2(o_2)\}\rangle \Rightarrow \langle m_2, p_2\rangle$ and $\langle m_1, \{r_2\text{->}\mathrm{op}_2(o_2)\}\rangle \Rightarrow \langle m_{12}, p_{12}\rangle$ and $\langle m_2, \{r_1\text{->}\mathrm{op}_1(o_1)\}\rangle \Rightarrow \langle m_{21}, p_{21}\rangle$ implies $m_{12} = m_{21}$ and $p_1 \uplus p_{12} = p_2 \uplus p_{21}$*

Definition 4 *$r_1\text{->}\mathrm{op}_1(o_1)$ and $r_2\text{->}\mathrm{op}_2(o_2)$ commute if $\forall m : r_1\text{->}\mathrm{op}_1(o_1)$ and $r_2\text{->}\mathrm{op}_2(o_2)$ commute in m.*

Theorem 1 *If $r_1 \neq r_2$ then $r_1\text{->}\mathrm{op}_1(o_1)$ and $r_2\text{->}\mathrm{op}_2(o_2)$ commute.*

Proof Sketch: Intuitively, if the receivers are different, the operations are independent because they access disjoint pieces of data.

4.2 Correspondence Between Parallel and Serial Semantics

We next present several lemmas that characterize the relationship between the parallel and serial operational semantics. Lemma 1 says that for each partial serial execution, there exists a partial parallel execution that generates an equivalent state.

Lemma 1 *If $\langle m, r\text{->}\mathrm{op}(o)\rangle \rightarrow \cdots \rightarrow \langle m', s\rangle$ then $\langle m, \{r\text{->}\mathrm{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m', s2p(s)\rangle$.*

Proof Sketch: At each step the parallel execution may execute the same operation as the serial execution.

An immediate corollary of Lemma 1 is that if the serial computation terminates, then one of the parallel computations must terminate with the same memory.

Corollary 1 *If $\langle m, r\text{->}\mathrm{op}(o)\rangle \rightarrow \cdots \rightarrow \langle m', \epsilon\rangle$ then $\langle m, \{r\text{->}\mathrm{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m', \emptyset\rangle$.*

We next establish the impact that commutativity has on the different parallel executions. Lemma 2 states that if all of the invoked operations in given computation commute and one of the parallel executions terminates, then all parallel executions terminate with the same memory. This lemma uses the function $gen : M \times A \rightarrow 2^A$, which tells which operations can be invoked as a result of invoking a given operation.

[3] Under the subset ordering on relations considered as set of pairs defined as follows. Consider two relations $\Rightarrow_1, \Rightarrow_2 \subseteq (M \times mst(A)) \times (M \times mst(A))$. $\Rightarrow_1 \leq \Rightarrow_2$ if $\Rightarrow_1 \subseteq \Rightarrow_2$.

Definition 5 $gen(m, r\texttt{->}\texttt{op}(o)) = \cup\{p : \langle m, \{r\texttt{->}\texttt{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m', p\rangle\}$

Lemma 2
If $\forall r_1\texttt{->}\texttt{op}_1(o_1), r_2\texttt{->}\texttt{op}_2(o_2) \in gen(m, r\texttt{->}\texttt{op}(o))$: $r_1\texttt{->}\texttt{op}_1(o_1)$ *and* $r_2\texttt{->}\texttt{op}_2(o_2)$ *commute, then* $\langle m, \{r\texttt{->}\texttt{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m_1, \emptyset\rangle$ *implies*

- *not* $\langle m, \{r\texttt{->}\texttt{op}(o)\}\rangle \Rightarrow \cdots$ *(i.e. there is no infinite parallel execution), and*
- $\langle m, \{r\texttt{->}\texttt{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m_2, \emptyset\rangle$ *implies* $m_1 = m_2$

Proof Sketch: If all of the invoked operations commute then the transition system is confluent, which guarantees deterministic execution[10].

Lemma 3 characterizes the situation when the computation may not terminate. It says that if all of the operations invoked in the parallel executions commute, then it is possible to take any two partial parallel executions and extend them to identical states.

Lemma 3
If $\forall r_1\texttt{->}\texttt{op}_1(o_1), r_2\texttt{->}\texttt{op}_2(o_2) \in gen(m, r\texttt{->}\texttt{op}(o))$: $r_1\texttt{->}\texttt{op}_1(o_1)$ *and* $r_2\texttt{->}\texttt{op}_2(o_2)$ *commute, then*
$\langle m, \{r\texttt{->}\texttt{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m_1, p_1\rangle$ *and*
$\langle m, \{r\texttt{->}\texttt{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m_2, p_2\rangle$ *implies*
$\exists m' \in M, p \in mst(A) : \langle m_1, p_1\rangle \Rightarrow \cdots \Rightarrow \langle m', p\rangle$ *and*
$\langle m_2, p_2\rangle \Rightarrow \cdots \Rightarrow \langle m', p\rangle$

Proof Sketch: If all of the invoked operations commute then the transition system is confluent, which guarantees deterministic execution[10].

An immediate corollary of these two lemmas is that if the serial computation terminates, then all parallel computations terminate with identical memories. Conversely, if a parallel computation terminates, then the serial computation also terminates with an identical memory.

5 Program Semantics

We next show how a program consisting of a sequence of operations defines the receiver effect function R and invoked operation function N that determine the behavior of the operations. We first define the abstract syntax for a simple object-based language. We assume a set of local variables $\texttt{u} \in \texttt{LV}$.

The grammar in Figure 3 defines the syntactic categories of programs PR, statements ST, arithmetic expressions EX and method invocation expressions MX. The defined set of expressions is standard except that it includes conditional expressions that would normally be used only for the symbolic analysis defined in Section 6.2. Finally, the symbolic analysis represents multiple invoked methods

$$
\begin{array}{lll}
\texttt{pr} \in \texttt{PR} & \texttt{::=} & \texttt{op(p)\{ST\}PR} | \epsilon \\
\texttt{st} \in \texttt{ST} & \texttt{::=} & \texttt{ST;ST} | \texttt{v=EX} | \texttt{u=EX} | \texttt{MX} | \epsilon \\
\texttt{ex} \in \texttt{EX} & \texttt{::=} & \texttt{EX+EX} | \texttt{-EX} | \texttt{if(EX,EX,EX)} | \texttt{v} | \texttt{u} | \texttt{c} \\
\texttt{mx} \in \texttt{MX} & \texttt{::=} & \texttt{EX->op(EX)} | \texttt{if(EX,MX)}
\end{array}
$$

Figure 3: Syntactic Categories for a Simple Programming Language

using sequences $\texttt{s} \in \texttt{S} = seq(\texttt{MX})$ of symbolic invoked method expressions $\texttt{s} = \texttt{mx}_1 \circ \cdots \circ \texttt{mx}_k$. We define the general set of expressions $\texttt{x} \in \texttt{X}$ by $\texttt{X} = \texttt{EX} \cup \texttt{MX} \cup \texttt{S}$.

