SVV 2005

Parallel Assignments in Software Model Checking

Murray Stokely [1]

Google

Sagar Chaki [2]

Carnegie Mellon University, Software Engineering Institute, Pittsburgh, USA

Joël Ouaknine [3]

Oxford University Computing Laboratory, UK

Abstract

In this paper we investigate 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, we review the method in which software abstractions are built using counterexample-guided abstraction refinement (CEGAR). 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. The proposed algorithms were implemented in the ComFoRT reasoning framework [12] and used to measure the improvement in the verification of real software systems. This improvement in time proved to be significant for many classes of software.

Key words: static analysis, software model checking, predicate abstraction, parallel assignment.

[1] Email: `mstokely@google.com`
[2] Email: `chaki@sei.cmu.edu`
[3] Email: `joel@comlab.ox.ac.uk`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

1 Introduction

In this paper we investigate how formal software verification systems can be improved by utilising parallel assignment statements. Specifically, we are interested in performing this static transformation on programs written in general-purpose programming languages such as C or Java. The modified program is then verified with an extended model checking tool which interacts with theorem provers and decision procedures to reason about software abstractions.

Verification tools such as MAGIC (**M**odular **A**nalysis of pro**G**rams **I**n **C**) [14,4] employ a framework known as **C**ounter**E**xample-**G**uided **A**bstraction **R**efinement (CEGAR) [8,5] to iteratively create more precise abstractions of the program until the desired properties can be proven or a real counterexample is generated.

MAGIC provides a compositional framework that can be used to verify concurrent C programs against a range of safety and liveness specifications [5,6,7]. In this context each atomic assignment is represented as a state in the software model and the cartesian product of this model is then taken with the specification Büchi automaton.

Our approach is to analyze the control-flow graph (CFG) and combine sequential assignments into parallel assignment blocks. Since the iterated CEGAR framework can result in a large number of passes over the same parts of the CFG, the cumulative savings obtained by this compression procedure can lead to significant speed-ups. Algorithmically, weakest precondition computations, which typically involve interacting with theorem provers, become more efficient. Moreover, in the context of concurrent C programs, the reduction in the number of states in individual components has a multiplicative effect on the number of states of the whole system.

1.1 Outline

In Section 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 Section 3, experiments are presented which show the level of assignment compression which can be achieved from a broad class of software. The algorithms from Section 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. Finally, we discuss future research directions in Section 4.

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. 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 section 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 precondition 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 [11] 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.

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

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.

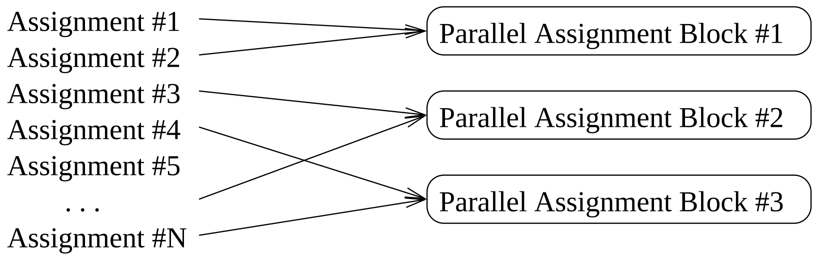


Fig. 1. Sequential Assignments transformed to Parallel Assignments

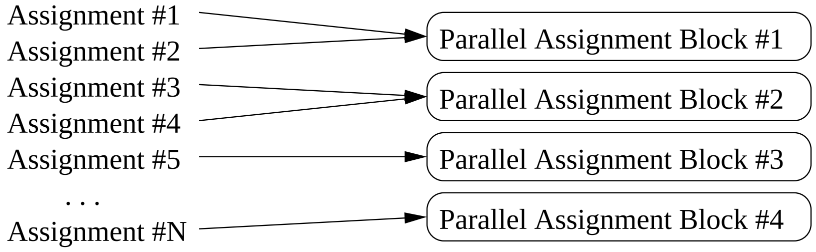


Fig. 2. Sequential Assignments transformed to Parallel Assignments without reordering.

Unfortunately, Sethi [16] 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.

2.2 Tractable General Parallel Assignment

Consider now 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 1 illustrates a transformation from sequential to parallel assignment statements involving reordering that would be allowed in the classical parallel assignment problem. Figure 2 shows a similar transformation with the additional condition preventing reordering.

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 $\|B_j\| < C$ for some constant C, then set inclusion 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. 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.

With these two lists as input, 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 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.

