The Theory of Deadlock Avoidance via Discrete Control *

Yin Wang Stéphane Lafortune
University of Michigan
{yinw,stephane}@eecs.umich.edu

Terence Kelly
Hewlett-Packard Labs
terence.p.kelly@hp.com

Manjunath Kudlur Scott Mahlke
University of Michigan
{kvman,mahlke}@umich.edu

Abstract

Deadlock in multithreaded programs is an increasingly important problem as ubiquitous multicore architectures force parallelization upon an ever wider range of software. This paper presents a theoretical foundation for dynamic deadlock avoidance in concurrent programs that employ conventional mutual exclusion and synchronization primitives (e.g., multithreaded C/Pthreads programs). Beginning with control flow graphs extracted from program source code, we construct a formal model of the program and then apply Discrete Control Theory to automatically synthesize deadlock-avoidance control logic that is implemented by program instrumentation. At run time, the control logic avoids deadlocks by postponing lock acquisitions. Discrete Control Theory guarantees that the program instrumented with our synthesized control logic cannot deadlock. Our method furthermore guarantees that the control logic is *maximally permissive*: it postpones lock acquisitions only when necessary to prevent deadlocks, and therefore permits maximal runtime concurrency. Our prototype for C/Pthreads scales to real software including Apache, OpenLDAP, and two kinds of benchmarks, automatically avoiding both injected and naturally occurring deadlocks while imposing modest runtime overheads.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

General Terms Algorithms, Languages, Theory, Verification

Keywords Dynamic Deadlock Avoidance, Discrete Control Theory, Concurrent Programming, Parallel Programming, Multithreaded Programming, Multicore Processors

1. Introduction

The multicore revolution in computer hardware is precipitating a crisis in computer software by compelling performance-conscious developers to parallelize an ever wider range of applications, typically via multithreading. Multithreading is fundamentally more difficult than serial programming because reasoning about concurrent or interleaved execution is difficult for human programmers, and unforeseen execution sequences can include data races. Programmers can prevent races by protecting shared data with mutual exclusion locks, but misuse of mutexes can cause deadlock. This creates yet another cognitive burden for programmers: Lock-based software modules are not composable, and deadlock freedom is a *global* program property that is difficult to reason about and enforce. These considerations motivate recent interest in mutex alternatives such as atomic sections, which guarantee atomic and isolated execution and which may be implemented using transactional memory (Larus and Rajwar 2007) or conventional locks (McCloskey et al. 2006). Major attractions of atomic sections include deadlock-freedom and composability.

This paper considers a different approach to restoring the composability that locks destroy and relieving the programmer of the obligation to reason about global deadlock freedom. We show how to automatically eliminate deadlocks in conventional lock-based multithreaded programs. Our strategy is to avoid deadlock through a combination of offline static analysis and runtime execution control: We first generate the control flow graph of a program, then enhance it and translate it into a formal model that captures salient features of possible program behaviors. Next, we employ analyses from *Discrete Control Theory* to identify potential deadlocks in the model. Finally, we use other Discrete Control Theory techniques to synthesize *feedback control logic* that provably avoids these deadlocks at runtime. The control logic is implemented by program instrumentation and lock function wrappers that postpone lock acquisitions as necessary to avoid deadlocks.

Discrete Control Theory (DCT) is a branch of control theory that considers systems with discrete state spaces and event driven dynamics (Cassandras and Lafortune 2007). The analysis and control synthesis techniques of DCT are model-based; finite automata and Petri nets are the two most popular modeling formalisms. Petri nets date back to the 1960's (Petri 1962) and are widely used to model nondeterministic concurrent systems. DCT originated with the seminal work of (Ramadge and Wonham 1987) on supervisory control of systems modeled by finite automata. A large body of theory has also been developed for controlling systems modeled by Petri nets (Holloway et al. 1997; Reveliotis 2005; Iordache and Antsaklis 2006). While classical control theory, which considers systems modeled by differential or difference equations, has been successfully applied to computer systems (Hellerstein et al. 2004), the application of DCT to computer systems problems is relatively recent (Wang et al. 2007). Generally speaking, classical control theory is better suited to problems where the specifications are quantitative in nature (e.g., throughput, delay, etc.). DCT is appropriate for problems with *qualitative* specifications, e.g., avoidance of undesirable system states; such specifications cannot be handled by classical control methods. The overall feedback control paradigm, however, is the same in DCT as in classical control: feedback control logic is automatically designed such that the *closed-loop* system, i.e., the original given system under the control of the feedback control logic, satisfies the given specifications.

Our work uses the Petri net modeling formalism because Petri nets allow for a compact representation of system dynamics that avoids explicit state enumeration. Petri nets can furthermore conveniently express the nondeterminism and concurrency of multithreaded programs. Most importantly, DCT control logic synthesis techniques for Petri nets are well suited to the problem of deadlock avoidance and these techniques facilitate concurrent control implementations that do not create global performance bottlenecks. Our Petri net models of multithreaded programs have special properties that allow us to customize a known control method for Petri nets

* The research of Wang, Kudlur, Lafortune, and Mahlke is supported in part by NSF grants ECCS-0624821, CCF-0819882, and CNS-0615261, and by an HP Labs Open Innovation award.

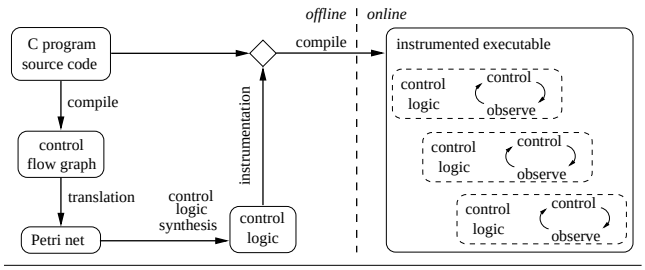


Figure 1. Program control architecture.

to our special subclass to achieve deadlock freedom and *maximally permissive* control. In the context of our problem, maximal permissiveness means that the control logic we synthesize postpones lock acquisitions only when necessary to avert deadlock in a worst-case future of the program's execution. With proper program modeling and control specification, maximal permissiveness maximizes runtime concurrency, subject to the deadlock-freedom requirement.

The main contribution of this paper is a detailed description of our customized control synthesis algorithm for Petri nets that model multithreaded programs. We also formally characterize the properties of programs to which our method has been applied, specifically, deadlock freedom and maximal permissiveness, all ensured by our methodology. For completeness, we briefly summarize our prototype implementation and experimental results involving randomly generated programs and real software; full details are available in a preliminary publication (Wang et al. 2008b) and a separate paper devoted to the prototype implementation and empirical evaluation (Wang et al. 2008a).

The remainder of this paper is organized as follows: Section 2 presents an overview of our approach and its characteristics. Section 3 presents our main results on the automatic synthesis of control logic for deadlock avoidance, and Section 4 describes several extensions. Sections 5 and 6 summarize our prototype implementation and experimental evaluations of its correctness, performance, and usability. Section 7 surveys related work, and Section 8 concludes.

2. Overview

Figure 1 illustrates the architecture of our approach, which proceeds in the following high-level steps:

1. Extract per-function Control Flow Graphs (CFGs) from program source code. We enhance the CFGs to facilitate deadlock analysis by including information about lock variable declaration and access, and lock-related functions and their parameters.
2. Translate the enhanced CFGs into a Petri net model of the whole program. The model includes locking and synchronization operations and captures realistic patterns such as dynamic lock selection through pointers. The model is constructed in such a way that deadlocks in the original program correspond to structural features in the Petri net.
3. Synthesize control logic for deadlock avoidance. Based on the special properties of our Petri net subclass, we customize a known Petri net control synthesis algorithm for this step. The output of this step is the original Petri net model augmented with additional features that guarantee deadlock avoidance in the original program.
4. Instrument the program to incorporate the control logic. This instrumentation ensures that the real program's runtime behavior conforms to that of the augmented model that was generated in the previous step, thus ensuring that the program cannot deadlock. Instrumentation includes code to update control state and wrappers for lock acquisition functions; the latter avoid deadlocks by postponing lock acquisitions at runtime.

