Efficient Predicate Abstraction Using Parallel Assignments for Software Verification Systems

Murray Stokely

Wadham College

University of Oxford

A thesis submitted for the degree of

Master of Science

Summer 2005

Abstract

This thesis shows how formal software verification systems can be improved by utilising parallel assignment in weakest precondition computations.

We begin with an introduction to modern software verification systems. Specifically, the method in which software abstractions are built using counterexample guided abstraction refinement (CEGAR). This approach consists of an iterative application of model construction, model checking, counterexample validation and model refinement steps.

In Chapter 2 parallel assignment constructs are introduced. The classical NP-complete parallel assignment problem is first posed, and then an additional restriction is added to create a special case in which the problem is tractable with an $O(n^2)$ algorithm. The parallel assignment problem is then discussed in the context of weakest precondition computations. In this special situation where statements can be assumed to execute truly concurrently, we show that any sequence of simple assignment statements without function calls can be transformed into an equivalent parallel assignment block.

Results of compressing assignment statements into a parallel form with this algorithm are presented for a wide variety of software applications in Chapter 3. The proposed algorithms were implemented in the ComFoRT Reasoning Framework [19] and used to measure the improvement in the verification of real software applications. This improvement in time proved to be significant for many classes of software.

Acknowledgements

I would like to thank Joel Ouaknine, my supervisor, for accepting me as one of his research students this summer. He has provided me with ideas, inspiration, and encouragement. I would also like to thank Tom Melham for first introducing me to model checking and computer aided formal verification.

Byron Cooke's talk at the Oxford University Computing Laboratory provided my first exposure to predicate abstraction and modern software verification tools. He was later kind enough to suggest fruitful directions of research for this thesis.

Sagar Chaki gave generously of his time to explain aspects of his papers and to integrate the algorithm from chapter two into his ComFoRT reasoning framework. He has nearly been a second supervisor for this work.

Wadham College has been a very enjoyable place to study. I am grateful to the college fellows and the Middle Common Room officers for encouraging the students to pursue their academic goals while also providing for an active graduate social life.

"The next rocket to go astray as a result of a programming language error may not be an exploratory space rocket on a harmless trip to Venus: It may be a nuclear warhead exploding over one of our own cities. An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations."

C.A.R. Hoare
ACM Turing Award Lecture, 1980

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

Modern society is increasingly reliant on the correct and efficient working of computer systems. Software malfunctions in critical computer systems have led to the disclosure of sensitive information, economic and environmental damage, and loss of life. Pioneering computer scientists such as Alan Turing and C.A.R. Hoare recognised the need to formalise programming languages and provide an axiomatic basis for computer programming on which formal correctness proofs can be built [18]. Unfortunately, the systems programming languages in common use today have made few improvements to guarantee the reliability and safety of programs expressed in those languages. As such, the primary methods for validating complex software systems are simulation, testing, and model checking [11].

Simulation and testing involve providing certain inputs as test cases and observing the corresponding output of the simulation or software product. Such tests can be an effective way to find many errors, but it is rarely possible to check all possible cases of interaction and input [11]. Recent research has engendered a new generation of software verification tools that operate directly on general purpose programming languages such as C or Java instead of those written in a restricted modelling language [8]. These tools are characterised by an extended model checking algorithm which interacts with theorem provers and decision procedures to reason about software abstractions.

We are interested in checking that a program respects a set of *temporal safety properties*. Safety properties are those that state “something bad does not happen”. An example is requiring that a lock is never released without first being acquired [3]. Verification of safety properties typically concentrates on the control flow of the program by performing reachability analysis for particular control points [9].

1.1 Model Checking

Model checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. Extremely large state-spaces can often be traversed in minutes. Techniques such as predicate abstraction [17, 9, 2] and partial order reduction [12] allow possibly infinite state systems, such as software applications, to be conservatively modelled by finite-state abstractions. Model checking normally involves an exhaustive search of the state space of the system to determine if some specification is true or not. This technique has been applied to complex real world protocols and application software.

The process of model checking involves several distinct tasks:

- specification;
- modelling;
- verification.

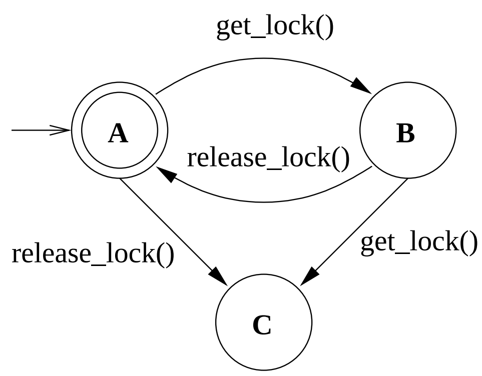


Figure 1.1: LTS for a simple locking protocol.

1.2 Specifications

In order to verify that a program acts according to specification, models of both the specification and the program must be available. Common specifications such as those for communication protocols or system application programming interfaces are often specified in terms of state machines.

Formalisms of state machines such as Kripke structures or *Labelled Transition Systems* (LTS) are naturally employed for such specifications. An LTS is a directed graph denoted by a 4-tuple $T = (S, s_0, \mathcal{L}, \rightarrow)$ with:

- a finite non-empty set of states S;
- an initial state $s_0 \in S$;
- a finite set of actions $\mathcal{L}$;
- and a transition relation $\rightarrow \subseteq S \times \mathcal{L}. \times S$

The edges of an LTS are labelled by elements of L, and as usual we write $s \xrightarrow{a} t$ to mean $(s, a, t) \in \rightarrow$ [24]. An example LTS that could be used to model the behaviour of a locking protocol is shown in Figure 1.1.

Formally, the LTS in Figure 1.1 would be represented as:

$$T_{locking} = \left\{ \begin{array}{l} \{A, B, C\}, \\ A, \\ \{\text{get\_lock}, \text{release\_lock}\}, \\ \{A \xrightarrow{get\_lock} B, B \xrightarrow{release\_lock} A, A \xrightarrow{release\_lock} C, B \xrightarrow{get\_lock} C\} \end{array} \right\}.$$

The initial state is A, and if our locking protocol does not allow double locks or double releases, then state C represents an error state. With this model, it would be natural to perform reachability analysis to determine if a program ever enters state C.

1.2.1 Temporal Logics

Temporal logics are often used to specify system behaviour. They describe the ordering of events in time without introducing time explicitly [11].

1.2.1.1 Linear-time Temporal Logic (LTL)

LTL-formulas over atomic propositions $p_1, \ldots, p_n$ are defined by recursion:

$$\begin{array}{rcl} \phi ::= & p_i & \text{atomic proposition} \\ | & \neg\phi & \text{negation} \\ | & \phi \wedge \psi & \text{conjunction} \\ | & \phi \vee \psi & \text{disjunction} \\ | & \mathbf{X}\phi & \textit{next} \\ | & \phi \; \mathbf{U} \; \psi & \textit{until} \end{array}$$

The next operator, $\mathbf{X}\phi$, intuitively means "ϕ is true *next* time". The until operator, $\phi\ \mathbf{U}\ \psi$, means "ϕ is true *until* eventually ψ is true".

Two additional operators can be built from this definition.

$$\begin{aligned} \mathbf{F}\phi &\equiv \text{true}\ \mathbf{U}\ \phi \\ \mathbf{G}\phi &\equiv \neg(\mathbf{F}\neg\phi) \end{aligned}$$

Where $\mathbf{F}\phi$ intuitively means "ϕ is *eventually* true" and $\mathbf{G}\phi$ means "ϕ is *always* true" [24].

If M is a labelled transition system and Φ is a temporal logic formula, then we say $M \models \Phi$ if every path $x_0x_1x_2\ldots$ through M satisfies Φ. In practice, software verification often centers on the goal of proving that a given error state can never be reached. This is equivalent to $M \models \mathbf{G}p$ for suitable model M and LTL proposition p.

1.3 Software Models

Since a program can, in general, be represented by an infinite-state model, existing tools do not directly check programs against specifications. Instead, a conservative finite state abstraction of the program is first generated.

Verification tools such as MAGIC (**M**odular **A**nalysis of pro**G**rams **I**n **C**) [22, 7] employ a framework known as **C**ounter**E**xample **G**uided **A**bstraction **R**efinement (CEGAR) [10, 8] to iteratively create more precise abstractions of the program until the desired properties can be proven or a real counterexample generated.

Chaki, et al. [7] summarise the CEGAR process as follows:

- **Step 1 (Model Creation).** Extract an LTS M_{Imp} from the program Π. The model is computed using the control flow graph (CFG) of the program in combination with an abstraction method called *predicate abstraction* [17, 2]. Properties such as the equivalence of predicates are decided with the help of a theorem prover.

- **Step 2 (Verification).** Verify that the abstraction M_{Imp} conforms to the specification, *Spec*. If this is the case, the verification is successful. Otherwise, i.e., if M_{Imp} does not conform to *Spec*, obtain a possibly spurious counterexample and perform step 3.

- **Step 3 (Validation).** Check whether the counterexample extracted in step 2 is valid. If this is the case, then we have found an actual bug and the verification terminates unsuccessfully. Otherwise construct an explanation for the spuriousness of the counterexample and proceed to Step 4.

- **Step 4 (Refinement).** Use the spurious counterexample *CE* from the previous step to construct an improved set of predicates. Return to step 1 to extract a more precise M_{Imp} using the new set of predicates instead of the old one. The new predicate set is constructed in such a way as to guarantee that all spurious counterexamples encountered so far will not appear in any future iteration of this loop.

1.3.1 Control Flow Graphs

A *control-flow graph* (CFG) is used to model the flow of control in the program. A CFG is a directed graph, G = (N, E) where each node $n \in N$ corresponds to a *basic block* in the program [14]. Generally, basic blocks begin with labelled statements and end with branches or jumps. The transitions between nodes represent possible transitions between the associated basic blocks, engendered by branch statements, gotos, function calls, or returns in the program code.

```
if (x == y)
  then {
    x := 7;
    y := z + 3;
  } else {
    z := 9;
  }
y := y + 1;
```

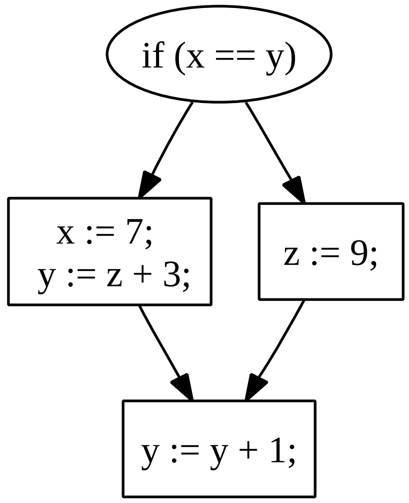


Figure 1.2: Example program source and the associated control flow graph.

