# Lock inference for systems software

John Regehr Alastair Reid
School of Computing, University of Utah
{regehr, reid}@flux.utah.edu

## ABSTRACT

We have developed task scheduler logic (TSL) to automate reasoning about scheduling and concurrency in systems software. TSL can detect race conditions and other errors as well as supporting *lock inference*: the derivation of an appropriate lock implementation for each critical section in a system. Lock inference solves a number of problems in creating flexible, reliable, and efficient systems software. TSL is based on a notion of asymmetrical preemption relations and it exploits the hierarchical inheritance of scheduling properties that is common in systems software.

## 1. INTRODUCTION

Embedded systems, operating systems, and Internet servers are fundamentally concurrent because they must respond to external events in real time. For people developing these systems, critical sections can be considered to be a functional aspect of software: they are used to maintain high-level program invariants. The implementation of critical sections, on the other hand, is a non-functional aspect — it affects response time and throughput.

In this paper we take the position that locks in systems software, which are usually named by referring to an instance of a particular lock implementation, should be specified at a higher level of abstraction. At system build time a whole-program analysis should be used to infer an appropriate lock implementation for each critical section. This has a number of benefits that can lead to the creation of robust, reusable systems software:

- Developers need not learn the complex rules that govern locking in systems software. For example, threads synchronize with interrupts by disabling interrupts, interrupts must not block, and non-preemptive event handlers are implicitly synchronized.
- Code maintenance and modification is made easier and less prone to bugs. For example, if an event handler is broken out into a preemptive thread to ensure that its response-time requirements can be met, resources that it shares with other handlers, which previously did not need protection by a lock, must now be protected. These resources can be automatically detected be TSL.
- In many cases a generic component, which implements correct locking, is instantiated in such a way that its locks serve no useful purpose, e.g., because it is accessed only by a single thread or because a component higher in the call graph provides sufficient serialization. In this case the locks can be dropped as an optimization.
- When the analysis finds a critical section where no available lock implementation works, a race condition has been detected and this should be brought to a developer's attention.
- Locks can be selected in such a way that their global side effects are desirable. For example, in a system where throughput is important, it might be the case that all locks should be implemented by disabling interrupts since this is very efficient. In another system where real-time deadlines must be met, it may be unacceptable to disable interrupts for more than a few microseconds because this delays unrelated, time-critical processing.
- Software components can be developed that are agnostic with respect to the execution environments in which they are instantiated. This is desirable because the environments in which a component executes depend on the call graph for a particular system. In effect, the execution environments in a system cross-cut its traditional modular decomposition.

These benefits are provided by task scheduler logic (TSL), a novel formalism for integrated reasoning about scheduling and concurrency in systems software. The key idea behind TSL is that the hierarchical inheritance of scheduling properties, when combined with modular specifications of schedulers and locks, can be used to formalize the rules that govern locking in systems software. These rules — previously informal and unchecked — are caused by complex relationships between multiple *execution environments*: software contexts such as kernel-supported threads, user-level threads, interrupts handlers, and event loops. Furthermore, as a side effect of deriving and checking rules about synchronization, it becomes possible to perform *lock inference*: the derivation of an appropriate lock implementation for each critical section.

Lock inference can be viewed as the top of a stack of useful capabilities for manipulating and analyzing concurrency aspects of systems software. At the bottom of the stack is externally visible and parameterizable locking — lock analysis and inference are difficult if locks are hard-wired into code. Many component systems, such as our Knit [10], have this capability. In the middle of the stack is the capacity to detect concurrency errors, which depends on a model of resources and concurrency as well as a mechanism for tracking the call graph. Many systems have done this, but TSL is the first we are aware of that can find concurrency problems in systems software where there are diverse execution environments. Finally, at the top of the stack is the ability to automatically infer an appropriate implementation for each critical section in a system. This is the primary contribution of TSL.