Our approach decomposes the overall deadlock avoidance problem into pieces that play to the respective strengths of existing compiler and Discrete Control techniques. Step 1 leverages standard compiler techniques, and Step 2 exploits powerful DCT results that equate *behavioral* features of discrete-event dynamical systems (e.g., deadlock) with *structural* features of Petri net models of such systems. These correspondences are crucial to the computational efficiency of our analyses. The control logic synthesis algorithm we use in Step 3 is called *Supervision Based on Place Invariants* (SBPI) and is the subject of a large body of theory (Iordache and Antsaklis 2006). To avoid deadlocks, SBPI augments the original Petri net model with features that constrain its dynamic behavior. The instrumentation of Step 4 can embed these features, which implement deadlock-avoidance control, into the original program using primitives supplied by standard concurrent programming packages (e.g., the mutexes and condition variables provided by the POSIX threads library). The control logic embedded in Step 4 is furthermore highly concurrent because it is decentralized throughout the program; it is *not* protected by a "big global lock" and therefore does not introduce a global performance bottleneck.

Our approach brings numerous benefits. As shown in the remainder of this paper, it eliminates deadlocks from the given program without introducing new deadlocks or global performance bottlenecks. It "does no harm," except perhaps to performance, because it intervenes in program execution only by temporarily postponing lock acquisitions; it neither adds new behaviors nor disables functionality present in the original program. (If deadlock avoidance is impossible for a given program, our method issues a warning explaining the problem and terminates in Step 3.)

Discrete Control Theory provides a unified formal framework for reasoning about a wide range of program behaviors (branching, looping, thread forks/joins) and synchronization primitives (mutexes, reader-writer locks, condition variables) that might otherwise require special-case treatment. Because DCT is model-based, the modeling of Step 2 is a key step in our methodology. Once modeling is done properly, the properties of the solution follow directly from results in DCT. The DCT control synthesis techniques that we employ guarantee maximally permissive control with respect to the program model, i.e., the control logic postpones lock acquisitions only when provably necessary to avoid deadlock. This property implies that, given a good program model, our approach permits maximum concurrency at run time.

The most computationally expensive operations in our approach are performed offline (Step 3), which greatly reduces the runtime overhead of control decisions. In essence, DCT control logic synthesis performs a deep whole-program analysis and compactly encodes "prepackaged decisions" that allow the runtime control logic to adjudicate lock acquisition requests quickly, while taking into account both current program state and worst-case future execution possibilities. The net result is low runtime performance overhead.

Like the atomic-sections paradigm that is the subject of much recent research, our approach ensures that independently developed software modules compose correctly without deadlocks. However our methods are compatible with existing code, programmers, libraries, tools, language implementations, and conventional lock-based programming paradigms. The latter is particularly important because lock-based code currently achieves substantially better performance than equivalent atomic-sections-based code in some situations. For example, Section 6 shows that lock-based code can exploit available physical resources more fully than atomic-based equivalents if critical regions contain I/O.

Our approach assumes full responsibility for deadlock in programs that it deems *controllable*, i.e., programs that admit deadlock avoidance control policies. However, there can be a performance tradeoff. Our approach allows a programmer to focus on common-case program logic and write straightforward code without fear of deadlock, but it remains the programmer's responsibility to use locks in a way that makes good performance possible.

3. Control Synthesis: Main Results

This section presents our main results on control logic synthesis. Throughout this section we illustrate our method using the dining philosophers program shown in Figure 2, where the main thread creates two philosopher threads that each grab two forks in a different order. The program deadlocks if each philosopher has grabbed one fork and is waiting for the other.

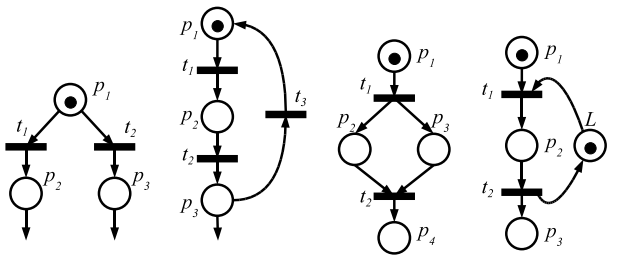
```
void * philosopher(void * arg) {
    if (RAND_MAX/2 > random()) {     /* grab A first */
        pthread_mutex_lock(&forkA);
        pthread_mutex_lock(&forkB);
    }
    else {                           /* grab B first */
        pthread_mutex_lock(&forkB);
        pthread_mutex_lock(&forkA);
    }
    eat();
    pthread_mutex_unlock(&forkA);
    pthread_mutex_unlock(&forkB);
}

int main(int argc, char *argv[]) {
    pthread_create(&p1, NULL, philosopher, NULL);
    pthread_create(&p2, NULL, philosopher, NULL);
}
```

Figure 2. Dining philosophers program with two philosophers

3.1 Petri Net Preliminaries

For the sake of completeness, we present a brief primer on Petri nets; see (Murata 1989) for a detailed discussion. Petri nets are bipartite directed graphs with two types of nodes: circles represent *places* and solid bars represent *transitions*. *Tokens* in places are shown as dots. The *marking* (state) of the Petri net is the number of tokens in each place. Transitions model events in the system that change the marking. Figure 3 shows how to model common patterns of program control flow using Petri nets. The arcs connecting places *to* a given transition represent the pre-conditions that are necessary for the event associated with that transition to occur. The arcs connecting places *from* a given transition represent the outcome of the event. For instance, transition t_1 in Figure 3(a) can occur only if its input place p_1 contains at least one token; in this case, we say that t_1 is *enabled*. Similarly, t_2 is enabled in this simple Petri net that models an `if/else` branch in a program. If transition t_1 occurs (or *fires* in Petri net terminology), then it "consumes" one token from p_1, and "produces" one token in its output place p_2. 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. Petri nets may model loops, as in Figure 3(b), where the firing of t_3 initiates another iteration of the loop. If two or more transitions are both enabled such that exactly one may fire, as with t_1 and t_2 in Figure 3(a), the Petri net does not specify *which* will fire, nor *when* a firing will occur. Petri nets are therefore well suited to modeling nondeterminism due to branching and processor scheduling in multithreaded programs.

Concurrency is also easily modeled in Petri nets. For example, in Figure 3(c), one can think of transition t_1 as the `thread_create` operation and t_2 as the `thread_join` operation. Firing t_1 generates two tokens representing the original and the child thread, in places p_2 and p_3, respectively. After t_2, the child thread joins the original thread in place p_4. In Figure 3(d), place L models a mutex lock, while t_1 and t_2 model lock acquisition and release operations, respectively. The token inside L represents the lock, whereas the token in p_1 represents the thread. After t_1 fires, a single token occupies p_2 and L is empty, meaning that the lock is not available and t_1 is disabled. If a new thread arrives at p_1 (via an arc not shown in Figure 3(d)), it cannot proceed. Firing t_2 returns a token to L, which means that the lock is again available.



(a) branch (b) loop (c) fork/join (d) lock/unlock

Figure 3. Basic Petri net models

Formally, we have the following definition.

Definition 1. *A Petri net $N = (P, T, A, W, M_0)$ is a bipartite graph, where $P = \{p_1, p_2, ..., p_n\}$ is the set of places, $T = \{t_1, t_2, ..., t_m\}$ is the set of transitions, $A \subseteq (P \times T) \cup (T \times P)$ is the set of arcs, $W : A \to \{0, 1, 2, ...\}$ is the arc weight function, and for each $p \in P$, $M_0(p)$ is the initial number of tokens in place p.*

The notation $\bullet p$ denotes the set of input transitions of place p: $\bullet p = \{t | (t, p) \in A\}$. Similarly, $p\bullet$ denotes the set of output transitions of p. The sets of input and output places of a transition t are similarly defined by $\bullet t$ and $t\bullet$. For example in Figure 3(a), $\bullet p_1 = \emptyset$, $p_1\bullet = \{t_1, t_2\}$, and $\bullet t_1 = \{p_1\}$. This notation is extended to sets of places or transitions in a natural way. A transition t in a Petri net is enabled if every input place p in $\bullet t$ has at least $W(p, t)$ tokens in it. When an enabled transition t fires, it removes $W(p, t)$ tokens from every input place p of t, and adds $W(t, p)$ tokens to every output place p in $t\bullet$. By convention, $W(p, t) = 0$ when there is no arc from place p to transition t. Throughout this section, our models of multithreaded programs have unit arc weights, i.e., $W(a) = 1, \forall a \in A$. Such Petri nets are called *ordinary* in the literature.

