Dynamic Deadlock Avoidance in Systems Code Using Statically Inferred Effects

Prodromos Gerakios [1] Nikolaos Papaspyrou [1] Konstantinos Sagonas [1,2] Panagiotis Vekris [1]

[1] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
[2] Department of Information Technology, Uppsala University, Sweden
{pgerakios, nickie, kostis, pvekris}@softlab.ntua.gr

Abstract

Deadlocks can have devastating effects in systems code. We have developed a type and effect system that provably avoids them and in this paper we present a tool that uses a sound static analysis to instrument multithreaded C programs and then links these programs with a run-time system that avoids possible deadlocks. In contrast to most other purely static tools for deadlock freedom, our tool does not insist that programs adhere to a strict lock acquisition order or use lock primitives in a block-structured way, thus it is appropriate for systems code and OS applications. We also report some very promising benchmark results which show that all possible deadlocks can automatically be avoided with only a small run-time overhead. More importantly, this is done without having to modify the original source program by altering the order of resource acquisition operations or by adding annotations.

1. Introduction

In shared memory concurrent programming, deadlocks typically occur as a consequence of cyclic lock acquisition between threads. Two or more threads are deadlocked when each of them is waiting for a resource, typically a lock, that has been acquired and is held by another thread. As deadlocks are a serious problem, several methods to achieve deadlock freedom have so far been proposed. In particular, approaches stemming from programming language research aim for static deadlock freedom guarantees by employing type systems that *prevent* deadlocks (e.g., [6, 15, 17]). Most such works often impose a strict (non-cyclic) lock acquisition order that must be respected throughout the entire program and/or require that locking is used in a block-structured way. The latter is not compatible with existing practices in systems code, at least without significant code rewrites (cf. Listing 1).

An alternative to deadlock prevention is to dynamically *avoid* deadlocks, taking into account information about future lock usage. We have recently developed a method to dynamically avoid deadlocks guided by information about the order of explicit lock and unlock operations collected statically by program analysis [9]. Our static analysis is based on a type and effect system that is general enough to be applicable regardless of how locking is used. One of the benefits of our method is that valid programs have no dangling unlock operations, which may cause unspecified program behavior and concurrency errors such as data races. Safe lock usage is not addressed in other tools (e.g., [10]) which can only prevent data races given the assumption that there are no dangling unlock operations.

```
static int udf_readdir(struct file *filp, void *dirent,
                       filldir_t filldir)
{
  struct inode *dir = filp->f_path.dentry->d_inode;
  int result;

  lock_kernel();

  if (filp->f_pos == 0) {
    if (filldir(dirent, ".", 1, ..., dir->i_ino, DT_DIR) < 0) {
      unlock_kernel();
      return 0;
    }
    filp->f_pos++;
  }

  result = do_udf_readdir(dir, filp, filldir, dirent);
  unlock_kernel();
  return result;
}
```

Listing 1. Code from `linux-2.6-kdbg.git/fs/udf/dir.c`

Building upon the technique formally developed in our previous work [9], in this paper we outline an extended implementation that can support locks residing in dynamically allocated data structures in a context-sensitive and field-sensitive manner, stack-allocated data structures containing lock handles, and several optimizations that minimize the run time overhead of deadlock avoidance.

We start by a brief overview that describes informally how our approach manages to avoid deadlocks when unstructured locking is used (Sect. 2). We then describe our analysis and its current implementation (Sect. 3), and evaluate its performance (Sect. 4). The paper ends with a brief review of related approaches to deadlock freedom (Sect. 5) and some concluding remarks.

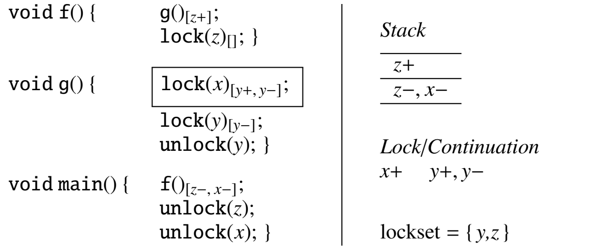
2. Overview of our Approach

The key idea of our technique is that granting a lock cannot lead to a deadlock when both the requested lock and its future lockset are available. The *future lockset* of a lock operation is the set of locks acquired between the lock and its matching unlock operation.