2.3 Concurrent Parallel Assignment

The algorithms described in the two previous sections are based on two assumptions. The first assumption is that we cannot 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

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

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 combine 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 $v \in \mathit{RHS}(s')$ **do**
  if $v = \mathit{LHS}(\tilde{s})$ for some $\tilde{s} \in g(S)$ **then**
    Replace v in s' with $\mathit{RHS}(\tilde{s})$
  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. ■

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 the following assignment list:

```
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 precondition computations, however, we can modify the assignment statements as necessary to ensure that all can be combined into one parallel assignment block:

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

2.3.1 Implementation

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

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

2.4 Parallel Assignment and Weakest Preconditions

For an assignment statement s and predicate ϕ, the weakest precondition $\mathcal{WP}(s, \phi)$ is 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 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 $\phi[a/b, c/a]$.

3 Experimental Evaluation

We implemented the atomiser algorithm inside both the ComFoRT reasoning framework from Carnegie Mellon and the Berkeley CIL [15] tool. The goals of this experimentation were as follows. The first goal was to determine

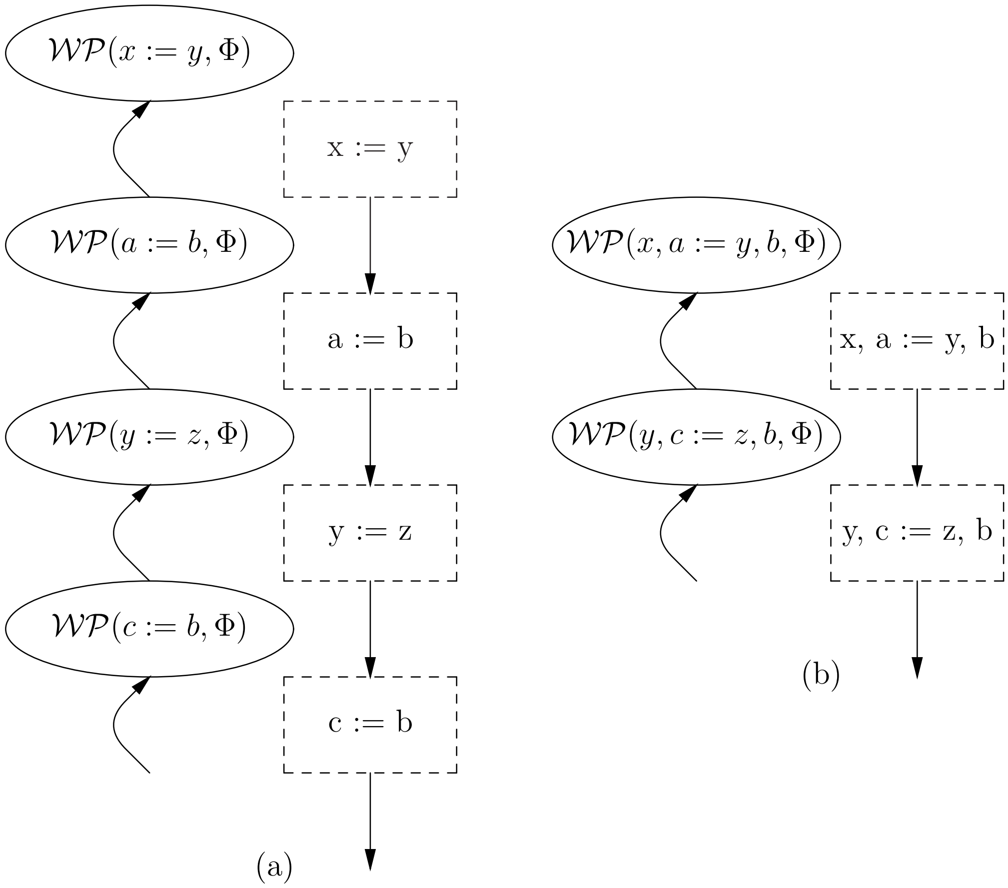


Fig. 3. (a) A sequence of four simple assignment statements and the associated weakest precondition computations that would be calculated in a CEGAR loop. (b) A shorter sequence of parallel assignment statements with fewer associated weakest precondition computations.

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 our parallel assignment compressor. The relative length and frequency of sequences of simple assignment statements varies with different software application domains. The experiments that follow were chosen because they represent a broad spectrum of relevant software applications.

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 1
Assignment Compression of Unix System Software

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 1 illustrates the results. The first column, Utility, provides the name of the utility. The second column, Source File, provides the name of the source file. The third column, LOC, 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, Loc1, lists the number of simple assignment statements in the source file. The fifth column, Loc2, lists the number of assignment statements in the new source file generated with the *Atomiser* algorithm. The sixth column, Loc3, lists the number of assignments in the new source file generated with the *ConcurrentAtomiser* algorithm.