The initial abstraction built in step 1 of the CEGAR loop is constructed from the CFG assuming that every branch in the program can be taken. Thus the first model is a conservative abstraction of the program's control flow. The model accepts a superset of the possible traces of the program [9].

A number of natural simplifications can be made to the C source program before the CFG is constructed. Typically this involves rewriting expressions with side-effect free statements, rewriting all loop statements with `if` and `goto`, and simplifying assignment statements. After these source transformations, the definition of a control location becomes straightforward. Each assignment, `goto`, and `return` statement gives rise to a control location with a unique successor. `if` statements yield a control location with exactly two successors [7]. An example code snippet with the associated control flow graph is illustrated in Figure 1.2.

Some tools, such as SLAM [26], can reason about recursive functions with the help of pushdown automata [13]. The rest of this introduction assumes only non-recursive functions, however.

1.3.2 Augmenting Control Flow Graphs with Data Abstraction

For verification purposes, the CFG is too imprecise because it ignores data (memory) and models only the control flow. It is computationally unfeasible to model the possible memory values explicitly. It is therefore necessary to augment the CFG with *abstract* memory state information [7].

Relevant properties about the memory state can be obtained from the C expressions used as branching conditions. For example, if the control flow graph contains a branch `if` $x > 0$ then in order to reason about the possible paths of control, we only need to know 1 bit of information rather than all 2^{32} possible values of x on a 32-bit computer.

If that were the only branch statement in a source function, and hence, the only relevant data property, then all states in the control flow graph would be split into two new states in an expanded CFG: one state where the property $x > 0$ is true, and one where it is false.

In general, if we have k data properties, each of which is either true or false, then each control location corresponds to 2^k possible states in the model. Thus there is a state corresponding to each possible valuation of the properties at each control location.

1.4 Predicate Abstraction

Predicate abstraction, first described by Graf and Saïdi, is a method for combining theorem proving and model checking techniques to prove properties of infinite state systems [17, 15]. In order to describe abstract memory states, a fixed set of properties $\mathcal{P} = \{P_1, \ldots, P_k\}$ must be obtained from the branch statements in the CFG as described in the previous section. We call these binary expressions *predicates*.

The relationship between the abstract memory state of our model and the concrete memory state is concisely defined by Chaki, et al [7] as:

> Given a concrete memory state m and a predicate P, we say that m satisfies P if and only if P evaluates to true during the execution of the given procedure when the memory state is m. A valuation for $\mathcal{P}$ is a vector $v_1, \ldots, v_k$ of Boolean values, such that v_i expresses the Boolean value of P_i. $\mathcal{V}$ denotes the set of all valuations, which is the set of all abstract memory states. Intuitively, a concrete memory state m is modelled by $v_1, \ldots, v_k$ if for $1 \leq i \leq k$, m satisfies P_i if and only if v_i is true.

Figure 1.3 shows a simple CFG with five states S_0, S_1, S_2, S_3, and S_4. The CFG has been augmented with the predicates $\mathcal{P} = \{(x >= 0), (x > y)\}$. The expanded CFG contains $5 * 2^2 = 20$ states, as each possible valuation of the predicates is represented for each of the original states.

The number of abstract states is thus exponential in the number of predicates. One large challenge, therefore, is to identify the minimal set of predicates that are necessary to prove a given property. The CEGAR loop begins with the initial abstraction and attempts to find counterexamples by iteratively searching and then refining the abstraction.

In the context of reachability analysis, if the verification step (Step 2) discovers a path through the abstraction to an error state, then this path must be validated in Step 3. If the abstraction is spurious, then a more precise refinement must be obtained by adding additional predicates to $\mathcal{P}$. Techniques have been developed [9] to guarantee that the set of predicates that eliminate all discovered counterexamples is minimal.

The abstractions are iteratively refined according to reachable successor states. Although termination cannot be guaranteed, since the model checking problem is in

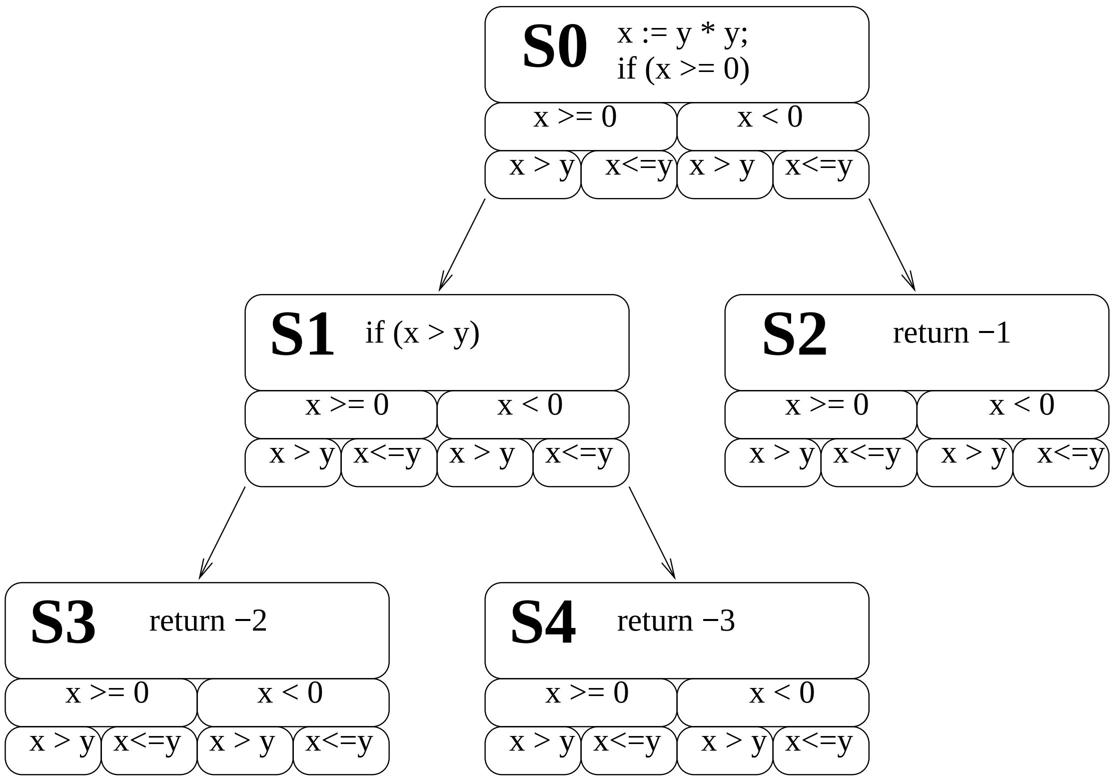


Figure 1.3: Example CFG extended with abstract memory states

general undecidable, in practice these successive abstractions often converge to a fixed point in a finite number of steps.

1.5 Weakest Preconditions

The refinement loop in the predicate abstraction process involves computing weakest preconditions of statements relative to a given predicate. In *Hoare Triple* notation, the required connection between a precondition (P), a program (Q), and a description of the result of its execution (R), is denoted:

$$P\{Q\}R$$

.

This is interpreted as "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion." [18]

Here we consider a statement s and a predicate ϕ, and let $\mathcal{WP}(s, \phi)$ denote the *weakest liberal precondition* of ϕ with respect to statement s. $\mathcal{WP}(s, \phi)$ is defined as the weakest predicate whose truth before s entails the truth of ϕ after s terminates.

For assignment statements, the weakest precondition of a predicate ϕ is obtained by replacing all occurrences of the left hand side of the assignment statement with the right hand side of the assignment.

For example, consider the assignment "$x = x + 1$" and the predicate "$x < 4$":

$$\mathcal{WP}(x = x + 1, x < 4) \quad = \quad (x + 1) < 4 \quad = \quad x < 3$$

The weakest precondition computations are a key part of the predicate abstraction process. Suppose that our set of predicates is $\mathcal{P} = \{(x = 1), (x < 4)\}$. We saw above that $\mathcal{WP}(x = x + 1, (x < 4)) = (x < 3)$, but the predicate $(x < 3)$ is not in our predicate set $\mathcal{P}$. In such a case a theorem prover may be called to *strengthen* the weakest precondition to an expression over the predicates in $\mathcal{P}$[2]. In this example, a theorem prover would show that $(x = 1) \rightarrow (x < 3)$. Therefore, if $(x = 1)$ is true before `x=x+1;` then $(x < 4)$ is true after. The improvements introduced in this thesis are designed to reduce the number of necessary calls to the theorem prover as part of this reasoning about weakest preconditions.

1.6 Thesis Outline

In Chapter 2, parallel assignment statements are introduced. The classical NP-complete parallel assignment problem is first considered, and then an additional restriction is added to create a special case in which the problem is tractable. The parallel assignment problem is then discussed in the context of weakest precondition computations. In this special situation where statements can be assumed to execute truly concurrently, we find an even better algorithm for compressing multiple sequential assignment statements into a single parallel assignment.

In Chapter 3, experiments are presented which show the level of assignment compression which can be achieved from a broad class of software. The algorithms from Chapter 2 have also been implemented in the ComFoRT Reasoning framework, and results are presented showing the time and memory space improvements for model checking a selection of applications. In Chapter 4, we conclude by providing a summary and directions of future work.

Chapter 2

Parallel Assignment

Sequences of assignment instructions are called *straight line programs* or *linear blocks*. Parallel assignment is a construct that permits the updating of multiple variables as a single atomic operation. As illustrated by Sethi [25], the Fibonacci sequence can be very cleanly generated with the parallel assignment $f_0, f_1 := f_1, f_0 + f_1$. When f_0 and f_1 are both initialised to 1, then repeated execution of this parallel assignment will lead to f_1 taking on the values of the Fibonacci sequence (2,3,5,8,...).

Some programming languages such as Algol 68 and Common Lisp provide support for expressing parallel assignment. For other languages, parallel assignment instructions can be implemented by a straight line program that may need to use additional temporary storage. For example, the simple swap of two variables can be expressed with the parallel assignment $x, y := y, x$. As a straight line program, we must store the value of x in a temporary variable before overwriting its contents, hence the linear block of three assignment statements: $t := x; x := y; y := t$.

