Static Versioning of Global State for Race Condition Detection

Steffen Keul

Dept. of Programming Languages, Universität Stuttgart,
Universitätsstr. 38, 70569 Stuttgart, Germany
steffen.keul@informatik.uni-stuttgart.de
http://www.iste.uni-stuttgart.de/ps/Keul

Abstract. The implementation of concurrent reliable software systems is very difficult. Race conditions on shared data can cause a program's memory state to become inconsistent and result in unpredictable behavior of the software. Much work has been published on analyses to identify access sites to shared data which do not conform to an accepted synchronization pattern. However, those algorithms usually cannot determine if a computation will use a consistent version of more than one shared data object. In this paper, we present a new static analysis algorithm to identify computations which can potentially load values that were stored independently of each other. These uses of global state are affected by race conditions and may yield undesired values during the execution of the program. We show applicability of an implementation of the analysis on several open-source systems.

Keywords: Concurrent Programming, Data Race Condition, Static Program Analysis.

1 Introduction

Concurrent programming is difficult due to subtle programming errors that are hard to find manually. Every interaction of one thread with shared memory needs to obey some global interaction protocol which the programmer has to keep in mind all the time. A violation of that protocol can introduce a race condition, possibly causing an inconsistent memory state. Special support from programming systems to specify and enforce the protocol is not widely available. Worse still, rigorous testing against concurrency bugs might prove in vain, because some of the issues only happen given very specific timing and data inputs, which might be extremely unlikely to encounter. For reliable concurrent software systems however, assuring the absence of harmful race conditions is essential to avoid intermittent misbehavior.

Today, there is a great number of analysis techniques to detect data races, both static and dynamic. Dynamic analyses are in general very precise, as they can identify issues that have happened (or may have happened given different scheduling) along the executed paths of execution [1], [2]. Unfortunately, dynamic

J. Real and T. Vardanega (Eds.): Ada-Europe 2010, LNCS 6106, pp. 111–124, 2010.

analysis can never come to the conclusion that a system is free of bugs, since it will always depend on specific input data into the system. To reach confidence about the absence of certain classes of race conditions, static analysis tools can be used. Static analysis can be conservative, meaning that all existing issues will be found, possibly including a number of false positive warnings.

Existing static race detection techniques usually identify all the places in the source code of a program where access conflicts might occur. Access conflicts occur, if two different threads access a common storage location, one of the threads writes to that location and there exists no ordering between the accesses. To prove the absence of access conflicts, many techniques verify that locking a common mutex variable provides mutual exclusion of the accesses [3], [4], [5], [6], [7], [8]. Artho et al. [9] analyze programs without access conflicts to verify that the sets of objects used inside of critical sections are consistent.

Other static analysis approaches use an extended type system to verify atomicity of functions, e.g. [10].

In this paper, we present a new approach that is able to identify uses of shared memory state that may be negatively influenced by a race condition. Our analysis technique focuses on the way in which global data is used by the program, taking mutex-based synchronization into account where it exists. This paper will refer to explicit mutex-lock and unlock functions, but the technique can be adapted to higher-level synchronization constructs like Java synchronized blocks or Ada protected units. In contrast to other work, we do not necessarily consider access conflicts to be programming errors, but identify calculations that are based on data which is not written into memory during an atomic operation.

The paper is organized as follows. In Section 2, we give some examples of concurrency issues to motivate this work. In Section 3 we outline how our intermediate representation is created, including an explicit representation of interprocedural, interthread data flow. In Section 4 we present the new analysis algorithm. Section 5 gives some measurements performed with an implementation of our algorithm, and Section 6 concludes.

2 Motivating Examples

Our work is motivated by the observation that data race detectors based on the lockset algorithm [5], [7], [8] produce undesired warnings in the following cases. We have encountered situations similar to the example in Listing 1 in industrial control devices. In this example, one thread `thread1` repeatedly produces a new value and stores it to a shared variable `g`. The threads `thread2`, and `thread3` repeatedly consume the value to perform some action. Both of the uses by `thread2` and `thread3` are conflict accesses because `g` can be assigned to and read concurrently without any synchronization. Assuming that reads and writes on `g` are performed atomically, `thread2` would perform as expected. In `thread3` however, there is a possibility that first the value of `g` is loaded, then a new value is written by `thread1` and then that new value is read again by `thread3`. The product `g*g` is then calculated from sensor values of two different read-cycles which is unlikely to be intended semantics.

```
1  int g;
2
3  void *thread1(void *p)
4  {  while (1) g = read_sensor_value();  }
5
6  void *thread2(void *p)
7  {  while (1) act_1(5 * g + 17);  }
8
9  void *thread3(void *p)
10 {  while (1) act_2(g * g);  }
```

Listing 1. Conflict accesses on `g` in `thread2` and `thread3`, but only `thread3` has an inconsistent expression

Using the traditional lockset algorithm, a static analyzer would warn about both calculations, in `thread2` and in `thread3`. Our technique however would accept `thread2` as intended by the programmer, because only one version of the global state is used during the calculation of its expression, but our technique would warn about the expression `g*g` in `thread3` because the uses of shared variables (of `g` in this case) might stem from different versions of the global state.