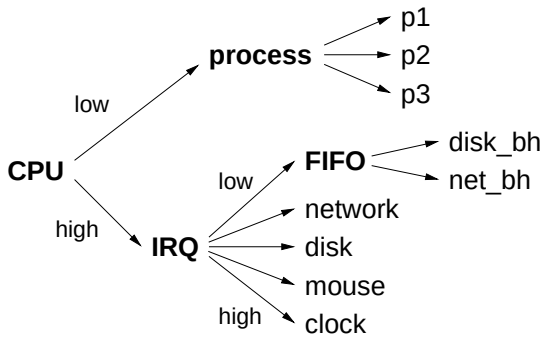


**Figure 1: A generic UNIX scheduling hierarchy**

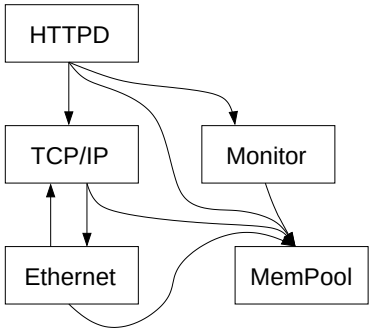


**Figure 2: A simple component-based monitoring system**

# 2. BACKGROUND

This section describes the hierarchical scheduling concepts that underlie TSL, the difficulties of creating component-based systems that motivate our work, and the lightweight program analysis that is a prerequisite for using TSL.

## 2.1 Hierarchical Scheduling

At a coarse granularity, the flow of control in a software system is determined by its schedulers. In this paper a task is a schedulable flow of control and a scheduler is any piece of software (or hardware) that controls the order of execution of tasks. Properties are imparted to a task by each scheduler that it runs under. For example, an interrupt handler cannot block and it is preemptively scheduled at higher priority than any user-mode code. If an event-processing loop is run in interrupt context, then event handlers scheduled by the loop inherit event properties, such as non-preemptive execution relative to other event handlers, in addition to all interrupt properties. The schedulers in a system create a variety of execution environments, each of which has its own rules for structuring code, sequencing operations, and interacting with other environments.

Figure 1 depicts the scheduling hierarchy for a typical UNIX-like operating system. The top-level scheduler, CPU, is implemented in hardware; it runs interrupts whenever possible and user-mode code otherwise. The IRQ scheduler preemptively schedules hardware interrupt handlers based on their priorities, as well as a software interrupt handler at the lowest priority. The software interrupt handler runs a FIFO scheduler that runs deferred bottom-half handlers disk_bh and net_bh with run-to-completion semantics. The process scheduler is the standard preemptive UNIX timesharing scheduler; it runs processes p1..p3.

We have found the hierarchical scheduling notation shown in Figure 1 to be quite useful and general for describing the execution environments provided by systems. For example:

- Linux, Windows 2000 [12], and most real-time operating systems are minor variations on the same theme.
- RTLinux [13] adds an additional level of scheduling above the CPU scheduler by virtualizing the interrupt handling structure of Linux.
- TinyOS [6] has no thread or process scheduler: its scheduling hierarchy includes only interrupts and an event loop.
- Internet servers, Java virtual machines, and other application-level systems software extend the scheduling hierarchy by implementing event loops, thread pools, and user-level threads.

TSL provides a uniform notation for modeling these and other collections of execution environments.

## 2.2 Component Based Systems Software

Figure 2 depicts the software for an embedded application that is designed to (1) monitor a system such as a pumping station on a remote section of an oil pipeline and (2) make information about the system available to HTTP clients. Upon request the $HTTPD$ component retrieves data from the $Monitor$ component and sends it out on the network using the $TCP/IP$ and $Ethernet$ components. All components make use of a memory allocator.

Although TSL applies generally to systems software, and makes no assumptions about the underlying component model, it is especially useful for analyzing component-based systems software. First, component software tends to expose interfaces for locking, making it easier to analyze and parameterize synchronization behavior. Second, component-based software is often hard to understand due to its many indirect connections between software modules. This complexity interacts poorly with the multiple execution environments that are created by a hierarchy of schedulers such as the one in Figure 1. For example, assume that the $MemPool$ component in Figure 2 reports an out-of-memory condition using a logging interface that is connected to a storage component (not shown). The storage component uses a thread mutex to protect its internal data structures. Since $MemPool$ can be called by the $Ethernet$ component while executing in interrupt context, the interrupt handler can indirectly attempt to acquire the mutex, leading to a system crash because interrupts are not permitted to acquire mutexes. This bug will not be obvious to a developer who simply wants to reuse these components and who does not have a detailed understanding of their internals. Furthermore, this bug will be very difficult to expose through testing since the allocator rarely runs out of storage. In our experience, creating correct systems using components like the ones in this example requires near-expert knowledge about component internals. Clearly there is room for improvement.

