WARLOCK
A Static Data Race Analysis Tool

Nicholas Sterling, SunSoft, Inc.

Concurrent programming is becoming available to the masses, bringing with it the potential for new types of errors such as deadlocks and data races. This paper describes a static data race analysis tool less ambitious than most, written for use with SunSoft's Solaris operating system. The basic algorithm is described, and a sample use of the tool is discussed. Some complicating factors of real code are presented, along with the means chosen to deal with them. The current status of the tool and some preliminary experiences are discussed.

1.0 Introduction

While concurrent programming has been explored in research communities for decades, it has only been available from vendors targeting various specialized markets. We are approaching a milestone in that such capabilities will soon be widely available on mainstream workstations. SunSoft's Solaris operating system is now multithreaded and allows application software access to the multithreading model through library calls. A new POSIX standard defines primitives which allow the development of portable multithreaded applications [POSIX]. Major software vendors are beginning to make use of threads in order to improve performance on both uniprocessors and multiprocessors.

In the multithreading model, a program consists of one or more threads of control which share a common address space and most other program resources. The various threads of control may execute in parallel on a multiprocessor; even on a uniprocessor the interleaving of progress on various threads is non-deterministic. Threads must acquire and release locks associated with shared data in order to reliably produce the intended results; where they fail to do so, data may become corrupted. This is a type of "data race" — a situation in which a program may produce different results when run repeatedly with the same input.

The benefits of multithreading technology are obvious: increased parallelism on multiprocessors, and the ability to better express the asynchronicity inherent in many problems. Multithreading can even improve the performance of programs run on uniprocessors. But the technology also brings with it significant challenges. Research on concurrent programming clearly shows that debugging and testing of concurrent programs is more difficult than it is for sequential programs, largely because of the non-determinism caused by differences in execution order of code within the various threads of control [MH89].

Data races are easy problems to introduce — simply accessing a variable without first acquiring the appropriate lock can cause one — and they are generally very difficult to find. Visible symptoms of a data race generally manifest only if two threads access the improperly protected data at nearly the same time; hence a data race may easily run for months without showing any signs of a problem. It would be extremely difficult to exhaustively test the various concurrency states for even a simple multithreaded program, so conventional testing and debugging are not an adequate defense against data races.

Furthermore, because threads which fail to use locks properly can generally interfere with one another at various points in their execution, the symptoms may be different each time problems do occur. To make things worse, post-mortem analysis of a crashed program will probably show which data are in an invalid state, but will generally offer no clues as to how they reached that state. When such problems are discovered in the field, which is likely for highly intermittent problems, customers often do not have time to help debug the problem.

Considerable research has focussed on new tools to address these problems [MH89]. One technique is to analyze the program source code, looking for potential problems. This technique is called "static analysis" to distinguish it from dynamic techniques, which involve running the program and analyzing its actual run-time behavior. Since dynamic analysis requires that the program be forced through the program states of interest, its use with multithreaded software is problematic.

Depending upon the semantics of the source language, static analysis can point out many types of potential problems, such as references to uninitialized variables, waiting for threads of control which have already terminated and been released, and references to variables with non-deterministic values [TO80]. Such analysis is often based on the creation of a concurrency graph, in which each node represents a unique possible state of the entire program — i.e. a unique set of locations for the various threads of control in their own individual flow graphs [Tay83]. For large programs with substantial concurrency this technique suffers from a combinatorial explosion in the size of the graph, although mechanisms can often be applied to deal with this explosion [YT88].

The tool described in this paper, called "warlock", is a static analysis tool less ambitious than the research activity mentioned. It is interesting primarily for its relevance to a widely available platform for concurrent programming and for the tradeoffs made in order to produce a meaningful tool in the context of the language and multithreading model. This paper is not so much about the science of static analysis as it is about the engineering of this particular tool.

2.0 Locking in Solaris

It is assumed that the reader has a rudimentary familiarity with the model of multithreading implemented in Solaris [Pow91, Kle92]. A brief review of the locking facilities follows.

Solaris supports simple mutex locks which are acquired and released with the calls

```
mutex_lock(mutex_t*)
mutex_trylock(mutex_t*)
mutex_unlock(mutex_t*)
```

These are the most efficient locks to use for simple mutual exclusion. **Mutex_trylock**() returns a failure code rather than blocking the thread if the lock is unavailable.