For the purpose of verification, we are interested in identifying sequential assignment statements in straight line code that can be replaced with equivalent parallel assignment statements. This operation compresses multiple control points for sequential assignment statements into a single parallel assignment control point. The new parallel assignment control point consists of a list of assignment statements.

In this chapter we consider a number of possible approaches to finding sequences of assignments suitable for parallel assignment.

- In Section 2.1, we require that each assignment in a parallel assignment block may be executed in any order without affecting the other assignment statements in that parallel block. In this scenario, the example $x, y := y, x$ would not be a valid parallel assignment because $x := y; y := x$ is different from $y := x; x := y$ whenever $x \neq y$.

- In Section 2.2, we add an additional restriction to the classical parallel assignment problem by disallowing reordering of the assignment statements. This produces a tractable problem for which efficient algorithms can be obtained.

- In Section 2.3, we see that we have additional flexibility in the context of weakest pre-condition computations. We can assume that the assignments in a parallel assignment block must all be executed concurrently.

2.1 Classical Parallel Assignment

The classical parallel assignment problem is stated by Garey and Johnson [16] as follows.

Instance: Set $V = \{v_1, v_2, \ldots, v_n\}$ of variables, set $A = \{A_1, A_2, \ldots, A_n\}$ of assignments, each A_i of the form "$v_i \leftarrow op(B_i)$" for some subset $B_i \subseteq V$, and a positive integer K.

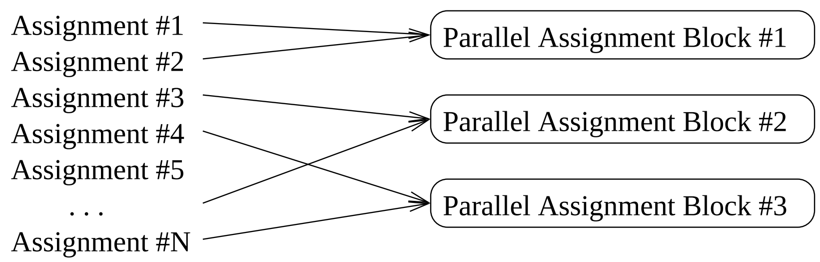


Figure 2.1: Sequential Assignments transformed to Parallel Assignments

$$\begin{aligned}
A_1 : v_1 &:= op(B_1)\\
A_2 : v_2 &:= op(B_2)\\
A_3 : v_3 &:= op(B_3)\\
&\vdots\\
A_n : v_n &:= op(B_n)
\end{aligned}$$

Question: Is there an ordering $v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}$ of V such that there are at most K values of i, $1 \leq i \leq n$, for which $v_{\pi(i)} \in B_{\pi(j)}$ for some $j > i$?

Thus our problem of compressing the sequential assignment statements into as few parallel assignment statements as possible would be equivalent to the optimisation problem of finding the minimum satisfying K.

Unfortunately, Sethi [25] showed that this problem is NP-Hard via a reduction from the feedback node set problem. In the next section we consider a greedy algorithm which identifies parallel assignments with the additional restriction that the sequential assignments must be adjacent. That is to say, no reordering of the assignments is allowed even if this would not disrupt the data dependencies. In Section 2.3 we consider the special circumstances of statements in weakest precondition computations to perform even better compression of single assignment statements.

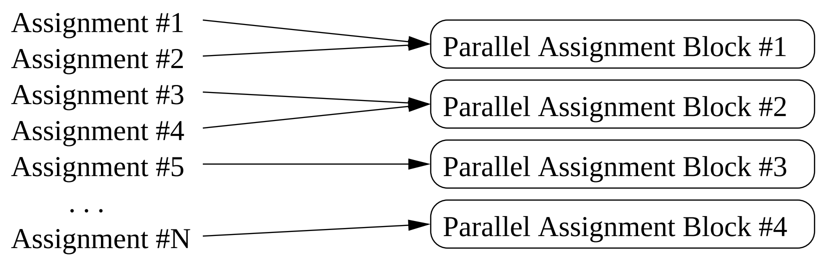


Figure 2.2: Sequential Assignments transformed to Parallel Assignments without reordering.

2.2 Tractable General Parallel Assignment

In this section we consider a modified version of the classical parallel assignment problem where reordering of the assignment statements is not allowed. The instance introduced in the previous section is still used, but the question becomes:

Question: Are there at most K values of i, $1 \leq i \leq n$, for which $v_i \in B_j$ for some $j > i$?

Figure 2.1 illustrates a transformation from sequential to parallel assignment statements involving reordering that would be allowed in the classical parallel assignment problem. Figure 2.2 shows a similar transformation with the additional condition preventing reordering. A more explicit example with 4 simple assignment statements is provided in Figure 2.3(a). The limitation described above that prevents reordering would allow us to transform this into Figure 2.3(b). If reordering were allowed, this could be written even more efficiently as in Figure 2.3(c).

2.2.1 Analysis

For each of the n assignments A_i, and for each $j, i < j \leq n$, we must test if $v_i \in B_j$. Therefore, we will need $(n-1)+(n-2)+(n-3)+\ldots+3+2+1 = \frac{(n-1)^2}{2}$ set inclusion operations. If each set B_j is represented as a bit field where the i^{th} bit represents inclusion of v_i and we have that $\|B_j\| < C$ for some constant C, then set inclusion

```
x = 1;
y = x;
u = 2;
v = u;
```

(a)

```
x = 1;
y = x ||| u = 2;
v = u;
```

(b)

```
x = 1 ||| u = 2;
y = x ||| v = u;
```

(c)

Figure 2.3: (a) shows a sequence of four sequential assignment statements. (b) shows the parallel assignment found by the atomiser algorithm. (c) shows the best possible parallel assignment that exists if we allow reordering as in the classical problem.

can be determined in constant time, yielding an $O(n^2)$ algorithm.

Recall that n will not be the number of control locations in the entire program. Instead, n is the number of assignments in a sequential list of assignment statements in one node of the control flow graph. As such, n is never a very large number.

2.2.2 Implementation

In order to reason about the variables on the right hand side of assignment statements, we need more information than what is provided by the control flow graph. The parse tree [1] provides the expression-level syntactic information we need to reason about individual assignments. We are not interested in a parse tree for the entire program source code, however. Instead, we expect the control flow graph to maintain a pointer to a parse tree for each individual assignment statement. Figure 2.4 shows what the parse tree would look like for the simple assignment statement `x := y + 1`. Given such a parse tree, one can easily build up lists of variables on the left-hand side (LHS) and right-hand side (RHS) of an assignment statement.

Given a control flow graph data structure that includes pointers to the parse trees for individual assignment statements, the process of creating a new CFG that utilises parallel assignment statements is described in Algorithm 1. This algorithm visits each node in the control flow graph and then follows a greedy strategy to build up lists of

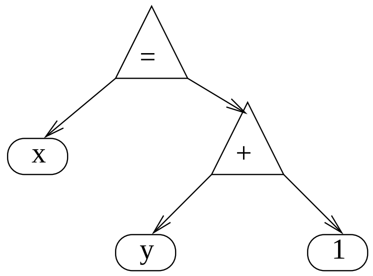


Figure 2.4: Example parse tree for an assignment statement.

parallel assignment statements.

Each assignment statement in the CFG node is compared to the running list of assignments in the parallel assignment block. If the assignment statement is not suitable for parallel assignment with all of the other assignments in the current parallel assignment block, then that assignment block is finished and a new one is started.

This algorithm relies on another algorithm to determine whether an assignment statement s_1 can be included in the block of parallel assignments P_1. Algorithm 2, *canParallelise*, illustrates the decision procedure in the simpler case of just two assignment statements.

2.2.2.1 CIL/OCaml Implementation

The C Intermediate Language (CIL) is a high level language representation along with a set of tools that permit easy analysis and source to source translation of C programs [23]. It breaks down certain complicated constructs of the C language into simpler ones. For example, CIL does not allow procedure calls inside the argument of another procedure call, expressions with side-effects (such as `x++`), or shortcut evaluations (such as `x && y`). It is coupled with a front end that can analyse and transform not just ANSI C, but also Microsoft C and GNU C extensions into this simplified C Intermediate Language.

The CIL tools are written in the OCaml extended functional programming lan-

Algorithm 1 **Atomise** accepts a CFG and loops over the assignment statements combining adjacent assignments into parallel assignment blocks whenever possible.

Input: A CFG
Output: A CFG in which assignment statements have been parallelised
for all $N \in CFG$ **do**
  if N contains a statement list S **then**
    Let $parallel\_list$ = first $s \in S$.
    for all statement $s \in S$ with successor statement s'. **do**
      if $canParallelise(parallel\_list, s')$ **then**
        append s' to $parallel\_list$.
      else
        Append $parallel\_list$ to new_list
        Initialise $parallel\_list$ with s'.
      end if
    end for
    Append $parallel\_list$ to new_list
    Replace statement list S in CFG node N with new_list.
  end if
end for

Algorithm 2 **CanParallelise** accepts a list of assignments suitable for parallel assignment and an additional assignment and determines if the new assignment can be safely added to the existing parallel assignment block.

Input: Assignment list l and an assignment statement s_1.
Output: A boolean answer as to whether the statements may be executed in parallel.
Let $LHS(s)$ be a function returning the variable on the left hand side of single assignment s.
Let $LHS\_List(l)$ be a function returning the variables on the left hand side of the assignments in assignment list l.
Let $RHS(s)$ be a function returning the list of variables on the right hand side of assignment s.
Let $RHS\_List(l)$ be a function returning the variables on the right hand side of the assignments in assignment list l.
if $LHS(s_1) \in RHS\_List(l)$ or $RHS(s_1) \cap LHS\_List(l) \neq \varnothing$ **then**
  return false
else
  return true
end if

guage. OCaml is a variant of ML that includes object oriented features. An implementation of Algorithm 1 in OCaml is provided in Appendix A. The code in this appendix is implemented as a visitor method for basic blocks of the Control Flow Graph. It makes use of the rich collection of data structures provided by CIL and can be compiled into the bundled `cilly` utility to compress sequential assignment statements in C source files and output compression statistics.