Let's see the analysis on an example. The analysis tracks *simple effects*, i.e., (possibly empty) sequences of lock and unlock events (e.g., $z+$ or $z-$). A *continuation effect* of an expression represents the effect of the code following that expression. Our analysis automatically annotates lock operations and function calls with their continuation effect. Even though lock operations are ordinary func-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLOS '11 October 23, 2011, Cascais, Portugal.
Copyright © 2011 ACM 978-1-4503-0979-0/11/10. . . $10.00



void f() { $\quad$ g()$_{[z+]}$;
$\quad$ lock(z)$_{[]}$; }

void g() { $\quad$ $\boxed{\texttt{lock}(x)_{[y+,\, y-]};}$
$\quad$ lock(y)$_{[y-]}$;
$\quad$ unlock(y); }

void main() { $\quad$ f()$_{[z-,\, x-]}$;
$\quad$ unlock(z);
$\quad$ unlock(x); }

Figure 1. An example program where lock and unlock operations are not in the same scope (left) and the run-time state of this program when the boxed term is executed (right).

tion calls in C programs, we treat them as distinct operations. Fig. 1 illustrates a program where lock operations and function calls are annotated with their continuation effect. Notice that lock and unlock operations reside in different scopes in this program. For instance, x is locked in function g, but it is unlocked in function main.

In contrast to continuation effects that are computed statically, the computation of future locksets is deferred until run time. At each function call, the associated continuation effect is pushed on the stack for the duration of the call. Figure 1 shows the run-time state of the program when control reaches the lock operation on x: the run-time stack (which grows upwards in the figure) contains the continuation effects of the f and g calls. The future lockset computation algorithm is straightforward given the continuation effect of x and the run-time stack: the algorithm starts with an empty future lockset and traverses the continuation effect until the matching $x-$ is found. While traversing the effect, other locations being locked are added to the future lockset. For instance, in Fig. 1 the algorithm adds y to the future lockset of x and then considers the continuation effects on the stack, from top to bottom. Thus, z is added to the future lockset and the matching unlock operation is found on the next element of the stack. The resulting lockset is $\{y, z\}$. During program execution the lock operation unblocks when lockset $\{x, y, z\}$ is available. Notice that we grant only a *single lock* for each lock operation (only x in this case) as opposed to acquiring prematurely the entire lockset with it ($\{y, z\}$), which would damage the program's degree of parallelism.

Having seen the idea, let's see the core ingredients of the analysis and how it deals with various programming language constructs.

Effects. Our static analysis is based on *effects*. *Simple effects* are sequences of lock and unlock events. A *continuation effect* of an expression represents the effect of the function code following that expression (i.e., our continuation effects are intra-procedural, in contrast to those of Hicks *et al.* [10] which are inter-procedural). Lock operations are automatically annotated with their continuation effect. As mentioned, the computation of future locksets is deferred until run time, allowing the calculation of future locksets to occur in a context-sensitive manner. More conservative (i.e., larger) locksets could be statically computed, but this approach would reduce parallelism by increasing the blocking time at lock operations. Within a single function, the future lockset computation algorithm is straightforward given the continuation effect of a lock acquisition operation for some lock l. We start with an empty future lockset and traverse the continuation effect until the matching unlock operation on l is found. The locations being locked while searching for the matching unlock operations are added to the future lockset. Notice that our algorithm deals with re-entrant locks so the search will only stop at the *matching* unlock operation rather than the first unlock operation. The runtime system utilizes the dynamically computed future locksets so that each lock operation can only proceed when its future lockset is *available* to the requesting thread.