A less obvious example is presented in Listing 2. Here, the assignments to `g1` and `g2` in Line 4 are both protected by locks `m` and `n`, respectively. Before the values of the global variables are used in line 13, both locks are acquired and might give a false sense of safety to a programmer. Since the assignments are not protected by a common lock, they can still be executed concurrently and, assuming atomic reads and writes, the reads in the main thread behave as if performed without locking. The program is free of access conflicts, but in contrast to traditional lockset-based data race checkers, our analysis is still able to identify the expression `g1+g2` to be run on potentially inconsistent versions.

An important issue are concurrent updates of global state if the updated value depends on the previous state. The simplest example is the increment of a shared counter by several threads. If two threads t_1, t_2 perform the operation $g := g + c$ where g is a global variable and c is some constant, then these operations might be performed in a way that one increment is lost: Let g have the initial value g_0. t_1 loads the value g_0 of g into an register $r_{1,1}$. Then t_2 loads the current value g_0 of g into the register $r_{2,1}$ concurrently. Both perform the addition $r_{i,2} := r_{i,1} + c$. Then the results $r_{1,2}$ and $r_{2,2}$ are both stored back to g in arbitrary order. The resulting value of g is thus the value $g_0 + c$ although given a different interleaving, $g_0 + 2c$ might have resulted.

Traditional data race checkers would fail to discover the potential error however, if the code were executed with separate locking around the read and the store of the shared variable as in Listing 3. In this example, the shared variable has consistent protection, and there is no access conflict. Our technique is able to identify the stale update nonetheless by comparing the versions of global state of the left-hand side (LHS) of the assignment in Line 6 to its right-hand

```
1  void *thread1(void *p)      void *thread2(void *p)
2  {  while (...)             {  while (...)
3     {  mutex_lock(&m);          {   mutex_lock(&n);
4        g1 = ...;                    g2 = ...;
5        mutex_unlock(&m);            mutex_unlock(&n);
6  }  }                        }  }
7
8  int main()
9  {  create(thread1); create(thread2);
10    while (...)
11    {  mutex_lock(&m);
12       mutex_lock(&n);
13       res = g1 + g2;
14       mutex_unlock(&n);
15       mutex_unlock(&m);
16 }  }
```

Listing 2. Free of data races, but the `mutex_lock`-calls around `g1+g2` have no effect

```
1  pthread_mutex_lock(&m);
2    int local = global;
3  pthread_mutex_unlock(&m);
4  local += 17;
5  pthread_mutex_lock(&m);
6    global = local;
7  pthread_mutex_unlock(&m);
```

Listing 3. Nonatomic increments

side (RHS). Only if both sides have the same version then it is certain that no other assignment to the LHS can have occurred since calculating the RHS.

3 Program Representation

Our analysis tool uses knowledge about the syntactical structure of the program as well as its control flow and data flow. Therefore we compile the source code into the high-level graph representation IML [11]. In IML, the program is represented as a graph that contains a statement tree for every function, control flow graphs, and data flow edges from "assignment" to "read" nodes. IML is general enough to be able to represent programs from different source languages, like C, C++, Ada or Java.

The frontend that translates source code into our intermediate representation decides which accesses to variables are atomic assignments and atomic reads. Non-atomic updates of objects are explicitly represented as sequences of read and separate store. In our experiments, we simply assume all reads and writes of objects of primitive types to be atomic. The frontend's implemention can be adapted for architectures where this assumption is not valid.

3.1 Data Flow

We use a context-insensitive, flow-insensitive, but field-sensitive points-to analysis like the one by Pearce [12]. The points-to analysis associates a set of abstract objects with each dereference of a pointer value in the program.

In the program, there are certain actions that start a new thread during runtime of the program (using pthreads, calls to `pthread_create`). We associate an abstract thread with each of these start-sites. Starting from the thread's main function, call graphs for every abstract thread are created. Using points-to sets, we can then calculate the set of abstract objects accessed by every function. Propagating this set along the call graph, from callee to caller, gives the set of nonlocal variables accessed by a function.