In addition to the expressive advantages, CIL was chosen in part because it is the parsing framework upon which the the **B**erkeley **L**azy **A**bstraction **S**oftware Verification **T**ool (BLAST) is implemented. BLAST [6] is a verification system for checking safety properties of C programs that uses a variant of the abstract – model check – refine loop described in Chapter 1.

Assignment compression results from this tool are described in the following chapter. By integrating this algorithm into BLAST, one could thus obtain further results about the affects of this assignment compression on the model checking process.

2.2.2.2 C Implementation

An implementation of Algorithm 1 in the C Programming Language is provided in Appendix B. The parser for this implementation was generated with the Bison LALR(1) parser generator [5] following the ANSI C language definition in [21]. The language translation and control flow graph implementation follows the general techniques outlined in [1, 14].

This implementation was integrated into the ComFoRT reasoning framework to obtain the model checking results presented in Chapter 3.

2.3 Concurrent Parallel Assignment

The algorithms described in the two previous sections are based on two assumptions. The first assumption is that we can not change the form of the individual assignment statements. The second assumption is that we must guarantee that the assignments in a parallel block can be executed in any order without affecting the result. In fact neither of these assumptions is necessary in the context of building parallel assignments for weakest precondition computations.

Consider the following example:

```
x := y
z := x
```

Algorithm 1 would not be able to combine these two assignment statements because the left hand side of one is present in the right hand side of the other. However, it is possible to change the second assignment without altering the result of the block.

```
x := y
z := y
```

With this modification, our existing algorithm would be able to combined these two assignments into a single parallel assignment block. It is also clear that the result is exactly the same as the original sequence of assignments.

In general, we can define a function that accepts a sequence of simple assignment statements S without pointers and without function calls and returns an equivalent parallel assignment statement.

Proof by Induction:

The base case of a single assignment, $S = \{s_1\}$, is vacuously true. $f(S) = S$ is the function.

Now, let S be a sequence of n sequential assignment statements and let S^+ denote the the sequence S and the successor of the last assignment in S, s'. Suppose a function g exists to transform the sequence S of assignments into an equivalent parallel assignment, $g(S)$. *(Inductive hypothesis)*

We build a new function $h(S^+)$ as follows:

for all $v \in RHS(s')$ **do**

  if $v = LHS(\tilde{s})$ for some $\tilde{s} \in g(S)$ **then**

    Replace v in s' with $RHS(\tilde{s})$

  end if

end for

Output (g(S),s')

By the replacement construction on s' we guarantee that it can be combined with $g(S)$ in a parallel block, thus proving our result inductively.

$\blacksquare$

With concurrent parallel assignment, the left hand side of all assignment statements are updated simultaneously. This means that instances of all variables in the parallel assignment block refer to the valuations before the parallel block is entered. If an assignment statement needs to utilise the valuation of a variable after another assignment statement, then that assignment must be rewritten with the procedure outlined in the previous proof.

As one final illustration, consider again the assignment list introduced in Figure 2.3(a).

```
x = 1;
y = x;
u = 2;
v = u;
```

The classical parallel assignment problem seeks to find the optimal ordering of the assignment statements so as to find a minimal set of parallel assignment statements, such as:

```
x = 1 ||| u = 2;
y = x ||| v = u;
```

In the context of weakest pre-condition computations, however, we can keep track of the before and after state of each variable. The following example shows how this could be calculated in our weakest precondition computations, where X_0 means the value of X before the parallel assignment block, and X_1 means the value after the block.

$$\mathtt{x}_1 = 1 \quad ||| \quad \mathtt{u}_1 = 2 \quad ||| \quad \mathtt{y}_1 = \mathtt{x}_1 \quad ||| \quad \mathtt{v}_1 = \mathtt{u}_1$$

$$\mathtt{x}_1 = 1 \quad ||| \quad \mathtt{u}_1 = 2 \quad ||| \quad \mathtt{y}_1 = 1 \quad ||| \quad \mathtt{v}_1 = 2$$

2.3.1 Implementation

The *ConcurrentAtomise* algorithm described in the previous section is presented in Algorithm 3.

2.4 Parallel Assignment and Weakest Preconditions

In section 1.5 we introduced weakest preconditions and described the computation of $\mathcal{WP}(s, \phi)$ for a statement s and predicate ϕ. For assignment statements, the weakest precondition of a predicate ϕ was obtained by replacing all occurrences of the left hand side of s with the right hand side of the assignment. This can be represented in replacement notation by $\phi[LHS/RHS]$.

This replacement operation extends naturally when s is a parallel assignment block. Each variable in ϕ that occurs on the left hand side of an assignment in

Algorithm 3 ConcurrentAtomise accepts a CFG and loops over the assignment statements modifying adjacent assignments as necessary to allow them to be combined into a single parallel assignment block.

Input: A CFG
Output: A CFG in which assignment statements have been parallelised
for all $N \in CFG$ **do**
  if N contains a statement list S **then**
    Let $parallel\_list$ = first $s \in S$.
    for all statement $s \in S$ with successor statement s'. **do**
      for all $v \in RHS(s')$ **do**
        if $v = LHS(\tilde{s})$ for some $\tilde{s} \in parallel\_list$ **then**
          Replace v in s' with $RHS(\tilde{s})$
        end if
      end for
      append s' to $parallel\_list$.
    end for
    Replace statement list S in CFG node N with $parallel\_list$.
  end if
end for

s is replaced with the corresponding right hand side. For example, the weakest precondition of parallel assignment $a, c := b, a$ and the same predicate ϕ would be denoted $\phi[a/b, c/a]$. Figure 2.5(a) shows a sequence of assignment statements and the associated weakest precondition computations. Figure 2.5(b) shows the same sequence of assignment statements after it has been compressed with the **Atomiser** algorithm into a smaller sequence of parallel assignment statements.

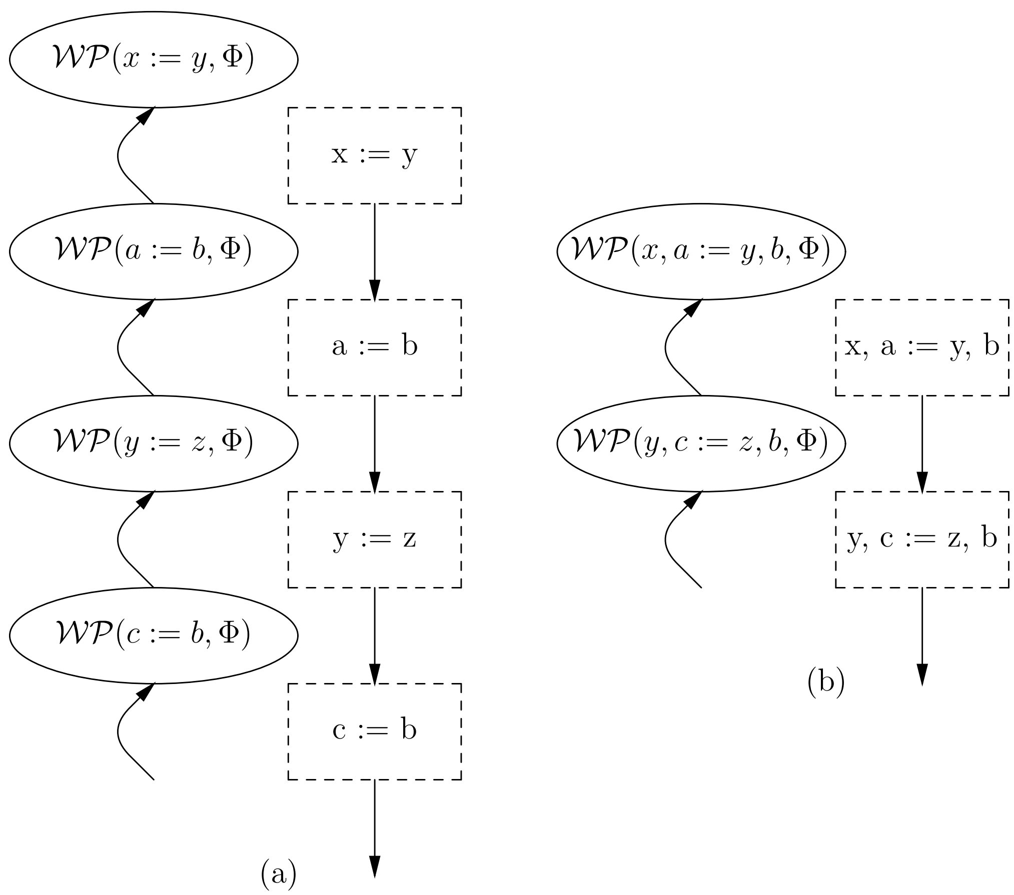


Figure 2.5: (a) A sequence of four simple assignment statements and the associated weakest precondition computations that would be calculated in a CEGAR loop. (b) A shorter sequence of parallel assignment statements with fewer associated weakest precondition computations.

Chapter 3

Experimental Evaluation

We implemented the atomiser algorithm inside both the ComFoRT Reasoning Framework from Carnegie Mellon and the Berkeley CIL tool. The goals of this experimentation were as follows. The first goal was to determine how much compression of assignment statements could be obtained for real programs in several different application domains. The second goal was to determine if this compression would in fact speed up the model checking process. The final goal was to characterise the class of software where model checking could benefit the most from utilisation of parallel assignment statements.

3.1 Assignment Compression Results

In this section we describe our results in the context of the first goal mentioned above, i.e, checking the effectiveness of the *Atomiser* and *ConcurrentAtomiser* algorithms at compressing the assignment control locations in real software source code.

The results in this section were obtained with the Berkeley CIL parser and the parallel assignment compressor, `atomiser.ml`, provided in Appendix A. The relative length and frequency of sequences of simple assignment statements varies with different software application domains. The experiments that follow were chosen because

Utility	Source File	LOC	Loc1	Loc2	Loc3
fsck	fsck.c	1208	102	72	62
ifconfig	ifconfig.c	2335	174	140	122
ifconfig	af_inet6.c	1436	76	61	56
mount	mount_ufs.c	227	10	6	5
ping	ping.c	3242	312	200	181
bdes	bdes.c	2357	284	253	220
gzip	trees.c	1221	299	192	147
gzip	deflate.c	477	103	65	59
gzip	inflate.c	1491	377	254	169
grep	search.c	2033	239	191	181
totals		16027	1976	1434	1202
average compression				72.6%	60.8%

Table 3.1: Assignment Compression of Unix System Software

they represent a broad spectrum of relevant software applications.