Following the standard convention in object-based languages, each method omits the declaration of the receiver. The method refers to the receiver using the variable $\texttt{r} \in \texttt{LV}$, and refers to its single parameter using the variable $\texttt{p} \in \texttt{LV}$. We assume that all programs also satisfy standard static consistency checks such as type checking.

We extract the R and N for a given program using standard techniques from denotational semantics. We first define expression evaluation, then define statement execution, then finally define how each program generates an R and N. The semantics uses a local variable function $l : \texttt{LV} \rightarrow O$, which gives the values of the local variables for the evaluation of expressions. We assume the existence of the undefined object $\perp \in O$.

We start with an inductive definition of the expression evaluation function $ev : \texttt{X} \times I \times L \rightarrow O \cup A \cup S$. $ev(\texttt{x}, i, l)$ is the value of the expression x in the context of the instance variable function i and local variable function l. Figure 4 contains a standard recursive definition of *ev*.

We continue with the definition of the statement execution function $ex : \texttt{ST} \times I \times L \times S \rightarrow I \times L \times S$. $ex(\texttt{st}, i, l, s)$ is the tuple $\langle i', l', s'\rangle$, where i' is the new instance variable function, l' is the new local variable function and s' is the new sequence of applied operations that result from executing the statement st in the instance variable function i, local variable function l and sequence of applied operations s. Figure 5 contains a standard recursive definition of *ex*.

We finally define the function *anpr*, which extracts the receiver effect function R and invoked operation function N from the program. $anpr(\texttt{pr}, R, N)$ is the tuple $\langle R', N'\rangle$, where R' is the receiver effect function and N' is the invoked operation function denoted by the program pr in the context of the receiver effect function R and invoked operation function N. Definitions 6 and 7 define two auxiliary functions. Figure 6 contains the definition of *anpr*.

Definition 6 $\forall \texttt{u} \in \texttt{LV} : l_\perp(\texttt{u}) = \perp$

Definition 7 $l_{ro} = l_\perp[\texttt{r} \mapsto r, \texttt{p} \mapsto o]$

$$ev(\mathtt{c}, i, l) = \mathtt{c}$$

$$ev(\mathtt{v}, i, l) = i(\mathtt{v})$$

$$ev(\mathtt{u}, i, l) = l(\mathtt{u})$$

$$ev(\mathtt{ex}_1\mathtt{+}\mathtt{ex}_2, i, l) = ev(\mathtt{ex}_1, i, l)\mathtt{+}ev(\mathtt{ex}_2, i, l)$$

$$ev(\mathtt{-ex}, i, l) = \mathtt{-}ev(\mathtt{ex}, i, l)$$

$$ev(\mathtt{if(ex,ex_1,ex_2)}, i, l) = \begin{cases} ev(\mathtt{ex}_2, i, l) & \text{if } ev(\mathtt{ex}, i, l) = 0 \\ ev(\mathtt{ex}_1, i, l) & \text{otherwise} \end{cases}$$

$$ev(\mathtt{if(cx,mx)}, i, l) = \begin{cases} \epsilon & \text{if } ev(\mathtt{ex}, i, l) = 0 \\ ev(\mathtt{mx}_1, i, l) & \text{otherwise} \end{cases}$$

$$ev(\mathtt{ex}_1\mathtt{->op}(\mathtt{ex}_2), i, l) = ev(\mathtt{ex}_1, i, l)\mathtt{->op}(ev(\mathtt{ex}_2, i, l))$$

$$ev(\mathtt{mx}\circ\mathtt{s}, i, l) = ev(\mathtt{mx}, i, l) \circ ev(\mathtt{s}, i, l)$$

$$ev(\epsilon, i, l) = \epsilon$$

Figure 4: Definition of *ev*

$$\frac{ex(\mathtt{st}_1, i, l, s) = \langle i_1, l_1, s_1\rangle, \quad ex(\mathtt{st}_2, i_1, l_1, s_1) = \langle i', l', s'\rangle}{ex(\mathtt{st}_1\mathtt{;}\,\mathtt{st}_2, i, l, s) = \langle i', l', s'\rangle}$$

$$\frac{i' = i[\mathtt{v}\mapsto ev(\mathtt{ex}, i, l)]}{ex(\mathtt{v=ex}, i, l, s) = \langle i', l, s\rangle}$$

$$\frac{l' = l[\mathtt{u}\mapsto ev(\mathtt{ex}, i, l)]}{ex(\mathtt{u=ex}, i, l, s) = \langle i, l', s\rangle}$$

$$\frac{s' = s \circ ev(\mathtt{mx}, i, l)}{ex(\mathtt{mx}, i, l, s) = \langle i, l, s'\rangle}$$

$$\frac{}{ex(\epsilon, i, l, s) = \langle i, l, s\rangle}$$

Figure 5: Definition of *exec*

$$\frac{\begin{array}{c}\forall r, o \in O, i \in I, \mathtt{op'} \in \mathtt{OP}: \\ R'(r\mathtt{->op'}(o), i) = \begin{cases} i' & \text{if } \mathtt{op'} = \mathtt{op} \\ R(r\mathtt{->op'}(o), i) & \text{otherwise} \end{cases} \\ N'(r\mathtt{->op'}(o), i) = \begin{cases} s' & \text{if } \mathtt{op'} = \mathtt{op} \\ N(r\mathtt{->op'}(o), i) & \text{otherwise} \end{cases} \\ \text{where } \langle i', l', s'\rangle = ex(\mathtt{st}, i, l_{ro}, \epsilon)\end{array}}{anpr(\mathtt{op(parm)\{st\}pr}, R, N) = anpr(\mathtt{pr}, R', N')}$$

$$\frac{}{anpr(\epsilon, R, N) = \langle R, N\rangle}$$

Figure 6: Definition of *anpr*

6 Analysis

Programmers define operations by writing methods. Each operation corresponds to a method invocation: to execute an operation, the machine executes the code in the corresponding method. The commutativity analysis algorithm determines if operations commute by analyzing the corresponding methods.

6.1 Overview

Recall that if an operation $r\mathtt{->op}(o)$ meets the conditions of Lemmas 2 and 3, it is possible to execute the operation in parallel without violating the serial semantics. Given a method, the compiler statically checks that all invocations of the method meet the conditions. If so, the compiler can generate parallel code.

The static checking algorithm has two major phases. Given a method with name `op`, the first phase computes a conservative, finitely representable approximation to $gen(m, r\mathtt{->op}(o))$ that is valid for all possible values of m, r and o. The second phase applies the commutativity testing conditions in Definition 3 to every pair of operations in the conservative approximation of $gen(m, r\mathtt{->op}(o))$ that may have the same receiver. If all of these pairs meet the commutativity testing conditions, it possible to execute $r\mathtt{->op}(o)$ in parallel for any r and o regardless of the memory m in which the operation executes. In this case the compiler can generate parallel code for the corresponding method.

To apply the commutativity testing conditions the compiler must represent and reason about both the new values of the receiver's instance variables and the multiset of operations invoked when two operations execute. The compiler represents these new values and invoked operations using syntactic expressions. To check if two method invocations commute, the compiler first uses symbolic execution to extract the relevant expressions under the two execution orders. It then compares corresponding expressions from the two different execution orders, using an expression simplification algorithm to facilitate the comparison. If the corresponding expressions are equivalent, the two method invocations satisfy the commutativity testing conditions.

