Runtime Checking of Multithreaded Applications with Visual Threads

Jerry J. Harrow, Jr.

Compaq Computer Corporation, Business Critical Servers Group,
110 Spit Brook Rd, Nashua, NH, USA
`Jerry.Harrow@Compaq.com`

Abstract. Multithreaded applications are notoriously difficult to design and build while avoiding defects. Many of Compaq's customers need to employ threads to implement high-performance, scalable applications that address their needs in business and science. In order to ensure their success using threads, Compaq provides a runtime debugging and analysis tool for multithreaded applications called Visual Threads. This paper describes the automatic runtime checking for multithreaded applications incorporated in Visual Threads.

1 Introduction

While the performance of computer systems continues to improve at a rapid rate, many problems in science and business continue to outstrip the ability of a single processor. This has given rise to computer systems built using Symmetric Multiprocessing (SMP), and programming interfaces to enable an application to enlist those processors. One such programming interface standard is `IEEE POSIX 1003.1-1996`, which is commonly referred to as *pthreads* [1]. The POSIX threads library enables an application to create and control additional threads of execution, and provides synchronization primitives to allow the threads to coordinate their work. Java ™ provides similar capabilities through its language syntax. All the threads in an application run inside the same process and therefore share the same address space, terminal, and open files. Depending upon the operating system[1], the threads may be scheduled across the available processors, thereby enabling true concurrent execution of the threads.

The challenges of threads are many. Primary of these challenges is the increased program complexity caused by the need to synchronize access to shared data structures, the potential for timing-dependent failures, errors using the programming interface, and the difficulty of debugging and optimizing the application.

[1] Compaq's *OpenVMS* and *Tru64*™ UNIX® operating systems both provide a POSIX threads library implementation that enables true concurrent execution of threads on systems with multiple processors.

2 Development of Visual Threads

The genesis of Visual Threads was the observation that many of the problems reported by our customers in their multithreaded applications were in fact programming errors on their part. The vision of the development manager for the threads group was that if we could help customers find these problems, it would greatly improve customer satisfaction, and reduce support costs. The goal of Visual Threads therefore became: *To help Compaq's customers succeed with threads*.

With almost no other competitive tools available in the industry, the development team brainstormed to build a list of all common thread-related programming errors that an application may experience. This list was then prioritized to reflect the relative usefulness of detecting each error to the programmer. After weighing the benefits of the various types of tools that could be built, the team decided to focus on automatic runtime-based checking. The primary factors influencing this choice are listed below.

– Can be applied to code already written
– Provides the most capability without user input
– Does not require buy-in to a particular design model
– Programming-language independence
– Addresses the widest range of errors
– Works when not all code is available as source code

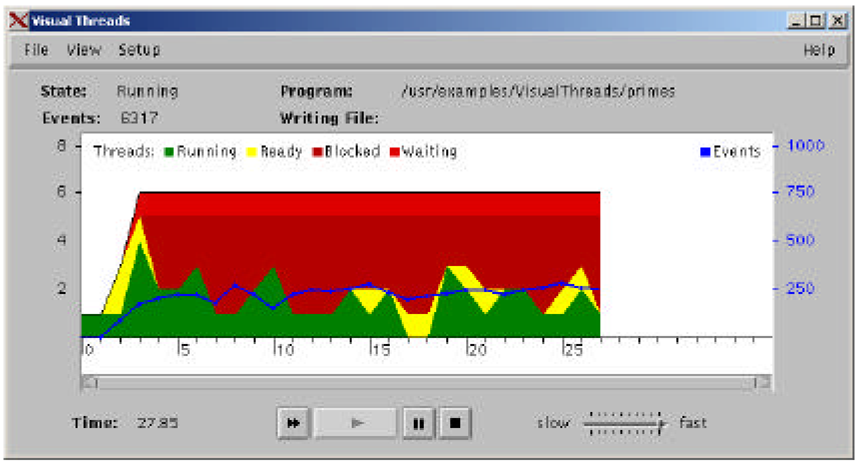
With the tool paradigm selected, the team started the detailed design of what specific errors this runtime-based checking tool would address. The set of analysis rules were chosen very pragmatically. The team commenced to design and implement the largest set of runtime checking that we could reasonably expect to complete within the time and resource constraints of ~~5~~6 engineers and ~~6~~12 months design and implementation. Additional analysis rules have been added in on-going releases.