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 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

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 2
Assignment Compression of Graphics Libraries

points.

3.2 Model Checking Results

The ComFoRT reasoning framework [12] uses model checking to predict whether software will meet specific safety and reliability requirements. The model checking engine is derived from MAGIC [5], a tool developed by the model checking group at Carnegie Mellon University. We integrated the atomiser algorithms into ComFoRT and ran it on a collection of Windows device drivers, OpenSSL, and Micro-C OS 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 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:

(i) no assignment parallelisation;

(ii) parallelisation with the *Atomiser* algorithm (individual assignments not changed)

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
OpenSSL benchmarks with ComFoRT model checker + Atomise

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 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 5
Micro-C OS benchmarks ComFoRT model checker + Atomise

(iii) parallelisation with with *ConcurrentAtomiser* algorithm (individual assignments changed as necessary)

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 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 OS

The final set of ComFoRT benchmarks was run on Micro-C OS. The results are presented in Table 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 be-

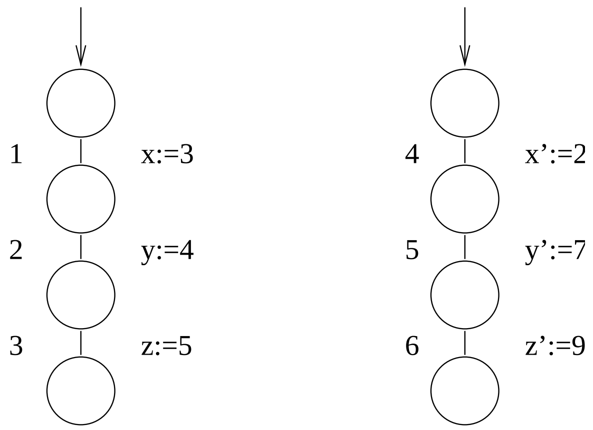


Fig. 4. Assignment states in two components of a compositional model

tween the *Atomiser* and *ConcurrentAtomiser* algorithms. Actual performance of the model checker does improve in many cases, in particular for SSL and Micro-C OS. 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

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

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 [10]. Concurrent events are modelled by allowing their execution in all possible orders relative to each other. Figure 4 shows 3 transitions (assignment statements) on each of two separate components. We assume here that the variables in each thread are not shared. 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 5.

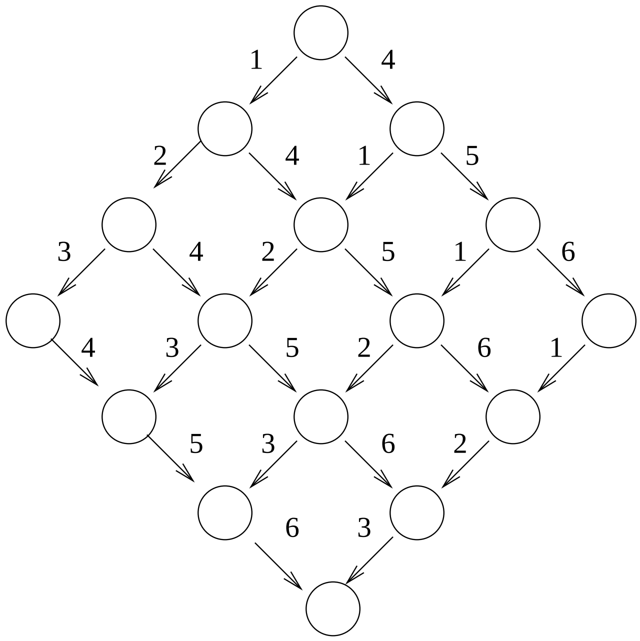


Fig. 5. Lattice of possible paths from transitions in two components without parallel assignment.

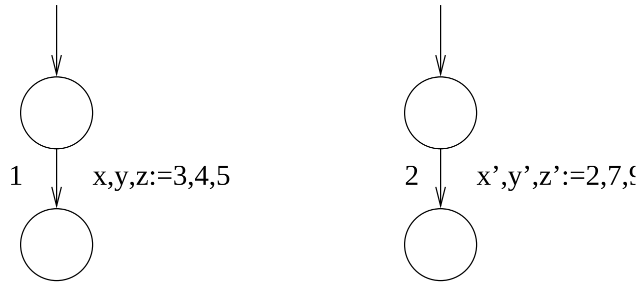


Fig. 6. Parallel Assignment states in two components of a compositional model

With parallel assignment statements, the 6 transitions of Figure 4 would be reduced to two transitions as in Figure 6. The much simpler associated lattice with parallel assignments is shown in Figure 7. The *ConcurrentAtomiser* algorithm allows for a special case of partial order reduction to eliminate the different equivalent interleaving orderings [9]. This has the effect of dramatically reducing the number 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 OS benchmarks in Table 5 provide another important illustration of situations in which the algorithms presented in this paper can be especially beneficial.

