On the Complexity of Commutativity Analysis

Oscar Ibarra, Pedro Diniz and Martin Rinard
{ibarra,pedro,martin}@cs.ucsb.edu
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106

Abstract

Two operations commute if they generate the same result regardless of the order in which they execute. Commutativity is an important property — commuting operations enable significant optimizations in the fields of parallel computing, optimizing compilers, parallelizing compilers and database concurrency control. Algorithms that statically decide if operations commute can be an important component of systems in these fields because they enable the automatic application of these optimizations. In this paper we define the commutativity decision problem and establish its complexity for a variety of basic instructions and control constructs. Although deciding commutativity is, in general, undecidable or computationally intractable, we believe that efficient algorithms exist that can solve many of the cases that arise in practice.

1 Introduction

Program analysis has been widely used to extract program properties of interest. In this paper we focus on a simple property between two program operations – commutativity. We say two operations A and B, each composed of a sequence of basic instructions, commute if they generate the same result regardless of the order in which they execute. Knowledge of commuting operations is of practical significance. In the context of optimizing compilers commuting program transformations can be used to reduce the search space for the optimal program transformation sequence, hence reducing the algorithmic complexity of the compiler optimization algorithms [22]. In the context of parallel computing commuting operations enable concurrent execution because they can execute in any order without changing the final result [23, 21]. Parallelizing compilers that recognize commuting operations can exploit this property to automatically generate parallel code for computations that consist only of commuting operations [20]. In the area of databases exploiting commuting operations can improve the performance of concurrency control algorithms by increasing the amount of concurrency in the transaction schedule [24].

This broad range of applications motivates the design of static analysis techniques capable of automatically detecting commuting operations — commutativity analysis. In this paper we focus on the theoretical aspects of commutativity analysis. We identify classes of programs for which commutativity analysis is undecidable, PSPACE-hard, NP-hard, polynomial, and probabilistically polynomial-time decidable (see [11] for definitions and motivations). For some cases we also show the class of programs to be complete for the corresponding complexity class. The results presented here rely on known complexity results from the area of theoretical computer science. They serve two purposes. First, they formally establish the complexity of commutativity analysis. Second, they should make researchers working in more applied areas aware of the inherent limitations of any commutativity analysis algorithm.

Although we show that commutativity analysis is, in the general case, undecidable or computationally intractable, we believe that it is possible to develop algorithms that successfully recognize many of the commuting operations that occur in practice. We believe that it is possible to recognize many common cases with simple algorithms that execute very quickly. When these algorithms fail, we believe that more complex algorithms with poor worst-case execution times may still execute efficiently for many of the cases that occur in practice. We therefore believe that it is feasible to develop effective practical algorithms that statically detect commuting operations.

The rest of the paper is structured as follows. In Section 2 we formally state the commutativity problem. In Section 3 we present the complexity results for commutativity analysis. In Section 4 we present practical algorithms that test for specific sufficient conditions for commutativity. A hierarchy of tests is described with varying degrees of accuracy and run-time complexity. In Section 5 we briefly describe the applications of commutativity analysis to the area of parallelizing compilers. We conclude in Section 6.

2 Problem Formulation

Commutativity analysis is designed primarily for programs written using a pure object-based paradigm. Such programs structure the computation as a sequence of operations on objects. Each object implements its state using a set of instance variables. Each operation consists of a sequence of basic instructions from the underlying language. Each operation has a receiver object; when the operation executes, its basic instructions can read or write the instance variables of the receiver and temporary variables.

Definition 1 *(Object Equivalence)*
Given two objects R_1 and R_2 with the same instance variables, we say that R_1 and R_2 have the same state if the value of each instance variable is the same in both objects.

The complexity of commutativity analysis depends on the instructions used in the operations. We will use the term $\mathcal{C}$-operation to denote the class of operations over the set of constructs $\mathcal{C}$. The commutativity problem is thus defined for any two $\mathcal{C}$-operations over a specific set $\mathcal{C}$ of instructions.

Definition 2 *($\mathcal{C}$-operation commutativity problem)*
Given two $\mathcal{C}$-operations O_1 and O_2 with the same receiver object, we say that O_1 and O_2 commute if the result of executing first O_1 then O_2 always leaves the receiver in the same state as executing first O_2 then O_1.

For convenience of the application of theoretical complexity results we recast commutativity between two $\mathcal{C}$-operations as the commutativity problem between two $\mathcal{C}$-programs P_1 and P_2 defined over the same set $\mathcal{C}$ of constructs. These programs, however, make use of two I/O instructions.