3 Visual Threads Architecture

Visual Threads integrates many distinct technologies into a single development tool to hide the underlying complexity, and make it easy to use. The primary components that make up Visual Threads are listed below.

– Graphical User Interface implemented in Java
– An optional connection from the user interface to the analysis engine running on a remote server via the Java Remote Method Invocation facility
– Analysis engine implemented in C++ and accessed via Java native method calls
– Shared-memory transport between analysis engine and application to be analyzed
– A binary instrumentation tool to add analysis code into an existing application
– A POSIX threads implementation that provides data gathering hooks

These technologies and components are used together to enable a programmer to apply the computation power of today's sophisticated computer systems toward solving their problem of producing robust, scalable, efficient code.


Visual Threads

File View Setup Help

State: Running

Events: 6317

Program: /usr/examples/VisualThreads/primes

Writing File:

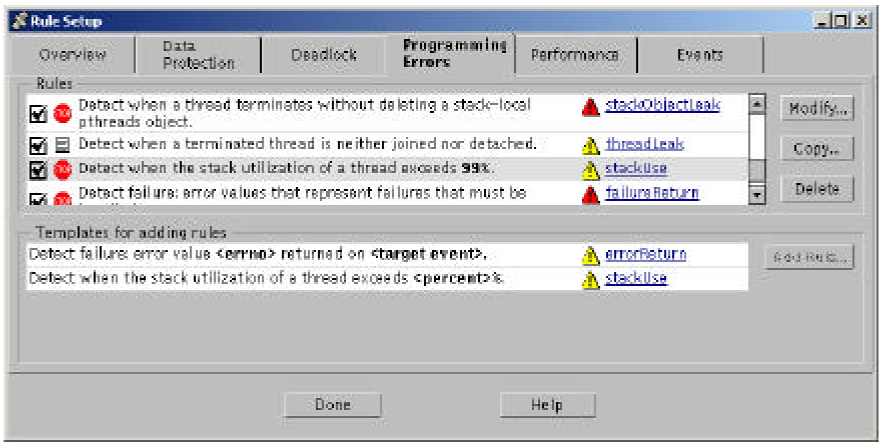
Time: 27.85

Fig. 1. The main window of the Visual Threads user interface provides execution controls, and a summary of thread activity over time. This graph is dynamically drawn in conjunction with the analysis of the application as it executes.

The primary unit of data processed by Visual Threads is the *event*. This data is provided directly from the library that implements the threads programming interface. A significant state change in the threads library results in an event being generated. This event is encoded as a small binary record and transmitted from the application process to Visual Threads via a shared memory ring buffer. Visual Threads uses these events to model the execution of the application via a state machine. The runtime checking is then applied to this state machine and any errors detected are reported via the user interface. In some cases, additional events are generated by other analysis code that Visual Threads injects into the application executable. In particular, the support for detecting race conditions on data shared between multiple threads without synchronization is performed by such injected code. If violations of this rule are detected, events are generated describing the violation.

4 Automatic Runtime Checking

The heart of Visual Threads' runtime checking is a set of rules that are applied to the threads-related activity in the application. The automatic error-detection rules can be classified into the categories of deadlock, data protection, and other programming errors. In addition to the error-detection rules, there are rule templates that allow a programmer to create customized rules to analyze the behavior of their particular application.