Function calls. As mentioned, our continuation effects are intra-procedural. Therefore, the matching unlock operation of some lock may not be located in a function's effect. Our static analysis resolves this issue, by automatically annotating function calls with their continuation effect. At run time, when a function call is encountered, its continuation effect is pushed on a stack of continuation effects for the duration of the function call. When the matching unlock operation is not found in the continuation effect, the lockset computation algorithm proceeds with the remaining continuation effects on the run-time stack. Because the static analysis guarantees that for each lock there exists a matching unlock operation, the lockset computation algorithm will find it and therefore terminate.

Conditionals. A shortcoming of representing effects as ordered events is that, when typing conditional expressions/statements, it is too restrictive to require that both branches have the same effect.

```
lock(x);
if (condition) {
  lock(y); data[i] = j; unlock(y);
}
unlock(x);
```

In this example, the "then" branch contains a lock operation on y followed by an unlock operation on y, whereas the "else" branch is empty. Requiring that both branches have the same effect, would unnecessarily reject common multithreaded C programs such as the above. Our analysis overcomes this issue by keeping track of the effects in both branches and the lockset calculation algorithm computes the lockset of the two branches separately. The resulting lockset is formulated by joining the two locksets. However, for each lock, the only restriction we impose is that the number of unmatched lock/unlock operations must be equal in both branches.

Loops and recursion. Additional problems need to be addressed when dealing with recursive function definitions (loops can be treated similarly). In this case, the effect of a recursive function name must equal the effect of its body. However, this is impossible, as the two effects cannot be structurally equivalent: the effect of a function name is *contained* in the effect of its body, due to the recursive call. To overcome this issue, our analysis assigns function names a *summary* of the effect of their bodies. Although in this paper we do not formally define what function summary does (see [9] for its definition), a possible (but conservative) choice here would be to "flatten" or merge all branches of the effect corresponding to the body of a recursive function, by placing all matched operations in a single "atomic" effect. Unmatched unlock and lock effects are placed at the end and the beginning of the summarized effect respectively. We have shown formally the soundness of our summary [9], which is a conservative approximation of the actual effect of recursive functions. Intuitively, the summary function guarantees that all lock operations included in a recursive function f will be included in the future lockset of a lock operation preceding the call of f. In fact, summaries are conservative as all unmatched unlock operations are placed at the end of the summarized effect.

3. Deadlock Avoidance Analysis

Our analysis is performed in two phases. The first phase takes place at compile-time and performs a field-sensitive and context-sensitive pointer analysis followed by continuation effect inference, and source code instrumentation with effects; finally, the instrumented program is linked with the runtime system. The second phase is purely dynamic and takes place when the original program requests a lock. The future lockset of the requested lock is computed by utilizing the inserted effects and the lock is only granted when both the lock and its future lockset are available. In this paper, we extend the implementation of our analysis so that it can handle locks residing in data structures allocated in the heap or the stack and introduce new optimizations that reduce the size of effects, thereby reducing the size of future locksets and the blocking time of lock operations.

3.1 Static Analysis

The analysis takes as input the program's abstract syntax tree and constructs a call graph, which is visited bottom-up. There are four main stages involved in the analysis of function declarations (i.e., nodes of the call graph) described in the following paragraphs.

Pointer analysis. First, a standard, off-the-shelf field-sensitive intra-procedural pointer analysis based on symbolic execution [18] is employed so as to formulate an abstract heap and stack state at each program point. We have customized the analysis so that it treats heap allocation in a context-sensitive manner. At the end of the first stage, we obtain a mapping for each expression to a set of abstract locations. Each abstract location r can be a formal parameter, a global variable or a heap-allocated location.

Effect inference. The effect for each function is computed by running a standard forward data flow algorithm on the function's control flow graph. Each node in the control flow graph is associated with an input, a current and an output effect. The input effect of a node is formulated by *joining* effects flowing from all its *front* edges. For instance, a node associated with effects $\gamma_1 \ldots \gamma_k$, will be assigned an input effect $\gamma_1 \, ? \, \ldots \, ? \, \gamma_k$, which denotes a choice between the alternative effects $\gamma_1 \ldots \gamma_k$. A distinction is made when encountering back edges; we defer the discussion regarding the treatment of back edges until the next paragraph.

