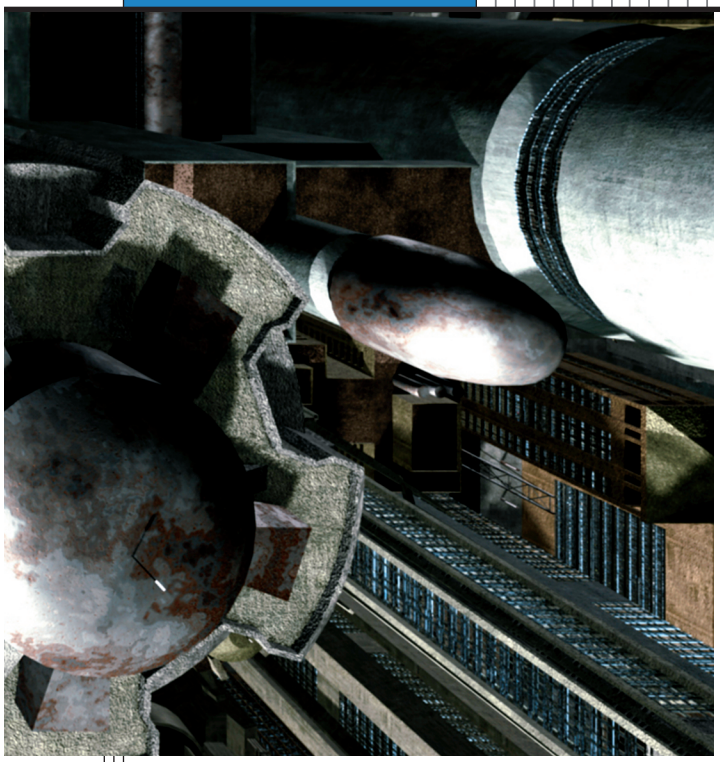


# ELIMINATING CONCURRENCY BUGS WITH CONTROL ENGINEERING

Terence Kelly and Yin Wang, *Hewlett-Packard Laboratories*
Stéphane Lafortune and Scott Mahlke, *University of Michigan*

**In the multicore era, concurrency bugs threaten to reduce programmer productivity, impair software safety, and erode end-user value. Control engineering can eliminate concurrency bugs by constraining software behavior, preventing runtime failures, and offloading onerous burdens from human programmers onto automatically synthesized control logic.**

Computer programming has never been easy, and the cost of errors has always been high. Software failures have claimed lives, and expensive software project failures are the stuff of industry legend. Over time, however, improvements in programming languages, development tools, and education have ameliorated the difficulties of ordinary serial programming. Average programmers circa 2004 were as productive and competent as their counterparts in other engineering domains. Recent hardware trends, however, threaten to erode software dependability, programmer productivity, and the industry's rate of economic value creation.

For decades, uniprocessor performance improvements made serial software run faster at no cost to the programmer. In recent years, however, uniprocessor performance has flattened out due to heat dissipation barriers. For the foreseeable future, multicore processors promise more cores, rather than faster cores, in successive hardware generations. Serial application performance therefore can no longer effortlessly piggyback on hardware performance gains. Only parallel software can exploit multicore hardware's full potential.

The problem is that parallel programming is far harder than serial programming. Reasoning about concurrency is formidably difficult even in shared-memory multithreaded programming, which partly preserves the style of sequential programming, due to the numerous possible interleavings of basic operations. Faulty reasoning can cause errors (data races[1]) or nontermination (deadlock), and such defects can easily survive testing with disastrous results in production—subtle concurrency bugs in thoroughly tested software caused the notorious Therac-25 fatalities.[2] Conservative programming practices such as coarse-grained locks lower the risk of bugs in new software but reduce concurrency and therefore impair performance, negating the main benefit of parallelization. Finally, multicore hardware will expose latent concurrency bugs in legacy multithreaded software that ran without frequent failure on uniprocessors.

The necessity and the difficulty of parallel programming together pose a serious challenge for the computer

 Published by the IEEE Computer Society 

industry. Users upgrade hardware and software to obtain richer functionality and better performance. If the challenges of parallel programming prove insurmountable, however, new software releases will be no more reliable, no faster, and hence no more valuable than their predecessors. Hardware value follows the same trend—Would you scrap a 32-core laptop if available software performs no better on a 64-core upgrade?—and longer replacement cycles imply corresponding slowdowns in economic value creation.[3]

To summarize the concurrency crisis: Multicore hardware will make shared-memory multithreaded software ubiquitous. Therefore, we must enable average programmers to do something they have never before been called upon to do: write correct and efficient multithreaded code in large volume and at reasonable cost in time and money. Leading observers see peril ahead for the entire IT industry and call for revolutionary solutions.[4]

We have found inspiration and useful technology to address the challenges of parallel programming in a seemingly unlikely quarter: *control engineering*. Classical control theory makes it possible to safely and efficiently control complex and potentially dangerous systems such as oil refineries and aircraft avionics. This theory has enjoyed remarkable successes in industrial applications for more than a century, and today it is pervasive in consumer applications that improve our everyday lives. Conventional control theory is best suited to physical systems with continuous state spaces and coupled-differential-equation dynamics—systems with little obvious resemblance to concurrent software. A lesser-known and more recent branch of control theory, however, deals with discrete state spaces and event-driven dynamics.