Every nonlocal object that is used by functions of different threads is flagged as a shared variable.

In the next step, lockset analysis is performed. Lockset analysis calculates a set of mutex-locks held at a point in the program during any execution of the program. We define contexts of function calls to be the pair of the lockset, which is active at the call site and the abstract thread which executes the call. Thus lockset analysis is context-sensitive with respect to this notion of a context, but it does not incur the complexity of treating functions like inline code at every call site.

To model data flow caused by concurrent assignments, ψ-nodes are inserted into the intermediate representation. At all places in the program, where the value of a shared variable v is used, an explicit "read"-node exists in IML. If the read of v is an access conflict, then an artificial assignment to the shared variable is inserted into the control flow graph directly in front of the read-node. That assignment is called conflict-ψ-node. It takes the form $v := \psi(v_s, v_1, \ldots v_n)$, where v_s is later replaced by the sequentially reaching definitions of v and the $v_1, \ldots, v_n$ represent all possibly reaching definitions of v from concurrent conflict assignments.

If a read of a shared variable v is not an access conflict, then that read happens inside of a critical section protected by a mutex-lock. In this case, a section-ψ-node is created at all control flow entrances into the critical section. The section-ψ-node links the effect of concurrent updates into the sequential data flow of the critical section. Reads of a consistently protected shared variable inside a critical section are only reached by sequential data flow edges. In the critical section case, the ψ-node's arguments for potentially concurrent data flow $v_1, \ldots, v_n$ do not reference assignment-nodes in other threads directly, but so called link-out nodes. Link-out nodes are created at all exists of a critical section to represent the fact, that the critical section's assignments to shared variables become visible at the point of the link-out node. Note that at every entrance into or exit from a critical section, only one section-ψ-node or link-out node is created. The single node summarizes the data flow of all shared variables accessed in the critical section.

Once ψ-nodes are inserted, data flow links from definitions to their uses need to be calculated and stored into the intermediate representation. In our implementation, this is done by a transformation into SSA-form, as in [13].

Note, that the output of lockset-based data races checkers can be extracted from our intermediate representation, for shared variables which are read in at least one place: simply enumerate all conflict-ψ-nodes.

3.2 Summary Edges

To speed up the analysis, as described in Section 4.1, we need to efficiently propagate data flow facts across function call sites without re-analyzing the entire call graph rooted at that call site. Therefore, we add summary edges to the data flow graph. Summary edges have been used successfully in interprocedural slicing algorithms [14]. If inside of a callee function f, the value of parameter p reaches a set T of output parameters of f, then a summary edge is created at every call site for f from the copy-in node that corresponds to p to every copy-out node that corresponds to any node in T. Similarly, summary edges are created for nonlocal objects accessed inside of f as if they were passed as parameters.

4 Analysis Algorithm

In this section, we will explain our new analysis. It reasons about the versions of global state that are used in expressions. The analysis is performed in two successive phases: in the first phase, all expressions are annotated with the version of the global state they operate on. In the second phase, warnings are generated for suspicious constructs.

4.1 State-Version Analysis

In our program representation, the execution of a ψ-node identifies one unique observation of the global state. If two values stem from the same run-time instance of a ψ-node, then those two values were stored during execution of the same critical section. This property is ensured by the placement strategy for ψ-nodes. The state-version analysis uses this property to identify versions of global state with versions of the ψ-nodes whose values flow into the expressions.

We number all ψ-nodes in the program from 1 to n. The analysis uses the state space $\mathcal{L}$ to describe the visible global state at a point in the program:

$$\begin{aligned} \mathbb{V} &= \{\bot, \top, \psi_1, \ldots, \psi_n\} \\ \mathcal{L} &= \{s \mid s : \mathbf{Var} \to \mathbb{V}\} \end{aligned}$$

In every state $m \in \mathcal{L}$, each abstract variable $v \in \mathbf{Var}$ of the program is mapped to its version from the set $\mathbb{V}$. The version can be either $\top$ if it is not influenced by any data flow originating in a different thread, $\bot$ if it can be influenced by more than one ψ-node, or $\psi_i \in \{\psi_1, \ldots, \psi_n\}$ if it can be influenced by only the most recent execution of the ψ-node with number i. Consequently, two variables a, b have a consistent version at a point in the program, if the analysis determines $(m\,(a) = m\,(b)$ and $m\,(a) \neq \bot)$ or $m\,(a) = \top$ or $m\,(b) = \top$.