The current effect of a node can be r^+ or r^-, when a lock or unlock operation is found, respectively. It can be $\texttt{malloc}\,\rho$, when a new reference is allocated dynamically and bound to the variable ρ. It can also be $\texttt{call}\ r\,(r_1, \ldots, r_n) : r'$ when a function is called, where r is a reference to the function, $r_1 \ldots r_n$ are references to the function's arguments, and r' is a reference that corresponds to the function's result. In this case, if our standard points-to analysis cannot determine a unique target for r (e.g., if a function is called indirectly through a pointer), then the original effect is replaced by a joined effect consisting of several alternative branches of the form $\texttt{call}\ f_i\,(r_1, \ldots, r_n) : r'_i$, for each alternative target function f_i.

The output effect is computed by appending the current effect to the input effect and is propagated to a node's successors until a fixed point is reached. A function's effect is computed by joining the output effects of nodes having no successors.

Loops. Effects flowing from back edges must be equivalent (with respect to the lock counts) to the input effect of the same node. This restriction allows us to soundly encode loop effects: a loop may have any number of lock or unlock operations provided that upon exit of a loop the counts of each lock match the counts before the loop was executed. Assuming that the effect of the loop body is γ, then we take $(\gamma \, ? \, \emptyset)$, $(\gamma \, ? \, \emptyset)$ as the effect of the entire loop. The empty effect on both branches compensates for the case where a loop is not executed. The duplication of $(\gamma \, ? \, \emptyset)$ is required so that lock operations can match between successive loop iterations. In cases where a loop effect does not contain unmatched lock operations, this duplication may be optimized away.

Effect optimizations. While computing effects, several optimizations are performed so as to compact/elide effects and in general minimize the repetitions of the identical effect segments in a function's effect. One of the optimizations for compacting an effect computes the common prefix and suffix of the effects included in a join operator, to decrease the size of branches. Another kind of optimization is to flatten effects that consist of nested join operators. For example, an effect of the form $(\gamma_1 \, ? \, \gamma_2) \, ? \, (\gamma_3 \, ? \, \gamma_4)$ can be reduced to $\gamma_1 \, ? \, \gamma_2 \, ? \, \gamma_3 \, ? \, \gamma_4$. In addition, multiple occurrences of the *empty effect* in alternative effects $\gamma_1 \ldots \gamma_n$ are substituted with a single empty effect. These two optimizations are run alternately until a fixed point is reached in the size of the effect.

Call effects are substituted by a *summary* of the effect corresponding to the function being called. In summarized effects multiple lock/unlock pairs for the same reference are redundant. The intuition behind this optimization is that the future lockset that will be computed dynamically will be the same, regardless of the number of times that a lock operation occurs.

Another important optimization attempts to minimize the size of the run-time effect stack, so that the future lockset calculation algorithm visits as few stack frames as possible. One way to achieve this is by disabling code instrumentation for functions that do not directly perform any lock operations and do not contain calls to functions that will visit their effect frame at run time. Finally, we invoke the data flow algorithm, which is CPU-intensive, only for functions that are known to contain lock operations (by performing an in-advance linear search), to avoid additional overheads.

3.2 Code Generation

Our main goal was to minimize the overhead induced by "effect accounting". A naïve implementation of the technique informally described earlier would simply allocate and initialize effect frames for each function call or lock operation, which would be unacceptable in terms of performance. The code generation phase statically creates a *single* block of initialization code for the effect of each function and inserts effect index update instructions (i.e., a single assignment) before each call and lock operation. Therefore, the overhead imposed for such operations is minimal. Each function is also instrumented with instructions for pushing and popping effects from the run-time stack at function entry and exit points respectively. This imposes a constant overhead to function calls.

Finally, *mappings* for stack and heap pointers are generated at run time as such locations cannot be known statically. A mapping binds an abstract location to a run-time address. An inverse mapping is also maintained for abstract heap locations. When a deallocation operation is performed such as `free`, the inverse mapping is searched using the physical address to be deallocated and the binding between the abstract heap location and the physical address is removed from the heap mapping. In this way, our analysis is able to deal with locks that are dynamically deallocated and thereby avoid invalid accesses to deallocated locks.

