*Appears in the Proceedings of the 2005 USENIX Annual Technical Conference*
*Anaheim, California, April 10–15, 2005*

# Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution

Tong Li[1], Carla S. Ellis[1], Alvin R. Lebeck[1], and Daniel J. Sorin[2]

[1]*Department of Computer Science*
[2]*Department of Electrical and Computer Engineering*
*Duke University*
{tongli,carla,alvy}@cs.duke.edu, sorin@ee.duke.edu

## Abstract

Deadlock can occur wherever multiple processes interact. Most existing static and dynamic deadlock detection tools focus on simple types of deadlock, such as those caused by incorrect ordering of lock acquisitions. In this paper, we propose *Pulse*, a novel operating system mechanism that dynamically detects various types of deadlock in application programs. Pulse runs as a system daemon. Periodically, it scans the system for processes that have been blocked for a long time (e.g., waiting on I/O events). To determine if these processes are deadlocked, Pulse speculatively executes them ahead to discover their dependences. Based on this information, it constructs a general resource graph and detects deadlock by checking if the graph contains cycles. The ability to look into the future allows Pulse to detect deadlocks involving consumable resources, such as synchronization semaphores and pipes, which no existing tools can detect. We evaluate Pulse by showing that it can detect deadlocks in the classical dining-philosophers and smokers problems. Furthermore, we show that Pulse can detect a well-known deadlock scenario, which is widely referred to as a denial-of-service vulnerability, in the Apache web server. Our results show that Pulse can detect all these deadlocks within three seconds, and it introduces little performance overhead to normal applications that do not deadlock.

## 1 Introduction

Concurrent programs are difficult to write and debug. One common problem is deadlock. Deadlock can occur wherever multiple processes (or threads) interact. A set of processes is deadlocked if each process is waiting for an event that only another process in the set can cause. Deadlock is a potential problem in all multithreaded programs. Timely detection of deadlock and its cause is essential for resolving the error and maintaining forward progress.

To address this problem, researchers have developed deadlock detection mechanisms. In practice, however, deadlock detection is often not performed in an effective manner. The drawbacks of the various existing approaches include restrictions on the deadlock-prone access patterns that can be handled and lack of information provided on the causes of deadlocks that arise. We classify existing approaches based on whether they are dynamic or static.

One common dynamic deadlock detection approach is to use timeouts. With timeouts, a process is assumed deadlocked after waiting for a shared resource longer than a certain amount of time. This approach is simple, but inaccurate because it cannot differentiate between processes that are deadlocked and processes that simply need a long time to acquire a resource. Even when they detect deadlock correctly, timeouts provide no information to the developer about why the deadlock occurred.

An alternative dynamic approach is based on graph modelling of process interactions. The "textbook" deadlock detection uses the general resource graph model [9], which models the state of a system as a directed graph. The nodes in the graph represent processes and resources, and the edges represent dependences among them. Given a general resource graph, deadlock can be detected by checking if the graph possesses certain properties (e.g., a cycle or a knot). The general resource graph model classifies resources into reusable and consumable. A reusable resource has a fixed number of units; one unit can be assigned to at most one process at

a time. A consumable resource has no fixed total number of units; when a unit is assigned to a process, it ceases to exist. Only a process that is designated as a producer of a consumable resource can produce units of the resource.

In practice, deadlock detection often assumes a simplified resource model: the system contains only reusable resources and there is only a single unit of every resource. This model makes deadlock detection simple to implement, but at the cost of detecting fewer types of deadlock. Under this model, a general resource graph takes a much simpler form as a wait-for-graph (WFG). The nodes in a WFG represent processes and the edges represent dependences between processes—there is an edge from node A to node B if process A is waiting for process B to release a resource. A cycle in a WFG indicates a deadlock. Constructing a WFG requires dynamically tracking the status of the resources. This includes tracking the owner of each resource and the processes that are waiting for the resource at any time.

A common disadvantage of all dynamic techniques is that their analysis only considers control flow paths actually taken. Static deadlock detection (e.g., RacerX [5]) does not have this problem because it performs analysis on all possible control flow paths. However, these methods depend upon programmer specification of lock semantics and availability of the entire code base. Considering all possible paths also forces static tools to face the issue of filtering out potentially large amounts of false positives.