Also supported are readers/writer locks, which decrease contention by allowing multiple readers to hold the lock at one time. The following calls are supported:

```
rw_rdlock(rwlock_t*)
rw_wrlock(rwlock_t*)
rw_tryrdlock(rwlock_t*)
rw_trywrlock(rwlock_t*)
rw_unlock(rwlock_t*)
```

Rw_tryrdlock() and **rw_trywrlock**() return a failure code rather than blocking the thread if the lock is unavailable.

Counting semaphores and condition variables are also available but, for reasons to be discussed later, warlock ignores them.

The granularity of data protected by locks is entirely up to the developer, and in practice varies considerably depending upon the amount of contention expected, the weight of the operations to be performed, and other factors. In one case an entire library may be completely serialized through the use of a single lock, while in another case a single structure may contain multiple locks, each of which protects a few members of the structure. Very commonly a structure will contain a single lock to protect its members, or a single lock will protect a collection of some kind, such as a linked list.

3.0 Warlock Background

Warlock was initially created as an aid to developers of the kernel and programs which comprise the Solaris operating system. A tool similar to the well known UNIX **lint** utility was visualized, with makefile targets set up to run the tool automatically. Also a static analysis tool, **lint** serves as a familiar model for C programmers.

For such a tool to do its analysis using a concurrency graph would have been problematic. Multithreaded kernel code is almost completely event-driven; so is a high percentage of application software, notably servers and window-based tools. In event-driven software, code is typically structured so that it is almost all concurrently executable. Such a breadth of concurrency, and the sheer size of some of this software, would tend to make the concurrency graphs intractably large.

Furthermore, a scalable mechanism was needed. Much of the code to be analyzed consists of libraries, which one would like to analyze independently of any particular program's use of them. Moreover, the development process itself involves a large number of engineers who cannot be called together to participate in the analysis procedure. For these reasons, as well as the time it would take to perform a large analysis, the tool would have to allow the code to be analyzed in smaller pieces.

Also, it was felt that the most interesting information to be gained from such a tool was potential data races. They were felt to be the problems most likely to remain undetected in code which seemed to work. Deadlocks were also of interest, but a separate internal tool had already been written to help find them.

As a result, a simpler approach to finding data races was chosen. Rather than explicitly looking for data races using a concurrency graph, which requires so much processing, warlock simply checks for consistency in the code's use of locks. In so doing it detects what was felt to be the most common cause of data races: failure to hold the appropriate lock while accessing a variable.

Addressing the problem in this manner presupposes that each lock is associated with, or "protects," an unchanging set of variables. While there is no requirement that locks be used in this manner (one could protect a variable with one lock during one phase of a program's execution, and then with a different lock during another phase), they almost always are. With a few degenerate exceptions, properly protecting variables when the correct method changes over time would be highly error-prone, especially in code which is nearly all concurrently executable. So warlock generally assumes that locks are used consistently, and provides special mechanisms for dealing with exceptions.

Warlock discovers locking problems basically by tracing the execution of every path through the code, noting which locks are held each time a variable is accessed. A symbol table is maintained for variables, and for each entry in that table a list is maintained of locks consistently held when that variable was accessed. After all execution paths have been traced, if a variable's list of locks consistently held is empty, then — as far as warlock can tell — that variable is not properly protected by a lock, and any code which accesses that variable is subject to data races.

Consider the following simple (and contrived) example:

```
mutex_t lock1, lock2;
int var1, var2;

func1()
{
    mutex_lock(&lock1);
    if (var1++)
        func3();
    mutex_unlock(&lock1);
}

func2()
{
    mutex_lock(&lock2);
    var2++;
    mutex_unlock(&lock2);
}

func3()
{
    var2 = 9;
}
```

Warlock recognizes that **func1**() and **func2**() are not called from anywhere, so it traces the execution of each, descending into **func3**() when it is called to trace its execution as well. Warlock traces parts of **func1**() twice, once with the expression controlling the **if** statement true and once with it false. When all paths have been traced, warlock discovers that **lock1** was held every time **var1** was accessed, but there is no lock which was held every time **var2** was accessed (during one access **lock1** was held, and during the other **lock2** was held). Therefore uses of **var2** are potential data races.

With real software, there is much to complicate this basic algorithm. There are also other useful results which can be produced.