Rules

✓	Detect when a thread terminates without deleting a stack-local pthreads object.	stackObjectLeak
✓	Detect when a terminated thread is neither joined nor detached.	threadLeak
✓	Detect when the stack utilization of a thread exceeds **99%**.	stackUse
✓	Detect failure: error values that represent failures that must be	failureReturn

Templates for adding rules

Detect failure: error value **<errno>** returned on **<target event>**.	errorReturn
Detect when the stack utilization of a thread exceeds **<percent>**%.	stackUse

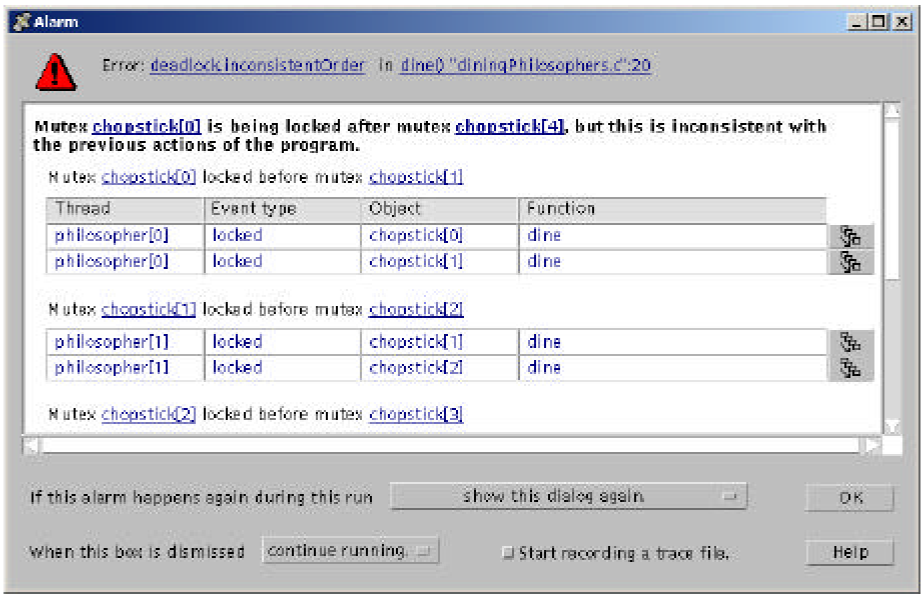
Fig. 2. Rules are enabled, disabled, and created via the Rule Setup dialog. The top section lists the currently defined rules in the *Programming Errors* category, if they are enabled, and what action to take when the rule is violated. The bottom section provides templates that can be used to create additional rules.

When a given rule is enabled, Visual Threads evaluates the rule condition relative to the thread events generated by the application. When a rule is violated, Visual Threads responds with a rule action (such as stopping the application or ignoring the violation) and provides the necessary data to help diagnose the error in the application. Consider the following naïve implementation of Dykstra's classic dining philosophers problem [2].

```
#include <pthread.h>

pthread_t philosopher[5];
pthread_mutex_t chopstick[5];

void *dine(void *arg) {
    long left = (long) arg;
    long right = (left + 1) % 5;
    while (1) {
        pthread_mutex_lock(&chopstick[left]);
        pthread_mutex_lock(&chopstick[right]);
        // Eating...
        pthread_mutex_unlock(&chopstick[left]);
        pthread_mutex_unlock(&chopstick[right]);
        sleep(1);
    }
}
```

```
void main() {
    // Create chopsticks
    for (int c = 0; c < 5; c++)
        pthread_mutex_init(&chopstick[c], NULL);

    // Create philosophers
    for (int p = 0; p < 5; p++)
        pthread_create(&philosopher[p], NULL,
                       &dine, (void *)p);
    sleep(100);
}
```

When executed under Visual Threads, the following alarm is generated.



Error: deadlock.inconsistentOrder in dine() "diningPhilosophers.c":20

Mutex chopstick[0] is being locked after mutex chopstick[4], but this is inconsistent with the previous actions of the program.

Mutex chopstick[0] locked before mutex chopstick[1]

Thread	Event type	Object	Function
philosopher[0]	locked	chopstick[0]	dine
philosopher[0]	locked	chopstick[1]	dine

Mutex chopstick[1] locked before mutex chopstick[2]

philosopher[1]	locked	chopstick[1]	dine
philosopher[1]	locked	chopstick[2]	dine

Mutex chopstick[2] locked before mutex chopstick[3]

If this alarm happens again during this run: show this dialog again

When this box is dismissed: continue running

☐ Start recording a trace file.