3.3 Current Limitations

Non C code. Our analysis can strictly handle the C language. Library code cannot be analyzed as it is not C code. We have assumed that by default library functions have an empty effect. However, it is possible to provide user-defined effect annotations for library functions. The analysis cannot deal with non-local jumps (including signals) and inline assembly.

Pointer analysis. The off-the-shelf pointer analysis module fails when encountering programs with pointer arithmetic involving locks (including arrays) and recursive data structures that contain or point to locks. Even though our analysis extends the standard pointer analysis with context-sensitive tracking of fresh heap locations, it fails to track heap allocation (for data structures containing or pointing to locks) at recursive functions and loops. This limitation is dual to the aforementioned limitation regarding unbounded data structures. In addition, expressions passed in lock functions must be assigned a *unique* abstract location. Finally, we require that lock pointers are mutated only before they are shared between threads, and that locks referenced with at least two levels of indirection (e.g., via double pointers) are not aliased at function calls.

Conditional execution. The analysis also currently rejects programs in which lock and their matching unlock operations are conditionally executed in distinct conditional statements having equivalent guards. For instance, the following program is rejected:

```
if (condition) lock(z);
if (condition) unlock(z);
```

3.4 Runtime System

The runtime system overrides the standard implementation of locking functions such as the pthreads functions pthread_mutex_lock and pthread_cond_wait.[1] If a lock is already held by the requesting thread then the lock's count is simply incremented. (This occurs only when re-entrant locks are used; however, re-entrant locks are needed in languages that support unrestricted lock aliasing.) Otherwise, the runtime system performs two steps: it computes the future lockset of the requested lock and verifies that all locks in the future lockset are available when the lock is acquired.

The future lockset calculation algorithm uses the effect index inserted by the instrumentation phase to calculate the future lockset of the requested lock. When the matching unlock operation is not found in the function's effect, the algorithm visits the effects on the run-time stack. Locks held by the requesting thread are excluded from the lockset. Effect traversal is performed efficiently as the majority of effects are represented by arrays. Each atomic effect is represented by two machine words.

The next step is to acquire the lock provided that all locks in its future lockset are available. We have implemented three different strategies for dealing with unavailable locks. The first one employs *futexes* [7], which in general allow a thread to wait in the kernel for an event. The remaining two strategies employ *busy waiting* or *yield* control to other threads. As the performance of futexes was clearly superior in almost all benchmarks, we quickly focused on this strategy and used it exclusively for all the results that we present in the next section.

Our algorithm initially checks if all locks in the future lockset are available. If some lock is unavailable, we perform a wait operation on it (using one of the above strategies) and retry. If all locks in the future lockset are available, we *tentatively acquire* the requested lock. Then, we check again the future lockset. If any lock in the future lockset is unavailable, we *release* the acquired lock, we perform a wait operation on the unavailable lock and repeat all the steps from the beginning. Otherwise, the lock acquisition operation is considered successful. This approach allows our locking algorithm to be more permissive compared to versioning schemes that check future locksets atomically and thereby yields a higher degree of concurrency for instrumented programs.

4. Performance Evaluation

We describe some experimental results, aiming to demonstrate that our approach can achieve deadlock freedom with low run-time overhead. The experiments were performed on a machine with four Intel Xeon E7340 CPUs (2.40 GHz), having a total of 16 cores and 16 GB of RAM, running Linux 2.6.26-2-amd64 and GCC 4.3.2.