Intraprocedural versioning. The intraprocedural analysis starts off with the optimistic assumption that a caller function does not propagate any versioned values into the callee function. This imprecision will be healed later during interprocedural analysis (see below in this section). Due to this, the functions can be analyzed separately in reversed topological order of the call graph. Before a caller function is entered, its callees will already be analyzed. An exception to this rule are cycles of the call graph which require iteration.

Consider, that versions of global state are created by either a ψ-node in a current function, or by a ψ-node in a callee function. Versions created in or propagated out of a callee are never equal to any version created in the current function, even if they originate from the same ψ-node (possible in case of recursion). Consequently, analysis of a call site can generate new versions of the global state. That information can be queried from the result of that function's analysis. The versions of nonlocal variables that are read in the callee, but not definitely updated, only need to be propagated across the call site using the summary edges (see Section 3.2).

The intraprocedural analysis is simple, if the ψ-nodes of the function are not contained in cycles of the control flow graph. In this case, every ψ-node can generate at most one version. If a ψ-node is contained in a cycle, then that node can generate a different version during every iteration of the cycle. To deal with a statically unknown number of different versions, the analysis simplifies the problem and tracks only the global version of the most recent execution of each ψ-node along every control flow path.

The data flow equations are solved by a standard monotone framework for an iterative forward data flow analysis. The initial value is $\iota \in \mathcal{L}$:

$$\iota = \forall v \in \mathbf{Var} : v \mapsto \top$$

At control flow confluence points, the states of the predecessor blocks $p_1, \ldots, p_n$ are joined by the meet operation $\bigsqcap\{p_1, \ldots, p_n\} = p_1 \sqcap \ldots \sqcap p_n$.

$$\forall x, y \in \mathbb{V} : x \diamond y = \begin{cases} \top, & \text{if } x = y = \top \\ \psi_x, & \text{if } x = \psi_x \wedge y \in \{\psi_x, \top\} \\ \psi_y, & \text{if } y = \psi_y \wedge x \in \{\psi_y, \top\} \\ \bot, & \text{otherwise} \end{cases}$$

$$a \sqcap b = \forall v \in \mathbf{Var} : v \mapsto a(v) \diamond b(v)$$

To determine the version of an expression, the versions $r_1, \ldots r_n \in \mathbb{V}$ of all variables that contribute values to the expression are joined using the operation $r = r_1 \diamond \ldots \diamond r_n$.

The intraprocedural analysis can be summarized by the following equations. We use $\mathrm{in}(s)$ to denote the information that flows into a statement s and $\mathrm{out}(s)$ for information that flows out of s. The set of predecessor statements of s is denoted $\mathrm{pred}(s)$.

$$\mathrm{in}(s) = \begin{cases} \iota, & \text{if } s \text{ is the function entry} \\ \bigsqcap_{p \in \mathrm{pred}(s)} \mathrm{out}(p), & \text{otherwise} \end{cases}$$

$$\mathrm{out}(s) = \begin{cases} f_{:=}\left(\mathrm{in}(s)\right), & \text{if } s \text{ is an assignment} \\ f_{\psi_i}\left(\mathrm{in}(s)\right), & \text{if } s \text{ is the } \psi\text{-node with number } i \\ f_{\mathrm{call}}\left(\mathrm{in}(s)\right), & \text{if } s \text{ is a call site} \\ \mathrm{in}(s), & \text{otherwise} \end{cases}$$

We distinguish between strong and weak updates of assignment-nodes. A strong update occurs, if the analysis can definitely decide that the left-hand side (LHS) abstract object of the assignment is a single object and the value is fully replaced by the assigned value. Weak updates happen whenever the LHS-object is not statically known to represent exactly one single object. This can happen in static analysis, for example, if assignments are made to array elements or to heap objects of which multiple instances might exist. At assignment-nodes, the version v_{RHS} of the expression on the right-hand side is calculated. If the assignment is a strong update, then the original version of the LHS can be killed and replaced by the new version. If the assignment is a weak update then the operation $\diamond$ is used to combine the versions of LHS and RHS.

$$f_{:=}(\mathrm{pre}) = v \mapsto \begin{cases} pre(v) \diamond v_{\mathrm{RHS}}, & \text{if weak update and } v \in \mathrm{LHS} \\ v_{\mathrm{RHS}}, & \text{if strong update and } v \in \mathrm{LHS} \\ pre(v), & \text{otherwise} \end{cases}$$