Several years ago, we formed a team of control engineers and systems/software and compiler specialists to bring to concurrent software the benefits that classical control brought to physical systems. Gadara[5,6] represents our latest foray into the intersection of concurrent software and control engineering. Gadara uses *discrete control theory* (DCT)[7] to analyze concurrent software and automatically repair an important class of concurrency bugs: deadlocks involving standard synchronization primitives, including circular-mutex-wait deadlocks. Because Gadara rests on a rigorous theoretical foundation, it can decompose the practical goal of deadlock elimination into well-studied formal problems, leverage a large body of proven methods, and deliver hard safety/correctness and performance guarantees.

## CONTROL ENGINEERING

Figure 1 illustrates the basic modeling-control paradigm of control engineering. We begin with a complex real-world system that we wish to control, conventionally

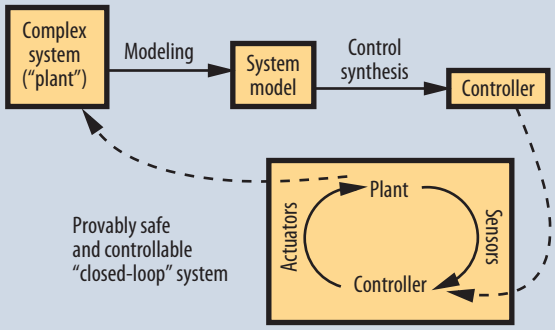



**Figure 1. Basic modeling-control paradigm of control engineering.**

termed the "plant." For example, the plant might be a boiler whose temperature must be kept constant; this might be difficult due to environmental perturbations such as temperature fluctuations.

We then develop a formal model of the system—for example, a set of differential equations describing the time evolution of state variables as functions of the conditions that influence them. Control theory supplies mathematical methods that synthesize a controller from the system model and whatever constraints we wish to enforce (such as "maintain a specified temperature"). Finally, when we couple the controller to the plant with sensors (say, thermometers) and actuators (say, heating elements), we obtain a controlled "closed-loop" system.

The control engineering paradigm confers several important benefits. It yields a controller that automatically manages a plant, a task that might be difficult or impossible for a human operator. The automatic controller is often welcome, even if manual control is possible, because it relieves the human operator of a tiresome burden (as in cruise control for automobiles). More importantly, controller design can rely heavily on rigorous, time-tested, standard design methodologies that place the power of control engineering in the hands of large numbers of practitioners rather than restricting it to highly trained specialists.

Most importantly, modern control theory guarantees that the closed-loop system will behave according to specification in the field. This hard correctness guarantee contrasts starkly with earlier unprincipled ad hoc/heuristic control approaches, which "work except when they don't." Decades of painful experience have taught control engineers to insist upon safety guarantees with principled mathematical foundations.

Control engineering is a remarkably successful paradigm, and control theory makes much of the modern world possible. Conventional control theory provides safe, efficient automation for industrial processes and consumer applications ranging from chemical plants to refrigerators. It is so widely deployed and trusted that it has become

invisible despite its ubiquity. One of the last frontiers for control theory, which until recently has resisted the control engineering paradigm's encroachment, is software reliability. It is this frontier that we have explored with a relatively new branch of control theory.

## DISCRETE CONTROL THEORY

Inspired by the impressive successes of conventional control theory, starting in the 1980s researchers began developing an analogous body of DCT for systems with discrete state spaces and event-driven dynamics.[7] At a high level, DCT shares with conventional control theory the basic control engineering paradigm: It begins with a "plant" model—typically a finite automaton or Petri net—that captures a real-world system's dynamics. It synthesizes control logic that, when connected to the plant's sensors and actuators, "closes the loop" to enforce specified behavioral restrictions (such as "steer clear of unsafe states") by postponing state transitions in the plant.