We build the incidence matrix D of a Petri net as follows: $D \in \mathbb{Z}^{n \times m}$ where $D_{ij} = W(t_j, p_i) - W(p_i, t_j)$ represents the net change in the number of tokens in place p_i when transition t_j fires. If the net has no self-loop, i.e., at least one of $W(p_i, t_j)$ or $W(t_j, p_i)$ is equal to zero, then: (i) a negative D_{ij} means there is an arc of weight $-D_{ij}$ from p_i to t_j; and (ii) a positive D_{ij} means there is an arc of weight D_{ij} from t_j to p_i. The incidence matrix of the Petri net in Figure 3(a) is

$$D = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \end{array} \begin{array}{c} \begin{array}{cc} t_1 & t_2 \end{array} \\ \begin{bmatrix} -1 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \end{array}$$

The marking (i.e., state) of a Petri net, which records the number of tokens in each place, is represented as a column vector M of dimension $n \times 1$ with non-negative integer entries, given a fixed order for the set of places: $M = \left[M(p_1) \cdots M(p_n)\right]^T$, where T

denotes transpose. As defined above, M_0 is the initial marking. For example, the marking of the Petri net in Figure 3(a) is $\begin{bmatrix}1 & 0 & 0\end{bmatrix}^T$; this is the number of tokens in the three places ordered as: p_1, p_2, p_3. If t_1 fires, the marking becomes $\begin{bmatrix}0 & 1 & 0\end{bmatrix}^T$.

The reachable state space of a Petri net is the set of all markings reachable by transition firing sequences starting from M_0. This state space may be infinite if one or more places may contain an unbounded number of tokens. Fortunately we need not consider the reachable state space because we employ techniques from DCT that operate directly upon the relatively compact Petri net rather than its potentially vast state space.

3.2 Modeling Multithreaded Programs

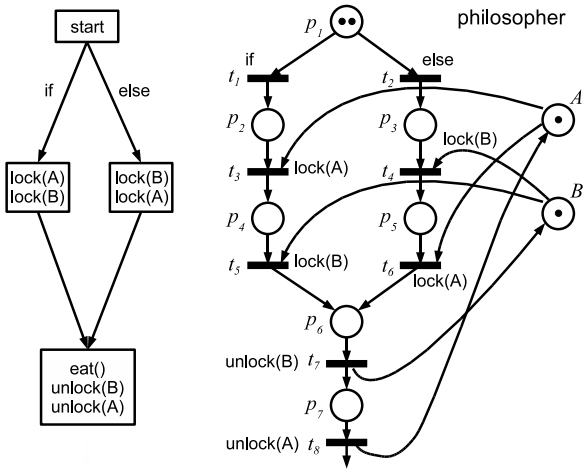
Our modeling methodology begins with the set of per-function CFGs extracted from the target C program. We augment these CFGs such that in addition to basic blocks and flow information, lock variables and lock functions are also included. Each (augmented) CFG is a directed graph. To obtain a Petri net we first create a place for each node (basic block) of this graph. For each arc connecting two nodes in the graph, we create a transition and two arcs in the Petri net: one from the place corresponding to the originating node to the transition, and one from the transition to the place corresponding to the destination node. Overall, a basic block-transition-basic block chain in the CFG is converted into a place-arc-transition-arc-place chain in the corresponding model; see for example Figures 3(a) and 3(b).

The execution of a thread is modeled as a token flowing through the Petri net. In order to model lock acquisition/release functions appropriately, we split a basic block that contains multiple lock functions into a sequence of blocks such that each block has at most one lock function associated with it. Therefore, after model translation, each lock operation is represented by a single transition in the Petri net. Similarly, a basic block containing multiple user-defined functions is split such that each function call is represented by one place in the Petri net. With this split, we can substitute the function call place with the Petri net model of the called function. A new copy of the called function's Petri net is substituted at each distinct call site. In other words, we build inlined Petri net models.

Functions that do not invoke lock operations need not be considered in the modeling phase. We further prune fractions of the inlined Petri net model that are irrelevant to deadlock analysis. Finally, if the program uses different sets of locks in different modules, we decompose the Petri net and apply the control synthesis algorithm to each subnet separately. These simple preprocessing techniques are highly effective in shrinking real program models to a manageable size. Additional details are available in (Wang et al. 2008b).

If recursions occur in the pruned Petri net, we handle them in a similar way as we model program loops. Each recursion is substituted by exactly one copy of every function involved in the recursion. Subsequent recursive calls inside these functions are linked back to themselves. Control synthesis treats these recursion loops as normal loops. Online instrumentation, however, must track recursive calls and know when the program leaves the recursion by augmenting parameters of functions involved in the recursion.

Modeling multithreaded synchronization primitives using Petri nets has been studied previously in the literature; see (Kavi et al. 2002). We apply these known techniques to model locking primitives. For example, thread creation and join are modeled as illustrated in Figure 3(c). To model mutex locks, we add a new place for each lock, called a *lock place*, with one initial token to represent lock availability. If a transition represents a lock acquisition call, we add arcs from the lock place to the transition. If a transition represents a lock release call, we add arcs from the transition to the lock place; see Figure 3(d).



(a) CFG (b) Translated Petri Net

Figure 4. Modeling the Dining Philosopher Example

With these modeling techniques, we are able to build a complete Petri net model of a given concurrent program. Figure 4(a) is the control flow graph of function `philosopher` in the example in Figure 2. There are four basic blocks, representing start, `if` branch, `else` branch and the rest of the function. Figure 4(b) is the translated Petri net model of the CFG. The structure is similar to the CFG, with lock places added. Basic blocks containing multiple lock functions are split into sequences of places and transitions such that each lock function is represented by a single transition in the net, as annotated.

3.3 Controlling Petri Nets by Place Invariants

The purpose of control logic synthesis for Petri nets is to avoid "undesirable" or "illegal" markings. Appropriate formal specifications that characterize these undesirable markings are needed. A common form of specification is the linear inequality

$$l^T M \geq b \tag{1}$$

where l is a weight (column) vector, M is the marking, and b is a scalar; b and the entries of l are integers. Equation (1) states that the weighted sum of the number of tokens in each place should be greater than or equal to a constant. We will show in Section 3.4 how to attack deadlock avoidance using such specifications.

Markings violating the linear inequality in Equation (1) must be avoided by control as they are illegal; all other markings are permitted. It turns out that this condition can be achieved by adding a new *control place* to the net with arcs connecting to transitions in the net. The control place blocks (disables) its output transitions when it has an insufficient token count. This method is formally stated as follows.

Theorem 1. *(Iordache and Antsaklis 2006) If a Petri net $N = (P, T, A, W, M_0)$ with incidence matrix D satisfies*

$$b - l^T M_0 \leq 0 \tag{2}$$

then we can add a control place c that enforces Equation (1). Let $D_c : T \to \mathbb{Z}$ denote the weight vector of arcs connecting c with the transitions in the net; D_c is obtained by

$$D_c = l^T D \tag{3}$$

The initial number of tokens in c is

$$M_0(c) = l^T M_0 - b \geq 0 \quad (4)$$

The control place enforces maximally permissive control logic, i.e., the only reachable markings of the original net N that it avoids are those violating Equation (1).

The above control technique is called Supervision Based on Place Invariants (SBPI). It maintains the condition in Equation (1) by building a *place invariant* with the newly added control place. The place invariant guarantees that for any marking M in N's set of reachable markings, $l^T M - M(c) = b$, where $M(c)$ is the number of tokens in the control place c. Since $M(c)$ is non-negative, the inequality in Equation (1) is satisfied. Equation (2) states that Equation (1) must be satisfied for M_0, otherwise there is no solution.

Equation (3) shows that SBPI operates on the net structure (incidence matrix) directly without the need to enumerate or explore the set of reachable markings of the net; this greatly reduces the complexity of the analysis. Equally importantly, SBPI guarantees that the controlled Petri net is maximally permissive, i.e., a transition is not disabled (by lack of tokens in control place c) unless its firing leads to a marking where the linear inequality is violated (Iordache and Antsaklis 2006). In other words, it enforces "just enough control" to avoid all illegal markings. SBPI is the basis for our deadlock avoidance control synthesis algorithm. Specifically, SBPI eliminates potential deadlocks that we discover via *siphon analysis*.

3.4 Deadlocks and Petri Net Siphons

To achieve the objective of deadlock avoidance in a concurrent program using SBPI, we need to express deadlock freedom using linear inequality specifications. This is done by means of siphon analysis.

Definition 2. *A siphon is a set S of places such that $\bullet S \subseteq S\bullet$.*

Intuitively, since the input transition set is a subset of the output transition set, if a siphon S becomes empty, every output transition in $S\bullet$ is disabled and therefore no input transition can fire. As a result, the set of places S will remain empty forever and the transitions in $S\bullet$ will never fire again. For example, the set of places $\{A, B, p_6, p_7\}$, marked by crosses in Figure 5, is a siphon. It becomes empty when each philosopher acquires one fork and waits for another. In this situation, no place in the siphon ever gains any token; indeed, we have a deadlock.