3.1.1 Unix System Software

The first benchmark set includes Unix system software from the FreeBSD 6.0 operating system. The utilities chosen include the file system consistency check utility (fsck), ifconfig, mount, ping, bdes, gzip, and grep.

Table 3.1 illustrates the results. The first column provides the name of the utility. The second column provides the name of the source file. The third column lists the number of lines of code in the source file. Specifically, this means the lines of code after the C pre-processor has been run and the CIL transformations performed but without counting any `#line` directives inserted by the pre-processor. The fourth column lists the number of simple assignment statements in the source file. The fifth column lists the number of assignment statements in the new source file generated with the *Atomiser* algorithm. The sixth column lists the number of assignments in the new source file generated with the *ConcurrentAtomiser* algorithm.

Library	Source File	LOC	Loc1	Loc2	Loc3
png	png.c	1108	87	60	57
png	pnggccrd.c	2835	511	262	229
png	pngrtan.c	6221	1859	930	629
jpeg	jmemmgr.c	1232	252	174	160
jpeg	jquant1.c	1361	257	125	96
jpeg	jquant2.c	1803	466	264	176
jpeg	transupp.c	3826	637	414	345
totals		18386	4069	2229	1692
average compression				54.8%	41.6%

Table 3.2: Assignment Compression of Graphics Libraries

3.1.2 Graphics Libraries

The second benchmark set includes the popular PNG and JPEG libraries used by most commercial and open source software to read and write those popular graphics file formats. Table 3.2 illustrates the assignment compression results for the largest source files of libpng v1.2.8 and libjpeg v6b.

3.1.3 Results Summary

On the body of software tested in this section, the *Atomiser* algorithm reduces the number of assignment statement control points to 63% of the original total. The *ConcurrentAtomiser* algorithm provides another 10% reduction in control points.

3.2 Model Checking Results

The ComFoRT Reasoning Framework [19] uses model checking to predict whether software will meet specific safety and reliability requirements. The model checking engine is derived from MAGIC [8], a tool developed by the model checking group at Carnegie Mellon University (CMU).

The source code for ComFoRT is not publicly available at this time, but Sagar

Name	LOC	Loc1	Loc2	Loc3	Time1	Time2	Time3	Mem1	Mem2	Mem3
Server	2483	207	172	171	9.8	8.8	8.4	135.3	136.2	133.8
Client	2484	175	145	144	17.5	11.7	12.4	128.9	128.1	127.7
Srvr-Clnt	locations are as above				165.8	136.7	128.4	201.1	194.7	192.3

Table 3.3: OpenSSL benchmarks with ComFoRT model checker + Atomise

Chaki from Carnegie Mellon was kind enough to integrate the atomiser algorithms into this tool and then run his benchmarks on a collection of Windows device drivers, OpenSSL, and Micro-C benchmarks. These benchmarks show the improvement in time and memory space that is provided by the assignment compression.

3.2.1 OpenSSL

The first set of benchmarks was run on the OpenSSL source code. The OpenSSL library implements the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols. It is widely used by web browsers, ssh clients, and other secure network applications on many different computing platforms.

Table 3.3 provides model checking results for the OpenSSL benchmarks. The *Server* test is the geometric mean of four benchmarks with same source code but different specifications. The *Client* test is the geometric mean of two benchmarks with same source code but different specifications. The *Srvr-Clnt* test is the geometric mean of sixteen benchmarks with same source code but different specifications.

Each test was run under three different model checking conditions:

1. no assignment parallelisation;
2. parallelisation with the *Atomiser* algorithm (individual assignments not changed)
3. parallelisation with with *ConcurrentAtomiser* algorithm (individual assignments changed as necessary)

Name	LOC	Loc1	Loc2	Loc3	Time1	Time2	Time3	Mem1	Mem2	Mem3
cdaudio	10171	2613	1447	1298	52.6	52.7	53.0	272.6	264.0	269.6
diskperf	4824	1187	719	617	15.9	15.8	15.7	176.3	176.3	175.0
floppy	9579	3478	1957	1845	130.4	130.5	129.3	468.8	468.8	470.4
kbfiltr	3905	560	331	286	1.9	1.9	1.8	129.1	128.7	126.3
parclass	26623	2840	1649	1450	74.5	73.7	72.3	335.5	335.5	340.0
parport	12431	4634	2935	2409	384.5	381.1	375.6	1102.3	1102.3	1127.2

Table 3.4: Windows device driver benchmarks ComFoRT model checker + Atomise

Name	LOC	Loc1	Loc2	Loc3	Time1	Time2	Time3	Mem1	Mem2	Mem3
Safety	6279	2699	1789	1589	35.5	35.7	36.0	229.2	229.2	223.5
Liveness	locations are as above				182.2	144.4	134.4	272.3	260.6	260.4

Table 3.5: Micro-C benchmarks ComFoRT model checker + Atomise

For each condition above, the number of assignments is listed (Loc) as well as the the time in seconds (Time), and the number of megabytes of memory (Mem) required for model checking.

3.2.2 Windows Device Drivers

The second set of ComFoRT benchmarks was run on a collection of Windows device drivers. The results are presented in Table 3.4 in the same format as the last section. Note that although significant assignment compression is achieved, the model checking time is not improved substantially.

3.2.3 Micro-C

The final set of ComFoRT benchmarks was run on Micro-C. The results are presented in 3.5. The same source code was used against two different specifications. One describing a Safety property and the other a Liveness property. The most striking result in this table is perhaps the fact that model checking of the Safety property is not improved with assignment compression, but the speed of Liveness property verification is significantly improved.

3.2.4 Results Summary

There is certainly a compression in terms of the number of control locations using either of the two atomiser algorithms. In general, the difference between no compression, and the *Atomiser* algorithm is more significant than that between the *Atomiser* and *ConcurrentAtomiser* algorithms. Actual performance of the model checker does improve is many cases, in particular for SSL and Micro-C. The improvement is marked for time, but somewhat marginal for space. The lack of improvement for the device drivers may be because of the relatively small number of predicates necessary to complete the verification. This means that the number of states does not decrease as dramatically with the reduction in the number of control locations as for the other benchmarks. More experiments with other examples may provide additional support for these observations.

3.3 Observations

The difference between the two assignment compression algorithms was more pronounced in the CIL implementation than in the ComFoRT implementation. This can be explained by the fact that the CIL tool performs additional simplifications to the source code before the atomiser algorithm is run. These transformations involve the creation of new temporary variables and assignments to those variables to simplify control flow and always provide unique return statements for procedures.

After examining the data, two scenarios can be seen as contributing to the observed speedup in model checking times with the Atomiser algorithms.

- **Compositionality and Partial Order Reduction**
- **Property size**

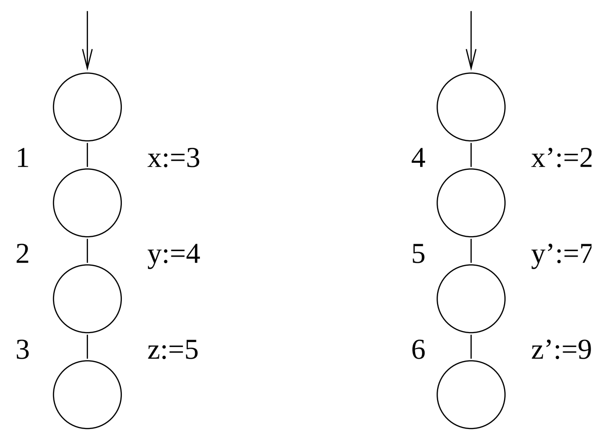


Figure 3.1: Assignment states in two components of a compositional model

3.3.1 Compositionality and Partial Order Reduction

Asynchronous systems such as the OpenSSL Srvr-Clnt benchmark are often described using an interleaving model of computation [11]. Concurrent events are modelled by allowing their execution in all possible orders relative to each other. Figure 3.1 shows 3 transitions (assignment statements) on each of two separate components. The transitions are labelled between 1 and 3 for the first component and between 4 and 6 for the second component. The sequence of control along each component is fixed, but there is no guarantee about the relative order, or interleaving, of the transitions of the two components. The model checker does not know that the interleavings do not matter, and so it will try all possible interleavings of the two for model checking. The lattice representing all possible transition interleavings is represented in Figure 3.2.

With parallel assignment statements, the 6 transitions of Figure 3.1 would be reduced to two transitions as in Figure 3.3. The much simpler associated lattice with parallel assignments is shown in Figure 3.4. The *ConcurrentAtomiser* algorithm allows for a special case of partial order reduction to eliminate the different equivalent interleaving orderings [12]. This has the effect of dramatically reducing the number

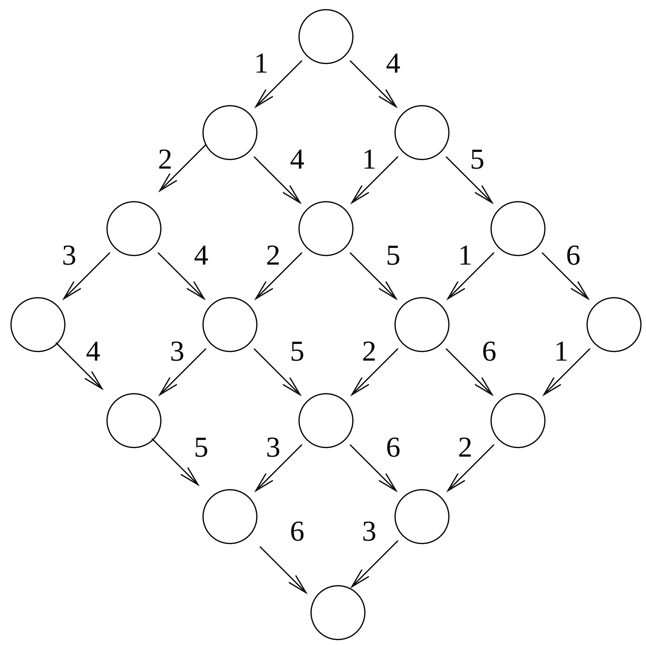


Figure 3.2: Lattice of possible paths from transitions in two components without parallel assignment.

of required calls to the theorem prover to reason about the predicates as part of the weakest precondition computations.

3.3.2 Property Size

The Micro-C benchmarks in Table 3.5 provide another important illustration of situations where the algorithms in this thesis can be especially beneficial.