## 2.3 Analyzing Systems

TSL requires static identification of tasks, schedulers, resources, critical sections, and the call graph for a program or system. There

are well-known techniques for obtaining this information; in practice we expect that a combination of annotations and language-based program analysis will be used. For example, in our prototype implementation (see Section 6) we learn about resources using annotations and obtain an approximation of the call graph by analyzing the component linking graph.

# 3. TASK SCHEDULER LOGIC

This section provides an overview of TSL.

## 3.1 Tasks and Schedulers

Tasks are sequential flows of control through a system; they are the fundamental unit of reasoning in TSL. Each task has a well-defined entry point and many tasks also finish by returning control to the scheduler that invoked them. Other tasks encapsulate an infinite loop and these never finish — control only returns to their scheduler through preemption. Throughout this paper the variables $t$, $t_1$, etc. range over tasks.

Schedulers are modeled in a modular way by specifying the preemption relations that the scheduler induces between tasks that it schedules. Preemption relations are represented asymmetrically: we write $t_1 \not\downarrow t_2$ when task $t_2$ can preempt task $t_1$. That is, if $t_2$ can start to run after $t_1$ begins to execute but before $t_1$ finishes.

The simplest scheduler, a non-preemptive event scheduler, does not permit any child to preempt any other child. For any two children $t_1$ and $t_2$ of such a scheduler, $\neg(t_1 \not\downarrow t_2) \wedge \neg(t_2 \not\downarrow t_1)$.

On the other hand, a preemptive scheduler, such as a UNIX time-sharing scheduler, potentially permits each child task to preempt each other child task. That is, for any two children of such a scheduler, $t_1 \not\downarrow t_2 \wedge t_2 \not\downarrow t_1$.

A third type of scheduler commonly found in systems software is a strict priority scheduler such as the interrupt controller in a typical PC. It schedules a number of tasks $t_1..t_n$ and it is the case that $t_j \not\downarrow t_i$ when $i < j$.

## 3.2 Resources, Races, and Locks

At each program point a task is accessing some (possibly empty) set of resources. The variables $r$, $r_1$, etc. range over resources, and we write $t \rightarrow r$ if a task $t$ uses a resource $r$. Resources represent data structures or hardware devices that must be accessed atomically.

A race condition may occur if task $t_1$ can be preempted by $t_2$ at a point where both tasks are accessing a common resource. Problematic preemption relations can be eliminated using locks; at each program point a task holds a (possibly empty) set of locks. We write $t_1 \not\downarrow_L t_2$ if parts of a task $t_2$ that hold a set of locks $L$ can start to run while a task $t_1$ holds $L$. For example, consider two threads that can usually preempt each other. If holding a thread lock $lk$ blocks a task $t_2$ from entering critical sections in $t_1$ protected by $lk$, then $(t_1 \not\downarrow_\varnothing t_2) \wedge \neg(t_1 \not\downarrow_{lk} t_2)$.

Every lock is provided by some scheduler; the kinds of locks provided by a scheduler are part of its specification. We write $t \multimap l$ if a scheduler $t$ provides a lock $l$, and require that each lock be provided by exactly one scheduler. There are two common kinds of locks. First, locks that resemble disabling interrupts: they prevent any task run by a particular scheduler from preempting a task that holds the lock. Second, locks that resemble thread mutexes: they only prevent preemption by tasks the hold the same instance of the type of lock.

Locks satisfy three important properties. First, if $t_1$ can be preempted while holding a set of locks, then $t_1$ can be preempted while holding fewer locks:

$$t_1 \not\downarrow_{L_1} t_2 \ \wedge \ L_1 \supseteq L_2 \ \Rightarrow \ t_1 \not\downarrow_{L_2} t_2$$

Second, if $t_1$ can be preempted by $t_2$ while holding either a set of locks $L_1$ or a set of locks $L_2$, then $t_1$ can be preempted by $t_2$ while holding both sets of locks.

$$t_1 \not\downarrow_{L_1} t_2 \ \wedge \ t_1 \not\downarrow_{L_2} t_2 \ \Rightarrow \ t_1 \not\downarrow_{L_1 \cup L_2} t_2$$

Finally, preemption is a transitive relation: if $t_1$ can be preempted by $t_2$ and $t_2$ can be preempted by $t_3$, then $t_1$ can be preempted by $t_3$.