6.2 Expression Extraction and Symbolic Execution

The symbolic analysis represents values using the syntactic category of expressions `X` defined in Section 5. It uses instance variable functions $\mathtt{i} : \mathtt{I} = \mathtt{IV} \to \mathtt{EX}$ and local variable functions $\mathtt{l} : \mathtt{L} = \mathtt{LV} \to \mathtt{EX}$ (here `LV` is a set of local variable names) to implement the symbolic variable bindings computed during the analysis. The symbolic expression evaluation function $\mathtt{ev} : \mathtt{X} \times \mathtt{I} \times \mathtt{L} \to \mathtt{X}$ and the symbolic statement execution function $\mathtt{ex} : \mathtt{ST} \times \mathtt{I} \times \mathtt{L} \times \mathtt{S} \to \mathtt{I} \times \mathtt{L} \times \mathtt{S}$ implement the basic symbolic execution. $\mathtt{ev}(\mathtt{x},\mathtt{i},\mathtt{l})$ is the expression denoting the value of the expression x when evaluated in the context of the instance variable function i and local variable function l. $\mathtt{ex}(\mathtt{st},\mathtt{i},\mathtt{l},\mathtt{s})$ is the tuple $\langle \mathtt{i'},\mathtt{l'},\mathtt{s'} \rangle$, where where i′ is the new instance variable function, l′ is the new local variable function and s′ is the new sequence of method invocation expressions that denote the sequence of methods invoked as a result of executing the statement st in the context of the instance variable function i, local variable function l and sequence of method invocation expressions s. Figure 7 contains the definition of ev and Figure 8 contains the definition of ex.

The program analysis function anpr analyzes a program to extract the symbolic receiver effect function $\mathtt{R} : \mathtt{OP} \to \mathtt{I}$ and symbolic invoked operation function $\mathtt{N} : \mathtt{OP} \to \mathtt{S}$. $\mathtt{R}(\mathtt{op})(\mathtt{v})$ is the expression that denotes the new value of the instance variable v after op executes. $\mathtt{N}(\mathtt{op})$ is the sequence of method invocation expressions that denotes the sequence of operations invoked as a result of executing op. The free variables in these expressions are r (the receiver), p (the parameter), constants and the instance variables of the receiver. An instance variable v represents the value of v before op executes. Figure 9 contains the definition of anpr.

$$\mathtt{ev(c,i,l)} = \mathtt{c}$$

$$\mathtt{ev(v,i,l)} = \mathtt{i(v)}$$

$$\mathtt{ev(u,i,l)} = \mathtt{l(u)}$$

$$\mathtt{ev(-ex,i,l)} = \mathtt{-ev(ex,i,l)}$$

$$\mathtt{ev(ex_1+ex_2,i,l)} = \mathtt{ev(ex_1,i,l)+ev(ex_2,i,l)}$$

$$\mathtt{ev(if(ex,x_1,x_2),i,l)} = \mathtt{if(ev(ex,i,l),ev(x_1,i,l),ev(x_2,i,l))}$$

$$\mathtt{ev(if(ex,mx),i,l)} = \mathtt{if(ev(ex,i,l),ev(mx,i,l))}$$

$$\mathtt{ev(ex_1\text{->}op(ex_2),i,l)} = \mathtt{ev(ex_1,i,l)\text{->}op(ev(ex_2,i,l))}$$

$$\mathtt{ev(mx{\circ}s,i,l)} = \mathtt{ev(mx,i,l){\circ}ev(s,i,l)}$$

$$\mathtt{ev}(\epsilon,\mathtt{i,l}) = \epsilon$$

Figure 7: The Definition of ev

$$\frac{\mathtt{ex(st_1,i,l,s)} = \langle \mathtt{i_1,l_1,s_1} \rangle,\quad \mathtt{ex(st_2,i_1,l_1,s_1)} = \langle \mathtt{i',l',s'} \rangle}{\mathtt{ex(st_1;st_2,i,l,s)} = \langle \mathtt{i',l',s'} \rangle}$$

$$\frac{\mathtt{i'} = \mathtt{i[v{\mapsto}ev(ex,i,l)]}}{\mathtt{ex(v=ex,i,l,s)} = \langle \mathtt{i',l,s} \rangle}$$

$$\frac{\mathtt{l'} = \mathtt{l[u{\mapsto}ev(ex,i,l)]}}{\mathtt{ex(u=ex,\ i,l,s)} = \langle \mathtt{i,l',s} \rangle}$$

$$\frac{\mathtt{s'} = \mathtt{s{\circ}ev(mx,i,l)}}{\mathtt{ex(mx,i,l,s)} = \langle \mathtt{i,l,s'} \rangle}$$

$$\frac{}{\mathtt{ex}(\epsilon,\mathtt{i,l,s}) = \langle \mathtt{i,l,s} \rangle}$$

Figure 8: The Definition of ex

$$\frac{\langle \mathtt{i',l',s'} \rangle = \mathtt{ex(st,\ Id,\ Id,}\ \epsilon),\quad \mathtt{R'} = \mathtt{R[op{\mapsto}i']},\quad \mathtt{N'} = \mathtt{N[op{\mapsto}s']}}{\mathtt{anpr(op(p)\{st\}pr,R,N)} = \mathtt{anpr(pr,R',N')}}$$

$$\frac{}{\mathtt{anpr}(\epsilon,\mathtt{R,N}) = \langle \mathtt{R,N} \rangle}$$

Figure 9: The Definition of anpr

6.3 Correspondence Between Program Semantics and Symbolic Analysis

We next establish the correspondence between the program semantics and the symbolic analysis. The main result is a theorem that identifies several conditions on the expressions generated by the symbolic analysis. If the expressions for two methods meet these conditions, all invocations of the two methods are guaranteed to commute.

We start with several lemmas that establish the correspondence between the symbolic analysis functions ev, ex and anpr and the program semantics functions *ev*, *ex* and *anpr*. The lemmas are readily proved via structural induction.