Fig. 3. This alarm reports the detection of an inconsistent lock hierarchy during the execution of the program. The white center area of the dialog is dedicated to describing the actions of the program that violate the *inconsistentOrder* rule. Each line in the tables displayed depicts an action in the program, and provides access to more information about each philosopher (thread), chopstick (mutex), the callstack, and the location in the source where the action occurred.

The entire Visual Threads system is targeted for efficient analysis of running programs. Under the default configuration all rules except those dealing with data protection are enabled. Each class of rules are described in the following sections.

4.1 Deadlock-Related Rules

The problem of deadlock is common in parallel programming. Deadlock can occur whenever multiple shared resources are required to accomplish a task. If not done correctly, two threads may end up each holding one resource but waiting for a second held by the other thread. If neither releases a resource until they complete their respective tasks, both will wait indefinitely. Visual Threads detects just this situation in the use of the POSIX threads programming interface. The set of resources analyzed by Visual Threads are mutexes, read-write locks (write portion), and threads (join operations). When a deadlock situation is detected, the error message provides all the relevant information necessary to diagnose the underlying cause of the error. For each thread involved in the deadlock, the location where the resource was acquired, and the location where the thread is waiting for the other resource is displayed.

Explicit Deadlock. Deadlock is a circularity in the dependency graph for the threads. It is detected via a simple recursive mark-search algorithm directly applied to the model representation of the program that Visual Threads constructs as the program executes. When a thread blocks on a synchronization point, a new marker value is allocated. A recursive routine is then invoked specifying the thread object and the marker value. The algorithm for this routine is described in pseudo code below.

```
CheckDeadlock (object, mark) {
   // If we find current mark, then cycle detected
   if object.mark == mark then report deadlock;

   object.mark = mark;

   for each o on which object depends
        CheckDeadlock (o, mark);
}
```

There are two noteworthy optimizations used to keep this processing efficient. The first of these is to invoke the algorithm only when a thread blocks. This avoids overhead in the case when a lock is not contended (a common case). Fortunately, the data collection hooks utilized by Visual Threads provide this level of detail, which is normally only available within the threads library implementation itself. A second optimization was to eliminate the need to clear previous marks. This was accomplished by always using a unique mark value on every search. Mark values are simply a 32-bit integer quantity incremented for every search. While it is theoretically possible for an application to be executed long enough to perform 2^{32} searches for deadlock, thereby causing mark values to be re-used, the probability of this is negligible.

Potential Deadlock. While detecting actual deadlock is useful, it is relatively obvious when it occurs because the application never completes. Visual Threads goes beyond this, however, to detect various conditions that may lead to deadlock. These are much more important to ensuring program correctness because they detect situations that may not typically have any visible symptoms, but at some point in the future may cause the application to fail. One such rule detects when locks (mutexes and/or read-write locks) are acquired in an inconsistent order sometime during the application run. Visual Threads does this by monitoring the lock acquisition order, and verifying that all future acquisitions are performed in the same order. If the locks are acquired inconsistently, there is the potential for the application to deadlock.

The algorithm for detecting inconsistent lock order maintains a set of must-not-be-locked-before relationship pairs. When a new lock is acquired a search is performed to find any existing lock order pairs involving the new lock and each of the other locks already held by the thread. Note that the must-not-be-locked-before relationship is transitive and therefore the search function recursively follows chains of relationships. If a match is found, an error is reported. The error message (see Figure 3 as an example) includes each of the locations in the source code that contributed to the observed lock order. If the search fails to find any inconsistencies with previous execution behavior, then new must-not-be-locked-before relationship pairs are created using the new lock and each lock currently held.

If the application was designed based upon a lock acquisition hierarchy (i.e. a consistent order in which to acquire locks is part of the application definition), then application-specific rules can be configured that enable Visual Threads to validate that the lock acquisition order specified by the design is in fact honored.