$$t_1 \not\downarrow_{L_1} t_2 \ \wedge \ t_2 \not\downarrow_{L_2} t_3 \ \Rightarrow \ t_1 \not\downarrow_{L_1 \cap L_2} t_3$$

The definition of a race condition is as follows:

$$\begin{aligned} \mathit{race}(t_1, t_2, r) \stackrel{\text{def}}{=} \ & t_1 \rightarrow_{L_1} r \\ \wedge \ & t_2 \rightarrow_{L_2} r \\ \wedge \ & t_1 \neq t2 \\ \wedge \ & t_1 \not\downarrow_{L_1 \cap L_2} t_2 \end{aligned}$$

That is, a race can occur if two tasks $t_1$ and $t_2$ use a common resource $r$ with some common set of locks $L_1 \cap L_2$, and if $t_2$ can preempt $t_1$ even when $t_1$ holds those locks. For example, if some task $t_1$ uses a resource $r$ with locks $\{l_1, l_2, l_3\}$ and another task $t_2$ uses $r$ with locks $\{l_2, l_3, l_4\}$ then they hold locks $\{l_2, l_3\}$ in common and a race occurs iff $t_1 \not\downarrow_{\{l_2, l_3\}} t_2$.

## 3.3 Hierarchical Scheduling

Each scheduler is itself a task from the point of view of a scheduler one level higher in the hierarchy. For example, when an OS schedules a thread, the thread is considered to be a task regardless of whether or not an event scheduler is provided by the thread. We write $t_1 \lhd t_2$ if a scheduler $t_1$ is directly above task $t_2$ in the hierarchy; $\lhd$ is the *parent* relation. Similarly, the *ancestor* relation $\lhd^+$ is the transitive closure of $\lhd$.

TSL gains much of its power by exploiting the properties of hierarchies of schedulers. First, the ability or lack of ability to preempt is inherited down the scheduling hierarchy: if a task $t_1$ cannot preempt a task $t_2$, then $t_1$ cannot preempt any descendent of $t_2$. A consequence is that if the *nearest common scheduler* in the hierarchy to two tasks is a non-preemptive scheduler, then neither task can preempt the other. This is a useful result when showing, for example, that a lock is not necessary to protect a resource that is accessed by a particular composition of components.

When a task that is the descendent of a particular scheduler requests a lock, the scheduler may have to block the task. It does this not by directly blocking the task, but by blocking its currently running child, which must be transitively scheduling the task that requested the lock. If a task attempts to acquire a lock that is not provided by one of its ancestors in the scheduling hierarchy then there is no child task for the scheduler to block — an illegal action has occurred. Using TSL we can check for a generalized version of the "blocking in interrupt" problem by ensuring that tasks only acquire blocking locks provided by their (possibly transitive) parents in the

scheduling hierarchy. We formalize this generalization as follows:

$$\begin{aligned} \mathit{illegal}(t, l) \stackrel{\text{def}}{=} \exists\, t_1 .\quad & t_1 \multimap l \\ \wedge\quad & \neg(t_1 \lhd^{+} t) \\ \wedge\quad & t \rightarrow_L r \\ \wedge\quad & l \in L \\ \wedge\quad & \mathit{blocking}(l) \end{aligned}$$

### 3.4 Using TSL

Software developers, who compose systems using new and existing components, do not need to directly interact with TSL. Rather, they create software as usual, but in addition to protecting critical sections with locks, they have the option of using a *virtual lock* that is a cue for TSL to infer an appropriate lock implementation. Developers who create new schedulers will need to specify their properties in TSL, but we expect that these programmers will be in the minority: most will reuse an existing scheduler and its attached TSL specification.

## 4. LOCK INFERENCE

Many of the benefits of TSL are provided by its ability to infer an appropriate lock implementation for each critical section. Recall that a lock assignment is legal if the lock is not a blocking lock or if it is provided by an ancestor of the task that contains the critical section. A brute-force algorithm for synchronization inference is to enumerate all legal assignments of locks to critical sections; the enumeration can stop once an assignment is found that eliminates all race conditions. If no such assignment exists, then there is a genuine race condition and the system cannot be built. No special algorithmic support for the elimination of unnecessary synchronization is required because synchronization inference subsumes synchronization elimination. It suffices to ensure that one of the locks available to each critical section is the "null lock" that has no effect on preemption relations and is implemented as a NOP.