Lemma 4

$$\forall \mathtt{v} \in \mathtt{IV} : ev(\mathtt{i}(\mathtt{v}), i_0, l_0) = i(\mathtt{v}) \textit{ and}$$
$$\forall \mathtt{u} \in \mathtt{LV} : ev(\mathtt{l}(\mathtt{u}), i_0, l_0) = l(\mathtt{u})$$

implies

$$\forall \mathtt{x} \in \mathtt{X} : ev(\mathtt{ev}(\mathtt{x},\mathtt{i},\mathtt{l}), i_0, l_0) = ev(\mathtt{x}, i, l)$$

Lemma 5

$$\forall \mathtt{v} \in \mathtt{IV} : ev(\mathtt{i}(\mathtt{v}), i_0, l_0) = i(\mathtt{v}) \text{ and}$$
$$\forall \mathtt{u} \in \mathtt{LV} : ev(\mathtt{l}(\mathtt{u}), i_0, l_0) = l(\mathtt{v}) \text{ and}$$
$$ex(\mathtt{st}, i, l, s) = \langle i', l', s' \rangle \text{ and}$$
$$\mathtt{ex}(\mathtt{st}, \mathtt{i}, \mathtt{l}, \mathtt{s}) = \langle \mathtt{i}', \mathtt{l}', \mathtt{s}' \rangle$$

implies

$$\forall \mathtt{v} \in \mathtt{IV} : ev(\mathtt{i}'(\mathtt{v}), i_0, l_0) = i'(\mathtt{v}) \text{ and}$$
$$\forall \mathtt{u} \in \mathtt{LV} : ev(\mathtt{l}'(\mathtt{u}), i_0, l_0) = l'(\mathtt{u}) \text{ and}$$
$$ev(\mathtt{s}', i_0, l_0) = s'$$

Lemma 6

$$\langle R', N' \rangle = anpr(\mathtt{op\,(p)\,\{st\}}, R, N) \text{ and}$$
$$\langle \mathtt{R}', \mathtt{N}' \rangle = \mathtt{anpr}(\mathtt{op\,(p)\,\{st\}}, \mathtt{R}, \mathtt{N})$$

implies

$$\forall o, r \in O, i \in I :$$
$$R'(r\text{->}\mathtt{op}(o), i) = \lambda \mathtt{v}.ev(\mathtt{R}'(\mathtt{op})(\mathtt{v}), i, l_{ro}) \text{ and}$$
$$N'(r\text{->}\mathtt{op}(o), i) = ev(\mathtt{N}'(\mathtt{op}), i, l_{ro})$$

Lemma 7

$$\forall o, r \in O, \mathtt{op} \in \mathtt{OP}, i \in I :$$
$$R(r\text{->}\mathtt{op}(o), i) = \lambda \mathtt{v}.ev(\mathtt{R}(\mathtt{op})(\mathtt{v}), i, l_{ro}) \text{ and}$$
$$N(r\text{->}\mathtt{op}(o), i) = ev(\mathtt{N}(\mathtt{op}), i, l_{ro}) \text{ and}$$
$$\langle R', N' \rangle = anpr(\mathtt{pr}, R, N) \text{ and}$$
$$\langle \mathtt{R}', \mathtt{N}' \rangle = \mathtt{anpr}(\mathtt{pr}, \mathtt{R}, \mathtt{N})$$

implies

$$\forall o, r \in O, \mathtt{op} \in \mathtt{OP}, i \in I :$$
$$R'(r\text{->}\mathtt{op}(o), i) = \lambda \mathtt{v}.ev(\mathtt{R}'(\mathtt{op})(\mathtt{v}), i, l_{ro}) \text{ and}$$
$$N'(r\text{->}\mathtt{op}(o), i) = ev(\mathtt{N}'(\mathtt{op}), i, l_{ro})$$

Given a program and initial receiver effect and method invocation functions, `anpr` and *anpr* extract the new receiver effect and method invocation functions that correspond to the symbolic and semantic meanings of the program in the context of the initial functions. The next two definitions specify the standard initial receiver effect and method invocation functions for `anpr` and *anpr*. Corollary 2 uses these definitions to characterize the relationship between the symbolic analysis and program semantics starting from the standard initial functions.

Definition 8 $\forall o, r \in O, \mathtt{op} \in \mathtt{OP}, i \in I : R_0(r\text{->}\mathtt{op}(o), i) = i$ *and* $N_0(r\text{->}\mathtt{op}(o), i) = \epsilon$

Definition 9 $\forall \mathtt{op} \in \mathtt{OP} : \mathtt{R}_0(\mathtt{op}) = \mathtt{Id}$ *and* $\mathtt{N}_0(\mathtt{op}) = \epsilon$

Corollary 2

$$\langle R, N \rangle = anpr(\mathtt{pr}, R_0, N_0) \text{ and}$$
$$\langle \mathtt{R}, \mathtt{N} \rangle = \mathtt{anpr}(\mathtt{pr}, \mathtt{R}_0, \mathtt{N}_0)$$

implies

$$\forall o, r \in O, \mathtt{op} \in \mathtt{OP}, i \in I :$$
$$R(r\text{->}\mathtt{op}(o), i) = \lambda \mathtt{v}.ev(\mathtt{R}(\mathtt{op})(\mathtt{v}), i, l_{ro}) \text{ and}$$
$$N(r\text{->}\mathtt{op}(o), i) = ev(\mathtt{N}(\mathtt{op}), i, l_{ro})$$

We next turn our attention to symbolic expressions that denote the new values of instance variables and invoked methods after the execution of two operations. Because the new value may depend on the parameters of the two operations, we introduce two new variables $\mathtt{p}_1$ and $\mathtt{p}_2$. $\mathtt{p}_1$ denotes the parameter of the first operation and $\mathtt{p}_2$ denotes the parameter of the second operation.

Given a symbolic receiver effect function `R` and symbolic invoked operation function `N`, the expression denoting the new value of an instance variable `v` after the symbolic execution of first $\mathtt{r\text{->}op_1(p_1)}$ then $\mathtt{r\text{->}op_2(p_2)}$ is $\mathtt{ev}(\mathtt{R}(\mathtt{op}_2)(\mathtt{v})[\mathtt{p}_2/\mathtt{p}], \mathtt{R}(\mathtt{op}_1), \mathtt{Id})[\mathtt{p}_1/\mathtt{p}]$. The symbolic expression for the sequence of invoked methods is $\mathtt{ev}(\mathtt{N}(\mathtt{op}_2)[\mathtt{p}_2/\mathtt{p}], \mathtt{R}(\mathtt{op}_1), \mathtt{Id})[\mathtt{p}_1/\mathtt{p}]$. Here the notation $\mathtt{x}[\mathtt{x}_1/\mathtt{x}_2]$ stands for `x` with all occurrences of $\mathtt{x}_2$ replaced by $\mathtt{x}_1$. According to the program semantics, the new value of `v` after first the execution of $r\text{->}\mathtt{op}_1(o_1)$ then the execution of $r\text{->}\mathtt{op}_2(o_2)$ is $R(r\text{->}\mathtt{op}_2(o_2), R(r\text{->}\mathtt{op}_1(o_1), i))(\mathtt{v})$. The sequence of invoked operations is $N(r\text{->}\mathtt{op}_2(o_2), R(r\text{->}\mathtt{op}_1(o_1), i))$. The next lemma proves the correspondence between the expressions from the symbolic analysis and the program semantics for the execution of two operations.