Clearly, warlock does not detect all data races. It is possible (but probably relatively rare) to write code which produces non-deterministic results even while all accesses are properly protected by locks, as does the following code:

```
mutex_lock(&foo->lock);
count = foo->count;
mutex_unlock(&foo->lock);
...
mutex_lock(&foo->lock);
foo->count = count + 1;
mutex_unlock(&foo->lock);
```

Furthermore some variables, while protected consistently, are protected by a mechanism more complex than simply acquiring a lock. For example, some data is effectively protected by semaphores and condition variables, while warlock only analyzes the use of mutex locks and readers/writer locks. However, mutex and readers/writer locks are typically used to protect most data because of their greater efficiency.

Mutex and readers/writer locks are suitable for analysis since a thread uses them by bracketing code with calls to acquire and release the lock. Semaphores, on the other hand, may be used in a variety of ways, frequently with one thread doing a P operation and another thread the corresponding V operation:

```
produce()
{
    static int next_empty_buffer = 0;
    sema_p(&empty_buffers);
    fill(next_empty_buffer);
    if (++next_empty_buffer >= NUM_BUFFERS)
        next_empty_buffer = 0;
    sema_v(&full_buffers);
}

consume()
{
    static int next_full_buffer = 0;
    sema_p(&full_buffers);
    if (++next_full_buffer >= NUM_BUFFERS)
        next_full_buffer = 0;
    empty(next_full_buffer);
    sema_v(&empty_buffers);
}
```

This makes it more difficult to identify which lines of code are actually under the protection of a semaphore; the rules could be different for each abstraction created using semaphores. Furthermore, it is difficult to distinguish between different elements of an array in static analysis. For example, it would be difficult for a static analysis tool to discern that in routine **consume**() above, because the call to **empty**() follows the increment of **next_full_buffer**, the wrong buffer is being accessed. As a result, the program might attempt to empty and fill a given buffer at the same time. Counting semaphores are often used with arrays in this manner.

Similarly, condition variables are often not used in a simple bracketing manner. In those cases where semaphores and condition variables *are* used in a bracketing manner, it is sometimes possible to modify the code so that warlock sees the locks as mutex locks, allowing the code to be analyzed.

4.0 Using Warlock — an Overview

In order to trace the execution of the program, warlock needs a representation of each function showing its activity relevant to the analysis. A modified version of SunPro's ANSI C compiler produces for each **.c** file not only a **.o** file, but also a **.wl** file. This **.wl** file contains information about the flow of control in each function, as well as each access to a variable or operation on a mutex or readers/writer lock.

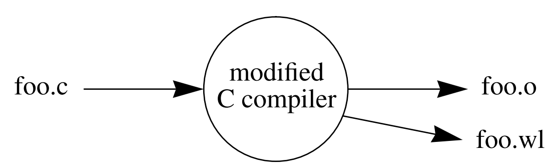

FIGURE 1. Creating Warlock Information Files

When all of the code to be analyzed has been compiled using this modified compiler, the user invokes warlock to perform the analysis. Used in this way, warlock is an interactive program. The user issues commands to load the relevant **.wl** files and gives other hints which will improve the analysis. In practice, these commands are generally stored in a start-up file and are executed automatically when warlock is invoked.

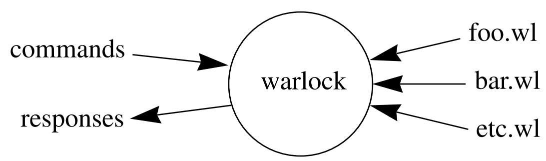

FIGURE 2. Interacting with Warlock

Then the user directs warlock to perform the analysis. The analysis may produce messages saying that a lock was released when it was not held, that a covered lock was acquired while its cover was not held, or the like. If a program has been running successfully, these particular messages usually don't reflect true problems in the code, but rather limitations in warlock's ability to understand the code. Through the use of conditional compilation one can present to warlock a somewhat simplified version of troublesome code fragments, allowing warlock to conduct a meaningful analysis.

Once the analysis completes without such errors, the user asks to see which variables are not consistently protected by locks. The user may make assertions to warlock about which variables are supposed to be protected by a lock and about which locks are supposed to be held whenever a function is called. Running the analysis with such assertions in place will show the user where the assertions are violated. Again, these violations may represent limitations in warlock's ability to understand the code; or they may represent actual problems in the code.