Priority Inversion. Another deadlock-related rule is the detection of priority inversion. Visual Threads detects high-priority threads that are waiting for a lock held by a low-priority thread when another medium-priority thread is currently executing. This rule, along with a warning about sharing locks between threads of differing priorities, helps to pinpoint other programming errors that can lead to poor performance or more drastic failures.

4.2 Data Protection Rules

The most powerful Visual Threads analysis rule finds data shared between multiple threads without the protection afforded by a mutex, read-write lock, or atomic hardware instruction sequence. In the rest of this section the generic term *lock* will be used to refer to all of those synchronization primitives. Unsynchronized access to data that is shared between multiple threads and is also modified can result in timing-related errors such as incorrect results, or memory corruption. The algorithm starts with the premise that the sharing of a read-write data item ***d*** should be governed by some lock ***l***, or by the implicit synchronization of thread creation or join (a join allows a thread to wait for the termination of another thread). Violations of this premise are reported as potential errors in the application being analyzed. There are three significant facets of the algorithm used to detect and report these potential errors.

– Memory-usage markings to handle initialization, and shared read data.
– Lockset refinement to detect that a lock does not protect the data.
– Thread segment identification to handle create/terminate synchronization

Basic Algorithm. Visual Threads data protection rules are based upon the algorithms and implementation of the Eraser tool. The complete details of the underlying Eraser algorithm are available in a separate paper [3], but it can be summarized as follows. At the start of the algorithm all data is marked as `NEW` (see Table 1), and is protected by a candidate *lockset* containing all the locks in the application.

Table 1. Memory-usage Markings

State	Description
`NEW`	Newly allocated memory, not yet accessed, no lock sets.
`EXCLUSIVE`	Memory is identified as being exclusively accessed by a particular thread, no lock sets.
`SHARED`	Identifies shared, read-only data. The set of locks in effect during all accesses is updated. No errors are reported.
`SHARED-MODIFIED`	Identifies shared, writable data. The set of locks in effect during all accesses is updated. If empty, an error is generated.

To account for initialization and read-only sharing of data, error reports are deferred until a data address reaches the `SHARED-MODIFIED` state, as depicted in Figure 4. Once a thread modifies the data, it is associated with that thread and marked `EXCLUSIVE`. If read or written by any other thread, the memory is then marked `SHARED` or `SHARED-MODIFIED` respectively.

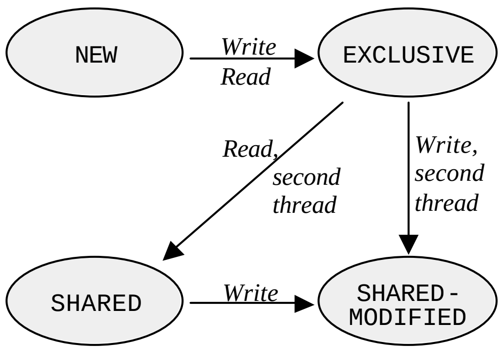

$$\begin{tikzcd}
\text{NEW} \arrow[r, "\textit{Write}", "\textit{Read}"'] & \text{EXCLUSIVE} \arrow[dl, "\textit{Read, second thread}"'] \arrow[d, "\textit{Write, second thread}"] \\
\text{SHARED} \arrow[r, "\textit{Write}"] & \text{SHARED-MODIFIED}
\end{tikzcd}$$

Fig. 4. Checking for consistent synchronization is deferred until the memory is actually shared between two or more threads and is being modified. Earlier state transitions accommodate memory allocated and initialized by a single thread, then shared with other threads either in a read-only fashion, or with synchronization.