We used a total of six benchmark programs, which are real applications whose source code is publicly available. Some of these programs use language features that could not be handled by the preliminary implementation of our system [9]. The performance results are shown in Table 1; the presentation order is alphabetical.

benchmark	run in	user	system	elapsed	ratio
curlftpfs	C	0.002	0.758	33.450	0.982
	C+da	0.000	0.680	32.862	
flam3	C	63.660	3.910	49.050	1.003
	C+da	67.860	3.640	49.200	
migrate-n	C	5545.311	4631.341	4138.070	1.118
	C+da	5334.921	5020.346	4625.670	
ngorca	C	124.846	0.126	8.270	0.996
	C+da	124.467	0.126	8.240	
sshfs-fuse	C	0.000	0.890	20.880	1.000
	C+da	0.000	0.950	20.880	
tgrep	C	13.238	11.639	5.190	1.191
	C+da	14.801	11.655	6.180	

Table 1. Performance of C vs. C+da (C plus deadlock avoidance).

curlftpfs [3] and sshfs-fuse [14]: are file system clients that access hosts via the FTP and SSH network protocol respectively. Both applications create threads on demand so as to serve concurrent read and write requests to the file systems, using two and three distinct locks respectively to synchronize data structures, logging and access to non thread-safe functions. In our experiments, we mount a remote directory over the corresponding file system and start a fixed number of concurrent threads, each of which is trying to download a number of large files. The total volume of data that is copied over the file systems is linear w.r.t. to the number of threads. In both cases, the instrumented program has almost identical performance to the original program. Both programs have approximately zero user time as they mainly invoke the kernel-space API of FUSE [8].

flam3: a multithreaded program which creates "cosmic recursive fractal flames", i.e., (animations consisting of) algorithmically generated images based on fractals [5]. A single lock is used to synchronize access to a shared bucket accumulator that merges computations of distinct threads. We measured the time required to generate a long sequence of fractal images. The results again were almost identical for the original and the instrumented version of flam3.

migrate-n: estimates population parameters, effective population sizes and migration rates of n populations using genetic data [12]. The program maintains a work list of Markov chains and uses a thread pool to execute tasks until the work list becomes empty. Two locks are employed for implementing the thread pool and for accessing shared variables. It is worth mentioning that locks in this program are dynamically allocated and several billion lock operations were executed during the program run. The instrumented program ran 11% slower than the original.

ngorca: a multithreaded password recovery tool using exhaustive key search for DES-encrypted passwords in Oracle databases [13]. The program achieves speedup by splitting the search space of each encrypted password across threads, using multiple locks for implementing logging, counters and condition variables. The results again were almost identical for the original and the instrumented version of ngorca.

tgrep: a multithreaded version of the utility program grep which is part of the SUNWdev suite of Solaris 10 [16]. The program achieves speedup by splitting the search space across threads, using multiple locks for implementing thread-safe queues, logging and counters. In our experiment, we looked for an occurrence of a six-letter word in a directory tree containing 100,000 files. The instrumented program is 19% slower than the original program. This is due to the fact that tgrep is not only

[1] Currently our tool has built-in support only for overriding the functions of the pthreads library, but it can easily be extended to support other locks. Its implementation and the set of benchmarks we used are available from http://www.softlab.ntua.gr/~pgerakios/deadlocks/.

lock-intensive (about 1.5 million lock operations were executed in our test run), but also it is by far the benchmark with the longest effects that we could find. The maximum effect size for a function is 54 and the average effect size is 19.5, which are both about five times higher than the second next benchmark (ngorca). Furthermore, the program employs seven distinct global locks; the dynamically calculated future lockset had a maximum size of five elements and an average size of 1.3, again about five times higher than the second next benchmark.

5. Deadlock Freedom and Related Work

Deadlock freedom can be obtained by either of the following three strategies. The first, *deadlock prevention*, ensures that programs are correct by design and can never have circular lock dependencies. In the deadlock prevention literature, one finds type and effect systems [2, 4, 6, 11, 15, 17] that guarantee deadlock freedom by statically enforcing a global lock-acquisition ordering, which must be respected by all threads. Second, *deadlock detection* and recovery strategies dynamically detect deadlocks and preempt some of the deadlocked threads, releasing (some of) their locks, so that the remaining threads can make progress. Third, type systems for *deadlock avoidance* (e.g., [1]) grant access to resources that cannot lead to deadlocked states. To achieve deadlock freedom, such systems employ run-time resource monitoring and utilize static information regarding thread resource allocation. The main advantage of systems based on deadlock avoidance is that a larger class of programs is accepted by the static analysis without requiring the insertion of manual annotations or changing the original program structure.