In this paper, we propose *Pulse*, an operating system mechanism that dynamically detects deadlock. Pulse is based on the general resource graph model. It uses high-level speculative execution [3, 6] to construct dependency information about processes that are blocked in the OS kernel. Throughout this paper, we use the terms *processes* and *threads* interchangeably to refer to the basic units of scheduling. Our goal is to detect a wide variety of deadlock situations, including those that can and cannot be detected by existing techniques. However, our intent is not to replace existing techniques, but to increase the types of deadlock that can be detected by developers. Pulse is complementary to existing techniques; when Pulse and the other tools are used together, they can provide the best coverage of deadlocks.

Our implementation of Pulse focuses on detecting deadlocks caused by bugs in application programs as opposed to bugs in kernel code. We have implemented Pulse in Linux kernel version 2.6.8.1. Our results show that Pulse can detect deadlock situations in incorrect solutions to the classical dining-philosophers and smokers problems, and a deadlock scenario in the Apache web server version 2.0.49.

Pulse runs as a daemon process and performs deadlock detection within the OS kernel when necessary. Pulse can be in one of three modes: *nap*, *monitor*, and *detection*. Initially, it is in the nap mode (i.e., the Pulse process sleeps in the kernel). Periodically, it awakens and enters the monitor mode. In this mode, Pulse checks if any process in the system has been asleep for a long time (how long is a tunable parameter). If none is found, Pulse returns to the nap mode. Otherwise, the sleeping processes might be deadlocked, and thus Pulse enters the detection mode. In this mode, Pulse identifies the events on which these long sleeping processes are waiting (e.g., a lock being free or a pipe being non-empty). To discover how these long sleeping processes depend on each other, Pulse forks each of them to create a *speculative process*. A speculative process first modifies its state such that it will not block again (e.g., by setting the status of the lock on which its parent is blocked to free in its own address space), and then executes ahead in its parent's program. To prevent a speculative process from changing the state of normal processes, Pulse leverages the copy-on-write mechanism in Unix fork and disallows a speculative process to perform I/O writes [6].

During the execution of a speculative process, Pulse records all the events it creates (e.g., releasing a lock or writing to a pipe). These events are then used to match against the events awaited by the sleeping processes. A match between processes $A$ and $B$ indicates that if process $A$ were unblocked, it would produce an event that unblocks process $B$. Based on this information, Pulse constructs a general resource graph in which the nodes denote the sleeping processes and the events on which they are waiting, and the edges denote the dependences Pulse discovers. Pulse detects deadlock by checking if this graph contains cycles. If it detects a cycle, it also prints out the entire graph to help programmers identify the causes of the deadlock.

Pulse differs from existing dynamic techniques in that it discovers dependency information by looking into the future, while existing techniques rely on past information (e.g., who owns a lock and who is waiting for it) to derive the dependences. Most existing techniques are WFG-based and they commonly restrict themselves to only detect deadlocks involving single-unit reusable resources such as locks. Fundamental in this resource model is the assumption that a busy resource can only be freed by the owner that currently holds the resource. This assumption makes it possible to discover process dependences based on information collected from the past, which includes the identities of the owners and

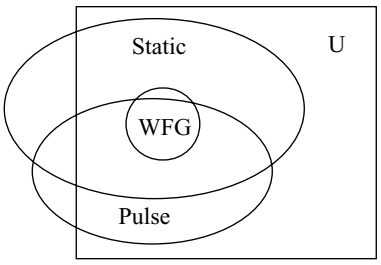


Figure 1: Venn diagram for deadlocks detectable by WFG-based techniques, static schemes, and Pulse. U is the universal set of all deadlocks. The regions outside the rectangle represent false positives.

waiters of each resource. However, consumable resources cannot be viewed as held by a process. For example, a semaphore used for synchronization (instead of mutual exclusion as a lock) may not be viewed as being held by a single process—any process can potentially perform an up operation on the semaphore and unblock another waiting process. In contrast, Pulse makes no assumption about the resource model. The ability to look into the future allows Pulse to directly discover what events a process can produce, as opposed to reasoning about it based on ownership information.