While the separate, interactive back end (Figure 2) provides the most powerful access to warlock's capabilities, warlock can be used in a more **lint**-like manner as well. The code to be analyzed can be annotated with special warlock directives which provide information warlock cannot glean from the code itself. Then a script can be run which compiles the source using the modified compiler, performs the analysis, and reports potential problems. Ultimately SunSoft plans to instrument all kernel code with appropriate directives and perform periodic analyses of the entire kernel by running the **lint**-like script from makefiles.

While the interactive use of the back end is interesting for its ability to answer questions about the behavior of a program relevant to locks, this paper will not discuss that interface. Instead the focus is on the use of source annotation to influence the analysis. By annotating common header files, the burden of the individual developer in running warlock is decreased and useful documentation is provided at the same time.

5.0 Some Complications

The example in section 3 was straightforward, but complications abound in real code. In fact, most of the effort both in developing warlock and in using warlock results from complications rather than the basic algorithm. This section discusses some of these complications and how they were addressed.

Frequently, the mechanism chosen for dealing with a complication was to make available to the user of warlock a way of annotating the program source code to inform warlock of things which it cannot glean from the source itself. These annotations, often called "directives," are in the form of comments with special keywords which the compiler has been modified to recognize. Source-annotation serves the same purpose in **lint**.

The intent was to have the comments provide documentation which would help a developer looking at the code for the first time. In effect, standard ways of expressing such ideas about locks have been defined, and the compiler modified for warlock has been taught to recognize them.

5.1 Uninteresting variables

Some variables are of no interest. Const variables and thread-local variables (identified to the compiler using a pragma) are ignored automatically by the modified compiler; no information about them is stored in the **.wl** file.

Automatic variables are ignored as well. An automatic may only be accessed from other threads via a pointer. Pointers are not ignored, so warlock handles this situation correctly unless the defining thread accesses the variable without using the pointer. A future release will provide support for this case.

The user may want other variables ignored as well, for various reasons. A variable might effectively be protected by a semaphore, while warlock doesn't analyze the use of semaphores. Some variables are used in such a way that no locking is required. For example, it is common for a variable to be written once before any other use, and only read thereafter.

Embedding the comment

```
/* VARIABLES PROTECTED BY "sem1": v1, v2 */
```

in the program source informs warlock that variables **v1** and **v2** are protected by some means other than a mutex or readers/writer lock (in this case semaphore **sem1**), and should therefore be ignored.

5.2 Initialization code

Earlier it was stated that in practice variables are protected consistently by locks. One degenerate exception to this is initialization code. A program starts life as a single thread, and often initializes variables before creating other threads. During this time, data races are impossible since only one thread is running, so the program may safely access data without holding the associated locks. Similarly, programs often finish with one thread waiting for all of the others to exit, and then doing some final work such as printing results; again, there is no need to acquire locks during this period. Special comments express this:

```
main()
{
    /* NO OTHER THREADS ARE RUNNING */
    initialize();
    /* OTHER THREADS ARE RUNNING */
    do_work_with_multiple_threads();
    /* NO OTHER THREADS ARE RUNNING */
    print_results();
}
```

Warlock ignores accesses to variables during periods when no other threads are running.

There is a second aspect to the initialization exception. A structure allocated off the heap is not globally accessible until the thread which allocates it places a pointer to it someplace where it can be seen by another thread. Commonly a routine will, after allocating a structure, initialize it without holding the corresponding locks. Again, special comments apprise warlock of the situation:

```
extern struct foo *global_foo;
struct foo *p;
p = (struct foo *) malloc(sizeof(*p));
/* LOCK UNNEEDED BECAUSE "p not shared":
        p->lock */
p->count = 0;
```

5.3 Pointers to functions

When a function is called through a pointer, warlock needs to descend into each of the functions which might actually be targeted by the call. Where the function pointer is part of a structure, the modified compiler watches for initializations of such structures and automatically records the initialization values as possible targets.

For all other function pointers, the user must enumerate the possible targets using a special comment:

```
void (*f)();
/* FUNCTIONS CALLED THROUGH POINTER f: a, b */
```

Even with this facility, function pointer arguments represent a problem. Consider the C library function **qsort**(), which takes a pointer to a comparison function. Certainly the implementation of **qsort**() could not contain the directive, since the callers of **qsort**() are different for each body of code analyzed. But for the calling code to use the directive, it would have to know the name of the variable **qsort**() uses to receive the function pointer. Furthermore, the caller would have to know whether **qsort**() passes the pointer to other functions, and the names those functions use to receive it.