**DCT's rigorous mathematical foundation provides hard safety guarantees, and its mode of operation minimizes online intrusiveness and overhead.**

The attractions of DCT are at least as compelling as those of classical control. DCT is arguably even more accessible to practitioners than classical control. It supports modeling formalisms such as Petri nets that can succinctly and conveniently model complex concurrent systems. DCT automatically synthesizes control logic from declarative behavioral specifications, and this control logic is provably safe and correct by construction. Further, it is *maximally permissive* in that it never intervenes in the plant unless intervention is necessary to enforce given behavioral specifications. Moreover, the online decision-making associated with DCT need not introduce performance bottlenecks because DCT offloads burdensome computations to offline control logic synthesis; the online safety checks required to enforce the given behavioral specifications are lightweight, fine-grained, decentralized, and highly concurrent. DCT's rigorous mathematical foundation provides hard safety guarantees, and its mode of operation minimizes online intrusiveness and overhead.

When we realized that DCT allows us to control the logical behavior of concurrent software, we began to apply it to software failure avoidance. Our early work focused on "workflow" programs for data center automation—very-high-level scripts in restricted languages that emphasize concurrent control flow rather than data manipulation. We found that DCT could synthesize control logic for real production workflows to avoid arbitrary user-specified forbidden execution states, including deadlocks.[8] The situation becomes much more difficult when we remove the severe restrictions of workflow languages and tackle concurrent programming in general-purpose languages.

## DEADLOCK IN THE MULTICORE ERA

We focus on deadlocks in shared-memory multithreaded programs that employ conventional mutual exclusion and synchronization primitives—for example, C programs that use the Posix threading library. Although alternatives such as transactional memory and lock-free data structures attract increasing attention, old-fashioned mutexes and condition variables will remain important in practice for the foreseeable future. One reason is that they often score higher in terms of performance, compatibility with I/O, and maturity of implementations. Further, existing lock-based programs and the developers who write them represent enormous investments that must be preserved going forward.

Conventional locks, however, are the root of a growing deadlock menace. It is difficult for human programmers to reason about nondeterministic, interleaved, lock-mediated concurrency. Locks also force a nasty tradeoff between correctness and performance: Particularly on multicore hardware, fine-grained locking typically offers performance advantages over coarse-grained locking, but the former is more error-prone.

The fundamental problem is that deadlock freedom is a global program property, so programmers cannot restrict themselves to local reasoning about individual modules but must consider a program in its entirety. Locks undermine modularity—and therefore divide-and-conquer problem solving—because they are noncomposable: Combining correct lock-based components does not necessarily yield correct composite software, as the "Hard-to-Diagnose Deadlocks" sidebar shows. A good solution to the deadlock problem will restore composability to lock-based software and eliminate the need for global reasoning by programmers.

Finally, obscure corner-case deadlocks occur even within single modules developed by individual expert programmers; such bugs are difficult to detect, and repairing them is costly, manual, time-consuming, and error-prone. In addition to preserving the value of legacy code, a good solution to the deadlock problem will improve new code by letting programmers focus on modular common-case logic rather than fragile global properties and obscure corner cases. Such a solution will empower programmers to write safe and efficient parallel code as confidently and productively as they wrote serial code.

Decades of study have yielded several approaches to prevent and detect deadlocks, but they collectively do not address the challenges of the multicore era. Static

deadlock prevention via strict global lock-acquisition ordering is straightforward in principle but prohibitively difficult in practice: Attempts to define and enforce such an ordering in software developed by loosely coupled teams separated by both time and geography frequently fail.

Static deadlock detection via program analysis has made impressive strides in recent years, but spurious warnings are common and the cost of manually repairing genuine deadlock bugs remains high. Dynamic deadlock detection identifies the problem too late, when recovery is awkward or impossible; automated rollback and reexecution can help, but irrevocable actions such as I/O can preclude rollback.

Generalizations of the Banker's algorithm[9] can in principle provide dynamic deadlock avoidance but require more information than is often available and involve expensive serial safety checks. The multicore era's stringent performance requirements demand that online deadlock-avoidance checks be very fast, and scalability demands that they be parallel.

Approaches that "learn" to avoid recurrences of past deadlocks[10] may help with those that recur with high probability, but a complete solution must also address the "long tail": the huge number of deadlock bugs that individually bite only rarely but that collectively cause deadlocks with unacceptable frequency.