The ability to detect deadlocks that involve consumable resources also distinguishes Pulse from existing static approaches, such as RacerX [5]. In Figure 1, we use a Venn diagram to illustrate the types of deadlock that WFG-based techniques, static schemes, and Pulse can detect. We draw the set of deadlocks detectable by the static schemes larger than the other sets, because static schemes detect deadlocks along all possible control flow paths, while WFG-based techniques and Pulse only detect deadlocks that occur during execution. There is one particular type of deadlock that Pulse cannot detect while WFG-based techniques and static schemes can. This happens when Pulse cannot discover all the dependences from running ahead in the program. For example, if a sloppy programmer forgets an unlock operation that can unblock a waiting process, then Pulse will not see this future event and thus will not be able to identify a cycle of dependences.

There are also types of deadlock that Pulse can detect but the other approaches cannot. These include deadlocks involving consumable resources (e.g., RacerX ignores deadlocks with synchronization semaphores) and variable aliasing (e.g., different variables may point to the same lock, which may not be detectable with static analysis). On the other hand, both static detection and Pulse can generate false positives, i.e., detect deadlocks that do not really exist. Static detection can generate more false positives because it cannot completely filter out control flow paths that are never taken in real execution. However, Pulse does create a unique set of false positives that other techniques do not encounter, which we discuss in Section 3.5.

Pulse can be viewed as complementary to the existing deadlock detection techniques. If Pulse and the other techniques are used together, they can provide the best coverage of deadlocks.

The remainder of this paper is organized as follows. In Section 2, we discuss related work. Section 3 describes how Pulse works and its limitations. We discuss how we implement Pulse in Section 4. In Section 5, we demonstrate that Pulse can detect deadlock situations in the classical dining-philosophers and smokers problems, as well as in the Apache web server. Existing techniques, including WFG-based schemes and RacerX, can only detect deadlock in the dining-philosophers problem, but cannot detect the other two deadlock situations. Finally, we conclude in Section 6.

## 2 Related Work

In this section, we provide more details about related work on dynamic and static deadlock detection and OS-level speculative execution.

**Dynamic deadlock detection.** Most dynamic schemes detect simple deadlocks that involve only lock-like resources. By tracking every acquire and release of the shared resources, these schemes can discover cyclic dependences among the processes. There is a large body of research on deadlock detection in distributed systems (see a survey by Singhal [11]). Some software systems also provide dynamic deadlock detection functionalities. For example, the Windows Driver Verifier and Java HotSpot VM both can track the use of locks and detect cyclic lock dependences. Database systems, such as Berkeley DB, MySQL, Oracle, and PostgreSQL, use timeouts and WFGs to detect deadlock. The Linux kernel can perform simple deadlock detection whenever the `fcntl` system call is invoked. Havelund [8] describes a dynamic deadlock detection mechanism as an extension to NASA's Java PathFinder 2 model checking system [13]. This mechanism records lock operations executed by each Java thread and performs post-mortem analysis to detect potential deadlocks. Different from all these mechanisms, Pulse does not assume only lock-like resources and can detect more general forms of deadlock.

**Static deadlock detection.** Model checking is a formal verification technique that searches in a program's state space for possible errors, including deadlock. Example model checking systems include Bandera [4], VeriSoft

[7], SPIN [10], and Java PathFinder [13]. However, model checking large, complex software systems is still impractical due to the state explosion problem.

Microsoft suggests modelling multithreaded Win32 applications as Petri Nets and using their DLDETECT tool to statically analyze programs for potential deadlock [1]. Sun Solaris provides the Crash Analysis Tool (CAT) that helps users statically analyze system crash dumps to identify simple lock-induced deadlocks. Similarly, Linux supports the Non-Maskable Interrupt (NMI) watchdog, which periodically prints out system information that can help statically identify deadlock.

Recently, Engler et al. [5] proposed RacerX, a static tool for checking data races and deadlocks in large software systems. RacerX annotates source code of the system being checked, constructs the whole-system control flow graph, and searches within the graph for possible deadlocks. An important part of RacerX is to rank the detected deadlocks in terms of how likely they are to occur and filter out inaccurate deadlock warnings. RacerX can only detect deadlocks involving lock-like resources. In contrast, Pulse can detect more complex deadlocks involving consumable resources.