A ψ-node ψ_i represents a potential assignment of the data flow originating in concurrent definitions to a set V of some variables. Each of the variables in V becomes consistent to the version ψ_i of the global state. But since the analysis only represents the most recent execution of a ψ-node, all variables $K = \mathbf{Var} \setminus V$ are not consistent to ψ_i anymore. This is reflected by setting all variables of K, which had the version ψ_i before, to $\bot$.

$$f_{\psi_i}(\mathrm{pre}) = v \mapsto \begin{cases} \psi_i, & \text{if } v \in V \\ \bot, & \text{if } v \in K \wedge \mathrm{pre}(v) = \psi_i \\ \mathrm{pre}(v), & \text{otherwise} \end{cases}$$

Function calls are handled as follows. Let $\mathrm{pre} \in \mathcal{L}$ denote the state that is active directly before the call site. That state is mapped to a new state $p' \in \mathcal{L}$ using summary edges (see Section 3.2). Thus, the state p' models the result of the call if all ψ-nodes inside of the callee are ignored. Similarly, let $R \in \mathcal{L}$ represent the meet of the results of the intraprocedural analysis of all possible callee functions without any influence from the caller. The following transfer function f_{call} calculates a conservative combination of p' and R. Any version propagated from the caller into the callee will always be different from any version that the caller produces. Therefore all variables reached by versions of the caller and by versions of the callee indicate that the callee only performs may-defs and leaves the variable in doubtable state. The variable's state is conservatively reduced to $\bot$. If the callee produces a different version of the same ψ-node (in case of recursion) then the version of the callee will always be newer, and the variables containing the older version are reset to $\bot$.

$$f_{\text{call}}(\text{pre}) = v \mapsto \begin{cases} \top & \text{if } p'(v) = R(v) = \top \\ \psi_j & \text{if } p'(v) = \top \wedge R(v) = \psi_j \\ \psi_i & \text{if } p'(v) = \psi_i \wedge R(v) = \top \wedge \{x \in \mathbf{Var} | R(x) = \psi_i\} = \emptyset \\ \bot & \text{otherwise} \end{cases}$$

It can be shown, that the transfer functions and meet operation only perform transitions that are monotonously decreasing with respect to the partial order defined here. For all states $x \in \mathcal{L}$, let $T_x = \{v | x(v) = \top\}$, let $\forall 1 \leq i \leq n : \Psi_{x,i} = \{v | x(v) = \psi_i\}$. The partial order of states is defined as:

$$\begin{aligned} a < b \Leftrightarrow & (T_a = T_b \text{ and } \forall 1 \leq i \leq n : \Psi_{a,i} \subseteq \Psi_{b,i} \text{ and } \exists 1 \leq j \leq n : \Psi_{a,j} \subsetneq \Psi_{b,j}) \\ & \text{or } (T_a \subsetneq T_b \text{ and } \forall 1 \leq i \leq n : \Psi_{a,i} \subseteq \Psi_{b,i}) \end{aligned}$$

Since all operations are monotonous, and a single smallest state exists ($\forall v \in \mathbf{Var} : v \mapsto \bot$), and the condition $|\mathcal{L}| < \infty$ holds, the analysis is guaranteed to reach the largest fixed point.

Interprocedural analysis. As described in the previous paragraph, all functions are analyzed separately, based on the assumption that all of the function's input data have version $\top$. Versions that are created inside of a function are propagated to the function's caller, but the propagation from the caller into its callees has been omitted so far. This section will outline how context information is added to complete the analysis.

The fundamental observation is that no version created in a caller-function can be equal to a version created in one of its callees. The following algorithm is performed separately for every function in topological order of the call graph to produce the final result for a function. Inside of cycles of the call graph, the functions are processed in arbitrary order and the cycle is iterated until one iteration yields the same results as the previous one.

1. Use the state $c_c \in \mathcal{L}$ directly before a call site,
2. initialize all nodes inside of the callee to $\forall v \in \mathbf{Var} : v \mapsto \top$,
3. perform copy-in assignments for parameters and nonlocal objects accessed in the callee-function to map c_c into the callee, producing the state $c \in \mathcal{L}$,
4. for every $v_c \in \mathbf{Var}$: if $c(v_c) \neq \top$, then propagate the version $c(v_c)$ along the intraprocedural data flow edges in the callee-function. Assume this propagation reaches a data flow node, which identifies a variable v_s and its state $s \in \mathcal{L}$. Produce a new state for that node: $(v_s \mapsto c(v_c), \forall v \neq v_s : v \mapsto s(v))$,
5. calculate the interprocedural meet $\Cap$ of the result of the intraprocedural analysis and the new states. In contrast to the intraprocedural meet operation $\sqcap$, a ψ_i originating from the intraprocedural analysis is never treated as equal to any ψ_j of the interprocedural analysis.