## GADARA: ELIMINATING DEADLOCKS WITH DCT

Gadara is our approach to automatically enable multithreaded programs to dynamically avoid deadlocks. It proceeds in four phases:

- compiler techniques extract a formal model from program source code;

## HARD-TO-DIAGNOSE DEADLOCKS

**Hard-to-diagnose deadlocks can arise in software developed over time by several programmers. Consider a large single-threaded legacy program containing tables, "container" structures that store nodes. Tables support the kinds of operations we would expect—insert a node, delete a node, find the node with a given key, and so on. Trouble starts when performance requirements dictate that the program be multithreaded. Because tables must now support concurrent accesses, locks must be retrofitted onto tables and nodes to prevent data races.**

**Formerly straightforward operations now require lock/unlock calls that are, for programmers, pesky annoyances unrelated to the objectives at hand. Maintenance programmer Andrew has therefore hidden the lock/unlock calls needed for a common operation in a DELETE macro, which avoids the overhead of yet another function call:**

```
#define DELETE(n, t) \
    do { \
        lock(n->L); \
        lock(t->L); \
        table_delete(t, n); \
        unlock(t->L); \
        unlock(n->L); \
    } while (0)
```

**His colleague Betty has written an analogous `UPDATE` macro:**

```
#define UPDATE(t, key, val) \
    do { \
        node_t *n; \
        lock(t->L); \
        n = table_find(t, key); \
        if (n) { \
            lock(n->L); \
            n->value = val; \
            unlock(n->L); \
        } \
        unlock(t->L); \
    } while (0)
```

**Unfortunately, Andrew and Betty failed to agree on lock acquisition order; the code can deadlock with `DELETE` holding a node lock and `UPDATE` holding a table lock.**

**Neither code fragment looks wrong—indeed, neither is wrong; the bug arises from their interaction. Deadlock occurs only during rare thread interleavings and therefore does not manifest during testing. However, when the program is released to a large user base, deadlock complaints pour in. Now yet another pain point becomes apparent because this single bug can manifest in myriad ways: The program contains numerous uses of `DELETE` and `UPDATE`, and every pair of uses is a potential deadlock.**

**The bewildering variety of manifestations—each of which yields a completely different "autopsy report" for the few end users diligent enough to generate core files and analyze per-thread stack traces—makes it impossible to unify the bug reports and link them to a single root cause. Most users simply complain about mysterious and unreproducible "hangs" under many different inputs and configurations, the difficulty of debugging stripped, highly optimized, function-inlined production executables having deterred them from further investigation.**

**The mystery deepens if the program in question is a server and deadlocks ensnare pairs of worker threads from a fixed-size pool. Clients report sporadic timeouts, but the bug appears to be transient if retries succeed. Meanwhile, server administrators observe a gradual decline in throughput as the pool of remaining worker threads dwindles, but restarting the program "fixes" the problem—which looks more like a resource leak than a deadlock.**

**For a real-world example of a single, fiendishly difficult-to-diagnose concurrency bug with at least 30 distinct manifestations, see M. Musuvathi et al., "Finding and Reproducing Heisenbugs in Concurrent Programs," *Proc. 8th Symp. Operating Systems Design and Implementation* (OSDI 08), Usenix, 2008, pp. 267-280.**

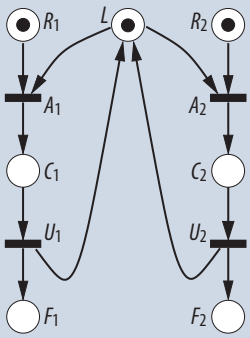



**Figure 2. Petri net. Petri nets are bipartite directed graphs containing two types of nodes: places (depicted as circles) and transitions (bars). The number of tokens (dots) in all places is the Petri net's marking (state). Transitions model events that change the marking. Arcs connecting places to a transition represent preconditions of the associated event.**

- DCT methods automatically synthesize control logic that dynamically avoids deadlocks in the model;
- instrumentation embeds the control logic in the program; and
- at runtime the control logic compels the program to behave like the deadlock-free controlled model.

The net effect is that Gadara intelligently postpones operations such as lock-acquisition attempts when necessary to ensure that deadlock cannot occur.

## Modeling programs

Gadara begins by extracting per-function control flow graphs (CFGs) from program source code using standard compiler techniques. It then enhances these CFGs with information about lock acquisition/release and synchronization function calls and translates the CFGs into Petri net models. Finally, Gadara merges the latter into a single whole-program Petri net model of control flow.

Petri nets are bipartite directed graphs containing two types of nodes: *places* (depicted as circles) and *transitions* (bars). The number of *tokens* (dots) in all places is the Petri net's *marking* (state). Transitions model events that change the marking. Arcs connecting places to a transition represent preconditions of the associated event.