Lemma 8

$$\langle R, N \rangle = anpr(\mathtt{pr}, R_0, N_0) \text{ and}$$
$$\langle \mathtt{R}, \mathtt{N} \rangle = \mathtt{anpr}(\mathtt{pr}, \mathtt{R}_0, \mathtt{N}_0)$$

implies

$$R(r\text{->}\mathtt{op}_2(o_2), R(r\text{->}\mathtt{op}_1(o_1), i)) =$$
$$\lambda \mathtt{v}.ev(\mathtt{ev}(\mathtt{R}(\mathtt{op}_2)(\mathtt{v})[\mathtt{p}_2/\mathtt{p}], \mathtt{R}(\mathtt{op}_1), \mathtt{Id})[\mathtt{p}_1/\mathtt{p}]), i, l_{r12})$$
$$\text{and}$$
$$N(r\text{->}\mathtt{op}_2(o_2), R(r\text{->}\mathtt{op}_1(o_1), i)) =$$
$$ev(\mathtt{ev}(\mathtt{N}(\mathtt{op}_2)[\mathtt{p}_2/\mathtt{p}], \mathtt{R}(\mathtt{op}_1), \mathtt{Id})[\mathtt{p}_1/\mathtt{p}], i, l_{r12}),$$
$$\text{where } l_{r12} = l_\perp[\mathtt{r} \mapsto r, \mathtt{p}_1 \mapsto o_1, \mathtt{p}_2 \mapsto o_2]$$

Proof We first establish the identity for the receiver effect functions R and `R`:

$$\lambda \mathtt{v}.ev(\mathtt{ev}(\mathtt{R}(\mathtt{op}_2)(\mathtt{v})[\mathtt{p}_2/\mathtt{p}], \mathtt{R}(\mathtt{op}_1), \mathtt{Id})[\mathtt{p}_1/\mathtt{p}]), i, l_{r12}) =$$
$$\lambda \mathtt{v}.ev(\mathtt{ev}(\mathtt{R}(\mathtt{op}_2)(\mathtt{v})[\mathtt{p}_2/\mathtt{p}], \mathtt{R}(\mathtt{op}_1), \mathtt{Id}), i, l_{ro2}),$$
$$\text{where } l_{ro2} = l_\perp[\mathtt{r} \mapsto r, \mathtt{p} \mapsto o_1, \mathtt{p}_2 \mapsto o_2]$$

We next apply Lemma 4, with $\mathtt{ex} = \mathtt{R}(\mathtt{op}_2)(\mathtt{v})[\mathtt{p}_2/\mathtt{p}]$, $\mathtt{i} = \mathtt{R}(\mathtt{op}_1)$, $\mathtt{l} = \mathtt{Id}$, $i_0 = i$, $l_0 = l_{ro2}$, $i = R(r\text{->}\mathtt{op}_1(o_1), i)$ and $l = l_{ro2}$:

$$\lambda \mathtt{v}.ev(\mathtt{ev}(\mathtt{R}(\mathtt{op}_2)(\mathtt{v})[\mathtt{p}_2/\mathtt{p}], \mathtt{R}(\mathtt{op}_1), \mathtt{Id}), i, l_{ro2}) =$$
$$\lambda \mathtt{v}.ev(\mathtt{R}(\mathtt{op}_2)(\mathtt{v})[\mathtt{p}_2/\mathtt{p}], R(r\text{->}\mathtt{op}_1(o_1), i), l_{ro2}) =$$
$$\lambda \mathtt{v}.ev(\mathtt{R}(\mathtt{op}_2)(\mathtt{v}), R(r\text{->}\mathtt{op}_1(o_1), i), l_{ro})$$

We next apply Corollary 2 with $\mathtt{op} = \mathtt{op}_2$ and $i = R(r\text{->}\mathtt{op}_1(o_1), i)$:

$$\lambda \mathtt{v}.ev(\mathtt{R}(\mathtt{op}_2)(\mathtt{v}), R(r\text{->}\mathtt{op}_1(o_1), i), l_{ro}) =$$
$$R(r\text{->}\mathtt{op}_2(o_2), R(r\text{->}\mathtt{op}_1(o_1), i))$$

The proof of equality for the invoked operation functions N and $\mathtt{N}$ is similar:

$$\begin{aligned}&ev(\mathtt{ev}(\mathtt{N}(\mathtt{op}_2)[\mathtt{p}_2/\mathtt{p}], \mathtt{R}(\mathtt{op}_1), \mathtt{Id})[\mathtt{p}_1/\mathtt{p}], i, l_{r12}) = \\ &ev(\mathtt{ev}(\mathtt{N}(\mathtt{op}_2)[\mathtt{p}_2/\mathtt{p}], \mathtt{R}(\mathtt{op}_1), \mathtt{Id}), i, l_{ro2}) = \\ &ev(\mathtt{N}(\mathtt{op}_2)[\mathtt{p}_2/\mathtt{p}], R(r\text{->}\mathtt{op}_1(o_1), i), l_{ro2}) = \\ &ev(\mathtt{N}(\mathtt{op}_2), R(r\text{->}\mathtt{op}_1(o_1), i), l_{ro}) = \\ &N(r\text{->}\mathtt{op}_2(o_2), R(r\text{->}\mathtt{op}_1(o_1), i))\end{aligned}$$

The following lemma simply applies the definition of $\Rightarrow$ twice to characterize the effect of executing two operations on the same object.

Lemma 9

$$\langle m, \{r\text{->}\mathtt{op}_1(o_1)\}\rangle \Rightarrow \langle m_1, p_1\rangle \textit{ and}$$
$$\langle m_1, \{r\text{->}\mathtt{op}_2(o_2)\}\rangle \Rightarrow \langle m_{12}, p_{12}\rangle$$

implies

$$\begin{aligned}&m_{12} = m[r \mapsto R(r\text{->}\mathtt{op}_2(o_2), R(r\text{->}\mathtt{op}_1(o_1), m(r)))] \\ &\textit{and } p_1 = \textit{s2p}(N(r\text{->}\mathtt{op}_1(o_1), m(r))) \textit{ and} \\ &p_{12} = \textit{s2p}(N(r\text{->}\mathtt{op}_2(o_2), R(r\text{->}\mathtt{op}_1(o_1), m(r))))\end{aligned}$$

Definition 10 establishes the notion of equivalence for expressions. This definition is used in Definition 11, which extends the notion of commutativity to operation names.

Definition 10 $\mathtt{x}_1 \approx \mathtt{x}_2$ *if* $\forall i \in I, l \in L : ev(\mathtt{x}_1, i, l) = ev(\mathtt{x}_2, i, l)$

Definition 11 $\mathtt{op}_1$ *and* $\mathtt{op}_2$ *commute if*

$$\begin{aligned}&\forall \mathtt{v} \in \mathtt{IV}: \\ &\quad \mathtt{ev}(\mathtt{R}(\mathtt{op}_2)(\mathtt{v})[\mathtt{p}_2/\mathtt{p}], \mathtt{R}(\mathtt{op}_1), \mathtt{Id})[\mathtt{p}_1/\mathtt{p}] \approx \\ &\quad \mathtt{ev}(\mathtt{R}(\mathtt{op}_1)(\mathtt{v})[\mathtt{p}_1/\mathtt{p}], \mathtt{R}(\mathtt{op}_2), \mathtt{Id})[\mathtt{p}_2/\mathtt{p}] \textit{ and} \\ &\mathtt{s2p}(\mathtt{N}(\mathtt{op}_1)[\mathtt{p}_1/\mathtt{p}]) \uplus \\ &\quad \mathtt{s2p}(\mathtt{ev}(\mathtt{N}(\mathtt{op}_2)[\mathtt{p}_2/\mathtt{p}], \mathtt{R}(\mathtt{op}_1), \mathtt{Id})[\mathtt{p}_1/\mathtt{p}]) \approx \\ &\mathtt{s2p}(\mathtt{N}(\mathtt{op}_2)[\mathtt{p}_2/\mathtt{p}]) \uplus \\ &\quad \mathtt{s2p}(\mathtt{ev}(\mathtt{N}(\mathtt{op}_1)[\mathtt{p}_1/\mathtt{p}], \mathtt{R}(\mathtt{op}_2), \mathtt{Id})[\mathtt{p}_2/\mathtt{p}])\end{aligned}$$

The following theorem establishes that if two operations satisfy the conditions in Definition 11, then all invocations of the methods commute. Because the conditions in Definition 11 only depend on the expressions extracted by the symbolic analysis, the theorem establishes the foundation for the use of equational reasoning to determine if operations commute.