However, this situation is simplified by writing for each library a set of functions which represent the behavior of interest to warlock in a consistent, simple manner (similar to the use of **lint** libraries). For example, the warlock version of **qsort**() might look like this:

```
qsort(char *a, int b, int c,
        int (*qsort_FP)())
{
    (*qsort_FP)();
}
```

Assume that by convention such routines are written to accept function pointers in variables called *function*_**FP**, and that such routines do not in turn pass those function pointers on to other functions. Then developers can enumerate targets of function pointer arguments as follows:

```
/* FUNCTIONS CALLED THROUGH POINTER
    qsort_FP: foo, bar */
```

This problem may someday be addressed automatically by implementing in warlock a limited form of data analysis to keep track of function pointers passed as arguments.

Incidentally, the majority of functions in the C library have no behavior of interest to warlock, and hence do not require that such counterparts be written. A warlock version is needed for a function only if it takes a function pointer as an argument, has locking side effects, or assumes locks are held on entry.

5.4 Anonymous data

Warlock, like most static analyzers, cannot distinguish between different instances of a data type accessed through pointers and/or array indexing. For structures, warlock employs the usual solution of treating all such instances of the structure as a single instance [Tay83]. In its reporting warlock refers to member *mbr* of an anonymous instance of the structure with tag *tag* as *tag*::*mbr*. This notation is borrowed from the C++ language.

Unfortunately, the C language allows a structure to be defined without assigning it a tag. In this event the compiler generally makes up a tag name for the structure. Warlock could not use the name fabricated by the particular compiler upon which warlock was based, since the name assigned was not consistent across programs sharing the definition of that structure. Therefore warlock assigns its own name for the tag, creating it from the filename and line number in which the structure was defined. For example, the tag name created for a tagless structure defined on line 10 of file **x.c** would be **x.c@10**. Clearly this approach works for structures defined in include files but fails if the definition is simply copied into multiple files.

Anonymous simple types are ignored by warlock. In the following example, the warlock compiler makes a record of the access to **foo::a**, but not for the access to ***ip**:

```
struct foo {
    int a;
};
bar(struct foo* foop, int* ip)
{
    foop->a = 2;
    *ip = 3;
}
```

The write to ***ip** is every bit as subject to data races as the write to **foop->a**, so it would be helpful to have warlock check whether accesses to it are consistently protected. However, while all **foo::a** are protected by a single lock (or at least, warlock assumes so), it is unlikely that all anonymous ints are protected by a single lock. Therefore if warlock were to include such accesses in its analysis, it would almost certainly report them as errors.

5.5 Scope of identifiers

C, like many block-structured languages, allows a variable to become hidden by the introduction of another variable by the same name in a more immediate scope. Within the program, the scope rules determine which variable by that name will actually be accessed, and the others are simply inaccessible. By embedding directives in source in the form of comments and then parsing the directives along with actual program code, scope rules can be applied to names within directives.

Function names used in the aforementioned `FUNCTIONS CALLED THROUGH POINTER` directive are an exception. C allows the use of function names for which no declaration exists — that is, names which are not in scope. These names cannot be checked for validity by the modified warlock compiler. Moreover, the user needs the ability to specify which of several possible functions by that name is intended. Therefore the syntax "*filename*":*function* is allowed for a function name.

5.6 Union members

When a member of a union is accessed, all other members of that union are accessed as well. For this reason union member names are all recorded using a single name: %.

This causes accesses to various members of a union to be treated as accesses to the same variable. The implementation errs on the side of caution when some of the union members are structures, but this does not seem to cause trouble.

5.7 Loops and recursion

If warlock finds itself looping or recursing while tracing the execution of a program, it ends the trace of that path through the program so as not to loop or recurse forever. This can cause warlock to miss possible problems. For example, in the code

```
for (i=0; i<10; i++) {
    mutex_lock(&foo_lock);
    foo = foo + i;
}
```

warlock will only trace one iteration of the loop, and therefore will not flag an error that an attempt is made to acquire **foo_lock** when it is already held.