For instance, in Figure 2, transition $A_1$ in the Petri net is enabled because its input places $R_1$ and $L$ each contain at least one token. Similarly, $A_2$ is enabled, but all other transitions are disabled. Place $L$ can thus represent a lock that is available if $L$ contains a token; this Petri net can model two threads $i = 1, 2$ that both request $L$ after reaching $R_i$ and must acquire the lock via transition $A_i$ before entering a critical section represented by $C_i$. The Petri net models the mutual exclusion property of locks because if transition $A_1$ *fires* (occurs), it consumes one token from each input place $R_1$ and $L$ and deposits one token in its output place $C_1$. In general, the firing of a transition consumes tokens from each of its input places and produces tokens in each of its output places; the token count need not remain constant. After $A_1$ fires, $A_2$ becomes disabled and must wait for $U_1$ (unlock by thread 1) before it becomes enabled again.

Petri nets can model other important program features, including branches, forks/joins, loops, thread creation, and recursive function calls.[6]

Figure 3a illustrates Gadara's operation on a highly simplified deadlock bug from the Apache webserver. This subtle bug arises from the interaction between condition synchronization and mutual exclusion: If the thread executing function `f()` reaches the condition wait call while holding lock `L` before the thread executing `g()` reaches its lock call, then both threads deadlock as indicated by the code comments: `f()` waits in vain for a signal that `g()` will never send, while `g()` waits eternally for the lock that `f()` will never release. (For simplicity, the mutex paired with the condition variable is not shown; `L` is an unrelated lock.) Figure 3b shows Gadara's Petri net model that represents the lock and condition variable interdependencies between `f()` and `g()`.

It is important to note that Gadara's modeling phase is more automated than that of most conventional control engineering exercises. Gadara constructs initial Petri net models directly from control flow information automatically extracted from program source code by standard compiler techniques. Gadara allows programmers to refine the initial models through simple local function annotations.

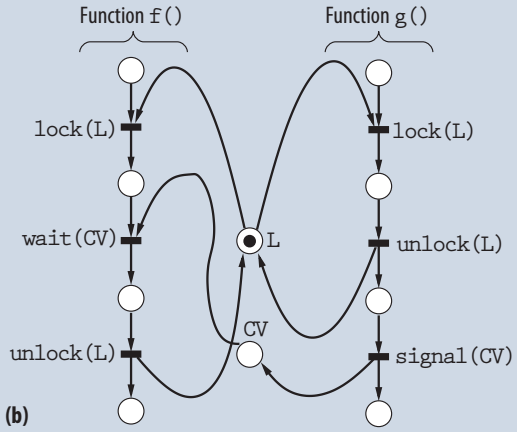
It has been said that "all models are wrong, but some are useful." Gadara's models are necessarily imperfect because perfect static analysis of program behavior is undecidable. Gadara errs on the side of caution by constructing conservative models that always capture deadlocks present in the target program but that possibly also contain spurious deadlocks. The net effect of such "false positives" can be unnecessary performance overhead, but in practice this overhead is tolerable.[5]

## Control logic synthesis

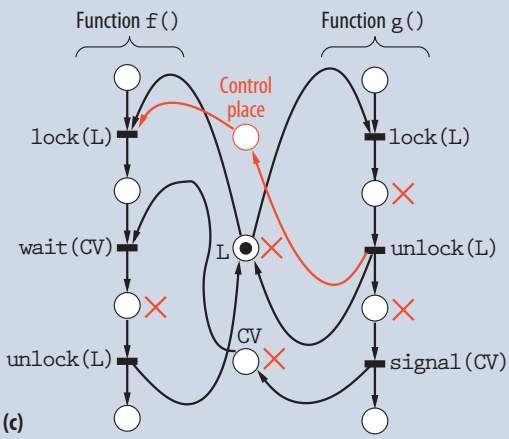
After constructing a whole-program model, Gadara next identifies potential deadlocks as structural features of the Petri net called *siphons*. A siphon is a set of places that can never regain a token once it is depleted of all tokens. Gadara establishes a correspondence between deadlocks in a program and empty siphons in its Petri net model and uses standard DCT analyses to identify siphons. The places constituting the siphon in our example are marked with red Xs in Figure 3c. Our deadlock-elimination problem therefore reduces to ensuring that siphons in the Petri net model are controlled to prevent them from draining empty of tokens and that the real-world program behaves like the controlled Petri net model.

```
f(...) {
    ...
    lock(L);
    ...
    wait(CV); /* deadlock */
    ...
    unlock(L);
    ...
}
g(...) {
    ...
    lock(L); /* deadlock */
    ...
    unlock(L);
    ...
    signal(CV);
    ...
}
```

(a)




(b)




(c)

```
f(...) {
    ...
    GADARA_LOCK_DEPLETE(L, ctrl_place);
    ...
    wait(CV);
    ...
    unlock(L);
    ...
}
g(...) {
    ...
    lock(L);
    ...
    unlock(L);
    GADARA_REPLENISH(ctrl_place);
    ...
    signal(CV);
    ...
}
```