Both the Safety and Liveness properties are sequential one-component systems here, so there is no benefit from reducing the interleaving paths as de-

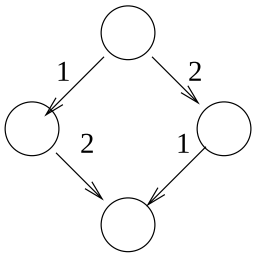
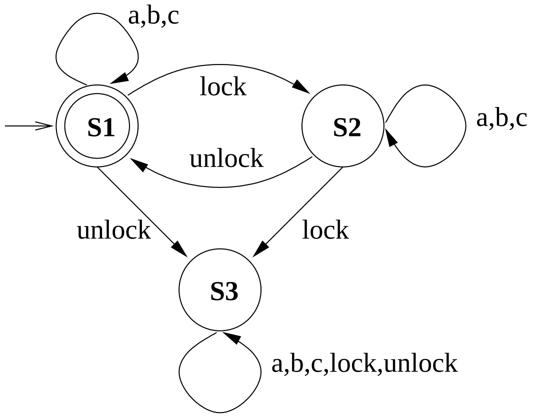


Fig. 7. Lattice of possible paths from transitions in two components with parallel assignment.

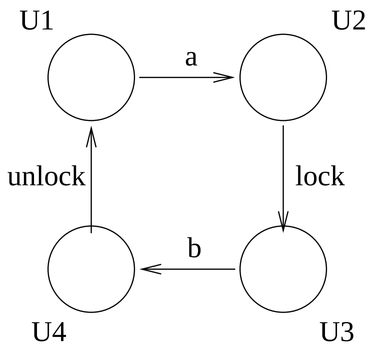
scribed in the previous section.

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 emptiness analysis is performed.

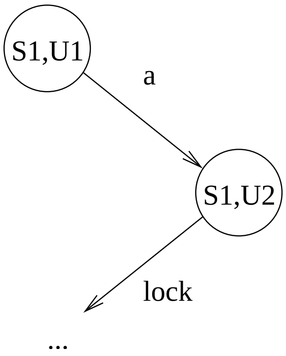
Consider the safety property $M \models$ "locks & unlocks alternate" and the event alphabet $\Sigma = \{a, b, c, lock, unlock\}$:



Suppose we also have an abstract model for our system:



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



In this way our LTL property is translated to emptiness testing with the cartesian product. With this cartesian product construction one finds that the size of the Büchi automaton of the property acts as a scaling factor for the size of the product automaton.

For the Micro-C OS safety property, the Büchi automaton is relatively simple with just 4 states. For the liveness property, however, the automaton 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 [3] and SLAM [2].

4 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 [13] 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.

References

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

[2] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*. ACM Press, 2002.

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

[4] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*. IEEE Computer Society, 2003.

[5] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 25(2-3):129–166, 2004.

[6] S. Chaki, E. M. Clarke, O. Grumberg, J. Ouaknine, N. Sharygina, T. Touili, and H. Veith. State/event software verification for branching-time specifications. In *IFM*. Springer LNCS, 2005.

[7] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing (to appear)*, 2005.

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[9] E. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *Int. J. on Soft. Tools for Tech. Transfer*, 2:279–287, 1999.

[10] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[11] M. Garey and D. Johnson. *Computers and Intractability*. Bell Telephone Laboratories, 1979.

[12] J. Ivers and N. Sharygina. Overview of ComFoRT: A model checking reasoning framework. Technical Report CMU/SEI-2004-TN-018, Carnegie Mellon Software Engineering Institute, 2004.

[13] R. Jhala and R. Majumdar. Path slicing. In *PLDI*. ACM Press, 2005.

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

[15] G. Necula, S.McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, volume 2304. Springer LNCS, 2002.

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