We currently use the brute-force algorithm to assign lock implementations to critical sections. Although it is tractable for systems that we have analyzed, we expect that we will want to develop improved algorithms. One avenue for improvement is to exploit qualities of the domain. For example, the search space can be narrowed by observing that it is probably not useful to attempt to use a different kind of lock, or a different instance of the same kind of lock, to protect different critical sections that access the same resource. In addition, for each resource the set of legal locks should be tried in an intelligent order, probably starting with a "strong" lock, like disabling interrupts, that eliminates many preemption relations. Another way to improve performance might be to cast the lock inference problem as an instance of the boolean satisfiability problem, for which very fast solvers exist [8].

Once a lock assignment that eliminates all races is found it may be desirable to optimize the choice of locks. Such optimization is outside the scope of TSL, which has no mechanism for preferring one lock assignment over another as long as both of them produce a system that is free of race conditions. In general, there is a tension between choosing an efficient lock for each critical section and picking locks that avoid unnecessarily delaying the execution of unrelated tasks.

## 5. REAL-TIME CONCERNS

Since lock choice has a pervasive effect on system performance, we plan to integrate synchronization inference with SPAK, a real-time scheduling tool that we have developed [9]. The negative effects that locks have on real-time tasks can be quantified by adding *blocking terms* — periods of time during which certain scheduling decisions cannot be made — to the schedulability analysis equations [11]. If a lock resembles disabling interrupts, it contributes blocking terms to all tasks run by the scheduler providing the lock. On the other hand, a lock that resembles a mutex contributes blocking terms only to tasks that may attempt to acquire the same lock. Blocking terms, like preemption relations, are inherited down the scheduling hierarchy.

Besides returning a binary result about overall system schedulability, SPAK can perform several useful functions that interact well with TSL. First, it can evaluate the *robustness* of a software system under timing faults: tasks that run for longer than their nominal worst-case execution times. This is useful because it can help TSL avoid creating systems that are brittle in the sense that a small perturbation in task execution will cause real-time deadlines to be missed. Second, SPAK has the capability to map a large number of design-time tasks onto a smaller number of run-time threads; this is useful for resource-constrained embedded systems because it reduces the amount of memory devoted to thread stacks and the amount of CPU time spent performing context switches. Synchronization inference and thread minimization interact favorably because strongly coupled collections of tasks, when aggregated into a single thread, will enable many locks to be eliminated. These tasks should be preferred targets for thread minimization when compared to collections of tasks that permit few locks to be eliminated.

## 6. APPLYING TSL

We have implemented a prototype TSL checker based on a forward-chaining evaluator: it takes a specification for a system and derives all possible consequences of the TSL axioms. Systems specified in TSL are finite; there are a limited number of tasks and preemption relations between tasks.

Our test environment is based on the Knit [10] component language and the OSKit [4], a library of systems software components. We extended Knit slightly to accommodate annotations about resources and locks, and we used Knit's component linking graph to generate a safe (though crude) approximation of the call graph.

Figure 3 provides a more detailed look at the embedded, web-based monitoring system from Figure 2. The scheduling hierarchy for the system is shown on the left side of the figure and application components are shown on the right side. For simplicity we have omitted many infrastructure components. The full system consists of 190,000 lines of code, 116 components, 1059 functions, 5 tasks, and 2 kinds of locks.

Before we can analyze the system with TSL we must label all the tasks (we name them $h1$, $h2$, $t$, $e$, and $m$), label all the schedulers (we name them $CPU$, $thread$, $IRQ$, and $FIFO$), and generate a TSL specification for each scheduler (not shown). We must also add resources (named $rh$, $rb$, $rt$, $re$, $rmon$, and $rmem$) and add their uses into the callgraph, add locks (named $cli$ and $lk$), attach locks to the scheduler that provides them, and label edges in the callgraph with locks acquired before the calls are made. Figure 3 shows these labels and relations. The example includes some errors in the use of locks that we shall discover using TSL. The schedulers are, of course, also components, but to keep the example to a reasonable size we do not show their resources, locks used to protect those resources, calls to the functions they export, etc.