(d)

**Figure 3. Example program based on Apache bug #42031. (a) Simplified original code; (b) Petri net model of original code; (c) controlled model resulting from siphon analysis and SBPI; (d) "Gadarized" code, with the Gadara instrumentation shown in red.**

We ensure that a siphon cannot drain by applying a DCT technique called *supervision based on place invariants* (SBPI).[11] The inputs to SBPI are a Petri net and a weighted linear constraint on its marking. To request that a siphon never drain, we simply specify that the total number of tokens in the places constituting the siphon must be at least 1. SBPI's control logic output is a *control place* to be added to the original Petri net; the control place alters the Petri net's dynamics and guarantees that the siphon cannot drain. Figure 3c shows in red the control place and incident arcs that Gadara's SBPI has added to address the deadlock bug in our example. The control place ensures that `f()` cannot acquire lock `L` until after `g()` has released it, thus effectively eliminating the deadlock in our Petri net model.

An important benefit of SBPI is that it generates *maximally permissive* control logic that provably postpones the progress of threads only when necessary to ensure that deadlock cannot occur in a worst-case future of the program's execution. The theoretical property of maximal permissiveness in Gadara's control logic thus contributes directly to the practical property of maximal runtime concurrency in the controlled program.

The addition of control places can create new potential deadlocks. However, Gadara can address these "control-induced deadlocks" by simply applying siphon detection and SBPI repeatedly until no uncontrolled siphons remain. In our experience to date, control logic synthesis for real software terminates after a single iteration of SBPI, which takes a few seconds.

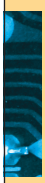
Gadara addresses the most common types of deadlocks, including circular-mutex-wait deadlocks involving standard portable locking primitives; our previous publications[5,6] define the scope precisely. Finally, some programs are uncontrollable in the sense that they cannot be prevented from deadlocking. In the simplest example, a thread repeatedly locks a single nonrecursive mutex. Gadara

detects uncontrollability during control logic synthesis and issues appropriate warnings.

### Instrumentation and dynamic control

The only remaining problem is to ensure that the original program's runtime behavior conforms to that of the deadlock-free controlled model that we obtained via siphon analysis and SBPI. Gadara solves this problem by instrumenting the original program: Our prototype performs a source-to-source transformation, which maximizes portability and requires no changes to threading libraries or other infrastructure.

Figure 3d illustrates how Gadara implements the control logic of the Petri net in Figure 3c. Gadara replaces the original `lock(L)` call in function `f()` with a wrapper, `GADARA_LOCK_DEPLETE()`, that atomically obtains the original lock `L` and also decrements a variable representing the token count in the control place of Figure 3c. If no token is present in the control place, the wrapper function waits for one to be deposited there; the wait is implemented with an ordinary condition variable (not shown in the figure). Gadara modifies function `g()` by adding a `GADARA_REPLENISH()` call that deposits a token in the control place and signals its condition variable, allowing `GADARA_LOCK_DEPLETE()` in `f()` to return.



**Gadara correctly identified and eliminated both previously reported and unknown deadlocks in the real software and similarly eliminated injected deadlock faults in the benchmark.**

Note that Gadara does not in any way meddle with three of the original four lock/unlock operations in Figure 3a. In real software, Gadara leaves the vast majority of all lock/unlock operations completely unaffected and therefore adds zero overhead to them. Further, the control logic that Gadara does add is

- *lightweight*—it adds only a simple condition variable wait/signal implementing the control place;
- *decentralized*—it affects only threads executing functions `f()` and `g()` and has no effect on other threads executing unrelated code; and
- *fine-grained*—it addresses a specific deadlock fault with a dedicated control place.

The net result is that Gadara's control logic is *highly concurrent*; it introduces no global serialization into software. This important property sets Gadara apart from deadlock-avoidance schemes involving a central "banker" that performs centralized safety checks upon every lock acquisition request.[9] In addition to being computationally expensive, such checks must be performed serially because they require exclusive access to the banker's central "account ledger." Yesterday's uniprocessors did not allow threads to execute in parallel, so the resulting serialization might have been acceptable on such hardware (if the safety checks were fast). On multicore hardware, however, globally serializing lock acquisitions creates a global performance bottleneck that prevents parallel software from fully exploiting the parallel hardware. Gadara and other scalable approaches do not globally serialize lock acquisitions.[10]

## EXPERIMENTAL EVALUATION

We have applied our Gadara prototype to several C/Pthreads programs, including the Apache webserver, the OpenLDAP directory server, the BIND name server, a client-server transaction processing benchmark application, and several other programs. "Gadarizing" software roughly doubles the time required to build a program from scratch: Model construction takes about as long as running `make`, and control logic synthesis takes a few additional seconds. For some programs, we locally annotated a small fraction of the program's functions to help Gadara perform siphon analysis more efficiently; this required a few minutes per function for a programmer unfamiliar with the software.