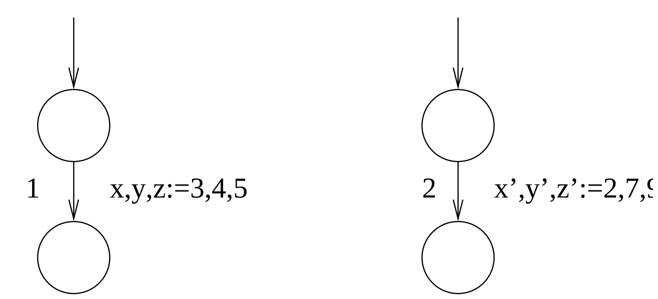


Figure 3.3: Parallel Assignment states in two components of a compositional model

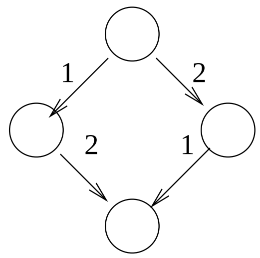
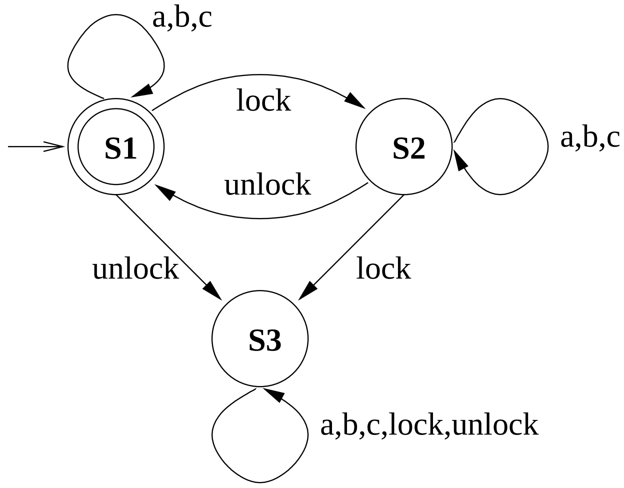


Figure 3.4: Lattice of possible paths from transitions in two components with parallel assignment.

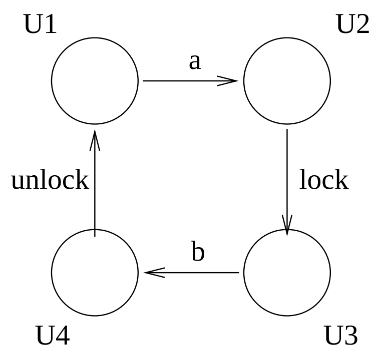
Both the Safety and Liveness properties are sequential one component systems here, so there is no benefit from reducing the interleaving paths as described in the previous section.

In the process of model checking a Büchi automaton for the negation of the property is constructed. This automaton is then synchronised with the abstract model of the software to obtain a new product automaton on which reachability analysis is performed.

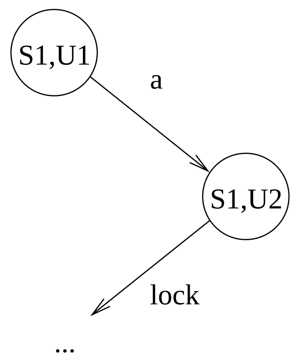
Consider the safety property $M \models$ "locks & unlocks alternate" and the event alphabet $\Sigma = \{a, b, c, lock, unlock\}$:



Suppose we also have an abstract model for our system:



We can then take the cross-product to define a new modified Kripke structure:



In this way our LTL property is translated to reachability with the cross product. With this cross product construction one finds that the size of the Büchi automata of the property acts as a scaling factor for the size of the product automata.

For the Micro-C safety property, the Büchi automata is relatively simple with just 4 states. For the liveness property, however, the automata has 51 states. Therefore any small reduction in the abstract software model size will be improved further by this factor. This explains why the same level of assignment compression has a significant effect for the liveness benchmark but not for the safety benchmark.

It would be interesting to see what improvements in time and memory could be obtained by implementing these algorithms into other model checking tools such as BLAST [6] and SLAM [4].

Chapter 4

Conclusions

4.1 Summary

The aim of this thesis has been to explore the use of parallel assignment in software verification systems.

We began in Chapter 1 with a description of how modern software verification tools use predicate abstraction, theorem provers, and model checkers to verify properties of software written in general purpose programming languages. In Chapter 2 we introduced parallel assignment statements. The classical NP-complete parallel assignment problem was first posed, and then an additional restriction was added to create a special case in which the problem is tractable with an $O(n^2)$ algorithm. The parallel assignment problem was then discussed in the context of weakest precondition computations. In this special situation where statements can be assumed to execute truly concurrently, we provided an inductive proof that any sequence of simple assignment statements without function calls can be transformed into an equivalent parallel assignment block.

Chapter 3 provided experimental results of the algorithms from Chapter 2 to identify sequences of assignment statements and combine those suitable for parallelisation. Results of this assignment compression were provided for a wide variety of software applications. We then provided results of implementing this algorithm into the ComFoRT model checker. The improvement in time was significant for some classes of software, while the improvement in memory space was somewhat marginal. We then analysed the relative speedups that the *Atomiser* algorithms provided for several different classes of software.

The primary contributions of this thesis can be summarised as follows.

- A survey of modern software verification tools for general purpose programming languages.

- A survey of parallel assignment transformations.

- A greedy algorithm, *Atomiser*, to implement general parallel assignment statements from straight line code.

- A concurrent parallel assignment algorithm, *ConcurrentAtomiser*, that can be used for software verification tools.

- Experimental results showing the compression of assignment statements in a broad class of software.

- Experimental results showing the improvement in speed and space of model checking systems augmented with the *Atomiser* and *ConcurrentAtomiser* algorithms.

4.2 Future Work

This work focussed on a single transformation of the control flow graph of the software before the abstraction and modelling steps took place. However, this is part of a much broader class of possible improvements to the software model checking process. Other static transformations may enable the further reduction of the number of necessary states. For example, recent work on pathslicing [20] illustrates how static analysis of the control flow graph can remove a large number of unnecessary states from the abstract model.

There may also be more fruitful applications of partial order reduction in modern software verification tools.

It would be interesting to implement the atomiser algorithm into other software verification tools to verify the observations made at the end of Chapter 3.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, New York, NY, USA, 2001. ACM Press.

[3] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *8th International SPIN Workshop*, pages 103–122, New York, NY, USA, 2001. ACM Press.

[4] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM Press.

[5] Bison website. `http://www.gnu.org/bison`.

[6] Blast website. http://www-cad.eecs.berkeley.edu/rupak/blast.

[7] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 385–395, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Sagar Chaki, Edmund Clarke, Alex Groce, Joel Ouaknine, Ofer Strichman, and K. Yorav. Efficient verification of sequential and concurrent c programs.

[9] Sagar Chaki, Edmund Clarke, Alex Groce, and Ofer Strichman. Predicate abstraction with minimum predicates.

[10] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[11] Edmund M. Jr. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[12] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:279–287, 1999.

[13] Stephen A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *J. ACM*, 18(1):4–18, 1971.

[14] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Elsevier Science, 2004.

[15] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *CAV '99: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 160–171, London, UK, 1999. Springer-Verlag.

[16] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Bell Telephone Laboratories, 1979.

[17] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83, London, UK, 1997. Springer-Verlag.

[18] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–80,583, 1969.

[19] James Ivers and Natasha Sharygina. Overview of comfort: A model checking reasoning framework. Technical Report CMU/SEI-2004-TN-018, Carnegie Mellon Software Engineering Institute, 2004.

[20] Ranjit Jhala and Rupak Majumdar. Path slicing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–47, New York, NY, USA, 2005. ACM Press.

[21] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[22] Magic website. `http://www.cs.cmu.edu/~chaki/magic`.

[23] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of Conference on Compiler Construction*, 2002.

[24] Luke Ong. Automata, logic & games lecture notes. Hilary Term 2005.

[25] Ravi Sethi. A note on implementing parallel assignment instructions. *Information Processing Letters*, 2:91–95, 1973.

[26] Slam website. http://research.microsoft.com/slam.