When the data becomes shared, the candidate lockset presumed to protect the data is updated. The lockset is reduced to the intersection of the previous lockset and the set of locks held by the thread on the current access. This progressively removes locks from the lockset that are only incidentally locked during some access to the data. If the lockset ever becomes empty an error is reported because no single lock protects all access to the data. A pseudo-code representation of the Eraser algorithm is depicted below.

```
Let locks_held(t) be the set of locks held by thread t.
Let update_use(d,a,t) update the marking of data d
    by thread t using access a as shown in Figure 4.
For each data item d:
    mark[d] := NEW
    lockset[d] := { all locks }

On each access a (read or write) to d, by a thread t:
    update_use(d,a,t)
    if mark[d] in { SHARED, SHARED-MODIFIED }
        lockset[d] := lockset[d] ∩ locks_held(t)
    if mark[d] = SHARED-MODIFIED and,
       lockset[d] = { }, then report error
```

Thread Segment Extension to Eraser Algorithm. Not all data shared between threads without the use of a lock is an application error. There are types of indirect synchronization that may eliminate the need for explicit locks. In particular, a very common paradigm is for the initial thread to allocate and initialize some data, create worker threads to perform some transformations on this data, and then after the threads have all completed, display the result. The original Eraser algorithm was extended to reduce the number of false reports due to this type of implicit synchronization, by introducing the concept of thread segment. A thread segment delineates time in addition to just thread identity. By utilizing the thread segment identifiers in the algorithm instead of simply using the thread identity, we can distinguish accesses that cannot happen concurrently. No thread segment spans beyond the creation of a new thread, or a join. When a parent thread creates a new child thread, the parent's thread segment id is updated, and the child's thread segment id is assigned. Similarly, after a join operation, the parent's thread segment id is updated (the child no longer exists). Each running thread is represented as a leaf in the graph. Figure 5 shows how thread segments are assigned over time.

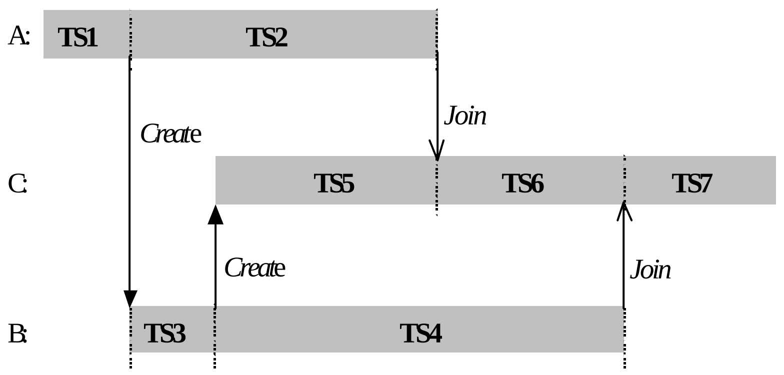


Fig. 5. A single thread may have many thread segments over time. Thread A creates Thread B, and Thread B creates Thread C. Thread C synchronizes with Thread A by waiting for it to terminate. Similarly, Thread C then waits for Thread B to terminate. This results in the seven thread segments **T1** through **T7**.

These thread segments are kept in a directed graph that indicates which thread segments cannot occur concurrent with each other. As you can see from Figure 5, **S1** happens before **TS2**-**TS7**, **TS3** happens before **TS5**-**TS7**, **TS2** happens before **TS6** and **TS7**, and so forth. The modification of the Eraser algorithm to incorporate thread segments is relatively straightforward.

1. When data ***d*** is marked as `EXCLUSIVE`, associate it with the thread segment id of the current thread instead of the thread id.
2. If data ***d*** is marked as `EXCLUSIVE` to thread segment **TS***i*, and is being touched by ***TSj*** (where $i \neq j$), and **TS***i* happens before **TS***j* in the graph, then instead of moving the data to one of the shared states, associate ***d*** with **TS***j*. The marking remains `EXCLUSIVE`.

This extension allows exclusive-use data to be "passed" from parent thread to child thread (and back) as long as the thread segments involved in the access cannot be concurrent due to the known points of thread creation and termination (as determined by the thread segment graph). If the two thread segments are potentially concurrent, then the data is marked as either `SHARED` or `SHARED-MODIFIED` as appropriate and the standard checking is applied.