Theorem 2 *If* $\mathtt{op}_1$ *and* $\mathtt{op}_2$ *commute then* $\forall r, o_1, o_2 \in O$: $r\text{->}\mathtt{op}_1(o_1)$ *and* $r\text{->}\mathtt{op}_2(o_2)$ *commute.*

6.4 Expression Simplification and Comparison

The expression comparison algorithm used to check the commutativity testing conditions consists of several different phases. These phases reduce the expressions to a simplified form, then apply a simple recursive isomorphism test to determine if the two expressions always denote the same value.

The first phase of the simplification algorithm applies simple scalar rewrite rules. It first distributes conditionals out of condition, scalar and array expressions using distributive rules such as $\mathtt{ex+if(ex,ex_1,ex_2)} = \mathtt{if(ex,ex+ex_1,ex+ex_2)}$.

The algorithm then applies standard arithmetic and logical rewrite rules such as distributive rules of the form $\mathtt{-(ex_1+ex_2)} = \mathtt{(-ex_1+-ex_2)}$ and simplification rules such as $\mathtt{--ex} = \mathtt{ex}$. When none of these rules can be applied to the current expression, the simplifier applies rules such as $\mathtt{((ex_1+ex_2)+ex_3)} = \mathtt{(ex_1+ex_2+ex_3)}$ and $\mathtt{(ex_1+(ex_2+ex_3))} = \mathtt{(ex_1+ex_2+ex_3)}$ that convert binary applications of commutative and associative operators into n-ary applications. It then sorts the operands according to an arbitrary, recursively defined total order on expressions. This sort facilitates the eventual comparison by making it easier to identify isomorphic operands of commutative and associative operators.

The last phase of the simplification algorithm builds a *condition table* for each expression. This table enables the equality testing algorithm to use a simple isomorphism test even for expressions containing conditionals. Each condition table contains the maximal condition-free subexpressions of the original expression. Each subexpression is stored under an index which consists of a conjunction of basic terms. If a subexpression is stored under a given index, it denotes the value of the original expression when all of the basic terms in the index are true. The algorithm builds the table by recursively traversing the outer conditional expressions to identify the minimal conjunctions of basic terms that select each maximal conditional-free subexpression as the value of the original expression. It is possible to further simplify the table using logic minimization techniques as proposed in [7].

To compare two expressions for equality the algorithm performs a simple recursive isomorphism test. The algorithm checks that the condition tables have the same indexes and that corresponding subexpressions in the table are isomorphic.

6.5 Execution Time and Limitations

In the worst case the expression manipulation algorithms may take exponential running time, although in practice we do not expect this worst-case behavior to prevent the application of commutativity analysis. Other researchers have successfully applied similar expression manipulation techniques in other analysis contexts[4]. Even with exponential running time, there are some expressions that always denote the same value but do not have the same simplified form. Given that it is, in general, undecidable to determine if two operations commute [11], such imprecision is an inherent limitation of any expression manipulation algorithm.

Another limitation of the presented algorithms is that they require commuting operations to leave the receiver object in identical states in both execution orders. This restriction is obviously too conservative: all that is required is that the operations leave the object in a state that is equivalent with respect to the remainder of the computation. Recognizing operations that satisfy this relaxed form of commutativity would expand the range of programs that the compiler could parallelize. In practice we have found that the current techniques work well for computations that traverse data structures. The more advanced techniques may be required to effectively parallelize a range of computations that build data structures.

6.6 Invoked Operations

Given a method, the compiler traverses the call graph to compute a conservative approximation to the set of methods invoked as a result of executing a given method. It constructs the set of potentially invoked methods by repeatedly choosing one of the methods in the set and adding all of the methods that it could potentially invoke to the set until it reaches a fixed point. In this section we formalize the algorithm described above. We first define notation for subsets of operation names: $\texttt{o} \subseteq \texttt{OP}$. We then define a function `invk` that, given a sequence of method invocation expressions, extracts the names of all of the potentially invoked operations. Figure 10 contains the definition of this function.

$$\begin{gathered}\texttt{invk}(\texttt{ex}_1\texttt{->op}(\texttt{ex}_2)) = \{\texttt{op}\}\\ \texttt{invk}(\texttt{if(ex,mx)}) = \texttt{invk}(\texttt{mx})\\ \texttt{invk}(\texttt{mx} \circ \texttt{s}) = \texttt{invk}(\texttt{mx}) \cup \texttt{invk}(\texttt{s})\\ \texttt{invk}(\epsilon) = \emptyset\end{gathered}$$

Figure 10: The Definition of `invoked`

We next define the notion of a set of operation names covering a set of operations. Given a method, the analysis algorithm will construct a set of operation names that covers the set of all operations potentially invoked as a result of executing the method.

Definition 12 $\texttt{o}$ *covers* s *if* $\forall r\texttt{->op}(o) \in s : \texttt{op} \in \texttt{o}$.

Lemma 10 *If* $\langle R, N\rangle = anpr(\texttt{pr}, R_0, N_0)$ *and* $\langle \texttt{R}, \texttt{N}\rangle = \texttt{anpr}(\texttt{pr}, \texttt{R}_0, \texttt{N}_0)$ *then* $\forall r, o \in O, i \in I, \texttt{op} \in \texttt{OP} : \texttt{invk}(\texttt{N}(\texttt{op}))$ *covers* $N(r\texttt{->op}(o), i)$.

The analysis algorithm uses a standard iterative algorithm to compute a set of operation names that covers the set of potentially invoked operations. The algorithm maintains a set of the names of potentially invoked methods. At every step it traverses methods whose names are in this set to discover the names of any additional potentially invoked methods whose names are not yet in the set. The $\leadsto$ function defined below specifies what happens at each iteration. To make the algorithm more efficient, it splits the set of names of potentially invoked methods into two sets: a set that was discovered in the last iteration, and a set that was discovered before the last iteration.