Our Petri net models have special properties that allow us to identify deadlocks in the original program by identifying siphons in the corresponding Petri net model. Recall from Section 3.1 that our Petri nets are *ordinary* (all arcs have unit weight). Let N_G denote the Petri net model of a concurrent program. Let the part of N_G that corresponds to the control flow graph be denoted by N_{CFG}; in other words, lock places are excluded in N_{CFG}. By construction, all the transitions in N_{CFG} have exactly one input place and exactly one output place. Clearly, the only siphon in N_{CFG} is the entire set of places, which cannot become empty during the execution of the program. Therefore, any siphon in N_G *must* include lock places. We build from the following known result in the literature.

Theorem 2. *(Reisig 1985) A totally deadlocked ordinary Petri net contains at least one empty siphon.*

In this theorem, "total deadlock" refers to a Petri net state in which no transition is enabled. In our analysis, we are interested in circular-mutex-wait deadlocks, not total deadlocks. However, in our class of Petri net models, the presence of a circular-mutex-wait deadlock implies that N_G contains an empty siphon. To see this, consider a Petri net state that models a program with a circular-mutex-wait deadlock. Consider only the subnet involved in the circular-mutex-wait deadlock, and only the tokens representing the deadlocked threads. This subnet has no enabled transition. According to Theorem 2, it contains at least one empty siphon. This siphon is also empty in the original Petri net state.

Consider next the reverse implication of Theorem 2: what if the net contains an empty siphon in some reachable marking? An empty siphon cannot gain any token back and therefore the corresponding transitions are permanently disabled. Since an empty siphon in our Petri net model must include lock places, these lock places remain empty as well, meaning that the threads holding these locks will never release them. This could be due to a thread that simply acquires a lock and never releases it. We handle the preceding scenario separately in our control logic synthesis. For the purpose of the present analysis, we assume that threads eventually release all the locks that they acquire. Under this assumption, empty siphons that include lock places correspond to circular-mutex-wait deadlocks. Combining this result with Theorem 2, we have the following important result:

Theorem 3. *The problem of deadlock avoidance in a concurrent program is equivalent to the problem of avoidance of empty siphons in its ordinary Petri net model N_G.*

Theorem 3 establishes a relationship between deadlock, which is a behavioral property, and siphons, which are structural features. The latter can be identified directly from the incidence matrix without exploring the set of reachable markings (Boer and Murata 1994).

In some cases, a siphon cannot become empty in any reachable marking. For example, places L and p_2 in Figure 3(d) form a siphon. Once empty, they remain empty forever. But with an initial token in L, these two places will never become empty. In fact, a token will always occupy one of the two places in any reachable marking. When synthesizing deadlock avoidance control logic, it is important to distinguish siphons that may become empty from those that cannot. Control need only be synthesized to address the former; the latter may safely be ignored.

3.5 Control Logic Synthesis for Deadlock Avoidance

Given Theorem 3, our objective is to control the Petri net model of a concurrent program in a manner that guarantees that none of its siphons ever becomes empty. For this purpose, it is sufficient to consider only *minimal* siphons, i.e., those siphons that do not contain other siphons. This goal is translated into specifications of the form in Equation (1) as follows: The sum of the number of tokens in each minimal siphon is never less than one in any reachable marking. SBPI adds a control place to the net that maintains Equation (1) for each minimal siphon. For example, consider again the Petri net in Figure 5 (without the place and arcs that are dashed); to prevent the minimal siphon $\{A, B, p_6, p_7\}$ in this net from being emptied, we define

$$l^T = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}, \; b = 1 \quad (5)$$

where the order of the places for vectors M and l is: $p_1, \ldots p_7, A, B$. In this case, Equation (1) means that the total number of tokens in places p_6, p_7, A, and B should not be less than 1. The incidence matrix of the net in Figure 5 is (with transitions ordered according

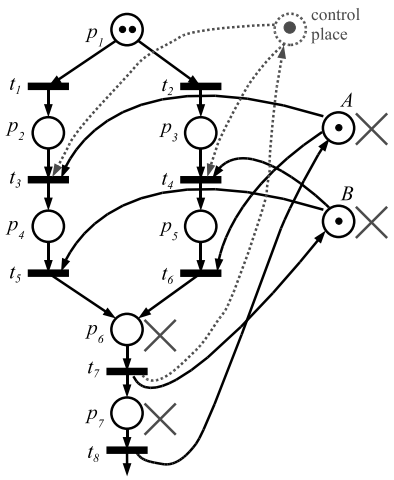


Figure 5. Controlled Dining Philosophers Example

to their subscripts):

$$D = \begin{bmatrix} -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & -1 & 0 & 1 & 0 \end{bmatrix} \quad (6)$$

Applying Equations (3 and 4), we have

$$D_c = \begin{bmatrix} 0 & 0 & -1 & -1 & 0 & 0 & 1 & 0 \end{bmatrix}, \; M_0(c) = 1 \quad (7)$$

which means that the control place has output arcs to transitions t_3 and t_4, and an input arc from transition t_7; all these arcs have weight one. The control place has one initial token. This control place and its associated arcs are shown with dashed lines in Figure 5. The Petri net including the control place is called the *augmented net*. From Theorem 1, we know that the place invariant $l^T M - M(c) = 1$ always holds for any reachable marking M, and therefore the siphon is never empty.

It would be wrong to conclude from this simple example that our approach simply "coarsens locking" or "adds meta-locks." This is a reasonable interpretation of the control place in Figure 5, but in general the control logic that our procedure synthesizes admits no such simple characterization, as we shall see when we consider how our approach handles real-world deadlock bugs in Apache and OpenLDAP.

A difficulty that arises in the preceding methodology is that the newly added control places (one per minimal siphon in the net) could introduce new siphons in the augmented net. Intuitively, SBPI avoids deadlocks at the last lock acquisition step, i.e., the lock acquisition that completes the circular wait. Sometimes this is too late. While the control place blocks the transition immediately leading to the deadlock, there may be no other transition the program can take. This is a deadlock introduced by the control place. Fortunately, this deadlock implies the existence of a new siphon in the augmented Petri net that includes the control place. Therefore, we can apply SBPI again and iterate until both the deadlocks in the original program and deadlocks introduced by control logic are avoided. The iterative procedure is defined in Figure 6. Step 4 refers to "redundant" control places. Those are control places that achieve

Input Petri net N_G that models the program

Output Augmented N_G with control places added

Step 1 Let R be the set of places representing mutex locks

Step 2 Find all minimal siphons in N_G that include at least one place in R and can become empty; if no siphon found, goto **End**

Step 3 Add a control place for every siphon found in **Step 2**

Step 4 Remove redundant control places added in **Step 3**; let R be the set of control places remaining; goto **Step 2**

End Output N_G with all control places added

Figure 6. Control Synthesis Algorithm

redundant control objectives as compared with control places added in earlier iterations. Details on how we check whether a siphon can become empty and how we remove redundant control places are described in (Wang et al. 2008b).

Combining the results of Sections 3.3 and 3.4 with the procedure in Figure 6, we have the following corollary:

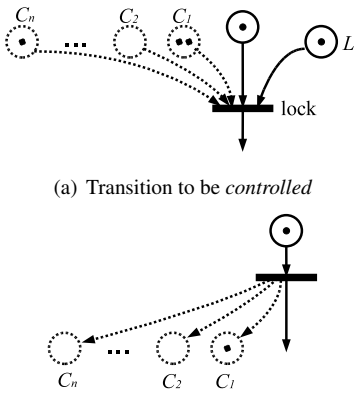
Corollary 4. *After the iterative procedure of Figure 6, we know that the augmented Petri net with control places has no reachable empty siphon. If the arcs connecting the added control places to the transitions of the original net all have unit weight, then by Theorem 3 we conclude that the augmented net models the deadlock-free execution of the original multithreaded program. Moreover, by Theorem 1, the behavior of the augmented net is the maximally-permissive deadlock-free sub-behavior of the original net.*

If a newly added control place has a non-unit-weight arc to a transition of the original net, then deadlock in the multithreaded program does not necessarily imply an empty siphon in the net as Theorem 2 is not directly applicable. Theorem 2 can be generalized to the case of non-unit arc weights; in this case liveness is not entirely characterized by empty siphons, but rather by the notion of "deadly marked siphons" from (Reveliotis 2005). In this case, further behavioral analysis of the siphons is necessary; details are omitted here. In practice, in our experiments so far with the special Petri net subclass modeling multithreaded programs, our iterative SBPI algorithm has converged quickly without introducing non-unit arc weights. This has been observed on both randomly generated programs and real-world software including Apache and OpenLDAP.