$$a \Cap b = v \mapsto \begin{cases} \top, & \text{if } a(v) = b(v) = \top \\ \psi_a, & \text{if } a(v) = \psi_a \wedge b(v) = \top \\ \psi_b, & \text{if } b(v) = \psi_b \wedge a(v) = \top \\ \bot, & \text{otherwise} \end{cases}$$

Note that the number of different contexts identified by a state c_c before a call site might become very large. However, it is easily possible to use the $\sqcap$ operation on several different contexts to bound the number of contexts at the expense of a more conservative analysis result that produces more false positive warnings. Since the non-recursive functions are processed in topological order, all contexts for a callee function are available before that function is analyzed. Inside of cycles of the call graph, a viable option is to allocate only one context per function.

Figure 1 gives an example of a run of the analysis on an example function. Assume that another thread is executed concurrently to the function `caller`. That other thread acquires the mutex `m`, updates the value of the global variable `g` and releases `m`. In the example, ψ-nodes are already inserted. In line 6, the value of `g` is linked into the critical section. In lines 24 and 26, ψ-nodes represent the conflict uses of `g`. The function `use` is omitted, it can be assumed to have an empty implementation. In the "intra" column, the fixed point reached by the intraprocedural analysis is displayed. The column "inter" displays the states created by the interprocedural part of the analysis and the column "result" the final result after intra- and interprocedural information has been joined. The column "xpr" states the version of the expression calculated in the line of source code, if any. The column "xpr" is consulted during the warning-generation pass. Note that the last three columns depend on context. If there are more than one call site for `f`, and the active state before those call sites differs, then multiple instances of these three columns will be created.

Note that in line 14 the version of `z` is $\bot$ because the loop can have been exited by the `break`-statement in line 10. If that were the case, `z`'s version might not be from the latest execution of the critical section. In contrast, the use of `z` in line 12 is always consistent with `x`'s version.

In line 30, the previous version ψ_4 of the global variable `g` is propagated across the call-site using summary edges. Since `f` only performs may-defs to `g`, `g` still has the version ψ_4 directly after the call-site after the intraprocedural versioning. In the exit-node of `f`, `g` has the version ψ_1, therefore its version is reduced to $\bot$ in the interprocedural phase.

In line 18, no warning would be created after the intraprocedural phase. In that line `x` has version ψ_1 and `y` has version $\top$, which results in ψ_1 for the expression. Only after context-information is propagated across the assignment in line 17 the conflict becomes visible. In this case, the formal parameter `q` has version ψ_4 which is assigned to `y` and therefore two different versions are joined in the expression `x + y`. A similar effect happens on the return value of `f`. In the intraprocedural phase, both `p` and `q` have version $\top$, but after the interprocedural propagation, the return value's version is reduced to $\bot$.

	intra						inter						result						xpr
	x	y	z	g	p	q	x	y	z	g	p	q	x	y	z	g	p	q	
1 `int f(int p, int q)`																			
2 `{`	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	4	3	4	⊤	⊤	⊤	4	3	4	
3 `int x, y, z;`																			
4 `do {`	1	1	1	1	⊤	⊤	⊤	⊤	⊤	4	3	4	1	1	1	⊥	3	4	
5 `mutex_lock(&m);`																			
6 `/*`$g = \psi_1(g, \dots)$`*/`	⊥	⊥	⊥	1	⊤	⊤	⊤	⊤	⊤	⊤	3	4	⊥	⊥	⊥	1	3	4	1
7 `x = g;`	1	⊥	⊥	1	⊤	⊤	⊤	⊤	⊤	⊤	3	4	1	⊥	⊥	1	3	4	1
8 `y = g;`	1	1	⊥	1	⊤	⊤	⊤	⊤	⊤	⊤	3	4	1	1	⊥	1	3	4	1
9 `mutex_unlock(&m);`																			
10 `if (cond2) break;`																			⊤
11 `z = x;`	1	1	1	1	⊤	⊤	⊤	⊤	⊤	⊤	3	4	1	1	1	1	3	4	1
12 `use(x + z);`																			1
13 `} while (cond1);`	1	1	⊥	1	⊤	⊤	⊤	⊤	⊤	⊤	3	4	1	1	⊥	1	3	4	⊤
14 `if (x + z)`																			⊥
15 `use(x + y);`	1	1	⊥	1	⊤	⊤	⊤	⊤	⊤	⊤	3	4	1	1	⊥	1	3	4	1
16 `else {`																			
17 `y = q;`	1	⊤	⊥	1	⊤	⊤	⊤	4	⊤	⊤	3	4	1	4	⊥	1	3	4	4
18 `use(x + y);`																			⊥
19 `}`																			
20 `return p + q;`	1	1	⊥	1	⊤	⊤	⊤	4	⊤	⊤	3	4	1	⊥	⊥	1	3	4	⊥
21 `}`																			
	a	b	g				a	b	g				a	b	g				
22 `void caller()`																			
23 `{`	⊤	⊤	⊤				⊤	⊤	⊤				⊤	⊤	⊤				
24 `/*`$g = \psi_3(g, \dots)$`*/`	⊤	⊤	3				⊤	⊤	⊤				⊤	⊤	3				3
25 `int a = g;`	3	⊤	3				⊤	⊤	⊤				3	⊤	3				3
26 `/*`$g = \psi_4(g, \dots)$`*/`	3	⊤	4				⊤	⊤	⊤				3	⊤	4				4
27 `int b = g;`	3	4	4				⊤	⊤	⊤				3	4	4				4
28																			
29 `use(a + b);`																			⊥
30 `use(f(a, b));`	3	4	⊥				⊤	⊤	⊤				3	4	⊥				⊥
31 `}`																			