Definition 13 $\leadsto$ *is the function on* $\texttt{OP} \times \texttt{OP}$ *defined by:*

$$\frac{\begin{array}{l}\texttt{o}_3 = \texttt{o}_1 \cup \texttt{o}_2,\\ \texttt{o}_4 = \bigcup_{\texttt{op} \in \texttt{o}_2} \{\texttt{op}' \in \texttt{invk}(\texttt{N}(\texttt{op})) : \texttt{op}' \notin \texttt{o}_1 \cup \texttt{o}_2\}\end{array}}{\langle \texttt{o}_1, \texttt{o}_2\rangle \leadsto \langle \texttt{o}_3, \texttt{o}_4\rangle}$$

We next present several lemmas and corollaries that establish several properties that will be used to show that repeated applications of the $\leadsto$ function compute a set of operation names that cover the set of potentially invoked operations. We also show that repeated applications of the $\leadsto$ function will eventually reach a fixed point.

Lemma 11 *If* $\langle \texttt{o}_1, \texttt{o}_2\rangle \leadsto \langle \texttt{o}_3, \texttt{o}_4\rangle$ *then* $\texttt{o}_1 \cup \texttt{o}_2 \subseteq \texttt{o}_3 \cup \texttt{o}_4$.

Lemma 12 *If* $\forall \texttt{op} \in \texttt{o}_1 : \texttt{invk}(\texttt{N}(\texttt{op})) \subseteq \texttt{o}_1 \cup \texttt{o}_2$ *and* $\langle \texttt{o}_1, \texttt{o}_2\rangle \leadsto \cdots \leadsto \langle \texttt{o}_3, \texttt{o}_4\rangle$ *then* $\forall \texttt{op} \in \texttt{o}_3 : \texttt{invk}(\texttt{N}(\texttt{op})) \subseteq \texttt{o}_3 \cup \texttt{o}_4$.

Corollary 3 *If* $\langle \emptyset, \texttt{o}\rangle \leadsto \cdots \leadsto \langle \texttt{o}_3, \texttt{o}_4\rangle$ *then* $\forall \texttt{op} \in \texttt{o}_3 : \texttt{invk}(\texttt{N}(\texttt{op})) \subseteq \texttt{o}_3 \cup \texttt{o}_4$.

Lemma 13 $\forall \texttt{op} \in \texttt{OP} \exists \texttt{o} \subseteq \texttt{OP} : \langle \emptyset, \{\texttt{op}\}\rangle \leadsto \cdots \leadsto \langle \texttt{o}, \emptyset\rangle$.

Because $\leadsto$ is a function, the $\texttt{o}$ for each $\texttt{op}$ is unique. We next define `gen(op)`, which computes the names of the operations potentially invoked as a result of executing an operation with name $\texttt{op}$.

Definition 14 *If* $\langle \emptyset, \{\texttt{op}\}\rangle \leadsto \cdots \leadsto \langle \texttt{o}, \emptyset\rangle$ *then* `gen(op)=o`.

We next establish the correspondence between `gen(op)` and $r\texttt{->op}(o)$.

Definition 15 $\langle m, s\rangle \Rightarrow_0 \langle m, s\rangle$. *If* $\langle m_1, s_1\rangle \Rightarrow_n \langle m_2, s_2\rangle$ *and* $\langle m_2, s_2\rangle \Rightarrow \langle m_3, s_3\rangle$, *then* $\langle m_1, s_1\rangle \Rightarrow_{n+1} \langle m_3, s_3\rangle$.

Definition 16 $\langle \texttt{o}_1, \texttt{o}_2\rangle \leadsto_0 \langle \texttt{o}_1, \texttt{o}_2\rangle$. *If* $\langle \texttt{o}_1, \texttt{o}_2\rangle \leadsto_n \langle \texttt{o}_3, \texttt{o}_4\rangle$ *and* $\langle \texttt{o}_3, \texttt{o}_4\rangle \leadsto \langle \texttt{o}_5, \texttt{o}_6\rangle$ *then* $\langle \texttt{o}_1, \texttt{o}_2\rangle \leadsto_{n+1} \langle \texttt{o}_5, \texttt{o}_6\rangle$.

Lemma 14 *If* $\langle m_0, s_0\rangle \Rightarrow \langle m_1, s_1\rangle$ *and* $\texttt{o}$ *covers* s_0 *and* $\langle \emptyset, \texttt{o}\rangle \leadsto \langle \texttt{o}_1, \texttt{o}_2\rangle$ *then* $\texttt{o}_1 \cup \texttt{o}_2$ *covers* s_1.

Lemma 15 *If* $\langle m_0, s_0\rangle \Rightarrow_n \langle m_n, s_n\rangle$ *and* $\circ$ *covers* s_0 *and* $\langle\emptyset, \circ\rangle \rightsquigarrow_n \langle \circ_1, \circ_2\rangle$ *then* $\circ_1 \cup \circ_2$ *covers* s_n.

Corollary 4 *If* $\langle\emptyset, \circ\rangle \rightsquigarrow_n \langle \circ_1, \circ_2\rangle$ *and* $\langle m_0, s_0\rangle \Rightarrow_i \langle m_i, s_i\rangle$ *and* $i \leq n$ *then* $\circ_1 \cup \circ_2$ *covers* s_i.

Theorem 3 $\forall r, o \in O, m \in M, \texttt{op} \in \texttt{OP} : \texttt{gen(op)}$ *covers gen*$(m, r\texttt{->op}(o))$.

6.7 Commutativity Testing

Given `gen(op)` for the method named `op`, the compiler can test that all pairs of potentially invoked methods commute. For each pair of methods whose names are in `gen(op)`, the compiler generates the expressions for the new instance variables and multiset of invoked methods as described in Section 6.2. It then simplifies the expressions and applies the commutativity testing conditions by comparing corresponding expressions for equality as described in Section 6.4. As an optimization it can first compare the unsimplified expressions, only simplifying if that comparison fails. If all pairs of invoked methods commute the compiler can generate parallel code.

6.8 Code Generation

The compiler determines which computations to attempt to parallelize by traversing the call graph of the program. At each method invocation site the compiler attempts to parallelize the computation generated by the execution of the method invoked at that site. Once the compiler encounters a parallelizable method, it does not traverse the subgraph rooted at the invocation site of the method.

If it is possible to parallelize the method the compiler generates two methods: a parallel method and a wrapper method that invokes the parallel method then waits until the computation has finished. The parallel method directly invokes the parallel versions of all of the invoked methods. The compiler also augments the data structure declaration of the receiver object with a mutual exclusion lock. The parallel method uses the lock to ensure that it executes atomically with respect to other methods that access the receiver. It acquires the lock before the first access to the receiver, then releases the lock before the first operation invocation site. The compiler may generate multiple lock release sites; only one will be executed per invocation. Figure 2 contains an example of the generated code.

7 Experimental Results

We have implemented a prototype parallelizing compiler that uses commutativity analysis as its basic analysis paradigm. The compiler uses several analysis techniques to extend the model of computation significantly beyond the basic model of computation presented in Section 3. It also performs several code generation optimizations that significantly extend the code generation strategy discussed in Section 6.8. We present a complete treatment of both of these extensions elsewhere [18, 6].