Definition 3 *($\mathcal{C}$-programs)*
A $\mathcal{C}$-program consists of a sequence of non-I/O instructions over the set $\mathcal{C}$ augmented by two `input` *and* `output` *instructions. If the program uses input values, the first instruction in the sequence is the unique program input statement* `input(`$x_1, \cdots x_r$`)`, *where $x_1, \cdots x_r$ are the program input variables. The last instruction in the sequence is the unique program output statement* `output(`x_i`)` *where x_i is one of the program's input variables, possibly modified.*

The set $\mathcal{C}$ of instructions defines the expressive power of the $\mathcal{C}$-programs. We defer the complete definition of the non-I/O constructs for each class of $\mathcal{C}$-programs to Section 3. For the time being the reader may consider control and arithmetic constructs such as $x \leftarrow x \oplus y$, where $\oplus$ is a binary operator, and constructs such as $x \leftarrow \ominus y$, where $\ominus$ is a unary operator. Besides the traditional arithmetic operators like addition and multiplication we will include in some sets $\mathcal{C}$ the following operators: monus or proper subtraction (denoted by $\dot{-}$), $\div$ as rational division, integer division with truncation (/) and the unary operator $sign(x) = 1$ if x is positive; else 0. Built-in arithmetic constants include 1 and 0. Other arithmetic constants can be computed by a finite sequence of instructions. Input variables assume values over an input domain $\mathcal{D}$ having a null element, the zero value. Possible input domains include, $\mathcal{R}$ - the set of all rational numbers, $\mathcal{Z}$ - the set of all integers, and $\mathcal{N}$ the set of all nonnegative integers.

Each $\mathcal{C}$-program may also use temporary variables, initially set to zero. Input variables and auxiliary variables (i.e., non-input variables) constitute the program variables.

Definition 4 *($\mathcal{C}$-program commutativity problem)*
Let P_1 and P_2 be $\mathcal{C}$-programs with input variables $x_1, \ldots, x_r$ and $y_1, \ldots, y_r$, respectively. Consider the program $P_1; P_2$, that is, P_1 followed by P_2. (Note that technically, $P_1; P_2$ is no longer a $\mathcal{C}$-program since it has two input statements and two output statements. It needs $2r$ input values and produces two output values.) We say that P_1 and P_2 commute if for all input assignments to $x_1, \ldots, x_r$, $y_1 = x_1, \ldots, y_r = x_r$, $P_1; P_2$ produces the same results as $P_2; P_1$. The (non) commutativity problem for $\mathcal{C}$-programs is the problem of deciding for two $\mathcal{C}$-programs P_1 and P_2 whether they (do not) commute. Note that P_1 and P_2 commute if and only if they are equivalent.

We now show the reduction from the $\mathcal{C}$-program commutativity problem to the $\mathcal{C}$-operations commutativity problem.

Theorem 1 *There is a linear-time algorithm that converts any two $\mathcal{C}$-programs P_1 and P_2 to $\mathcal{C}$-operations O_1 and O_2 such that P_1 and P_2 are equivalent if and only if O_1 and O_2 commute.*

Proof: Given the two $\mathcal{C}$-programs with the same number of input variables $x_1 \cdots x_r$ the transducer generates two operations O_1 and O_2 over objects with $r+1$ instance variables $x_1, \cdots, x_r, y$. The transducer copies the code from programs P_1 and P_2 to the operations O_1 and O_2, respectively, performing the following transformations. The input statement (if any) is replaced by a sequence of instructions that saves the object instance variables to temporary variables $t_1, \cdots, t_r$. The output statement is replaced by an instruction that assigns y to the $\mathcal{C}$-program output variable, followed by a sequence that uses the temporary variables to restore the original values of object instance variables. By construction the two operations only modify the instance variable y. The transformation preserves the original values of instance variables $x_1, \cdots, x_r$ after the operation's execution. It is thus clear that if programs P_1 and P_2 commute then the corresponding operations O_1 and O_2 commute. On the other hand if operations commute, i.e. under both execution orders the instance variable y always has the same final value, then it must be the case that programs P_1 and P_2 commute. In addition the transformation can be done in linear time with respect to the length of the input $\mathcal{C}$-programs as the transformation only adds a constant amount of instructions to the length of the input $\mathcal{C}$-program. □

We can hereafter refer to the commutativity of two $\mathcal{C}$-programs P_1 and P_2 as the commutativity of the two corresponding $\mathcal{C}$-operations O_1 and O_2.

3 Complexity Results

In this section we present the complexity results for the commutativity problem. In what follows and as mentioned in the previous section, we refer to the commutativity problem between two programs P_1 and P_2 as the commutativity problem between the two corresponding operations O_1 and O_2.

For each class of $\mathcal{C}$-programs we define the instruction set $\mathcal{C}$ and domain of program input variable values $\mathcal{D}$ for which the result holds. All complexity results are with respect to the size (i.e., length) of the longer of the two $\mathcal{C}$-programs being tested for commutativity.

The undecidable/intractable results we present are the best possible we can prove at present - the programs make use of "very restricted" arithmetic/control instructions and a limited numbers of input and auxiliary variables. The results are stated without proofs as the proof techniques are similar to the ones used for proving undecidability/intractability of the equivalence problem for programs cited in the references.