Implementation Details. The basic technique used to detect this type of error involves monitoring every load and store of all global and heap data in the application. The code for monitoring memory access is injected into an existing application executable using a binary code modification tool called Atom [4]. Information is maintained for each data address in a corresponding *shadow* address. Since access to the state information is very frequent, the shadow data is efficiently determined by a table lookup and offset calculation. The table lookup is kept small by using large shadow segments (16MB) to minimize search time on the table. This table lookup was added for improved robustness necessary in a product, even at the cost of some additional execution overhead. The data maintained in the shadow area is listed below.

- Type of sharing (as listed in Table 1)
- Set of locks protecting this data, if marked `SHARED` or `SHARED-MODIFIED`.
- Owner thread segment, if marked `EXCLUSIVE`

Protecting Unsafe Libraries and Functions. In addition to detecting unsynchronized access to shared data, Visual Threads also allows the definition of application-specific rules that identify particular libraries or functions that are known not to be thread safe. There are many extensively used libraries (such as user-interface toolkits) that are not thread safe. Typically, the programmer must surround any call to such a library with a lock to prevent other threads from calling into the library concurrently. Programmers can then identify this requirement, and have Visual Threads verify that the application has been coded properly.

4.3 Other Programming Error Rules

The remaining error-detection rules validate various facets of using the POSIX threads programming interface correctly. For the most part, the rules listed below are simply checks that are performed during state changes in the model maintained by Visual Threads.

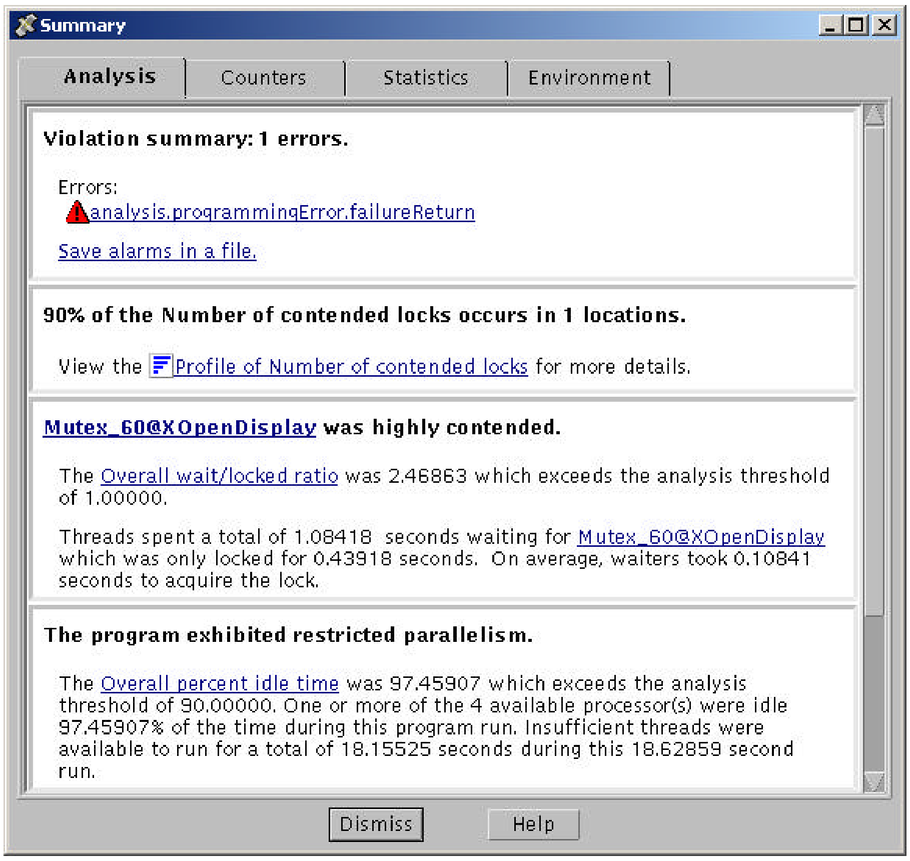
- Detect attempts to relock a non-recursive mutex.
- Detect attempts to unlock a lock the thread did not previously lock.
- Detect when a condition variable is associated with more than one mutex.
- Detect when mixed scheduling policy is used.
- Detect attempts to wait on a condition variable when the mutex is not locked.
- Detect when a thread terminates while holding a mutex or read-write lock.
- Detect when a thread terminates without deleting a stack-local threads object.
- Detect when a terminated thread is neither joined nor detached.
- Detect when stack utilization of a thread exceeds 99%.
- Detect when the threads programming interface returns an error value that represents a failure that must be handled.