3.6 Control Logic Implementation

The output of the control logic synthesis algorithm is an augmented version of the input Petri net, to which have been added control places with incoming and outgoing arcs to transitions in the original Petri net. An outgoing arc from a control place will effectively delay the target transition until a token is available in the control place; the token is consumed when the transition fires. An incoming arc from a transition to a control place replenishes the control place with a token when the transition fires. Outgoing arcs from control places always link to lock acquisition calls, which are the transitions that the runtime control logic *controls*. Incoming arcs originate at transitions corresponding to lock release calls or branches, which are the transitions the control logic must *observe*.

A lock acquisition transition that needs to be controlled has one or more incoming arcs from control places, as illustrated in Figure 7(a), where L is the "real" lock in the program that has to be acquired, and $C_1, C_2, ...C_n$ are control places that link to the transition. A transition that needs to be observed has one or more outgoing arcs to control places, as illustrated in Figure 7(b). For the



(a) Transition to be *controlled*

(b) Transition to be *observed*

Figure 7. The control logic implementation problem

sake of generality, Figures 7(a) and 7(b) show several control places connected to a given transition. In practice, the number of control places connected to a transition is very small, typically only one.

As Figure 7 suggests, control places resemble lock places, and therefore can be implemented with primitives supplied by standard multithreading libraries, e.g., libpthread.

Controlled Transitions For a lock acquisition transition that needs to be controlled, the control logic must check the token availability of all input places to that transition. These include the lock place in the original net model as well as all the control places that were added by the procedure in Figure 6, as depicted in Figure 7(a). We replace the native lock-acquisition function with our wrapper to implement the required test for these transitions. The wrapper internally uses two-phase locking with global ordering on the set of control places to obtain the necessary tokens. If a control place does not have enough tokens, the wrapper returns all tokens it has obtained from other control places, and waits on a condition variable that implements this control place; this effectively delays the calling thread. Once the token becomes available, the wrapper starts over again to acquire tokens from all control places.

Observed Transitions For a transition that needs to be observed, i.e., with outgoing arcs to control places as shown in Figure 7(b), we insert a control logic update function that increases the token count and signals the condition variables of the corresponding control places.

Figure 8 is the lock wrapper implementation for Figure 7(a) using the Pthread library. Each control place C_i is implemented as a three-tuple {`n[i]`, `l[i]`, `c[i]`}, where `n[i]` is an integer representing the number of tokens in C_i, `l[i]` is the mutex lock protecting `n[i]`, and `c[i]` is the condition variable used when a thread is waiting for token in the control place.

The following theorem establishes that the above-described implementation of control places does not introduce livelock into the instrumentation: Two or more threads cannot become permanently "stuck" executing the outer loop in the wrapper function code of Figure 8.

Theorem 5. *With the implementation of Figure 8 and global ordering of $l[i]$, if a set of threads competes for tokens in control places, at least one thread will acquire all required tokens from the control places and succeed in firing the corresponding transition.*

Proof. Assume $TD = \{T_1, T_2, ...T_u\}$ is the set of threads competing for tokens in the set of control places $CP = \{C_1, C_2, ...C_v\}$. Without loss of generality, let us also assume every thread in TD is at the "start" label of the lock wrapper in Figure 8, i.e., no thread has consumed any token in CP yet, and every other thread is either sleeping or waiting on some (real) lock. Then CP must have enough tokens for at least one thread in TD to go through. Otherwise all threads are permanently waiting and this is a deadlock, which is provably avoided by our control synthesis algorithm.

Assume CP has enough tokens for T_1 to get through. If T_1 failed to get a token from a control place, say C_1, some other thread in TD, say T_2, must have acquired the token in C_1 before T_1 attempted to acquire it. If T_2 failed to get the token in a control place, say C_2, there are two cases: (1) C_2 does not have any tokens at all to start with or (2) some other thread in TD has temporarily acquired it first. In the first case, T_2 will sleep and not compete with threads in TD anymore. Furthermore, T_2 will release the token to C_1 and wake up T_1 before it goes into sleep. Then we can repeat the analysis for T_1 all over again. In the second case, the token in C_2 cannot be temporarily acquired by T_1 because of the assumed global ordering on control places. Assuming T_3 has temporarily acquired the token in C_2, we could follow the same analysis performed on T_2. Eventually, either some thread in TD gets all tokens needed or every thread other than T_1 goes to sleep, in which case T_1 will be awakened and obtain all tokens needed. □

```
start:
pthread_mutex_lock(&L);         /* acquire real lock */
for (i=1; i<=n; i++) {
  pthread_mutex_lock(&l[i]));           /* check Ci */
  if (0 < n[i])  {                /* has token in Ci */
    n[i]--;                        /* take one token */
    pthread_mutex_unlock(&l[i]));
  }
  else {                             /* no token in Ci */
    pthread_mutex_unlock(&L);   /* release real lock */
    for (j=i-1; j>=1; j--) { /* replenish all tokens */
      pthread_mutex_lock(&l[j]);
      n[j]++;
      pthread_cond_signal(&c[j]);
      pthread_mutex_unlock(&l[j]);
    }
    pthread_cond_wait(&c[i], &l[i]);   /* wait on Ci */
    pthread_mutex_unlock(&l[i]));
    goto start;          /* start over once signaled */
  }
}
```

Figure 8. Lock wrapper implementation for the example of Figure 7(a). A control place C_i is associated with integer `n[i]` representing the number of tokens in it; lock `l[i]` and condition variable `c[i]` protect `n[i]`. These are global variables in the control logic implementation.

As shown in Figure 8, our current controller implementation does not address the scheduling of threads onto locks; underlying infrastructure (threading library and OS) is responsible for this. Whether our control logic introduces scheduling issues (e.g., priority inversion, starvation) depends on the semantics provided by the underlying infrastructure.

4. Control Synthesis: Extensions

Section 3 presented the main results of our deadlock avoidance methodology for concurrent programs. This section discusses extensions to the basic method and additional topics relevant to our problem domain.

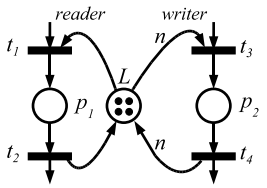


Figure 9. Reader-Writer Lock

4.1 Model Extensions

Our control synthesis algorithm is not limited to circular-mutex-wait deadlocks. Rather, it depends on what is included in the Petri net model of the program. With the rich representation capabilities of Petri net models, it is relatively easy to model other multithreaded synchronization primitives and thereby to automatically address deadlocks involving them. We discuss a few examples.

Semaphores A semaphore is essentially a lock with multiple instances that can be "acquired/released" by different threads repeatedly through `down/up` operations. Therefore, semaphores share the same model as locks except that the initial number of tokens in a semaphore place may exceed one.

Reader-Writer Locks Modeling reader-writer locks is illustrated in Figure 9. The initial number of tokens in the lock place represents the maximum number of readers allowed. A reader can acquire the lock as long as at least one token is available, while a writer must acquire all of the tokens. When the maximum number of readers is not specified by the program, we can use a sufficiently large initial number of tokens, e.g., greater than the number of threads allowed. Note that the right-hand arcs in Figure 9 both have weight n. Theorem 2 presented earlier requires unit arc weights as an assumption. As was mentioned in Section 3.5, Theorem 2 can be generalized to the case of non-unit arc weights. This complicates the procedure of generating the control logic for deadlock avoidance and is not discussed in this paper.

Condition Variables Condition variable models include a place that models the signal variable and transitions that model `wait`, `signal`, and `broadcast` calls. We use a separate place, with no initial token, to represent each signal. The place gains tokens with `signal/broadcast` transitions and loses tokens with `wait` transitions. In addition to the input arc from the signal place, the `wait` transition must represent the fact that the mutex lock is released during wait and reacquired once the signal is available. In addition, a complete model should also include signal loss (when the thread to be awakened is not waiting on the condition variable); see (Kavi et al. 2002) for further discussion.

Condition variables are another major source of deadlocks in multithreaded programs, and it is very difficult to reason about them. However, once condition variables are included in our Petri net models, condition-variable deadlocks can be identified through siphon analysis in the same manner as mutex deadlocks are found.