**Speculative execution.** Chang and Gibson [3] proposed a design for automatically transforming applications to perform speculative execution and issue hints for their future I/O read accesses. Fraser and Chang [6] improved this design by leveraging existing OS features to perform speculative execution. To ensure safety, they sever the ordinary relationships between speculative processes and their parents, and disallow speculative processes to execute potentially unsafe system calls. Pulse employs similar techniques to ensure safety.

## 3 Deadlock Detection with Pulse

In this section, we present an overview of Pulse (Section 3.1) and then describe its design in detail (Section 3.2–Section 3.4). Finally, we discuss how Pulse can be extended and its current limitations (Section 3.5). Section 4 presents our Linux implementation of Pulse.

### 3.1 Overview

Pulse exploits the observation that if a set of processes is deadlocked, the processes often sleep within the OS kernel, each waiting for events that can only be produced by another process in the set. Thus, when Pulse sees a set of processes blocked for a long time, it considers that a deadlock might have occurred.

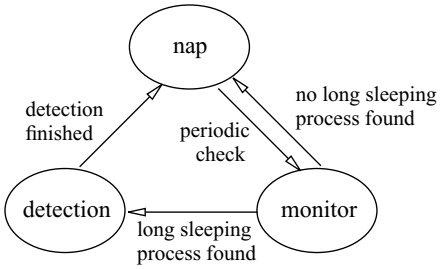


Figure 2: Pulse state transition diagram.

| Process $P_1$ | Process $P_2$ |
|---|---|
| lock(A) | lock(B) |
| lock(B) | lock(A) |
| unlock(A) | unlock(B) |

Figure 3: A circular lock example.

Pulse runs as a daemon process that can be in one of three modes: nap, monitor, and detection. Figure 2 is a state diagram that shows how Pulse transitions between these modes. For most of the time, Pulse is in the nap mode in which it sleeps in the OS kernel. Periodically, it awakens and enters the monitor mode to check if any process in the system has been asleep for a threshold amount of time, where the threshold value is a tunable parameter (e.g., five minutes). If no such processes are found, Pulse returns to the nap mode, thus incurring low overhead. Otherwise, Pulse enters the detection mode and performs deadlock detection for these sleeping processes. We show in Section 5.4 that a threshold as low as one minute introduces negligible performance overhead. However, if the threshold is too large, Pulse may miss certain deadlocks that can be broken by mechanisms such as timeouts, as we explain in Section 3.5.

Pulse uses the general resource graph model [9] to detect deadlock. To illustrate the idea, we use the code in Figure 3 as a running example. Suppose that the two processes execute their second `lock` statements simultaneously. Thus process $P_1$ blocks on lock $B$ and process $P_2$ blocks on lock $A$, and they deadlock. When Pulse detects that processes $P_1$ and $P_2$ have been asleep longer than the threshold, it enters the detection mode.

In the detection mode, Pulse identifies the events on which the sleeping processes are waiting (Section 3.2). In our example, process $P_1$ is waiting for lock $B$ to be free and process $P_2$ is waiting for lock $A$ to be free. For each sleeping process, Pulse constructs a *process node* in a general resource graph. For each event it identifies, Pulse constructs an *event node*. We use the term event node instead of resource node [9] to emphasize that a

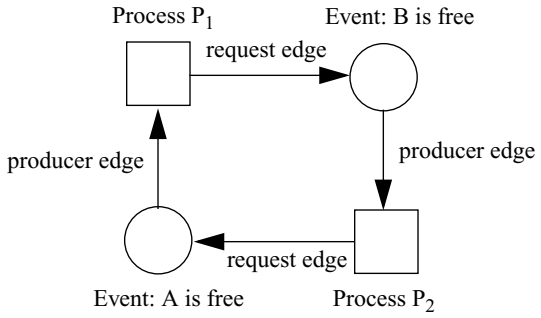

Figure 4: Resource graph for the circular lock example.

process often waits for an event involving a resource (e.g., a lock being free), as opposed to the resource itself (e.g., the lock). Pulse also constructs a *request edge,* directed from a process node to an event node, if the process is waiting for that event.