In this section we will make use of the two programs, hereafter named ONE and ZERO. ONE always outputs the value 1 and is defined as $input(x); x \leftarrow 1; output(x)$. ZERO always outputs the value 0 and is defined as $input(x); x \leftarrow 0; output(x)$.

3.1 (Un-)decidable Problems

The idea behind the proofs of the undecidability results in this subsection is an intricate reduction of Hilbert's Tenth Problem to the commutativity problem. The proofs are similar to the ones in [14] for proving the undecidability of program equivalence.

> **Result #1.** Let $\mathcal{C} = \{x \leftarrow 1, x \leftarrow x + y, x \leftarrow x/y\}$ and input domain $\mathcal{D} = \mathcal{Z}$. It is undecidable to determine, given a $\mathcal{C}$-program P with three input variables and nine auxiliary variables, whether it commutes with ONE. We say, in this case, that *commutativity with ONE is undecidable.* The result holds even if we restrict the problem to only programs P that compute total 0/1-functions (i.e., programs with output range $\{0, 1\}$ that are defined for all inputs). The result is also valid when the input domain $\mathcal{D} = \mathcal{N}$, but at present we can only give a proof for the case when the numbers of input and auxiliary variables are ten and four, respectively.

If, in the above $\mathcal{C}$, $x \leftarrow x + y$ is replaced by $x \leftarrow x - y$, the number of input variables of P can be reduced to two (the number of auxiliary variables is nine), and the result holds for both input domains $\mathcal{D} = \mathcal{Z}$ and $\mathcal{D} = \mathcal{N}$. This is the best possible, since it can be shown that if $\mathcal{C} = \{x \leftarrow 1, x \leftarrow x + y, x \leftarrow x - y, x \leftarrow x * y, x \leftarrow x/y\}$, the commutativity problem for $\mathcal{C}$-programs with one input variable (but unrestricted number of auxiliary variables) is decidable.

> **Result #2.** Let $\mathcal{C} = \{x \leftarrow 1, x \leftarrow x - y, x \leftarrow x * y, x \leftarrow x/2\}$. Over $\mathcal{D} = \mathcal{Z}$ or $\mathcal{D} = \mathcal{N}$, commutativity with ONE is undecidable for $\mathcal{C}$-programs with nine input variables and three auxiliary variables. On the other hand, if $\mathcal{C} = \{x \leftarrow 1, x \leftarrow x+y, x \leftarrow x*y, x \leftarrow x/d \mid d \geq 2\}$, commutativity with ONE (for programs with unrestricted numbers of input and auxiliary variables) is decidable.

> **Result #3.** Let $\mathcal{C} = \{x \leftarrow 1, x \leftarrow x+y, x \leftarrow x-y, x \leftarrow x*y, x \leftarrow x \div y\}$. Over $\mathcal{D} = \mathcal{Z}$, the commutativity problem is decidable for $\mathcal{C}$-programs that compute total functions. When one of the programs being tested for commutativity is an arbitrary program (i.e., it may or may not be defined for all inputs), commutativity with ONE is undecidable. See the next result.

Result #4. Let $\mathcal{C} = \{x \leftarrow 1, x \leftarrow x + y, x \leftarrow x \div y\}$. Over $\mathcal{D} = \mathcal{Z}$, commutativity with ONE is undecidable for $\mathcal{C}$-programs with 28 input variables and three auxiliary variables. In contrast, and even for $\mathcal{C} = \{x \leftarrow 0, x \leftarrow 1, x \leftarrow x + y, x \leftarrow x * y, x \leftarrow x \div y\}$, the commutativity problem for $\mathcal{C}$-programs (with an unrestricted number of input and auxiliary variables) is decidable when the input domain is $\mathcal{D} = \mathcal{N}$. This latter result is not true when the instruction $x \leftarrow x + y$ is replaced by $x \leftarrow x - y$ since it can be shown that if $\mathcal{C} = \{x \leftarrow 1, x \leftarrow x - y, x \leftarrow x \div y\}$, commutativity with ONE is undecidable for $\mathcal{C}$-programs with nine input variables and three auxiliary variables over input domain $\mathcal{D} = \mathcal{N}$.

Result #5. Let $\mathcal{C} = \{x \leftarrow 1, x \leftarrow y, x \leftarrow x \dot{-} y, x \leftarrow x/y\}$. Over $\mathcal{D} = \mathcal{N}$, commutativity of $\mathcal{C}$-programs with two input variables and six auxiliary variables is undecidable. However, the problem becomes decidable if one of the programs being considered computes a total function (the programs may have arbitrary numbers of input and auxiliary variables).

If, in the above, $x \leftarrow 1$ is replaced by $x \leftarrow x + 1$, we get

Result #6. Let $\mathcal{C} = \{x \leftarrow x + 1, x \leftarrow y, x \leftarrow x \dot{-} y, x \leftarrow x/y\}$. Over $\mathcal{D} = \mathcal{N}$, commutativity with ONE is undecidable for $\mathcal{C}$-programs with two input variables and seven auxiliary variables. The result holds even if we consider only programs that compute total 0/1-functions.