Figure 10 shows a condition variable deadlock from the Apache bug database (Apache). This deadlock is introduced by a mutex together with condition variables. The listener thread waits on a condition variable while holding the `timeout` mutex. The worker thread acquires then releases `timeout`, then signals the listener. If the listener thread is already waiting before the worker thread acquires `timeout`, the signal is never sent and the two threads deadlock in the calls indicated by comments.

Figure 11 is the Petri net model of the code in Figure 10. For simplicity we show only basic signal operations. Details like the release and reacquisition of locks with the `wait` call are not shown. Places marked by crosses form the siphon corresponding to the

```
listener_thread(...) {
  ...
  apr_thread_mutex_lock(timeout_mutex);
  ...
  rv = apr_thread_mutex_lock(queue_info->idlers_mutex);
  ...
  rv = apr_thread_cond_wait(queue_info->wait_for_idler,
                            queue_info->idlers_mutex); /**/
  ...
  rv = apr_thread_mutex_unlock(queue_info->idlers_mutex);
  ...
  apr_thread_mutex_unlock(timeout_mutex);
  ...
}

worker_thread(...) {
  ...
  apr_thread_mutex_lock(timeout_mutex); /**/
  ...
  apr_thread_mutex_unlock(timeout_mutex);
  ...
  rv = apr_thread_mutex_lock(queue_info->idlers_mutex);
  ...
  rv = apr_thread_cond_signal(queue_info->wait_for_idler);
  ...
  rv = apr_thread_mutex_unlock(queue_info->idlers_mutex);
  ...
}
```

Figure 10. Apache deadlock, bug #42031.

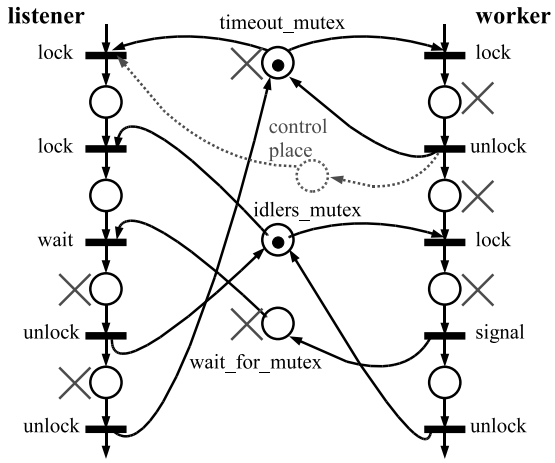


Figure 11. Simplified Petri net model for the Apache deadlock, bug #42031.

deadlock bug. The control place added guarantees that the siphon will never empty. The control place prevents the listener thread from acquiring the `timeout` mutex until the worker thread has released it and is able to signal the listener. This control logic is maximally permissive as it allows the listener thread to proceed after the worker thread releases the `timeout` mutex.

4.2 Partial Controllability and Observability

So far, we have assumed that every transition in the Petri net is *controllable*, i.e., it can be prevented from firing if we append a control place to its set of input places. Therefore, if a transition has an incoming arc from a control place after the control synthesis procedure, the control logic effectively blocks that transition when

```
ldap_pvt_thread_rdwr_wlock(&bdb->bi_cache.c_rwlock);
/* LOCK(A) */
...
ldap_pvt_thread_mutex_lock( &bdb->bi_cache.lru_mutex );
/* LOCK(B) */
...
ldap_pvt_thread_rdwr_wunlock(&bdb->bi_cache.c_rwlock);
/* UNLOCK(A) */
...
if ( bdb->bi_cache.c_cursize>bdb->bi_cache.c_maxsize ) {
  ...
  for (...) {
    ...
    ldap_pvt_thread_rdwr_wlock(&bdb->bi_cache.c_rwlock);
    /* LOCK(A) */
    ...
    ldap_pvt_thread_rdwr_wunlock(&bdb->bi_cache.c_rwlock);
    /* UNLOCK(A) */
    ...
  }
}
...
ldap_pvt_thread_mutex_unlock(&bdb->bi_cache.lru_mutex);
/* UNLOCK(B) */
```

Figure 12. OpenLDAP deadlock, bug #3494.

the control place has an insufficient token count. In our problem, however, not every transition is controllable. For example, transitions representing `if/else` branches or loops are not controllable. We cannot "force" the program to take one branch instead of the other. In our application, the only controllable transitions are those representing lock acquisitions.

Partial controllability refers to the situation where not every transition in the net is controllable. When a control place added by the control synthesis algorithm has *outgoing* arcs to an uncontrollable transition in the net, then the corresponding control logic is not implementable. Synthesizing control logic for a partially controllable net in general requires correctness-preserving *linear constraint transformation* (Iordache and Antsaklis 2006). The transformed constraints guarantee that control places added by SBPI have output arcs to controllable transitions only, while satisfying the original linear inequality specifications. The synthesis of control logic under partial controllability is in general more conservative than under the case of full controllability. A version of maximal permissiveness can still be achieved in this case, in the sense that the control logic should not block any transition unless the execution of that transition can lead to an undesired state *unavoidably*, i.e., through a sequence of uncontrollable transitions.

We illustrate the controllability issue with an actual OpenLDAP bug (OpenLDAP) shown in Figure 12, where clarifying comments are inserted. The deadlock may occur if thread 1 locks A and B, then releases A. Before thread 1 reacquires A, thread 2 executes the same code, which acquires A then B. Assuming full controllability, the control logic would allow thread 2 to enter and acquire lock A, then force thread 1 to jump out of the `for` loop if thread 2 acquires A first. With partial controllability, the control logic immediately forbids other threads from entering once thread 1 is executing the code, *even if lock A is available*. If thread 1 branches over the body of the `if`, or if it leaves the `for` loop, the control logic knows that thread 1 cannot be involved in this deadlock bug and therefore permits other threads to enter.

Another issue to consider in practice is that of partial observability. A transition is not *observable* if we cannot observe its firing. If a synthesized control place has *incoming* arcs from an unobservable transition in the net, then the control logic is not implementable as the control logic does not know when to replenish tokens in the control place. In our application, one could in principle observe every transition by proper instrumentation of the program. However, if source code modifications are not allowed, e.g., only binaries are available, we can still control the program by the technique of library interposition, by intercepting all lock acquisition/release calls. In this case however, the evolution of the program is not fully observable.

Synthesizing control logic for a partially observable Petri net can also be solved by the technique of constraint transformation, as described in (Iordache and Antsaklis 2006). Transformed linear inequality constraints guarantee that control places added have incoming arcs from observable transitions only. The control logic is in general more conservative than the one assuming full observability. In the example of Figure 12, if we can observe only `lock` and `unlock` calls, the control logic must wait until the first thread releases `lru_mutex` at the end before allowing another thread to enter this critical region, which effectively serializes the whole critical region. Assuming full observability, as discussed above, the control logic allows another thread to enter as soon as the first thread jumps out of the `for` loop.

4.3 Current Limitations

Our current prototype implementation does not perform alias or pointer analysis when building the Petri net model of a concurrent program. We represent lock pointer variables by their type names, i.e., the structure type that encloses the primitive lock variable. This approximation is adopted by a few static analysis tools as well (Engler and Ashcraft 2003). It may lead to conservative control logic but does not miss any deadlock unless the program violates type safety conventions by illegal pointer casting. More sophisticated pointer analysis methods, such as the one used in (Cherem et al. 2008), could be incorporated into our framework, thereby resulting in more fine-grained control logic.

Another source of conservatism in our control synthesis phase is the lack of data flow information in our current prototype, which is also shared with static analysis tools. Figure 13 illustrates the "false paths" problem (Engler and Ashcraft 2003). With only control flow information, the control synthesis algorithm does not know that the two conditional branches are paired up if `x` is not modified in between, and therefore it mistakenly concludes that this code might acquire the lock but not release it. The algorithm might respond by adding unnecessary control logic. This example illustrates the fact that our control logic is maximally permissive *with respect to the program model*, as formalized by Theorem 1. More accurate program models, e.g., from data flow analysis, can result in better control and improved performance by reducing instrumentation overhead and by allowing more concurrency.

```
if (x)
  lock(L)
...
if (x)
  unlock(L)
```

Figure 13.

Some deadlocks are simply unavoidable. For example, a thread may repeatedly lock a nonrecursive mutex. In the terminology of DCT, such programs are *uncontrollable*, and SBPI responds to the corresponding Petri net models by emitting negative coefficients where positive ones are expected (e.g., for arc weights and markings). Our prototype implementation issues warnings for uncontrollable deadlocks.