To discover dependences between the sleeping processes, Pulse uses speculative execution. For each sleeping process, Pulse forks a speculative process that executes ahead in its parent's program. Speculative execution allows Pulse to discover the events that a blocked process would produce if it were not blocked. In our example, Pulse discovers that process $P_1$ would unlock $A$ and process $P_2$ would unlock $B$, if they were not blocked. Thus Pulse constructs a *producer edge*, directed from the event that process $P_1$ creates, i.e., lock $A$ becoming free, to process $P_1$. Similarly, Pulse constructs a producer edge from the event node that $P_2$ creates, i.e., lock $B$ being free, to process $P_2$.

Figure 4 shows the graph that Pulse finally obtains. We use squares to denote process nodes and circles to denote event nodes. Since this graph contains a cycle, Pulse outputs that deadlock exists. It also outputs the entire graph to help developers debug the deadlock.

The general resource graph model allows us to detect whether a deadlock exists, and if so, which processes are involved in the deadlock and why they are deadlocked. To construct a general resource graph, our design needs to address the following questions:

1. How do we construct nodes in the graph? In other words, how do we identify the processes and events involved in a potential deadlock? (Section 3.2)
2. How do we construct edges in the graph? That is, how do we identify the dependences among the processes and events? (Section 3.3)

In the rest of this section, we describe how our design addresses these questions in detail.

## 3.2 Constructing nodes

When Pulse enters the detection mode, it has already identified a set of processes that have been asleep for a long time. These processes are potentially deadlocked; thus, they constitute the process nodes in the general resource graph.

The events on which these processes are blocked constitute the event nodes in the graph. To identify the events for which a sleeping process is waiting, Pulse requires all blocking system calls to be modified. Within these calls, we add new code to record the events for which the caller process is waiting. Pulse characterizes an event by a (*resource*, *condition*) pair. The *resource* field identifies the resource on which the process is blocked (e.g., a lock or a pipe). The *condition* field describes the condition about the resource for which the process is waiting (e.g., the lock being free, or the pipe being non-empty).

## 3.3 Constructing edges

Pulse models all resources as consumable resources. This resource model enables Pulse to detect deadlocks involving more than just single-unit reusable resources. However, it also causes Pulse to generate certain false positives, as we will see in Section 3.5.

There are two sets of edges that Pulse needs to construct in a graph: request edges and producer edges. The request edges can be constructed at the same time when Pulse constructs the event nodes. As discussed in Section 3.2, when Pulse identifies a sleeping process and its awaited events, it can construct a request edge from the process to each event for which it is waiting.

To construct the producer edges, Pulse uses speculative execution. For each sleeping process, Pulse forks a copy of the process, called a *speculative process*. The speculative process first creates the events awaited by its parent process, if this does not affect any other process. For example, if the parent is blocked on a busy lock, then the speculative process can set the lock variable (often a user-space memory location) to free within its own address space, not affecting any other process. On the other hand, if the parent has been waiting for an I/O, the speculative process is not allowed to create the awaited event because it could affect the state of other processes. Regardless of whether the events are created or not, the goal of Pulse here is to enable the speculative process to unblock and thus run ahead in its program. The next step for the speculative process is to return from the system call that caused its parent to sleep, pretending the call was successful, and then to resume its

parent's user-level code after the blocking system call, all within the speculative process's own context (i.e., the parent is unchanged and it continues to sleep).

The execution of a speculative process should be safe, i.e., it should not modify the state of any other process. The Unix fork mechanism implements copy-on-write, which naturally protects a speculative process from modifying the user-space memory state of its parent (and other processes). Some systems support forking a process (or thread) that shares the same address space with the parent. Pulse should avoid using fork in such a way and it should always invoke fork with copy-on-write enabled. Similar to Fraser and Chang [6], we do not allow a speculative process to write to the file system (thus no I/O writes) or deliver a signal to any other process. In Section 4, we discuss in detail the extra measures that our implementation takes to ensure the safety of the kernel state. These safety measures, however, may cause Pulse to produce false positives and/or false negatives of deadlock, as we explain next.