Result #7. Let $\mathcal{C} = \{x \leftarrow 1, x \leftarrow x \dot{-} y, x \leftarrow x * y\}$. Over $\mathcal{D} = \mathcal{N}$, commutativity with ONE is undecidable for $\mathcal{C}$-programs with nine input variables and three auxiliary variables.

Suppose we are only interested in deciding whether two programs commute over a limited range of inputs, and not for all inputs. Consider, e.g., the case when the input domain $\mathcal{D} = \{0\}$; this is equivalent to saying that all program variables are initially 0 (i.e., there is no input variable). The problems are now called *commutativity over zero input* and *commutativity with ONE over zero input.*

The following result follows from the undecidability of the halting problem for 2-counter machines which can simulate Turing machines.

Result #8. Let $\mathcal{C} = \{x \leftarrow 0, x \leftarrow x+1, x \leftarrow x \dot{-} 1, \textit{if } x = 0 \textit{ goto } t\}$. Commutativity with ONE over zero input is undecidable for two-variable $\mathcal{C}$-programs. Here t denotes a label; note that a program halts if and only if it executes the output statement, which is the last instruction in the program.

3.2 Nonelementary Recursive Problems

Result #8 relies on the fact that the program being tested for commutativity with ONE may not halt. Suppose we only consider programs that always halt. Then the (unique) output can clearly be evaluated; hence, commutativity with ONE over zero input is decidable. However, its complexity is enormous. The proof of the next result uses the time hierarchy theorem for Turing machines. Let $f_1(n)$, $f_2(n)$ and $f_3(n)$ be functions on the positive integers defined as follows: $f_1(n) = 2n$ and for $i = 1, 2$ $f_{i+1}(n) = f_i^{(n)}(n)$ where $f_i^{(n)}$ is the n^{th} fold composition of f_i. Note that $f_3(n) \geq 2^{\cdot^{\cdot^{2^n}}}$ (n levels of 2's) is nonelementary recursive. A function $g(n)$ is elementary recursive if $g(n) \leq 2^{\cdot^{\cdot^{2^k}}}$ (k levels of 2's for some fixed k).

In the next result the $\mathcal{C}$-programs are allowed to have *do* loops of the form *do* x $\cdots$ *end*. In this context we assume the loop variable x to be bound upon the loop entrance, hence guaranteeing loop termination.

Result #9. Let $\mathcal{C} = \{x \leftarrow 1, x \leftarrow x + y, x \leftarrow x \dot{-} y, do\ x\ \cdots\ end\}$. There are rational constants c and d such that the time complexity of testing commutativity with ONE over zero input for $\mathcal{C}$-programs with no nesting of loops has lower and upper bounds of $f_3(cn)$ and $f_3(dn)$, respectively where n is the program length. The lower bound holds even for five-variable programs [17].

If we replace $x \leftarrow x + y$ and $x \leftarrow x \dot{-} y$ by simpler constructs, we can show the following:

Result #10. Let $\mathcal{C} = \{x \leftarrow 0, x \leftarrow x + 1, x \leftarrow x \dot{-} 1, x \leftarrow y, if\ x = 0\ then\ goto\ t, goto\ t, do\ x\ \cdots\ end\}$. Commutativity over zero input is polynomial-time decidable for $\mathcal{C}$-programs with no nesting of loops, where t denotes a "forward" label not in the scope of any do-loop (do-statements however can be labeled) [10].

3.3 PSPACE-Hard Problems

If in result #10 we allow "forward" labels to be inside the do-loops, then the problem becomes PSPACE-hard. In fact, we can show the following by coding the computation of a deterministic linear-bounded automaton.

Result #11. Let $\mathcal{C} = \{x \leftarrow 0, x \leftarrow x + 1, if\ x = 0\ then\ y \leftarrow 0, if\ x = 0\ then\ y \leftarrow 1, do\ x\ \cdots\ end\}$. Commutativity with ONE over zero input is PSPACE-hard for $\mathcal{C}$-programs with no nesting of loops. This result also holds for $\mathcal{C} = \{x \leftarrow 0, x \leftarrow x+1, x \leftarrow x \dot{-} 1, if\ x = 0\ then\ y \leftarrow 1, do\ x\ \cdots\ end\}$ [17].

Note that if $\mathcal{C} = \{x \leftarrow 0, x \leftarrow x + 1, x \leftarrow x \dot{-} 1, x \leftarrow y, if\ x = 0\ then\ y \leftarrow z, do\ x\ \cdots\ end\}$ commutativity over zero input for $\mathcal{C}$-programs with no nesting of loops is in PSPACE. This follows from the observation that if P is a program and n is the length of P, then during the execution of P, no variable in P can have value greater than 2^n. Hence the computation of P can be simulated using $O(n)$ bits of memory.