5.8 Data dependencies

Currently warlock makes no attempt whatsoever to keep track of the values of variables. This can cause warlock to analyze paths through the code which could not really be taken. In the following code fragment, for example, if the lock is acquired, then it is released, and if it is not acquired, then it is not released.

```
Bool we_locked_it = FALSE;
if (...) {
    mutex_lock(&lock);
    we_locked_it = TRUE;
}

...

if (we_locked_it)
    mutex_unlock(&lock);
```

But because warlock does not keep track of the value of variable **we_locked_it**, warlock sees four possible paths through this code rather than two. In one of those paths the program unlocks the lock without having locked it — clearly an error — and in the other the program leaves the lock locked, causing the function to appear to have inconsistent side-effects on locks.

Such data dependencies can be circumvented by presenting to warlock a simpler view of the code. This can be done using macro **WARLOCK**, which is defined during the creation of the **.wl** file but not during the creation of the object file. The example above might be altered to look like this:

```
Bool we_locked_it = FALSE;
#ifdef WARLOCK
mutex_lock(&lock);
#else
if (...) {
    mutex_lock(&lock);
    we_locked_it = TRUE;
}
#endif

...

#ifdef WARLOCK
mutex_unlock(&lock);
#else
if (we_locked_it)
    mutex_unlock(&lock);
#endif
```

Dealing with data dependencies is one of the more problematic aspects of using warlock. Not only does it require that the user modify source code, but the resulting code is also harder to read. In the future, limited forms of data analysis may be employed in order to automatically handle some of the common problems.

5.9 Partial analyses

Warlock traverses the call graph for the code being analyzed to identify the "root" functions — those which are not called from any other function being analyzed. **Main**() and signal handlers would typically be root functions for a user program. The trace of the program's execution proceeds from each of the root functions, down into the functions it calls.

Because one frequently wants to analyze a library or a subsystem of a very large program (like the kernel), a non-root function may also be called from outside the set of functions currently being analyzed. Such a function should be considered a root function in the context of the set of functions being analyzed. This situation arises mostly with analysis of libraries, where a function which is part of the library's public interface is also called by a function within the library.

One could envision a directive which informs warlock that a function should be treated as though it were a root, but that would not solve the problem. The need to identify a particular function as a root function comes not from the nature of the function itself, but rather from the functions one has chosen not to include in the analysis. An annotation would cause the function to be treated as a root in any analysis, whereas the function really needs to be treated as root or not depending upon which other functions are analyzed along with it.

One way of dealing with the problem is to write a dummy function which calls each of the public functions in the library, and analyze it along with the library:

```
main()
{
    public_func1();
    public_func2();
    ...
}
```

Another problem with partial analyses arises when a function assumes that a lock is acquired by the caller. If no caller is provided in the analysis, it will appear to warlock that the function accesses variables without holding the appropriate locks. A special comment exists to communicate the function's assumption to warlock:

```
foo(struct bar* b)
{
    /* LOCK HELD ON ENTRY: b->lock */
```

If the function is a root, warlock automatically acquires the specified lock before it begins the analysis of this function as a root. If the function is ever called from another function, warlock checks that the specified lock is held at the time the call is made.

5.10 Hierarchical locking conventions

A readers/writer lock may be used to control access to other locks, allowing one to set up lock hierarchies. In warlock parlance, the controlling lock is said to "cover" the other locks. While holding the cover for write access, it is unnecessary to acquire any of the covered locks. It is an error to hold a covered lock while not also holding its cover for at least read access. This technique is sometimes used, for efficiency reasons, to control access to a set of related data structures. For example, in the following code, a readers/writer lock controls access to a linked list of **foo_t** structures, each of which may be individually locked as well.

```
delete_foo(foo_t *foo)
{
    /* lock the entire list */
    rw_wrlock(&list_lock);
    <prepare foo for deletion>
    /* unlink; no need to lock prev/next */
    <unlink foo>
    rw_unlock(&list_lock);
}

update_foo(foo_t *foo)
{
    /* have to get read access to list */
    rw_rdlock(&list_lock);
    /* now lock that foo and update it */
    mutex_lock(&foo->lock);
    <update foo>
    mutex_unlock(&foo->lock);
    rw_unlock(&list_lock);
}
```

Deleting a **foo_t** requires that other threads be prevented from using the list; acquiring **list_lock** for write access accomplishes this, obviating locking individual **foo_t** structures in the list. Updating a **foo_t**, on the other hand, does not preclude other threads from using the list — but it does require that other threads be prevented from using the **foo_t** being updated. Hence it is necessary to acquire the mutex lock protecting that entry. But before the **foo_t** itself can be locked, we must hold **list_lock** for read access.