While many of these errors may seem minor, often they have indeterminate effects upon the application state. This can lead to unexpected behavior at some later time, which can be extremely difficult to debug due to the distance from the original problem.

4.4 Detecting Errors by Observation and Heuristics

While this paper has focused primarily on the program validation aspects, Visual Threads also provides visualization of the state of thread and synchronization objects in the application, as well as statistical analysis of overall execution behavior. Unlike the relatively absolute errors detected by rule checking, some errors are recognized only by the degree of their severity. For this class of errors, Visual Threads provides a heuristic-based summary of the program execution as shown in Figure 6. The types of analysis reported in this manner are listed below.

– Locks with high levels of contention
– Locks with granularity that is too coarse
– Level of processor utilization
– Mutex with the highest percentage of contended locks
– Mutex with the highest total wait time
– Mutex with the highest number of concurrent waiters
– Inefficient use of mutex attributes



Violation summary: 1 errors.

Errors:
analysis.programmingError.failureReturn

Save alarms in a file.

90% of the Number of contended locks occurs in 1 locations.

View the Profile of Number of contended locks for more details.

Mutex_60@XOpenDisplay was highly contended.

The Overall wait/locked ratio was 2.46863 which exceeds the analysis threshold of 1.00000.

Threads spent a total of 1.08418 seconds waiting for Mutex_60@XOpenDisplay which was only locked for 0.43918 seconds. On average, waiters took 0.10841 seconds to acquire the lock.

The program exhibited restricted parallelism.

The Overall percent idle time was 97.45907 which exceeds the analysis threshold of 90.00000. One or more of the 4 available processor(s) were idle 97.45907% of the time during this program run. Insufficient threads were available to run for a total of 18.15525 seconds during this 18.62859 second run.

Fig. 6. The Analysis Summary is automatically displayed when the program completes. In addition to summarizing errors, it generates observations about the program execution.

Often such analysis is helpful in performance tuning, but it can also highlight coding or design errors that result in high lock contention, poor lock granularity, or poor processor utilization.

5 Summary

Visual Threads provides extensive automatic runtime validation of a running multithreaded application. Through pragmatic selection of targeted analysis and efficient algorithms, even the most complex application domain can benefit from automated program checking. While far from exhaustively validating the overall correctness of a multithreaded application, it can be invaluable in detecting programming errors that may lead to program instability and unexpected failures.

Visual Threads is available on Compaq's *OpenVMS* and *Tru64*™ UNIX® operating systems which run on the 64-bit Alpha microprocessor. It is able to analyze multithreaded programs whether written in C, C++, Fortran, or the Java programming language.

References

1. 9945-1:1996 (ISO/IEC) [IEEE/ANSI Std 1003.1 196 Edition] Information Technology – Portable Operating System Interface (POSIX)—Part 1: System Application: Program Interface (API) [C Language] (ANSI), IEEE Standards Press, ISBN 1-55937-573-6, 1996.
2. Dijkstra, E.W. Co-operating Sequential Processes. *Programming Languages*, Genuys, F. (ed.), Academic Press, 1965.
3. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, E.: Eraser: A dynamic data race detector for multi-threaded programs. ACM Transactions on Computer Systems (TOCS), 15(4): 391-411, November 1997. Also appeared in Proceedings of the Sixteenth ACM Symposium on Operating System Principles, October 5-8, 1997, St. Malo, France, Operating System Review 31(5), ACM Press, 1997, ISBN 0-89791-916-5, pp 27-37.
4. Compaq Computer Corporation: Compaq Tru64 UNIX Programmers Guide V5.0, July 1999.