Appendix A

OCaml code for functional atomiser algorithm

The following code is suitable for integration into the Berkeley CIL[23] transformation tool. It is written in the OCaml functional programming language.

```
(*
 * Atomizer
 *
 * A tool to conservatively utilize parallel assignment statements.
 * Implemented as a visitor method for statement objects through the
 * Berkeley CIL tool.
 *
 *)

open Pretty
open Cil

(** An atomizer to combine parallel assignment statements. *)

(* Statistics Counters *)
let stat_parassign = ref 0
let stat_assign = ref 0

(** The function that will be called with parallelizable assignment
    statements as arguments.  This can be defined as a MACRO to put it
    into whatever format is desired. *)
```

```
let atomize_fun = emptyFunction "CAN_ATOMIZE"
let atomize_exp = (Lval((Var(atomize_fun.svar)),NoOffset))

(** Convert a list of instructions into a list of expressions with
    lvalues on the even numbered points of the list, and the assignment
    expressions on the odd numbered points of the list. *)
let mkExpList (l : instr list) =
  (** Convert an assignment instruction to an lvalue/expression pair. *)
  let mkExp (i : instr) : exp list =
match i with
| Set (lv,e,l) -> stat_assign := !stat_assign + 1; Lval lv :: e :: []
| _ -> []
  in
List.concat (List.map mkExp l)

(** Create an instruction from a list of instructions that can be
    evaluated in parallel.  If there is more than one element in the list,
    output a call to the atomize function with each of the assignments
    passed as an argument to the atomizefunction.  If there is only one
    element in the list, simply output that instruction. *)
let myInstr (l : instr list) =
if List.length l > 1 then begin
stat_parassign := !stat_parassign + 1;
(Call(None,atomize_exp,mkExpList l,locUnknown))
end else
List.hd l


(** Recursively builds a list of lvalues present in the given expression. *)
let rec buildVarList (e : exp) : Cil.lval list =
match e with
   | Lval l -> l :: []
   | UnOp (u, expr1, t) ->  buildVarList expr1
   | BinOp (b, expr1, expr2, t) -> buildVarList expr1 @ buildVarList expr2
   | _ -> []
```

```
let rec arrayHelper (off: offset) : Cil.lval list =
match off with
| Index (exp,offset2) -> buildVarList exp @ arrayHelper offset2
| _ -> []

(** Recursively build a list of lvalues present in array indices *)
let rec buildArrayVarList (e : exp) : Cil.lval list =
match e with
   | Lval (host,offset) -> arrayHelper offset
   | UnOp (u, expr1, t) ->  buildArrayVarList expr1
   | BinOp (b, expr1, expr2, t) -> buildArrayVarList expr1 @ buildArrayVarList expr2
   | _ -> []

let arrayHelperLHS (host, offset) = (arrayHelper offset)

let buildRval (i: instr) : lval list =
match i with
| Set(lval, exp, loc) ->
  buildVarList exp @ buildArrayVarList exp @ arrayHelperLHS lval
| _ -> []

let rec compare_lval_offset offset1 offset2 : bool =
match offset1,offset2 with
| NoOffset,_ -> true
| _,NoOffset -> true
| Field(fieldinfo1,newoffset1),Field(fieldinfo2,newoffset2) ->
(fieldinfo1.fname = fieldinfo2.fname) &&
(compare_lval_offset newoffset1 newoffset2)
| Field(fieldinfo1,newoffset1),Index(exp2,newoffset2) -> false
| Index(exp1,newoffset1),Field(fieldinfo2,newoffset2) -> false
| Index(exp1,newoffset1),Index(exp2,newoffset2) -> true
(* This is conservatively wrong, we need to traverse the expression and see if it refers to the same part of the array.
(fieldinfo1 = fieldinfo2) &&
(compare_lval_offset newoffset1 newoffset2)
*)
```

```
(** Compare two lvalues and determine if they refer to the same
    variable.  This may need to recurse in case there are structures or
    arrays involved, as we should consider 'mystruct.x' and 'mystruct.x.y'
    as referring to the same variable here. *)
let rec compare_lval (lhost1, loffset1) (lhost2, loffset2) : bool =
if (compare lhost1 lhost2) = 0 then compare_lval_offset loffset1 loffset2
(* this means a.x and a are the same. but also a.x.p and a.x.q, which is wrong *)
else false

(* let rec compare_lval (lhost1, loffset1) (lhost2, loffset2) : bool =
if (lhost1 = lhost2) then true
else false
*)

(* Remember: equality between cyclic data structures does not terminate! *)
let rec intersect (l1: 'a list) (l2: 'a list) : bool =
if (l1 = []) then false
else (List.exists (function x -> compare_lval (List.hd l1) x) l2) ||
     (intersect (List.tl l1) l2)

(** A class to visit statements in the abstract syntax tree and
    parallelize those sequential assignment statements that the dependency
    graph allows. *)
class blockAtomizeVisitor = object (self)
  inherit nopCilVisitor

  (** The list of lvalues present in the running list of sequential
      assignments that can be parallelized. *)
  val mutable lhsList : Cil.lval list = []
  val mutable rhsList : Cil.lval list = []

  (** Initializes the running list of lvalues on the left hand side of
      the list of parallelized assignment statements.  If the first
      instruction is a Set, then the list is initialized with the lvalue on
      the left hand side of the Set.  Otherwise, it is initalized to the
      empty list. *)
```

```
  method private initLists (s1: instr) : unit =
match s1 with
| Set(lval, exp, loc) ->
lhsList <- lval :: [];
rhsList <- buildRval s1
| _ -> lhsList <- []

  (** Decision function that returns true if the two given
      instructions are both assignment statements and the dependency graph
      of the variables allows them to be parallelized.  If so, it also
      updates the running list of lvalues on the left hand side of the
      parallelized assignment statement. *)
  method private canAtomize (s1: instr) (s2: instr) : bool =
match s1, s2 with
| Set(lval, exp, loc), Set(lval2, exp2, loc2) ->
let rval1 = buildRval s1
in
let rval2 = buildRval s2
in
if ((intersect lhsList rval2) ||
    (List.exists (function x -> compare_lval lval2 x) rhsList))
then begin (* can't atomize, set lhs for next iteration *)
lhsList <- lval2 :: [];
rhsList <- rval2;
false
end else begin (* can atomize, append to lhs *)
if !Errormsg.verboseFlag then
ignore (warn "can atomize:\n%a@!--and--\n%a@!" d_instr s1 d_instr s2);
lhsList <- lval2 :: lhsList;
rhsList <- rval2 @ rhsList;
true
end;
| _ -> false;

  (** Visitor method for statement objects.  For those statements that
      are lists of instructions, this visitor seeks out atomizable
```

```
      sequential assignment statements and rebuilds the statement in
      parallelized form. *)
  method vstmt (s: stmt) : stmt visitAction =
let myList = ref []
in
let myParList = ref []
in
match s.skind with
(* Here, we need to iterate over the list building up a new
   list of atomized instructions. *)
| Instr(l) ->
(* if s.labels = [] then () else Printf.printf "We have labels!\n"; *)
if (List.length l > 0) then begin
let myinstrs = Array.of_list l
in
let i = ref 0
in
if !Errormsg.verboseFlag then
Printf.printf " - len of array = %d\n" (Array.length myinstrs);

(* Maintain a new list of instructions, including list of lists of
   instrs for parallel case, and then update the stmt to point
   to this new list of instrs after the while loop. *)

self#initLists myinstrs.(!i);
myParList := myinstrs.(!i) :: [];
while !i < (Array.length myinstrs - 1) do
if (self#canAtomize myinstrs.(!i) myinstrs.(!i + 1))
then begin
myParList := myinstrs.(!i + 1) :: !myParList;
end
else begin

(* Output the parallel list which has ended, and begin
                   a new parallel list.  That is, append a new
   instruction which is a parallelized version of
```

```
   myParList onto MyList *)

myList := (myInstr !myParList) :: !myList;
myParList := myinstrs.(!i + 1) :: [];
end;
i := !i + 1
done;

(* In either case we have a myParList with some
   instructions, so we should generate a new
   instruction of myParList and append it to myList. *)

myList := (myInstr !myParList) :: !myList;

if !Errormsg.verboseFlag then
Printf.printf "mylist.length : %d\n" (List.length !myList);

(* We cannot create a new statement and use ChangeTo
   here, or else any goto statements that pointed to
   this statement would get confused and print out
   __invalid_label. *)

s.skind <- Instr (List.rev !myList);
DoChildren
end
else DoChildren
| _ -> DoChildren

end

let feature : featureDescr =
  { fd_name = "atomize";
    fd_enabled = Cilutil.atomize;
    fd_description = "parallelize sequential assignment statements";
    fd_extraopt = [];
    fd_doit =
```

```
    (function (f: file) ->
if not !Cilutil.makeCFG then begin
  Errormsg.s (Errormsg.error "--doatomize: you must also specify --domakeCFG\n")
end;
      let blockAtomizer = new blockAtomizeVisitor in
      visitCilFileSameGlobals blockAtomizer f;
Printf.printf "Total Assignments: %d\nParallel Assignments: %d\n" !stat_assign !stat_parassign;
);
    fd_post_check = true;
  }
```