During the execution of a speculative process, Pulse records all the events produced by the process that could potentially unblock other processes. This requires modifying the system calls that counterpart the blocking system calls. For example, a write system call can block when trying to write to a full pipe (i.e., the pipe buffer is full), and can be unblocked only if a process reads data from the pipe. Thus, a pipe read system call counterparts a blocking pipe write system call. We modify all system calls that can unblock a process such that, if called by a speculative process, they record the resources being manipulated and the conditions being produced by the speculative process. As for the pipe example, the modified read system call would record a unique resource identifier for the pipe and some indicator representing that the pipe is being read. For each speculative process, Pulse maintains an *event buffer* that records the events the process produces during its execution. A speculative process adds an event to its event buffer only if the event is not already in the buffer. If the buffer is full, the process ignores the event (i.e., does not add it to the buffer). As we see in Section 5, a buffer of ten events is sufficient for all of our experiments.

A speculative process terminates if one of the following conditions is true:

1. It exits normally.
2. Its event buffer is full.
3. $T$ seconds have passed since the creation of the speculative process, where $T$ is adjustable by the user.

After all the speculative processes terminate, Pulse matches their produced events against the events awaited by the sleeping (parent) processes. If the speculative version of process $A$ produces an event that process $B$ is waiting for, then Pulse constructs a producer edge from this event to process $A$. A speculative process could produce events that are not awaited by any sleeping process. We do not include these events in the graph, because this reduces the graph size and does not affect the correctness of deadlock detection.

## 3.4 Putting it all together

The nodes and edges that Pulse constructs collectively form a general resource graph. If the graph contains cycles, Pulse outputs that a deadlock exists. To facilitate debugging, Pulse also prints out the entire resource graph. It is also possible to perform symbol table lookups such that the application programmer can map the graph to specific points in the code. Based on all this information and knowledge of the applications, the programmer could easily verify whether a deadlock indeed exists, and, if so, identify its cause.

Pulse requires OS kernel support (e.g., for speculative execution). Compared to user-space solutions, Pulse provides a general solution for all applications, thus freeing programmers from the burden of designing deadlock detection for each individual application. On the other hand, for applications that already have some deadlock detection built-in, they can use Pulse together with their own mechanisms to obtain the best coverage of deadlock.

## 3.5 Discussion

In this section, we first briefly describe our design plans for handling deadlocks involving spinning processes and deadlocks due to kernel bugs, but we leave a full evaluation of these designs as future work. We then discuss in detail the limitations of Pulse.

**Spin deadlocks.** A process can wait for synchronization events via spinning (a.k.a., busy-waiting). Although most commercial software puts a waiting process to sleep (possibly after spinning for a short period), spinning can be prevalent in scientific applications. To detect deadlocks involving spinning processes, we need to dynamically identify the spinning processes and the events for which they are waiting. We can achieve this by instrumenting synchronization libraries. After this information is obtained, Pulse can detect deadlock in the same way as it does for sleeping processes.

**Kernel deadlocks.** Applying Pulse to detect deadlocks due to kernel bugs is difficult, because allowing processes to speculatively execute within the kernel can cause unwanted changes to kernel structures and even crash the system. However, with the help of virtual machine technologies, such as VMware [2] or Xen [14], we could perform speculative execution on different virtual machines, making Pulse possibly applicable to detecting kernel deadlocks.

**Limitations of Pulse.** Pulse can output false positives, i.e., deadlocks that do not actually exist. There are three reasons why false positives can occur. First, because the execution of a speculative process must be safe, it cannot perform potentially unsafe operations, such as writing to a file. This could cause a speculative process to execute unrealistic program paths, making Pulse obtain incorrect dependences. Such false positives also exist in static detection tools because they often cannot identify unrealistic program paths statically.

Second, as we discussed in Section 3.3, Pulse models all resources as consumable resources. Thus it could construct more than one producer edge for resources (e.g., locks) that can be held and freed by only a single owner. The extra edges could cause Pulse to detect cycles that do not actually exist. Such false positives cannot occur in existing static and dynamic detection tools because they model only reusable resources.

Third, for systems with resources more complex than single-unit reusable resources, a cycle in a general resource graph is only a necessary, but not sufficient condition for deadlock [9]. For example, Pulse may detect a cycle when multiple processes block on a set of synchronization semaphores. However, a new process may later enter the system and perform an up operation on one of the semaphores, thus breaking the "deadlock". Such false positives are unique for Pulse; all the existing approaches do not consider resources more complex than single-unit reusable resources and thus do not have such false positives.