Gadara correctly identified and eliminated both previously reported and unknown deadlocks in the real software and similarly eliminated injected deadlock faults in the benchmark. Not surprisingly, deadlocks in mature, widely used open source software tend to occur in infrequently executed corner-case code rather than "hot" code paths. Gadara's deadlock-avoidance instrumentation therefore also executes infrequently, and its runtime overhead is typically negligible. For example, when Gadara eliminated a known deadlock in OpenLDAP's application-level cache insertion/deletion functions, the overhead was negligible under normal configuration because these functions are infrequently exercised.

We had to configure OpenLDAP in a highly unconventional way—with a very small cache size and database disk synchronization disabled—to trigger measurable Gadara overheads; even under these adverse conditions, the negative impact on throughput and response time never exceeded 10 percent. We also injected a deadlock fault into a common-case code path in our client-server benchmark application to ensure that every transaction triggered Gadara overhead. The impact on response times was negligible under normal workload. We observed substantial performance degradation (18 percent reduction in throughput) only under extreme oversaturation.[5]

## HOW TO ERR?

Perfect static program analysis is impossible: Static deadlock detection cannot both guarantee to find only genuine deadlock bugs and also guarantee to find all deadlock bugs. Gadara errs on the side of caution: It detects and remedies all deadlocks actually present in a program, and possibly also spurious deadlocks. The control logic that it adds to address the latter may impair performance, but in practice this overhead is typically very small.

DCT methods could be used in very different ways to achieve different tradeoffs. For example, DCT could provide an alternate implementation for a deadlock "exhibiting/healing" system.[10] First, we would observe or trigger a runtime deadlock in an unmodified program; recent techniques make it easier to trigger deadlocks by perturbing thread interleavings[10] or exploring them systematically.[12] Next, siphon analysis would yield siphons representing every potential deadlock in the program. The remaining problem would then be to identify the one siphon responsible for the observed runtime deadlock. A siphon corresponds to a set of basic blocks, and standard debugging tools could reveal the basic blocks where threads are deadlocked. Finally, SBPI would generate control logic to address the guilty siphon alone, ignoring all other siphons. The result would be to eliminate only genuine deadlock bugs that have actually occurred but not those that have not yet been observed.

Intuitively, the strategy of addressing only previously observed deadlocks is appealing for relatively stable software whose deadlock bugs would otherwise recur often, especially when the bugs facilitate remediation by manifesting consistently: An investment in eliminating such bugs, automatically or manually, yields frequent dividends. The situation is more complicated if we face large numbers of deadlock bugs that individually manifest rarely but collectively strike with unacceptable frequency, or deadlock bugs that thwart root-cause diagnosis and remediation by manifesting in numerous different guises, or a steady stream of new deadlock bugs introduced by rapid software development. Gadara's comprehensive approach of eliminating all deadlocks attacks the long tail of low-probability deadlock bugs and those that manifest inconsistently, and can keep pace with newly written deadlock bugs.

Our experience with Gadara convinces us that discrete control theory offers attractive benefits for solving concurrency problems. DCT enables Gadara to provably eliminate all instances of a broad class of deadlocks from a program without introducing new deadlocks, silently disabling functionality, or affecting program correctness. It also shifts expensive deadlock-avoidance computations offline and computes maximally permissive control logic that allows maximal runtime concurrency. The relatively few online checks required to avoid deadlocks are lightweight, decentralized, fine-grained, and highly concurrent.

Our approach relieves programmers of the burden of global reasoning about deadlock freedom, restoring the composability that locks destroy. It is compatible with legacy programs, programming practices (such as I/O in critical sections), and infrastructure (such as threading libraries), and requires no retraining or conceptual reorientation. Our ongoing work is developing a more robust and usable Gadara prototype, which we plan to eventually release to the public.

Looking beyond deadlocks, we believe that DCT might provide a unified framework for eliminating other concurrency bugs. By constructing Petri net models of programs, we hope to address other classes of bugs using different control specifications. For example, if static or dynamic analysis identifies data races, we might use SBPI to eliminate them by specifying that the number of tokens (threads) in the places (basic blocks) affected by the race should not exceed one. Given code containing programmer-specified `atomic{}` sections, we believe it is possible to use DCT to automatically assign locks to protect them. Thanks to the maximal permissiveness that DCT offers, the net result might be a more concurrent enforcement of the atomicity requirements than prior approaches to lock-based `atomic{}` sections.