Such hierarchical use of readers/writer locks is simply a convention — there is nothing in the code to indicate to warlock that this relationship exists, so warlock reports that the variables protected by such locks are not consistently protected. A special comment informs warlock of the relationship:

```
/* LOCKS COVERED BY list_lock: foo_t::lock */
```

Given this information, warlock knows to expect different locking behavior with these locks. Moreover, warlock can verify that no covered lock is ever held while the cover is not held.

5.11 Functions with locking side effects

Part of warlock's analysis involves calculating the net side effects each function has on locks. It is easy to accidentally code a function to have a locking side effect:

```
foo()
{
    mutex_lock(&lock1);
    if (phase_of_moon == BLUE)
        return;
    do_something();
    mutex_unlock(&lock1);
}
```

In the above example, the programmer has probably forgotten to unlock **lock1** in the conditional exit. Functions which intentionally leave locks in different states from their entry states do occur, but they are rare, so it is important for warlock to be able to check whether such side effects are valid. Special comments inform warlock of the programmer's intent:

```
/* LOCK ACQUIRED AS SIDE EFFECT: p->lock */
/* LOCK RELEASED AS SIDE EFFECT: lock1 */
```

Any discrepancy between the side effects declared in this manner for a function and the side effects warlock computes for the function is reported as an error.

5.12 Conditional locking calls

Some of the locking calls recognized by warlock are not guaranteed to succeed; their return value indicates whether or not the operation was done. If that return value is used to control branching, warlock has to rewrite the code in terms of unconditional operations. For example,

```
if (mutex_trylock(&lock1) == 0) {
    /* success */
} else {
    /* failure */
}
```

gets rewritten as

```
if () {
    mutex_lock(&lock1);
    /* success */
} else
    /* failure */
}
```

Because warlock does no data analysis, there are limits to warlock's ability to decipher the use of such return codes. For example, warlock warns that it can't figure out the following:

```
int return_val = mutex_trylock(&lock1);
if (return_val == 0) {
    /* success */
} else {
    /* failure */
}
```

5.13 Unprotected sampling

Consider a garbage-collector which examines reference counts and frees objects to which no references remain. Although updating the reference count requires that a lock be held, the garbage-collector need not hold the lock in order to sample the reference count.

Warlock does not provide direct support for handling such sampling. In order to stop warlock from complaining about the unprotected read, one must employ conditional compilation to change the code warlock sees.

It would not be sufficient to simply provide a way of telling warlock that a particular variable may be safely read without holding a lock. When such a variable is updated, a lock must be acquired before reading the variable and held until after the variable is written. Warlock would need to be able to tell when a read and write of a variable are part of an update — that is, when the value written is a function of the value read. This is currently beyond warlock's capabilities.

6.0 Status and experience

The ability to influence warlock's analysis through source annotation was only recently added to warlock. Most use of warlock involved the interactive back end, which is somewhat less friendly. Considerable development has been done since warlock's first uses, spurred by the difficulties found.

Even so, it is appropriate to think of the existing tool as a prototype for something larger. Too many code changes are required to get warlock to give a program a clean report; various forms of data analysis will be required to reduce the use of comment directives and **#ifdef warlock**.

Warlock currently only analyzes C source, while the use of C++ is growing. Deadlock detection needs to be added to warlock so that separate tools need not be run on all code.

Reaction to warlock has been mixed. Some users have praised it highly, reporting that it led them to problems in their code which would otherwise have been very difficult to find. Others have had trouble understanding what they need to do to their code to eliminate the spurious messages. Also, as one might expect, busy developers have difficulty finding the time to invest in warlock. About a dozen developers have chosen to try warlock during its development, and interest seems to be growing gradually as the tool matures.

Certainly warlock has the potential of catching errors which would otherwise be found in the field, and only after considerable investigation. After an X-windows library was made safe for multithreading use, warlock analysis resulted in eight further changes to the code. On the other hand, part of the kernel's virtual memory subsystem was run through warlock and, after all the complaints had been investigated, no true errors were discovered. While this latter exercise did not improve the code, it raised the developer's confidence in the code's correctness. No doubt the fact that the virtual memory system had been heavily exercised for months made it less likely to contain errors.