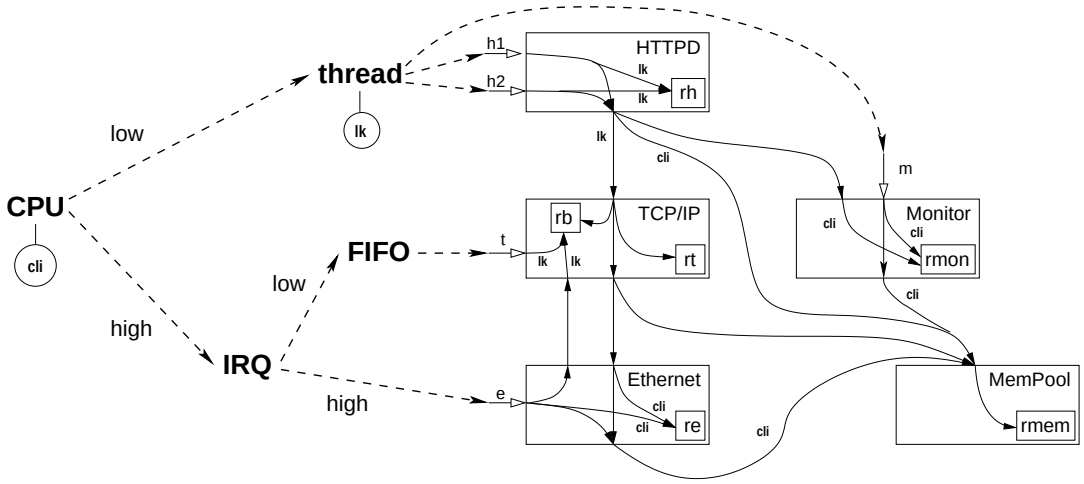


**Figure 3: A simple component-based monitoring system (right) and its scheduling hierarchy (left)**

## 6.1 Checking for illegal locking

To detect cases of illegal locking our implementation computes a list of all the resources accessed by each task with a given set of locks. For example, from the callgraph and locks shown in the figure we generate the following table:

$$\begin{array}{rll} h1, h2 & \rightarrow_{lk} & \{rh, rt, rb, rmem\} \\ h1, h2 & \rightarrow_{lk,cli} & \{re, rmem\} \\ h1, h2 & \rightarrow_{cli} & \{rmon, rmem\} \\ m & \rightarrow_{cli} & \{rmon, rmem\} \\ t & \rightarrow_{lk} & \{rb\} \\ e & \rightarrow_{lk} & \{rb\} \\ e & \rightarrow_{cli} & \{re, rmem\} \end{array}$$

Given this table and the knowledge that $lk$ is a blocking lock, it is straightforward to apply the definition of *illegal* to generate a list of all the illegal lock uses:

$$\begin{array}{l} \mathit{illegal}(t, lk) \\ \mathit{illegal}(e, lk) \end{array}$$

Both problems are caused by using the lock $lk$ to protect the resource $rb$ which is accessed by hardware and software interrupts. They can be resolved by changing the lock to $cli$.

Although these errors can be easily found by inspecting Figure 3, the real system has many more components and interconnections and is difficult to debug by inspection.

## 6.2 Checking for races

A race occurs when two tasks may access a resource simultaneously. TSL provides a list of potential race conditions and can be used to examine the scheduler hierarchy and call chain to diagnose the cause of problems.

For example, from the scheduler hierarchy we can deduce that the following preemption relations hold:

$$\begin{array}{rcl} h1, h2, m & \not\downarrow_{\varnothing} & h1, h2, m \\ h1, h2, m & \not\downarrow_{\varnothing} & t \\ t & \not\downarrow_{\varnothing} & e \end{array}$$

Combining this with resource use and the definition of *race*, we obtain the following race conditions.

$$\begin{array}{ll} \mathit{race}(h1, h2, rmem) & \mathit{race}(h2, h1, rmem) \\ \mathit{race}(h1, m, rmem) & \mathit{race}(h2, m, rmem) \\ \mathit{race}(h1, e, rmem) & \mathit{race}(h2, e, rmem) \end{array}$$

These can be fixed by acquiring the $cli$ lock when calling from $TCP/IP$ to $MemPool$.

## 6.3 Synchronization elimination and inference

The system in Figure 3 does not contain any redundant locks. However, consider what would happen if, due to memory constraints, the developer could only instantiate a single thread for the $HTTPD$ component. In this case the locks protecting $rh$ could be safely eliminated as could the thread lock providing atomic access to the top half of the $TCP/IP$ component.