Fig. 1. Versioning analysis applied on example thread. Another thread performs updates on the shared variable `g`, protected by mutex `m`.

4.2 Suspicious Pattern Recognition

After version numbers have been annotated to the expressions of the program, Warnings are generated for suspicious ones. The warning generator traverses the statement trees of all functions in post-order. The statement tree contains simple nodes for accesses to variables and composite nodes for statements or expressions. A node can be marked to express that a warning has been generated for it. The version determined for a node, and its marking-state are propagated upwards in the tree.

Whenever a "read"-node on some shared variable is encountered, that shared variable's version is loaded. When a composite node is evaluated, its version is

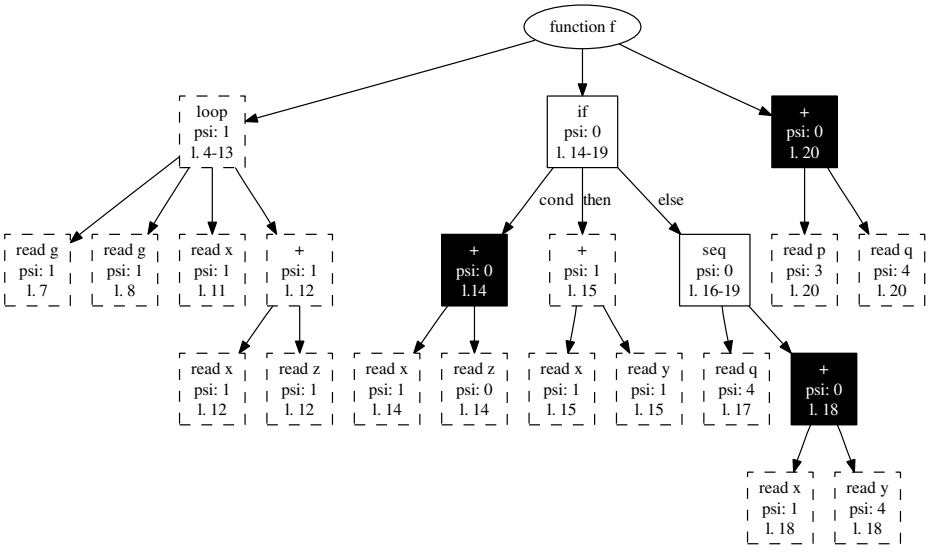

Fig. 2. Warning generation for the function `f` from Figure 1

calculated as the meet of all of its child nodes' versions. If the meet of only the unmarked child nodes of a composite node is $\perp$ then a warning will be generated for the composite node. If a warning is created for a composite node or if all of its child nodes are marked, then the composite node will be marked.

The marking of tree nodes after a warning has been generated suppresses multiple warnings about the same issue. However, if for example, two statements from a sequence of statements generate individual warnings, then the warning about the entire sequence of statements might be suppressed. As a consequence, when inspecting source code, warnings must be considered within the individual syntactical context they were created for.

In Figure 2, the warning generation process is illustrated for the function `f` from Figure 1. The black nodes represent nodes that generate a warning. Nodes with solid borders do not generate warnings on their own, because of their children's marking. Nodes with dashed borders are considered intended code. They do not generate a warning and are ignored.