Prior to the annotation capability, it typically took a developer four to eight days to read about warlock and then use it successfully with a significant body of code, such as a complex driver or a library. The source annotation capability might improve this situation slightly by providing a more natural interface. Also, once source annotation is in place, standard header files can be annotated once and for all, reducing the workload for individual developers. Similarly, warlock versions of libraries can be written once for all to use.

As one might expect, many of the problems found by warlock are simply cases where a block of code which primarily manipulates one data structure has embedded in it one easily missed reference to another data structure, for which the appropriate lock is not held.

Other problems result from overly aggressive attempts to improve the performance of code. In order to obtain the best performance on a multiprocessor, it makes sense to hold locks for the smallest interval possible. In some cases it is possible to avoid acquiring a lock at all for certain code paths. Occasionally, though, it takes a bit of thought to understand what can safely be done in parallel, and one can easily go a bit too far. In an X-windows library a routine which unrealizes a widget was modified to look something like this simplified version:

```
unrealize_widget(Widget *widget)
{
    if (!widget->realized)
        return;
    mutex_lock(&lock1); /* added */
    <unrealize widget and free structure>
    mutex_unlock(&lock1); /* added */
}
```

Clearly the routine was designed to correctly handle multiple calls to unrealize a widget. However, in a concurrent environment two threads could make the call at about the same time, and both threads could pass the test at the entrance to the function, resulting in one of the threads trying to use and free a structure which had already been freed. The fix, of course, was simply to acquire the lock before checking to see whether the widget is currently realized.

The following example depicts another recurrent theme:

```
get_bar()
{
    mutex_lock(&lock1);
    foo = ...
    ...
    mutex_unlock(&lock1);
    return foo->bar;
}
```

Depending upon the circumstances, it may or may not be safe to access foo->bar without holding the lock. In some cases it seems surprisingly difficult to decide.

Recall that warlock detects locking side effects, as this function exhibits on its conditional return:

```
foo()
{
    mutex_lock(&lock1);
    if (phase_of_moon == BLUE)
        return;
    do_something();
    mutex_unlock(&lock1);
}
```

It is somewhat surprising that no errors of this type have been found to date. Perhaps this is simply due to the fact that there is run-time checking to flag a second call to **mutex_lock**() by a given thread as an error, so such errors would tend to be caught quickly. By contrast, code which suffers from data races can execute correctly for years before a problem manifests, and when it does manifest there may be little to indicate the source of the problem.

7.0 Future work

Near-term efforts for the project involve utilizing the annotation capability in system header files and writing warlock versions of common libraries. It is time to provide an environment, using the functionality currently available, in which the work in common is already done. An immediate goal is to provide an environment in which drivers can be analyzed with minimal effort, since there are many drivers which could be analyzed if a set of header files and libraries were prepared.

The ability to detect potential deadlocks will probably be added to warlock at some point, since warlock already does much of the work required to prepare for such analysis. Also, there may be ways to check for the proper use of semaphores and condition variables, at least in certain situations.

Warlock shows good potential for finding many — but not all — data races in multithreaded code written for Solaris. There is substantial opportunity to improve warlock's ability to understand code through the implementation of limited forms of data analysis. This will further improve the results of warlock's analysis, and reduce the need for conditional compilation.

8.0 References

[POSIX] POSIX P1003.4a Draft 5, IEEE.

[MH89] McDowell, C. E., and Helmbold, D. P. 1989. Debugging Concurrent Programs. ACM Computing Surveys 21, 4 (December), 593-622.

[Tay83] Taylor, R. N. 1983. A General-Purpose Algorithm for Analyzing Concurrent Programs. CACM 26, 5, 362-376.

[TO80] Taylor, R. N., and Osterweil, L. J. 1980. Anomaly Detection in Concurrent Software by Static Data Flow Analysis. IEEE Trans. Softw. Eng. SE-6, 3 (May), 265-278.

[YT88] Young, M., and Taylor, R. N. 1988. Combining Static Concurrency Analysis with Symbolic Execution. IEEE Trans. Softw. Eng. 14, 10 (October), 1499-1510.

[Pow91] Powell, M. L., Kleiman, S. R., Barton, S., Shah, D., Stein, D., and Weeks, M. 1991. SunOS Multi-thread Architecture. USENIX Conference Proceedings, January, 65-79.

[Kle92] Kleiman, S., Smaalders, B., Stein, D., and Shah, D. 1992. Writing Multithreaded Code in Solaris. IEEE CompCon Proceedings, February, 187-192.