A different kind of uncontrollability problem can arise if our control logic prevents concurrent execution of two program fragments that must run concurrently in order for execution to proceed. This can occur if one fragment enters a blocking call (e.g., a read on a pipe) whose return is contingent upon the other fragment (e.g., a write to the same pipe). To address this problem, we must include in our program model all blocking calls, including those whose return is triggered by phenomena *not* explicitly modeled. (Condition variable signal/broadcast is modeled in our current prototype, but

blocking system calls are not.) It is then straightforward to identify cases where our control logic potentially precludes the return of blocking calls and issue appropriate warnings. If our experience with real software is any guide, such scenarios are rare in practice: we have never observed deadlocks caused by a combination of control logic and, e.g., interprocess communication.

If a program does not merit one or another sort of uncontrollability warning, we guarantee correct execution; our method never silently introduces deadlocks or disables program functionality.

In general, our overall approach is well suited to languages that admit the static modeling, analyses, and control synthesis that we require. Dynamic constructs do not preclude our approach but may lead to conservatism, e.g., we handle function calls through pointers by assuming that the callee may be any function with the appropriate type signature. Annotations help clarify the ambiguity caused by dynamic constructs. Our current C/Pthreads prototype implementation does not model `setjmp()/longjmp()`, exception handling, or signal handling.

5. Implementation

We briefly discuss in this section several issues regarding the implementation of our methodology, which is the subject of a separate paper (Wang et al. 2008a) devoted to our prototype and empirical evaluation.

5.1 Control Flow Graph

We modified the open source compiler OpenIMPACT (OpenImpact) to construct an augmented control flow graph (CFG) for each function in the input program. Lock variables (globally declared, locally declared, or dynamically allocated) are included in the CFG. In addition, functions calls are listed in each basic block, together with their argument name and type information. We recognize the standard Pthread functions automatically. We use programmer annotations to recognize wrapper functions, if such functions are used for the primitive Pthread functions. Basic blocks that contain these wrapper functions are marked and will be handled appropriately during the model translation phase.

As discussed in Section 4.3, patterns like the one illustrated in Figure 13 may lead to spurious deadlock detection. Since our control synthesis algorithm avoids all potential deadlocks, in large programs, numerous false control flow paths could result in very conservative control logic. Appropriate user annotation alleviates this problem. We found that function level annotation is highly effective against false paths. The annotation marks whether a particular lock type is *always* or *never* held upon return from a particular function. We use a variant of lockset analysis (Savage et al. 1997) to identify ambiguous functions automatically. In practice, the set of ambiguous functions is very small and a programmer unfamiliar with the source can annotate a function in a few minutes. For example, the function enclosing the code in Figure 13 could be annotated to indicate that lock L is never held after the function returns. Our experience with real-world programs suggests that path-sensitive data flow analysis tools could eliminate the need for most of the annotations we added.

5.2 Model Translation

As explained in Section 3.2, we translate each function CFG into a separate Petri net. Each place in the function Petri net corresponds to a basic block in the function, and control transfer from one basic block to another is represented by a transition. Lock places link lock acquisition/release transitions in these function Petri nets. All of these model translations are straightforward, but with real-world programs the difficulty lies in modeling common software practices that may obfuscate a program's locking behavior, e.g., locks accessed through pointers and primitive lock data types enclosed in wrapper structures that are passed to lock/unlock function wrappers.

With the lock name and type information in the augmented CFGs, as discussed in the previous subsection, we are able to identify whether the lock variable in a lock function argument is a static reference to a lock instance or a dynamic choice of a certain lock type. A lock type is defined as its wrapper structure type, i.e., the type of the argument of the wrapper function. In the case of chained pointers/references as the function argument, we could either be conservative and consider only the type of the last node (the default setting), or take the whole chain as the lock type. In the latter case, a false negative is possible when two different chains actually refer to the same lock in the end. False negatives are unacceptable, so we prefer the former approach.

When a lock variable is a static reference, we model the lock *instance* as a lock place and link it to the acquisition/release transitions as discussed in Section 3.2. When a lock variable is a dynamic choice through pointers, we approximate the model by using a lock place to represent the lock *type* rather than the actual instance. All dynamic references of this lock type share the same lock place, and there is exactly one initial token in the place. If a lock is accessed by both static address reference and pointers, we still approximate the model using a lock place to represent the lock type. In our experience, such mixed references are rare in practice.

Since we model dynamically selected locks by their types, different instances of the same lock type could give false positives for deadlock, and the end result would be impaired performance. In real programs, however, most deadlocks reported in bug databases and change logs involve cyclic wait on *different* lock types (Lu et al. 2008), i.e., locking hierarchy violations as in the OpenLDAP bug of Figure 12. Cyclic wait on locks of the same type is uncommon. Our model simplification matches this typical deadlock pattern well.

5.3 Offline Control Synthesis

The input to the offline control synthesis module is a set of Petri nets that represents each function in the program. Since a deadlock may cross function boundaries and involve different parts of the program, we need to inline these Petri nets for control synthesis, as described in Section 3.2. However, whole program inlining is not practical for real programs because of the extensive use of function pointers and recursions. We instead inline only functions related to critical regions, i.e., functions that can be called by threads holding locks. In other words, we collapse regions of the global inlined Petri net that contain no lock-related operations because they are irrelevant. With this correctness-preserving performance optimization, the inlined call depth is typically no more than three.

5.4 Online Control

We have two implementations of the control logic for online control of the program: library interposition and program instrumentation. Library interposition intercepts library calls, typically lock acquisition/release functions, and postpones lock acquisition calls whenever necessary. Library interposition does not modify program source code, and therefore can work directly with binaries. However, as explained in Section 4.2, the control synthesis algorithm must account for the partial observability problem, which possibly reduces concurrency because of the limited set of observable transitions. Program instrumentation, on the other hand, inserts control logic code into the program as needed, and therefore can in principle observe the complete program execution state.

For both implementations, we must correlate program execution state with corresponding Petri net state. The current execution function and line number are not sufficient as one function might be called at different locations and therefore result in different control

actions. We need the call stack to map the current program state to transitions in the Petri net. Our library interposition approach walks up the call stack from the intercepted library call and identifies the current transition. With the program instrumentation approach, for reasons of performance and portability, we instead instrument functions as necessary with additional parameters that encode the execution state.

6. Experiments

This section summarizes experimental evaluations of our prototype implementation. Greatly expanded results on randomly generated programs, the publish-subscribe benchmark, and OpenLDAP are available in (Wang et al. 2008b,a).

Randomly Generated Programs Our first test involved randomly generated programs reminiscent of the dining philosophers problem. These programs repeatedly acquire or release a randomly selected lock then sleep for a random interval; they deadlock readily. After we apply our deadlock avoidance technique to them, however, they run indefinitely without deadlock. A comparison of our customized control synthesis algorithm with a naïve application of standard SBPI shows that our approach offers scalability benefits in terms of the off-line computational cost of control synthesis (Wang et al. 2008b).

Publish-Subscribe Benchmark Our next test involved a highly concurrent network server benchmark. We implemented a simple multithreaded publish-subscribe server that can be compiled in three ways: deadlock-free, deadlock-prone, and atomic-section. The first employs fine-grained locks correctly and the second acquires locks out of order. The third is compiled using the Intel prototype software transactional memory compiler (Intel). The server software must perform I/O within critical sections to satisfy an application-level consistency requirement. It runs on a machine with a dual-core CPU and four network cards connected to four client emulators.

As expected, our deadlock avoidance technique automatically eliminates deadlocks in the deadlock-prone variant. The main purpose of the benchmark test is to evaluate performance rather than correctness. From the clients, we measured throughput under heavy load (server saturation) and transaction response times under light load. Our first result is that our deadlock avoidance approach has a negligible effect on light-load response times and reduces saturation throughput by roughly 18% compared to the deadlock-free variant. Our second result is that the atomic-sections version suffers far *worse* performance overheads: it achieves only about half of the heavy-load throughput of the dynamic-deadlock-avoidance version and suffers a $6\times$ increase in response times. This result initially surprised us until we determined that all I/O was serialized in the atomic-sections version. This is not a shortcoming in the Intel compiler but rather a fundamental consequence of atomic sections: atomic sections containing I/O *must* be serialized lest covert I/O channels violate isolation among them. Our results show that locks sometimes permit better exploitation of available physical resources than atomic sections. Our dynamic deadlock avoidance technique preserves this benefit of locks while eliminating deadlocks and restoring the composability that locks alone destroy.