3.4 NP-Hard Problems

The proofs of the NP-hardness results in this subsection use a rather complex encoding of the satisfiability problem for Boolean expressions and the fact that this latter problem is NP-hard.

Result #12. Let $\mathcal{C} = \{x \leftarrow 2x, x \leftarrow x/2, x \leftarrow x + y\}$ and input domain $\mathcal{D} = \mathcal{Z}$. The commutativity problem for $\mathcal{C}$-programs with one input variable and one auxiliary variable is NP-hard. The result also holds when $x \leftarrow x + y$ is replaced by $x \leftarrow x - y$ [18].

Commutativity, nevertheless, is decidable. In fact, consider the following set $\mathcal{C}^+$ of instructions: $\mathcal{C}^+ = \{x \leftarrow 0, x \leftarrow c, x \leftarrow x * c, x \leftarrow x/c, x \leftarrow x + y, x \leftarrow x - y, skip\ t, if\ p(x, y)\ then\ skip\ t\}$ where c denotes any positive integer (different c's can be used in the body of a program), t is any nonnegative integer, and $p(x, y)$ is a predicate of the form $x > y$, $x \geq y$, $x = y$, $x \neq y$, $x \leq y$, or $x < y$. $skip\ t$ causes the $(t + 1)^{st}$ instruction following the current instruction to be executed next. It can be shown that the non-commutativity problem for $\mathcal{C}^+$-programs is in NP [18]. Thus commutativity can be decided in exponential time.

Next, we have

Result #13. Let $\mathcal{C} = \{x \leftarrow 0, x \leftarrow x/2, x \leftarrow x - y\}$ and input domain $\mathcal{D} = \mathcal{Z}$. Commutativity with ZERO for $\mathcal{C}$-programs with one input variable and two auxiliary variables is NP-hard [18].

The above result is the best possible since it can be shown that commutativity with ZERO is decidable in polynomial time for $\mathcal{C}$-programs with two program variables (both variables may also be input variables) for $\mathcal{C} = \{x \leftarrow 0, x \leftarrow c, x \leftarrow -c, x \leftarrow x*c, x \leftarrow x/c, x \leftarrow x+c, x \leftarrow x-c, x \leftarrow x-y\}$ [18].

> **Result #14.** Let $\mathcal{C} = \{x \leftarrow 1, x \leftarrow x+y, x \leftarrow x \dot{-} y\}$. Over $\mathcal{D} = \mathcal{N}$, commutativity with ZERO is NP-hard for $\mathcal{C}$-programs with one input variable and two auxiliary variables [15].

One can strengthen this result by requiring the use of only one auxiliary variable, but the instruction set is now $\mathcal{C} = \{x \leftarrow 1, x \leftarrow 2x, x \leftarrow x+1, x \leftarrow x+y, x \leftarrow x \dot{-} y, x \leftarrow y \dot{-} x\}$. This is true also for the class $\mathcal{C} = \{x \leftarrow 1, x \leftarrow 2x, x \leftarrow x+y, x \leftarrow x \dot{-} y, x \leftarrow sign(x)\}$.

Considering now commutativity of programs over a limited range of inputs, we get NP-hardness, even for a very simple class of codes.

> **Result #15.** Let $\mathcal{C} = \{x \leftarrow 1, x \leftarrow y+z, x \leftarrow y-z, x \leftarrow y*z\}$. Over $\mathcal{D}$ = *finite* subset of $\mathcal{Z}$ with at least two elements, commutativity with ZERO is NP-hard for $\mathcal{C}$-programs [13].

Note that the non-commutativity problem for the above programs is clearly in NP.

We also have the following result.

> **Result #16.** Let $\mathcal{C} = \{x \leftarrow x*c, x \leftarrow x/2\}$ and input domain $\mathcal{D} = \mathcal{N}$. Consider only $\mathcal{C}$-programs with one program variable, which is also the input/output variable. Such a program computes a simple arithmetic expression of the form $xT_1T_2\cdots T_k$, where x is an integer variable and each T_i is a multiplication by a positive integer constant or integer division by 2. It is NP-hard to decide, given two $\mathcal{C}$-programs P_1 and P_2 and a positive integer m, whether P_1 and P_2 commute for all non-negative integer values of $x < m$ [16].

In this result, the complexity is with respect to the maximum of the length of P_1, length of P_2 , and length of the binary representation of m. In contrast, and rather counter-intuitive, when we do not restrict the range of the inputs for checking commutativity, we have:

> **Result #17.** Let $\mathcal{C} = \{x \leftarrow x*c, x \leftarrow x/2\}$ and input domain $\mathcal{D} = \mathcal{N}$. There is a polynomial-time algorithm to decide, given two $\mathcal{C}$-programs P_1 and P_2 whether P_1 and P_2 commute. [16]

3.5 Probabilistic Polynomial-Time Problems

If in result #15, $\mathcal{D}$ is the set of all integers or the set of all rational numbers, commutativity is probabilistically decidable in polynomial time. This means that there is a polynomial-time algorithm which uses a random number generator to decide if two given programs P_1 and P_2 commute. If the algorithm outputs *yes*, then P_1 and P_2 probably commute with probability of error $\leq 1/2$. If the algorithm outputs *no*, then P_1 and P_2 do not commute for sure. Clearly, a probability of error $\leq 1/2^k$ may be obtained by running the algorithm k times. Since it is conjectured that no NP-hard problem can be solved in probabilistic polynomial time, this result contrasts that of result #15 and has the same flavor as result #17.