All locks in our example refer to specific implementations. However, if the $cli$ locks in the $Monitor$ component in Figure 3 were declared as virtual locks then TSL would inform us that acceptable lock implementations are $cli$ and $lk$.

# 7. APPLICABILITY AND LIMITATIONS

TSL applies to static systems where tasks, schedulers, critical sections, and the call graph are known in advance. Although this is a good match for most embedded software we would like to extend TSL to handle systems with dynamic components. One way to do this would be to use static analysis or dynamic checking to bound the behavior of the dynamic part of the system. For example, if we guarantee that a particular resource cannot be accessed by a dynamic part of the system, then it is permissible to remove locks protecting this resource provided that this is otherwise a valid optimization. In general, tighter bounds on the behavior of the dynamic part of a system permit more effective analysis and optimization of the static part.

Although TSL cannot yet be used to check systems for risk of deadlock, we are exploring ways to permit this. If locks were represented as an ordered multiset, rather than as an unordered set, then TSL could be used to enforce an ordering on lock acquisitions, leading to a system that is guaranteed to be free of deadlock.

Furthermore, this would permit TSL to check for recursive lock acquisition — this is legal for some lock implementations but not for others.

## 8. RELATED WORK

Model checkers such as SPIN [7] and Bandera [1] represent a promising approach to bringing the benefits of concurrency theory to developers. Model checkers are more powerful than TSL in that they can reason about deadlock and liveness. However, TSL adds value over model checkers by specifically supporting the hierarchical inheritance of scheduling properties that occurs in systems software — this permits effective reasoning across multiple execution environments. Also, model checkers provide no support for lock inference.

The trend towards inclusion of concurrency in mainstream language definitions such as Java and towards strong static checking for errors is leading programming language research in the direction of providing annotations [2, 5] or extending type systems to model locking protocols [3]. These efforts are complementary to our work on reasoning about concurrency across execution environments; we believe that TSL and extended type systems would be a very powerful combination.

Early versions of our Knit toolchain [10] had a primitive mechanism for tracking top/bottom-half execution environments. It did not model locks and locking, but could check for the "blocking in interrupt" error that is particularly easy to make in component based systems. In the earlier version of Knit we could move components from one environment to another and check the resulting systems, but we could not add new execution environments or even model all of the environments in systems that we built.

## 9. CONCLUSION

TSL is a new logic that supports integrated reasoning about scheduling and concurrency; it supports lock inference as well as the detection of concurrency errors and elimination of redundant locking. Binding critical sections to lock implementations too early is the source of many problems in developing flexible, reliable, and efficient systems software. We believe that TSL, or something like it, is necessary to create next-generation software systems where components can be flexibly and correctly instantiated in a variety of execution environments.

## Acknowledgments

The authors would like to thank Eric Eide, Jay Lepreau, and the reviewers for providing valuable feedback on drafts of this paper.

This work was supported, in part, by the National Science Foundation under award CCR-0209185 and by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory under agreements F30602-99-1-0503 and F33615-00-C-1696.

## 10. REFERENCES

[1] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. of the 22nd Intl. Conf. on Software Engineering*, Limerick, Ireland, June 2000.

[2] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, Palo Alto, CA, December 1998.

[3] Cormac Flanagan and Martin Abadi. Types for safe locking. In S.D. Swierstra, editor, *ESOP'99 Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer-Verlag, March 1999.

[4] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 38–51, Saint-Malô, France, October 1997.

[5] Aaron Greenhouse and William L. Scherlis. Assuring and evolving concurrent programs: Annotations and policy. In *Proc. of the 24th Intl. Conf. on Software Engineering*, pages 453–463, Orlando, FL, May 2002.

[6] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. of the 9th ASPLOS*, pages 93–104, Cambridge, MA, November 2000.

[7] Gerard J. Holzmann. The Spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[8] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of the 39th Design Automation Conference*, Las Vegas, NV, June 2001.

[9] John Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proc. of the 23rd IEEE Real-Time Systems Symp.*, Austin, TX, December 2002.

[10] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation*, pages 347–360, San Diego, CA, October 2000.

[11] Lui Sha, Ragunathan Rajkumar, and John Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[12] David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, third edition, 2000.

[13] Victor Yodaiken. The RTLinux manifesto. In *Proc. of The 5th Linux Expo*, Raleigh, NC, March 1999.