From approaches that combine static and dynamic techniques, a tool that is quite similar to ours is Gadara [19]. Gadara employs whole program analysis to model programs and discrete control theory to synthesize a concurrent logic that avoids deadlocks at run time. Gadara targets C/pthreads programs and claims to avoid deadlocks quite efficiently because it performs the majority of its deadlock avoidance computations offline. (The tool is not publicly available.) Similarly to our future locksets, Gadara uses the notion of *control places* to decide whether it is safe to admit a lock acquisition. More precisely, a lock acquisition can only proceed when all the control places associated with the lock are available. The mostly static approach followed by Gadara, as well as the lack of alias analysis, results in an over-approximation of the set of run-time locks associated with a control place.

6. Concluding Remarks

Deadlocks are an important problem especially for systems code written in languages that employ non block-structured locking. In this paper, we presented implementation aspects of a novel tool that dynamically avoids deadlock states for multithreaded C programs. The key idea is to utilize statically computed information regarding lock usage at run time in order to avoid deadlocks. We described the main aspects of our static analysis and its obvious limitations: it is necessarily imprecise and cannot support unbounded data structures containing locks. However, we showed that our approach is applicable to several multithreaded C programs containing systems code and our evaluation results reveal that it imposes only a modest run-time overhead, induced by the future lockset computation and by blocking threads more often (i.e., when the requested lock is available but something in its future lockset is not). Nevertheless, we think that its run-time overhead is reasonable for guaranteed deadlock avoidance.

Acknowledgement

This research is partially funded by the programme for supporting basic research (ΠΕΒΕ 2010) of the National Technical University of Athens under a project titled “Safety properties for concurrent programming languages.”

References

[1] G. Boudol. A deadlock-free semantics for shared memory concurrency. In *Proceedings of the International Colloquium on Theoretical Aspects of Computing*, volume 5684 of *LNCS*, pages 140–154. Springer, 2009.

[2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, New York, NY, USA, Nov. 2002. ACM Press.

[3] A FTP filesystem based on cURL and FUSE. http://curlftpfs.sourceforge.net/.

[4] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 237–252, New York, NY, USA, 2003. ACM.

[5] flam3.com. Cosmic recursive fractal flames. http://flam3.com/.

[6] C. Flanagan and M. Abadi. Types for safe locking. In *Programming Language and Systems: Proceedings of the European Symposium on Programming*, volume 1576 of *LNCS*, pages 91–108. Springer, 1999.

[7] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Proceedings of the Ottawa Linux Summit*, pages 479–495, 2002.

[8] A filesystem in userspace. http://fuse.sourceforge.net/.

[9] P. Gerakios, N. Papaspyrou, and K. Sagonas. A type and effect system for deadlock avoidance in low-level languages. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 15–28, New York, NY, USA, 2011. ACM Press.

[10] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing*, June 2006.

[11] N. Kobayashi. A new type system for deadlock-free processes. In *International Conference on Concurrency Theory*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

[12] A tool that estimates population size and migration rate. http://popgen.sc.fsu.edu/Migrate/Migrate-n.html.

[13] A password recovery tool for Oracle Database. http://code.google.com/p/ngorca/.

[14] SSH FileSystem. http://fuse.sourceforge.net/sshfs.html.

[15] K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *Asian Symposium on Programming Languages and Systems*, volume 5356 of *LNCS*, pages 155–170. Springer, 2008.

[16] Multithreaded grep. Part of Sun Microsystems’ *Multithreaded Programming Guide*, available at http://docs.sun.com/app/docs/doc/806-5257.

[17] V. Vasconcelos, F. Martin, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, volume 17 of *EPTCS*, pages 95–109, 2010.

[18] J. W. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 205–214, New York, NY, USA, 2007. ACM.

[19] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 281–294. USENIX Association, 2008.