> **Result #18.** Let $\mathcal{C} = \{x \leftarrow 1, x \leftarrow y+z, x \leftarrow y-z, x \leftarrow y*z\}$. Over $\mathcal{D} = \mathcal{Z}$ or $\mathcal{D} = \mathcal{R}$, the commutativity problem for C-programs is decidable in probabilistic polynomial time [13].

One can show that, if in result #15, $\mathcal{D}$ has exactly one element, then the commutativity problem is probabilistically decidable in polynomial time [13].

4 Practical Algorithms for Commutativity Analysis

Despite the undecidable/intractable results presented in the previous section, we believe it is possible to develop algorithms that can recognize many of the commuting operations that occur in practice. In this section we present several of these algorithms, each designed to recognize a specific case that we have identified. We expect software systems such as parallelizing compilers that recognize commuting operations to combine these algorithms to recognize many of the cases that occur in practice. These algorithms can be organized into a hierarchy, with fast, relatively simple algorithms used first and the less efficient, more complex algorithms used only if the simple algorithms fail.

4.1 Identical Operations

If both operations are the same they commute because they are indistinguishable. In practice this case arises in graph traversal algorithms.

4.2 Independent Operations

Two operations are independent if no operation writes an instance variable that the other accesses. If two operations are independent then they obviously commute. For straight-line codes it is simple to test in polynomial time if two operations are independent. The algorithm constructs for each method two sets of instance variables, the *read* set and the *write* set. The *read* set consists of the instance variables the method reads; the *write* set consists of the instance variables the method writes. If neither method writes an instance variable the other method either reads or writes, the method are independent. Clearly this algorithm can be implemented in polynomial time with respect to both the method's length and the number of instance variables using set representation techniques [1].

4.3 Reduction

A common operation in many applications is to reduce many values into one accumulator variable by applying a commutative and associative operator such as $+$ or $*$. The code that computes these reductions contains assignment statements of the form $x = x + e$ where e is an expression denoting the accumulated value. Two operations that reduce values into the same accumulator variable commute if neither value depends on the order in which the operations execute. If the two operations meet the following condition then they commute:

- The only assignments are of the form $x = x + e$, where e is an arbitrary expression. There is no data dependence from any variable x that either operation modifies to the accumulated expression e.

4.4 Symbolic Execution

It is possible to increase the ability to detect commuting operations by using a more sophisticated symbolic analysis algorithm. This algorithm symbolically executes the operations A and B in both execution orders to construct two expressions for each instance variable. Each of these two expressions denotes the variable's final value in one of the execution orders. The algorithm then checks if the corresponding expressions denote the same value. If so, the operations commute. This algorithm first applies a set of rewrite rules designed to simplify the expressions. These rules apply standard algebraic properties such as distributivity and associativity. The expression comparison algorithm simply applies a recursive isomorphism test. It is theoretically possible for the application of distributive rewrite rules to generate an exponential increase in the size of the expressions. Despite this worst-case scenario we, do not expect the analysis to exhibit this exponential behavior in practice. Other research that uses related symbolic analysis techniques supports this hypothesis [6].

4.4.1 Straight-line Codes

For straight-line codes the algorithm constructs the instance variable expressions by symbolically executing each statement. At each point in the program each variable is bound to an expression denoting its value at that point. To symbolically execute a statement, the algorithm uses the current set of variable bindings to compute the expression denoting the new value of the assigned variable. When the symbolic execution completes, each variable is bound to an expression that denotes its final value after the execution of both operations.

4.4.2 Conditional Constructs

It is possible to integrate conditional constructs into the symbolic execution framework. When an instance variable's value depends on the flow of control through the operation, the expressions representing its new values contain conditionals. A conditional of the form $\mathtt{if}(\mathtt{cx}, \mathtt{ex}_1, \mathtt{ex}_2)$ denotes the expression $\mathtt{ex}_1$ if $\mathtt{cx}$ is true and $\mathtt{ex}_2$ if $\mathtt{cx}$ is false. The expression equivalence algorithm for two expressions that contain conditionals is more elaborate than the algorithm described in the previous section. The 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 conditional-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 [9].

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.