OpenLDAP We next applied our method to OpenLDAP, a popular open-source implementation of the Lightweight Directory Access Protocol. We tested on OpenLDAP version 2.2.20, which has a confirmed circular-mutex-wait deadlock bug (OpenLDAP). The bug was fixed in 2.2.21 but returned in 2.3.13 when new code was added. The whole program has 1,795 functions and 41 lock types (i.e., distinct types of structures that contain locks). We annotated OpenLDAP's internal wrapper lock functions and six other pairs of lock/unlock functions that operate on file or database locks or call OpenLDAP's wrappers through pointers. Our implementation's first pass took a few seconds and reported 25 ambiguous functions (i.e., the set of locks held on exit was ambiguous). We annotated 21 manually; this took slightly more than an hour. The second pass reported four potential deadlocks: the known deadlock, two previously unreported ones, and a false positive that is due to limited data flow analysis. We disabled deadlock avoidance instrumentation for the latter and enabled it for the three real deadlocks; control synthesis terminated in a few seconds, after a single iteration. Performance tests involving three different synthetic workloads initially showed negligible performance overheads. We had to modify the standard OpenLDAP server configuration substantially in order to trigger adverse performance consequences for the server instrumented with deadlock-avoidance control logic. Worst-case overheads on client-measured throughput and response times ranged from 3–10%, depending on the workload.

Apache We also applied our method to the most recent release of Apache, version 2.2.8. This version of `httpd` contains 2,264 functions and twelve lock types. Our prototype's first pass found 28 functions containing false-paths ambiguities, nearly all of which involve error checking in lock/unlock functions (if the attempt to acquire a lock fails, they return immediately). After we appropriately annotate these functions, the second pass shows no circular-mutex-wait deadlock. This finding is consistent with the Apache bug database, which reports no such deadlocks in version 2.2.8. When condition variables are included in the model, our analysis identifies the known deadlock bug in Figure 10 and automatically synthesizes the control logic depicted in Figure 11.

7. Related Work

This section reviews prior research in Discrete Control Theory and in atomic sections implemented with transactional memory and with conventional locks. See (Wang et al. 2008a) for a detailed review of four traditional approaches to deadlock (static detection, static prevention, dynamic detection, and dynamic avoidance) and two new proposals ("healing" and "immunity").

Discrete Control Theory has matured rapidly since the seminal work of (Ramadge and Wonham 1987). Comprehensive textbooks facilitate graduate-level education in Discrete Control (Cassandras and Lafortune 2007) as the research community expands the frontier of DCT results. The prior work closest our own involves deadlock avoidance in manufacturing systems (Li et al. 2008), which admit restricted models that facilitate analysis and control synthesis; however such models are inappropriate for concurrent software. Furthermore, much research in this area either fails to achieve maximal permissiveness or requires explicit exploration of the reachable state space. Our contributions are to leverage Discrete Control as a principled foundation for dynamic deadlock avoidance in general-purpose software, and to achieve both scalability and maximal permissiveness.

Several recent approaches allow programmers to define atomic sections that are guaranteed to execute atomically and in isolation. Transactional Memory (TM) implements atomic sections by optimistically permitting concurrency, detecting conflicts among concurrent atomic sections and resolving them by rolling back execution (Larus and Rajwar 2007). Rollback is not an option if irrevocable actions such as I/O occur within transactions, but such transactions can be supported as long as they are serialized (Welc et al. 2008). This is the approach taken in the Intel prototype TM compiler (Intel). Unfortunately, such serialization can degrade performance and can prevent software from fully exploiting available physical resources (Wang et al. 2008a).

An alternative approach to implementing atomic sections uses conventional locks rather than transactions and attempts to associate locks with atomic sections in such a way as to maximize concurrency (McCloskey et al. 2006; Isard and Birrell 2007; Emmi

et al. 2007; Cherem et al. 2008). In the simplest case, all locks associated with an atomic section are acquired upon entry of the section and released upon exit, which reduces concurrency. More fine-grained locking strategies acquire locks lazily and/or release locks eagerly; however, lazy acquisition immediately prior to accesses of protected variables can imply incorrect lock ordering and thus deadlock.

In contrast to the paradigm of atomic sections, our approach brings benefits to legacy lock-based code, imposes no performance penalty on I/O within critical sections, and exploits detailed knowledge of all possible whole-program behaviors to maximize concurrency. Locks provide a more nuanced language for expressing allowable concurrency than existing implementations of atomic sections, and our approach preserves this benefit. At the same time, our approach restores the composability that locks destroy and ensures deadlock freedom, just as atomic sections do.

8. Conclusions

This paper has demonstrated that Discrete Control Theory provides a formal foundation for dynamic deadlock avoidance in multithreaded software. We construct program models with structural features (siphons) corresponding to undesirable runtime behaviors (deadlocks), and use DCT to synthesize runtime control logic that provably avoids the latter by constraining the former. Our approach effectively eliminates deadlocks from the original program without silently introducing new deadlocks or global performance bottlenecks. The control logic that we synthesize is maximally permissive, ensuring that runtime concurrency is maximized. Our approach furthermore reduces runtime overheads by performing the most computationally expensive steps (siphon analysis and SBPI) offline, which minimizes the online costs associated with our control logic. In essence, DCT control logic synthesis performs a deep whole-program analysis that compactly encodes context-specific foresight, allowing the runtime control logic to adjudicate lock acquisition requests quickly, based on current program state and worst-case future execution possibilities.

Extensive experiments with a C/Pthreads prototype confirm that our approach scales to real software, eliminates both naturally occurring and injected deadlock faults, and adds negligible to modest performance overhead. Like atomic sections, our approach restores composability and thereby reinstates the cornerstones of programmer productivity, divide-and-conquer problem decomposition and software modularity. Unlike atomic sections, our approach is backward compatible with legacy code and programmers. Because it neither forbids nor penalizes arbitrary I/O in critical sections, it sometimes enables software to exploit available physical resources more fully than atomic sections.

Acknowledgments

We thank Eric Anderson, Hans Boehm, Pramod Joisha, Hongwei Liao, Spyros Reveliotis, and the anonymous reviewers for many helpful comments.

References

Apache. Apache bug database, 2008. `https://issues.apache.org/bugzilla/index.cgi`.

E. R. Boer and T. Murata. Generating basis siphons and traps of Petri nets using the sign incidence matrix. *IEEE Trans. on Circuits and Systems—I*, 41(4):266–271, April 1994.

C. G. Cassandras and S. Lafortune. *Introduction to Dsicrete Event Systems*. Springer, second edition, 2007.

S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI*, June 2008.

M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL*, 2007.

D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley, 2004.

L. Holloway, B. Krogh, and A. Giua. A survey of Petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 7(2):151–190, 1997.

Intel. Intel C++ STM Compiler, Prototype Edition, January 2008.

M. V. Iordache and P. J. Antsaklis. *Supervisory Control of Concurrent Systems: A Petri Net Structural Approach*. Birkhäuser, 2006.

M. Isard and A. Birrell. Automatic mutual exclusion. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, May 2007.

K. M. Kavi, A. Moshtaghi, and D. Chen. Modeling multithreaded applications using Petri nets. *International Journal of Parallel Programming*, 30(5):353–371, October 2002.

J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2007.

Z. Li, M. Zhou, and N. Wu. A survey and comparison of Petri net-based deadlock prevention policies for flexible manufacturing systems. *IEEE Trans. on Systems, Man, and Cybernetics—Part C*, 38(2):173–188, March 2008.

S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.

B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *POPL*, 2006.

T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

OpenImpact. OpenIMPACT, 2008. `http://www.gelato.uiuc.edu/`.

OpenLDAP. OpenLDAP Issue Tracking System, 2008. `http://www.openldap.org/its/`.

C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriffen des IIM Nr.3, 1962.

P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1), 1987.

W. Reisig. Petri nets. In *EATCS Monographs on Theoretical Computer Science*, volume 4. Springer-Verlag, Berlin, 1985.

S. A. Reveliotis. *Real-Time Management of Resource Allocation Systems: A Discrete-Event Systems Approach*. Springer, New York, NY, 2005.

S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 15(4):391–411, November 1997.

Y. Wang, T. Kelly, and S. Lafortune. Discrete control for safe execution of IT automation workflows. In *EuroSys*, 2007.

Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, 2008a.

Y. Wang, T. Kelly, M. Kudlur, S. Mahlke, and S. Lafortune. The application of supervisory control to deadlock avoidance in concurrent software. In *Workshop on Discrete Event Systems*, May 2008b.

Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA*, June 2008.