5 Empirical Results

We have implemented the algorithm described in the previous sections in the Bauhaus system [11]. We have used the analysis tool on different open source programs as shown in Figure 3. Manual inspection of the results for `aget-0.4` showed that our tool was able to issue a warning about a race condition in a comparison in the function `updateProgressBar`, which can result in undesired output.

		results		efficiency		
Tool	SLoC	warnings	ψ-nodes	s. edges	intra	inter
aget-0.4	0.8k	10	39	<0.2s		
linuxdown	1.4k	29	109	<0.4s		
aaxine	46k	621	2,599	6.5s	0.8s	6.1s
clamd	66k	6,667	13,062	25s	92s	879s

Fig. 3. Measurements on different open source programs

In Figure 3 our measurements are presented. "SLoC" are measured using `SLOCCount` [15]. "warnings" is the number of warnings generated by the process explained in Section 4.2, "ψ-nodes" is the number of ψ-nodes inserted into IML, "s. edges" is the time needed to create summary edges, "intra" is the time for the intraprocedural analysis phase, "inter" the time for the fully context-sensitive interprocedural phase. Note that the interprocedural phase could be done much more efficiently if less contexts were distinguished as explained in Section 4.1.

The measurements show that our analysis runs fast on real programs. We expect to improve precision of the shared variable classification using escape analysis [16], resulting in better overall efficiency. At present, the number of false positive warnings is still too high for a complete manual inspection.

6 Conclusion

We have presented a new static analysis that identifies expressions in a concurrent shared-memory program which depend on potentially inconsistent global state. The analysis is able to find bugs, that previous techniques could not. It can handle atomic reads and writes of shared variables and the decision if an access is atomic can be configured by the frontend. The analysis supports mutex-synchronization or other schemes that can be transformed into an explicit data flow representation. Using the hierarchical warning generation, intuitive warnings can be generated. We have found that the analysis is very efficient and able to handle larger programs.

Acknowledgments. The author would like to thank Erhard Plödereder and the anonymous reviewers for valuable comments on an earlier version of this paper.

References

1. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
2. Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and precise datarace detection for multithreaded object-oriented programs. In: PLDI 2002, pp. 258–269. ACM Press, New York (2002)

3. Kahlon, V., Sinha, N., Kruus, E., Zhang, Y.: Static data race detection for concurrent programs with asynchronous calls. In: ESEC/FSE 2009, pp. 13–22. ACM Press, New York (2009)
4. Raza, A., Vogel, G.: RCanalyser: A flexible framework for the detection of data races in parallel programs. In: Kordon, F., Vardanega, T. (eds.) Ada-Europe 2008. LNCS, vol. 5026, pp. 226–239. Springer, Heidelberg (2008)
5. Voung, J.W., Jhala, R., Lerner, S.: RELAY: Static race detection on millions of lines of code. In: ESEC-FSE 2007, pp. 205–214. ACM, New York (2007)
6. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Context-sensitive correlation analysis for race detection. In: PLDI 2006, pp. 320–331. ACM, New York (2006)
7. Engler, D., Ashcraft, K.: RacerX: Effective, static detection of race conditions and deadlocks. In: SOSP 2003, pp. 237–252. ACM, New York (2003)
8. Sterling, N.: WARLOCK – a static data race analysis tool. In: Proceedings of the USENIX Winter 1993 Technical Conference, San Diego, CA, USA, pp. 97–106 (1993)
9. Artho, C., Havelund, K., Biere, A.: High-level data races. Software Testing, Verification and Reliability 13(4), 207–227 (2003)
10. Flanagan, C., Freund, S.N., Lifshin, M., Qadeer, S.: Types for atomicity: Static checking and inference for java. ACM Trans. Program. Lang. Syst. 30(4), 1–53 (2008)
11. Raza, A., Vogel, G., Plödereder, E.: Bauhaus - a tool suite for program analysis and reverse engineering. In: Pinho, L.M., González Harbour, M. (eds.) Ada-Europe 2006. LNCS, vol. 4006, pp. 71–82. Springer, Heidelberg (2006)
12. Pearce, D.J., Kelly, P.H.J., Hankin, C.: Efficient field-sensitive pointer analysis for c. In: PASTE 2004, pp. 37–42. ACM, New York (2004)
13. Staiger, S., Vogel, G., Keul, S., Wiebe, E.: Interprocedural static single assignment form. In: WCRE 2007, October 2007, pp. 1–10. IEEE Computer Society, Los Alamitos (2007)
14. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI 1988, pp. 35–46. ACM, New York (1988)
15. Wheeler, D.A.: SLOCCount. v2.26, `http://www.dwheeler.com/sloccount`
16. Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for java. SIGPLAN Not. 34(10), 1–19 (1999)