4.4.3 Array Variables

It is possible to integrate array variables in the symbolic execution framework. An array expression consists of a variable followed by a list of updates. Each update contains an index and a new value and corresponds to an assignment to an array position. For instance the assignment $a[e_1] = e_2$ generates the symbolic update expression $a[e_1 \rightarrow e_2]$. Multiple assignments generate multiple array updates. The array simplification algorithm applies a rule that eliminates redundant updates and two rules that simplify array accesses; Figure 1 presents these rules.

The array simplification algorithm also attempts to sort the update list, using the indexes as the sort key and an arbitrary, recursively defined total order on expressions as the sort order. The following observation is the foundation of the update sorting algorithm:

$$
\begin{array}{lcl}
\mathtt{ax}[\mathtt{ex}_1 \rightarrow \mathtt{ex}_2] \cdots [\mathtt{ex}_3 \rightarrow \mathtt{ex}_4] & = & \mathtt{ax} \cdots [\mathtt{ex}_3 \rightarrow \mathtt{ex}_4] \text{ if } \mathtt{ex}_1 = \mathtt{ex}_3 \\
\mathtt{ax}[\mathtt{ex}_1 \rightarrow \mathtt{ex}_2][\mathtt{ex}_3] & = & \mathtt{ex}_2 \text{ if } \mathtt{ex}_1 = \mathtt{ex}_3 \\
\mathtt{ax}[\mathtt{ex}_1 \rightarrow \mathtt{ex}_2][\mathtt{ex}_3] & = & \mathtt{ax}[\mathtt{ex}_1] \text{ if } \mathtt{ex}_1 \neq \mathtt{ex}_3
\end{array}
$$

Figure 1: Array Expression Simplification Rules

Observation 1 $\mathtt{ax}[\mathtt{ex}_1 \rightarrow \mathtt{ex}_2][\mathtt{ex}_3 \rightarrow \mathtt{ex}_4] = \mathtt{ax}[\mathtt{ex}_5 \rightarrow \mathtt{ex}_6][\mathtt{ex}_7 \rightarrow \mathtt{ex}_8]$ *if*

- $\mathtt{ex}_1 = \mathtt{ex}_3$ *implies* $\mathtt{ex}_3 = \mathtt{ex}_7, \mathtt{ex}_5 = \mathtt{ex}_7, \mathtt{ex}_4 = \mathtt{ex}_8$ *and*
- $\mathtt{ex}_1 \neq \mathtt{ex}_3$ *implies either* $\mathtt{ex}_3 = \mathtt{ex}_7, \mathtt{ex}_1 = \mathtt{ex}_5, \mathtt{ex}_2 = \mathtt{ex}_6, \mathtt{ex}_4 = \mathtt{ex}_8$ *or* $\mathtt{ex}_3 = \mathtt{ex}_5, \mathtt{ex}_1 = \mathtt{ex}_7, \mathtt{ex}_2 = \mathtt{ex}_8, \mathtt{ex}_4 = \mathtt{ex}_6$

Given two array expressions in the form specified by Observation 1 to check for equality, the algorithm first assumes that $\mathtt{ex}_1 = \mathtt{ex}_3$, then attempts to check that $\mathtt{ex}_3 = \mathtt{ex}_7, \mathtt{ex}_5 = \mathtt{ex}_7$, and $\mathtt{ex}_4 = \mathtt{ex}_8$ under this assumption. Before checking the equality conditions, the algorithm first applies any array expression simplification rules enabled by the assumption. It goes through a similar process when it checks the second condition and assumes that $\mathtt{ex}_1 \neq \mathtt{ex}_3$.

The update sorting algorithm attempts to replace adjacent updates whose indexes are not in the sort order with updates whose indexes are in the sort order. The algorithm can replace the updates with any two updates that meet the conditions in Observation 1. Because each update is generated by an assignment to an array element, the algorithm constructs the updates that correspond to executing the assignments in the reverse order. If the assignments commute, the updates meet the conditions in Observation 1 and the algorithm can replace the original unsorted pair of updates with the new sorted pair.

Given an array expression $\mathtt{ax}[\mathtt{ex}_1 \rightarrow \mathtt{ex}_2][\mathtt{ex}_3 \rightarrow \mathtt{ex}_4]$ whose indexes $\mathtt{ex}_1$ and $\mathtt{ex}_3$ are unsorted, the algorithm generates the new array expression $\mathtt{ax}[\mathtt{ex}_5 \rightarrow \mathtt{ex}_6][\mathtt{ex}_7 \rightarrow \mathtt{ex}_8]$, where $\mathtt{ex}_3 = \mathtt{ex}_5, \mathtt{ex}_1 = \mathtt{ex}_7, \mathtt{ex}_6 = \mathtt{ex}_4[\mathtt{ax}/\mathtt{ax}[\mathtt{ex}_1 \rightarrow \mathtt{ex}_2]]$ and $\mathtt{ex}_8 = \mathtt{ex}_2[\mathtt{ax}[\mathtt{ex}_5 \rightarrow \mathtt{ex}_6]/\mathtt{ax}]$. It then checks if $\mathtt{ax}[\mathtt{ex}_1 \rightarrow \mathtt{ex}_2][\mathtt{ex}_3 \rightarrow \mathtt{ex}_4]$ and $\mathtt{ax}[\mathtt{ex}_5 \rightarrow \mathtt{ex}_6][\mathtt{ex}_7 \rightarrow \mathtt{ex}_8]$ meet the conditions in Observation 1. If so, it replaces the original pair of updates with the new pair. When the algorithm can reorder no pair of updates without violating the conditions in Observation 1, we say that the array expression is maximally ordered. The array expression simplification algorithm replaces pairs of updates until the array expression is maximally ordered. The array expression simplification rules in Figure 1 also compare expressions for inequality. The inequality comparison algorithm is currently very simple. It merely checks that the two expressions are equal to two expressions that it has already assumed (as a result of applying the conditions in Observation 1) to be unequal.