Appendix B

C code for imperative atomiser algorithm

```
/*
 * Atomize
 *
 * An algorithm to identify and combine sequential assignment
 * statements that are suitable for parallelization.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "common.h"
#include "cfg.h"

extern int f_verbose;
extern char string_tbl[]; /* string space, and pointers into it */

/* Data structure for binary parse tree */

typedef struct s_parsetree {
        int thistoken;
        attr_t attr;            /* contains type, value for consts,
                                 * and symbol table entry for
```

```
                                 * variables. */

        struct s_parsetree *child1;
        struct s_parsetree *child2;
        struct s_parsetree *next;
        struct s_parsetree *prev;
} parsetree_t;

/* Data structure for holding the list of variables and parse tree of
 * a given expression. */

typedef struct s_parlist {
        parsetree_t *root; /* root of this assignment statement
                                    * syntax tree. */

        struct varlist_entry *lhs; /* linked list of vars on LHS */

        struct varlist_entry *rhs; /* linked list of vars on RHS */
        struct s_parlist *next;
} parlist_t;


typedef struct s_expression {
        parsetree_t *tree; /* parse tree, with '=' as root */

        struct s_expression *next;
} expression_t;

typedef enum {stmt_expression, stmt_expression_list, stmt_labeled} statementtype_t;

typedef struct s_expressionlist {
        int length;
        expression_t *first;
        expression_t *last;
} expressionlist_t;
```

```
typedef struct s_statement {
        statementtype_t type;
        union {
                expression_t *exp;
                expressionlist_t *explist;
        } ptr;

        struct s_statement *next;
} statement_t;

typedef struct s_statementlist {
        int length;
        statement_t *first;
        statement_t *last;
} statementlist_t;


/*
 * atomizeCFG(cfgNode_t *cfg)
 *
 * Atomize a basic block (node of a CFG).  The basic block is assumed
 * to not contain any labels or branches.
 */

void
atomizeCFG(cfgNode_t *cfg) {
        statementlist_t *stmtlist;
        statement_t *stmt, *temp;
        parlist_t *parList, *nextparList;

        if (cfg == NULL)
                return;
        if (cfg->stmtlist == NULL)
                return;
```

```
stmtlist = cfg->stmtlist;
stmt = stmtlist->first;

while (stmt->next != NULL) {
        if ((stmt->type != stmt_expression) ||
            (stmt->next->type != stmt_expression))
                continue;

        /* For each expression, generate a list of variables
         * that is read from or written to in the assignment.
         * For example, in 'array[x] = y' we would have
         * 'array' on the lhs list and 'x' and 'y' on the rhs
         * list. */

        parList = mkParList(stmt->ptr.exp->tree);
        nextparList = mkParList(stmt->next->ptr.exp->tree);

        if (can_atomize(parList, nextparList)) {

                /* Greedy algorithm.  Combine these two, then
                 * compare amalgamation with next
                 * statement. */

                stmt->ptr.exp->tree = mkTree(MY_PARALLEL_OP,
                    stmt->ptr.exp->tree, stmt->next->ptr.exp->tree);

                temp =  stmt->next->next;

                /* Cleanup */
                free(stmt->next->ptr.exp);
                free(stmt->next);
                freeParList(parList);
                freeParList(nextparList);

                /* Rather than freeing all of this stuff, the
                 * first list can be reused so we don't have
```

```
                         * to calculate it again in the next iteration
                         * of the loop, as in the older algorithm at
                         * the bottom of this file.  */

                        stmt->next = temp;
                } else {
                        stmt = stmt->next;
                }
        }
}

/*
 * can_atomize(list1, list2) - returns a boolean decision as to
 * whether two assignment statements can be parallelized.
 *
 * list1 and list2 contain variables used in the first and second
 * assignment statements.  Each list has both an lhs and rhs member.
 * Intitially, the lhs contains only one variable that is being
 * assigned to.  However, if this assignment has already been
 * parallelized then it may contain more than one variable in the lhs
 * list.  i.e.  x = y + z ||| w = u - v, then {x,w} in LHS and
 * {y,z,u,v} in RHS.
 */

int
can_atomize(parlist_t *list1, parlist_t *list2) {
        if ((list2 == NULL) || (list1 == NULL))
                return 0;

        return !(compare_sides(list1->lhs, list2->rhs) ||
            compare_sides(list2->lhs, list1->rhs));
}

/*
 * compare_sides(varlist_entry *lhs, varlist_entry *rhs)
 *
```

```
 * returns 1 if any variable from the left hand side list occurs in
 * the right hand side list.
 */

int
compare_sides(struct varlist_entry *lhs, struct varlist_entry *rhs) {
        struct varlist_entry *temp_lhs;
        struct varlist_entry *temp_rhs;

        /* foreach variable on the left hand side. */
        for (temp_lhs = lhs; temp_lhs != NULL; temp_lhs = temp_lhs->next) {
                for (temp_rhs = rhs; temp_rhs != NULL; temp_rhs = temp_rhs->next) {
                        if (compare_var(temp_lhs, temp_rhs))
                                return 1;
                }
        }
        return 0;
}

/*
 * compare_var(varlist_entry *left, varlist_entry *right)
 *
 * returns 1 if the variables are the same.  For simple scalars this
 * is easy, but is slightly more involved for structures, where for
 * example we may have 'myStruct->member->a' and 'myStruck->member'
 * being compared.
 */

int
compare_var(struct varlist_entry *left, struct varlist_entry *right) {
        if ((left == NULL) || (right == NULL))
                return 0;

        if (left->type != right->type) {
                return 0;
        } else {
```

```
                if (left->type == structure_t) {
                        return compare_struct(left->structure,
                            right->structure);
                } else {
                        return (left->symb == right->symb);
                }
        }
}

/*
 * Compare if two structures are the same or not.  return 1 if the same.
 *
 * At this point, just check the base struct name, not any members.
 */

int
compare_struct(struct varlist *left, struct varlist *right) {

        if ((left == NULL) || (right == NULL) ||
            (left->head == NULL) || (right->head == NULL))
                return 0;

        return compare_struct_helper(left->head, right->head);
}

/*
 * This screams out for a recursive solution.  Recursively check
 * deeper levels of structure/member referencing until one is
 * different.
 */

int
compare_struct_helper(struct varlist_entry *left, struct varlist_entry *right) {
        /* check this level */
        if (left->symb != right->symb)
                return 0;
```

```
        /* same at this level, i.e   a.b and a.c */

        /* what if one of the next levels is null */
        if ((left->next == NULL) || (right->next == NULL))
                return 1;       /* they they are the same. */

        /* otherwise we have more levels to check. */
        return compare_struct_helper(left->next, right->next);
}


void
append_varlist(struct varlist_entry *list, struct varlist_entry *new) {
        struct varlist_entry *temp;

        if (list == NULL) {
                list = new;
                return;
        }
        temp = list;
        while (temp->next != NULL)
                temp = temp->next;

        temp->next = new;
}

void
merge_varlist(struct varlist *vlist, struct varlist *add) {
        struct varlist_entry *temp;

        if ((add == NULL) || (add->head == NULL))
                return;

        if (vlist->head == NULL) {
                vlist->head = add->head;
```

```
		vlist->tail = add->tail;
	} else {
		temp = vlist->head;
		while (temp->next != NULL)
			temp = temp->next;
		temp->next = add->head;
		vlist->tail = add->tail;
	}
}


/*
 * push a variable onto the parallelized list.
 */

void
push_varlist(struct varlist *varlist, struct symbol_rec *var, type_t type) {
	struct varlist_entry *temp;

	if (varlist->head == NULL) {
		varlist->head = (struct varlist_entry *)
			malloc(sizeof(struct varlist_entry));
		memset(varlist->head, 0, sizeof(struct varlist_entry));
		varlist->head->type = type;
		varlist->head->symb = var;
		varlist->head->next = NULL;
		varlist->tail = varlist->head;
	} else {
		temp = varlist->head;
		while (temp->next != NULL) {
			temp = temp->next;
		}
		temp->next = (struct varlist_entry *)
			malloc(sizeof(struct varlist_entry));
		memset(temp->next, 0, sizeof(struct varlist_entry));
		temp = temp->next;
```

```
                temp->symb = var;
                temp->next = NULL;
        }
}

/*
 * push a structure onto the parallelized list.
 */

void
push_varlist_struct(struct varlist *varlist, struct varlist *structure) {
        struct varlist_entry *temp;

        if (varlist->head == NULL) {
                varlist->head = (struct varlist_entry *)
                        malloc(sizeof(struct varlist_entry));
                memset(varlist->head, 0, sizeof(struct varlist_entry));
                varlist->head->type = structure_t;
                varlist->head->structure = structure;
                varlist->head->next = NULL;
        } else {
                temp = varlist->head;
                while (temp->next != NULL) {
                        temp = temp->next;
                }
                temp->next = (struct varlist_entry *)
                        malloc(sizeof(struct varlist_entry));
                memset(temp->next, 0, sizeof(struct varlist_entry));
                temp = temp->next;
                temp->structure = structure;
                temp->next = NULL;
        }
}


/*
```

```
 * mkVarList(list, tree) - builds a list of variables used in an
 *  expression (tree).
 */

struct varlist *
mkVarList(parsetree_t *tree) {
        struct varlist *structure;
        struct varlist *varlist;

        if (tree == NULL)
                return NULL;

        varlist = (struct varlist *)malloc(sizeof(struct varlist));
        memset(varlist, 0, sizeof(struct varlist));


        if (tree->attr.type == structure_t) {
                structure = (struct varlist *)malloc(sizeof(struct varlist));
                memset(structure, 0, sizeof(struct varlist));

                mkStruct(structure, tree);
                push_varlist_struct(varlist, structure);
                return varlist;
        }

        if (tree->attr.type == array_t) {
                if (f_verbose)
                        printf("array!\n");
                /* push the name of the array onto the stack */

                /* This doesn't work properly if this array is in a
                 * structure, for example a.b[3].  This need to
                 * integrate with the symbol table code better. */

                merge_varlist(varlist, mkVarList(tree->child1));
```

```
                /* These variables should be on RHS even if they are
                 * on LHS however!  array[x] = 9 means x is not
                 * modified */

                return varlist;
        }

        if ((tree->child1 == NULL) && (tree->child2 == NULL)) {
                if (tree->attr.var == 1) {
                        push_varlist(varlist, tree->attr.val.pval, int_t);
                        return varlist;
                }

        } else {
                merge_varlist(varlist, mkVarList(tree->child1));
                merge_varlist(varlist, mkVarList(tree->child2));
                return varlist;
        }
        return varlist;
}

/*
 * Return a list of variables in the array indices of tree.
 */

struct varlist *
arrayIndexList(parsetree_t *tree) {
        struct varlist *varlist;

        if (tree == NULL)
                return NULL;

        varlist = (struct varlist *)malloc(sizeof(struct varlist));
        memset(varlist, 0, sizeof(struct varlist));

        if (tree->attr.type == array_t) {
```

```
                merge_varlist(varlist, mkVarList(tree->child2));
                return varlist;
        }
        merge_varlist(varlist, arrayIndexList(tree->child1));
        merge_varlist(varlist, arrayIndexList(tree->child2));
        return varlist;
}


void
freeParList(parlist_t *list) {
        if (list == NULL)
                return;

        freeVarEntry(list->rhs);
        freeVarEntry(list->lhs);
        free(list);
}

void
freeVarList(struct varlist *varlist) {
        struct varlist_entry *temp;

        if (varlist == NULL)
                return;

        temp = varlist->head;
        freeVarEntry(temp->next);
        free(temp);
        free(varlist);
}

void
freeVarEntry(struct varlist_entry *var) {
        if (var == NULL)
                return;
```

```
        freeVarEntry(var->next);
        free(var);
}


/*
 * mkParList(tree) - Takes an abstract syntax tree for an assignment statement
 *  and returns a list of variables used on the LHS and the RHS.
 */

parlist_t *
mkParList(parsetree_t *tree) {
        parlist_t *list;
        struct varlist *varlist;

        if (tree == NULL)
                return NULL;

        list = (parlist_t *)malloc(sizeof(parlist_t));
        memset(list, 0, sizeof(parlist_t));

        list->root = tree;

        /* even for a single assignment, there can be multiple
         * variables on LHS.  For example, a[x], and other expressions
         * with variables for array indices. No, x acts like RHS in
         * such a case. */

        varlist = mkVarList(tree->child1);
        list->lhs = varlist->head;
        free(varlist);

        varlist = mkVarList(tree->child2);

        merge_varlist(varlist, arrayIndexList(tree->child1));
```

```
        merge_varlist(varlist, arrayIndexList(tree->child2));

        list->rhs = varlist->head;
        free(varlist);

        return list;
}

/*
 * mkStruct - return a structure datatype that describes the structure
 * variable passed in through the parse tree 'tree'.  The root of
 * 'tree' is expected to be a TOK_ARROW or TOK_DOT token that
 * separates one part of an identifier (the base structure) from the
 * field.
 */

void
mkStruct(struct varlist *vlist, parsetree_t *tree) {
        struct varlist_entry *temp;

        if (tree == NULL)
                return;

        /* if current node is a leaf, add to stack, otherwise call on
         * children in order. */

        if ((tree->child1 == NULL) && (tree->child2 == NULL)) {
                temp = vlist->tail;
                vlist->tail = (struct varlist_entry *)
                        malloc(sizeof(struct varlist_entry));
                memset(vlist->tail, 0, sizeof(struct varlist_entry));
                vlist->tail->symb = tree->attr.val.pval;

                if (temp == NULL) {
                        vlist->head = vlist->tail;
```

```
                } else {
                        temp->next = vlist->tail;
                }
        } else {
                mkStruct(vlist, tree->child1);
                mkStruct(vlist, tree->child2);
        }
        return;
}
```