We used the compiler to automatically parallelize two applications: the Barnes-Hut hierarchical N-body code[2] and Water, which evaluates forces and potentials in a system of water molecules in the liquid state. We briefly present several performance results; we provide a more complete description of the applications and the experimental methodology elsewhere [18].

Figure 11 presents the speedup curve for the Barnes-Hut on two input data sets; this graph plots the running time of the sequential version running with no parallelization overhead divided by the the running time of the automatically parallel version as a function of the number of processors executing the parallel computation. The primary limiting factor on the speedup is the fact that the compiler does not parallelize one of the phases of the computation; as the number of processors grows that phase becomes a bottleneck.

Figure 12 presents the speedup curve for Water running on two input data sets. The limiting factor on the speedup is contention for shared objects updated by multiple operations.

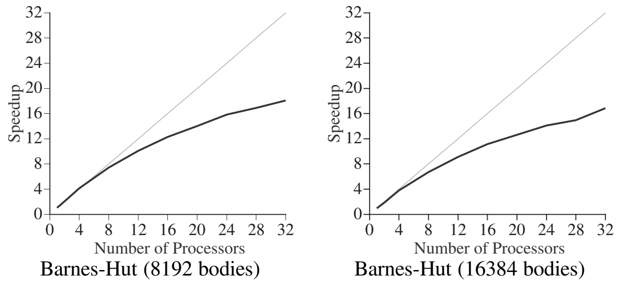

Figure 11: Speedup for Barnes-Hut

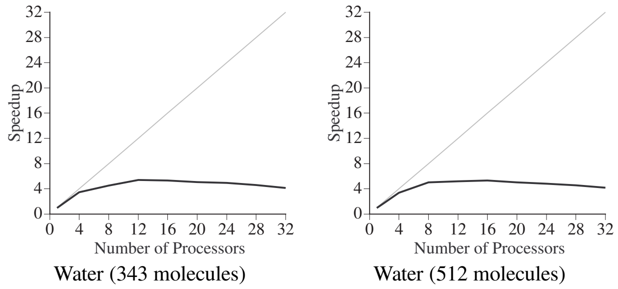

Figure 12: Speedup for Water

8 Related Work

Other researchers have recognized the value of including support for commuting operations in parallel computing systems[23, 3, 22]. These systems, however, focus on exploiting commuting operations and rely on some external mechanism, typically the programmer, to specify when the operations actually commute. The techniques presented in this paper, on the other hand, automatically detect commuting operations.

Several existing compilers can recognize when a loop performs a reduction of many values into a single value[7, 15, 5]. These compilers recognize when the reduction primitive (typically addition) is associative. They then exploit this algebraic property to eliminate the data dependence associated with the serial accumulation of values into the result. The generated program computes the reduction in parallel. Commutativity analysis can also recognize when operations commute because of the algebraic properties of the employed primitives. But unlike reduction analysis, commutativity analysis does not rely solely on these properties. We present a set of general conditions that allows the compiler to recognize commuting operations even when the operations employ no primitives with special algebraic properties. Knowledge of the algebraic properties therefore enhances the effectiveness of, rather than enables, the presented techniques.

Research performed in the context of database concurrency control has shown that it is possible to take advantage of commuting operations to increase the amount of concurrency available in transaction processing systems[24]. The presented approach is based on using abstract specifications of the behavior of the operations to expose the commutativity. It is the responsibility of the programmer to ensure that the actual implementation preserves the semantics of the specification. The techniques presented in this paper, on the other hand, automatically detect commuting operations given the implementation of the operations and do not rely on higher-level specifications of their behavior.

Programmers who parallelize computations by hand have always used commuting operations. For example, four (Water, MP3D, LocusRoute and Cholesky) of the six parallel applications in the SPLASH benchmark suite [21] and three of the four parallel applications described in [20] violate the data dependences of the original serial program and rely on commuting operations for their correct execution. While the techniques presented in this paper fail to recognize many of the commuting operations in these programs, [4] this applications experience does show how important it is to support commuting operations.

[4]The primary reason is that, as discussed in Section 6.5, the compiler currently requires commuting operations to leave the receiver object in identical states rather than relaxing the condition to allow commuting operations to leave the object in states that are equivalent with respect to the rest of the computation.

9 Conclusion

Existing parallelizing compilers all preserve the data dependences of the original serial program. We believe that this strategy is too conservative: compilers must recognize and exploit commuting operations if they are to effectively parallelize a range of applications. This paper presents a general algorithm that can statically recognize and exploit commuting operations to automatically generate parallel code. It states a general set of conditions that is sufficient to guarantee commutativity and proves that if operations meet these conditions then their executions commute. It also presents a static analysis algorithm that uses symbolic execution and expression manipulation to determine if operations commute and shows how a parallelizing compiler can exploit commuting operations to generate parallel code. Finally, it presents experimental results from two complete scientific applications that were successfully and automatically parallelized by a prototype compiler that uses commutativity analysis as its basic analysis paradigm. Both applications exhibit good parallel performance.

References

[1] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[2] J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, pages 446–449, December 1976.

[3] P. Barth, R. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture*, pages 538–568. Springer-Verlag, August 1991.

[4] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 357–363, Santa Barbara, CA, April 1995.

[5] D. Callahan. Recognizing and parallelizing bounded recurrences. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[6] P. Diniz and M. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. Technical Report TRCS96-07, Dept. of Computer Science, University of California at Santa Barbara, May 1996.

[7] A. Fisher and A. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, Orlando, FL, June 1994.

[8] W.L. Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, October 1989.

[9] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[10] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[11] O. Ibarra, P. Diniz, and M. Rinard. On the complexity of commutativity analysis. Technical Report TRCS95-18, Dept. of Computer Science, University of California at Santa Barbara, October 1995.

[12] D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[13] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[14] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, June 1990.

[15] S. Pinter and R. Pinter. Program optimization and parallelization using idioms. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*, Orlando, FL, January 1991.

[16] M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford, CA, 1994.

[17] M. Rinard and P. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the SIGPLAN '96 Conference on Program Language Design and Implementation*, Philadelphia, PA, May 1996. (An extended version of this document can be found at the authors' web address: http://www.cs.ucsb.edu/$\sim${martin,pedro}).

[18] M. Rinard and P. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. Technical Report TRCS96-08, Dept. of Computer Science, University of California at Santa Barbara, May 1996.

[19] M. Rinard and P. Diniz. Commutativity analysis: A technique for automatically parallelizing pointer-based computations. In *Proceedings of the 10th International Parallel Processing Symposium*, Honolulu, HI, April 1996.

[20] D. Scales and M. S. Lam. An efficient shared memory system for distributed memory machines. Technical Report CSL-TR-94-627, Computer Systems Laboratory, Stanford University, July 1994.

[21] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[22] J. Solworth and B. Reagan. Arbitrary order operations on trees. In *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Portland, OR, August 1993.

[23] G. Steele. Making asynchronous parallelism safe for the world. In *Proceedings of the Seventeenth Annual ACM Symposium on the Principles of Programming Languages*, pages 218–231, San Francisco, CA, January 1990.

[24] W. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.