5 Applications in Parallelizing Compilers

We have studied and analyzed complete applications whose computations perform multiple commuting updates to the underlying data structures. These applications include the Barnes-Hut [3] hierarchical N-body algorithm and the molecular dynamics code Water [1].

In the Barnes-Hut application, the algorithm maintains and performs multiple traversals on a spatial pointer-based tree data structure. Each such traversal reads data from nodes in the tree and

[1] A FORTRAN language variant of the Water code used in our analysis can be found in the PERFECT Benchmark [4] set of applications under the name of MDG.

generates commuting updates to the body nodes at the leaves. The Water application evaluates forces and potentials in a system of water molecules in the liquid state. The molecules are organized in an array data structure. The algorithm computes all pairs of interactions between the molecules. Each intermolecular interaction performs an accumulation of values in each of the intervening molecule data structures.

Detecting the commuting operations in these applications exposes a vast amount of concurrency. A compiler using commutativity analysis can exploit these opportunities to automatically and effectively parallelize such applications.

6 Conclusions

In this paper we have established the complexity of commutativity analysis. We have shown that, in general, the commutativity problem is undecidable or computationally intractable. Despite this negative general result, we believe that it is possible to develop algorithms that successfully recognize many of the commuting operations that occur in practice. We have outlined several such algorithms with varying degrees of success and run-time complexity. We therefore believe that it is feasible to develop effective practical algorithms that statically detect commuting operations.

References

[1] A. Aho, J. Hopcroft and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1982.

[2] A. Aho, R. Sethi and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

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

[4] M. Berry et al. The PERFECT Club Benchmars: Effective Performance Evaluation of Supercomputers. Technical Report CSRD-827, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May, 1989.

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

[6] W. Blume and R. Eigenmann Symbolic Range Propagation. In *Proceedings of the Ninth IEEE Int. Parallel Processing Symposium*, pp. 357–363, April, 1995

[7] P. Cousot and R. Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on the Principles of Programming Languages*, Los Angeles, CA, January 1977.

[8] P. Diniz. and M. Rinard. Exploiting Commutating Operations in Parallelizing Serial Programs Department of Computer Science, Technical Report TRCS95-11, January, 1995

[9] 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.

[10] E. Gurari and O. Ibarra. The Complexity of the Equivalence Problem for Simple Programs. In *Journal of the ACM*, 28(3):535-560, July 1981.

[11] M. Garey and D. Johnson. *Computer and Intractability. A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[12] 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.

[13] O. Ibarra and S. Moran. Probabilistic Algorithms for Deciding Equivalence of Straight-Line Programs. In *Journal of the ACM*, 30(1):217-228, January 1983.

[14] O. Ibarra and B. Leininger. On the Simplification and Equivalence Problems for Straight-Line Programs. In *Journal of the ACM*, 30(3):641-656, July 1983.

[15] O. Ibarra and B. Leininger. On the Zero-Inequivalence Problem for Loop Programs. In *Journal of Computer and System Sciences* , 26(1):47-64, February 1983.

[16] O. Ibarra, B. Leininger, and S. Moran. On the Complexity of Simple Arithmetic Expressions. In *Theoretical Computer Science* , 19:17-28, 1982.

[17] O. Ibarra, B. Leininger, and L. Rosier. A Note on the Complexity of Program Evaluation. In *Mathematical Systems Theory*, 17:85-96, 1984.

[18] O. Ibarra and B. Leininger. The Complexity of the Equivalence Problem for Simple Loop-Free Programs. In *SIAM Journal on Computing*, 11(1):15-27, February 1982.

[19] M. Lam. Software Pipelining. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1988.

[20] M. Rinard and P. Diniz. Automatically Parallelizing Serial Programs using Commutativity Analysis. (submitted for publication)

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

[22] R. Sethi. Testing for the Church-Rooser Property. In *Journal of the ACM*, 21(4):671-679, October 1974.

[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.

[25] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[26] M. Wolfe. Optimizing Supercompilers for Supercomputers. MIT Press, Cambridge, MA, 1989.