Finally, we believe that the broader prospects for DCT in computing systems are not confined to concurrency control in multithreaded software. We expect that for a wide range of problems, including access control and communication protocols, the benefits of control engineering will outweigh the additional effort required to exploit this paradigm. Because it is a model-based method, DCT requires model building to bridge the gap between theory and practice, and modeling is usually the most difficult part of the overall exercise. However, our experience with Gadara, which leverages existing compiler techniques to construct models, leads us to suspect that many other problems may be amenable to DCT. After investing effort in modeling a system that we wish to control, DCT offers handsome returns by exploiting a wide range of mature and powerful control techniques to constrain its behavior.


## Acknowledgments

The ongoing Gadara project at the University of Michigan is supported by National Science Foundation grant CCF-0819882 and by an Innovation Research Program award from Hewlett-Packard Laboratories.


## References


1. J.W. Voung, R. Jhala, and S. Lerner, "RELAY: Static Race Detection on Millions of Lines of Code," *Proc. 6th Joint*

*Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. Foundations of Software Eng.* (ESEC/FSE 07), ACM Press, 2007, pp. 205-214.
2. N.G. Leveson and C.S. Turner, "An Investigation of the Therac-25 Accidents," *Computer*, July 1993, pp. 18-41.
3. M. Herlihy and V. Luchangco, "Distributed Computing and the Multicore Revolution," *ACM SIGACT News*, Mar. 2008, pp. 62-72.
4. K. Asanovic et al., "A View of the Parallel Computing Landscape," *Comm. ACM*, Oct. 2009, pp. 56-67.
5. Y. Wang et al., "Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs," *Proc. 8th Symp. Operating Systems Design and Implementation* (OSDI 08), Usenix, 2008, pp. 281-294.
6. Y. Wang et al., "The Theory of Deadlock Avoidance via Discrete Control," *Proc. 36th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages* (POPL 09), ACM Press, 2009, pp. 252-263.
7. C.G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 2nd ed., Springer, 2007.
8. Y. Wang, T. Kelly, and S. Lafortune, "Discrete Control for Safe Execution of IT Automation Workflows," *Proc. 2nd ACM SIGOPS/European Conf. Computer Systems* (EuroSys 07), ACM Press, 2007, pp. 305-314.
9. R.C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys*, Sept. 1972, pp. 179-196.
10. Y. Nir-Buchbinder, R. Tzoref, and S. Ur, "Deadlocks: From Exhibiting to Healing," *Runtime Verification*, LNCS 5289, Springer, 2008, pp. 104-118.
11. M.V. Iordache and P.J. Antsaklis, *Supervisory Control of Concurrent Systems: A Petri Net Structural Approach*, Birkhäuser, 2006.
12. M. Musuvathi et al., "Finding and Reproducing Heisenbugs in Concurrent Programs," *Proc. 8th Symp. Operating Systems Design and Implementation* (OSDI 08), Usenix, 2008, pp. 267-280.

***Terence Kelly*** *is a senior researcher in the Exascale Computing Lab at Hewlett-Packard Laboratories. His research applies discrete control theory to failure avoidance/elimination in computing systems. Kelly received a PhD in computer science from the University of Michigan. He is a senior member of the IEEE and the ACM. Contact him at terence.p.kelly@hp.com.*

***Yin Wang*** *is a researcher at Hewlett-Packard Laboratories. His research interests include control and diagnosis of discrete event systems, modeled by automata or Petri nets, and their applications to computer systems. Wang received a PhD in electrical engineering from the University of Michigan. He is a member of the IEEE and the ACM. Contact him at yin.wang@hp.com.*

***Stéphane Lafortune*** *is a professor in the Electrical Engineering and Computer Science Department at the University of Michigan where he leads the Discrete Event Systems Group (UMDES: www.eecs.umich.edu/umdes/) in the Systems Laboratory. His research interests are primarily in discrete event systems, including modeling, analysis, supervisory control, optimal control, and diagnosis. Lafortune received a PhD in electrical engineering from the University of California, Berkeley. He is a Fellow of the IEEE. Contact him at stephane@eecs.umich.edu.*

***Scott Mahlke*** *is an associate professor in the Electrical Engineering and Computer Science Department at the University of Michigan, where he leads the Compilers Creating Custom Processors group (http://cccp.eecs.umich.edu). The CCCP group delivers technologies in the areas of compilers for multicore processors, application-specific processors for mobile computing, and reliable system design. Mahlke received a PhD in electrical engineering from the University of Illinois at Urbana-Champaign. He is a member of the IEEE Computer Society and the ACM. Contact him at mahlke@umich.edu.*