There are two types of deadlock that Pulse cannot detect (i.e., false negatives). First, some applications employ a timeout mechanism on operations that take too much time. For example, as we will see in Section 5, Pulse can detect a deadlock scenario due to cyclic pipe access dependences in the Apache web server. However, Apache can abort the pipe operations after they take longer than five minutes. The timeout effectively breaks the deadlock, although it also silently fails the client's request without providing any information about what has happened. Pulse could miss such deadlocks if its threshold value for entering the monitor mode is too large. However, such false negatives can be avoided by adjusting the threshold, possibly at the cost of impacting application performance.

The second type of false negative is due to Pulse's reliance on the future events it discovers. Pulse is able to detect deadlock because it can discover the future events that the sleeping processes could produce if awakened. However, if such events are unavailable, Pulse will not detect the deadlock. There are four scenarios in which the future events can be unavailable.

1. Such events do not exist in the application program. For example, the programmer forgets the unlock statements in the code in Figure 3.
2. Speculative processes do not run long enough to discover the events.
3. The event buffers fill up before speculative processes see the relevant events.
4. Speculative processes may execute unrealistic program paths that do not manifest the relevant events.

The first scenario is a fundamental limitation of Pulse. However, the second and third scenarios are not fundamental limitations; they can be avoided by increasing the run time and event buffer sizes of speculative processes. The reason for the fourth scenario is that Pulse does not allow speculative processes to execute potentially unsafe system calls. This scenario may be considered as a fundamental limitation of Pulse, if our implementation chooses to run speculative processes on the same operating system on which the normal processes run (which is what we do in this paper). However, we may avoid some cases of this scenario if each speculative process can run on a different operating system, e.g., using VMware [2] or Xen [14].

# 4 Implementation

In this section, we discuss how we implement the design described in the previous section. We implement Pulse by modifying Linux kernel version 2.6.8.1. We add a new system call that allows Pulse to be invoked from the user level.

## 4.1 Constructing process nodes

To construct the process nodes, Pulse needs to identify long sleeping processes. We add a flag, `was_asleep`, to each process's `task_struct`, which is set to false when the process is created. When Pulse enters the monitor mode, it scans the system for processes that satisfy the following three conditions:

- The process is asleep.

- The process was put to sleep by one of our modified system calls (see Section 4.2).
- The process's `was_asleep` flag is true, which indicates that the process was asleep when Pulse checked it last time.

If a process satisfies the first two conditions, but not the last one, Pulse sets the process's `was_asleep` flag to true. This flag is reset to false when the Linux scheduler switches this process to run, i.e., when it is awakened. For each process that satisfies all the three conditions, Pulse constructs a process node for it.

Some system daemon processes (e.g., automount) sleep in the kernel for a long time. If they satisfy the above conditions, Pulse will construct process nodes for them. Alternatively, we could implement a mechanism to disable the construction of nodes for these processes, if the user of Pulse (e.g., a system administrator) believes that these processes do not deadlock.

## 4.2 Constructing event nodes

To identify the events for which a sleeping process is waiting, we need to modify all system calls that can block. For the purposes of this paper, we have modified only three blocking system calls: `futex`, `write`, and `poll`. In each call, before putting the caller to sleep, we add code to construct the following two lists and store them in a structure pointed to by the caller process.

- A resource list, ($resource_1$, $resource_2$, …), where $resource_i$ is an integer that uniquely identifies a resource being manipulated by the system call.
- A condition list, (<$op_1$, $val_1$>, <$op_2$, $val_2$>, …), where the pair <$op_i$, $val_i$> encodes the condition about $resource_i$ that can unblock the caller process.

The `futex` and `write` system calls involve only one resource and thus their resource and condition lists consist of only one element. The `poll` system call, however, can operate on multiple file descriptors. Thus it may record more than one resource and condition. We now describe how we modify these three system calls.

**Futex.** Futex stands for fast user-space mutex. The `futex` system call is the basis of various synchronization primitives in the Native POSIX Thread Library (NPTL), which has been integrated in the recent versions of glibc. For the purposes of demonstrating Pulse, we consider NPTL's mutex and semaphore primitives in glibc 2.3.2. A thread acquiring a busy mutex or semaphore blocks via the `futex` system call. Each NPTL mutex or semaphore corresponds to a user-space memory address, which is passed to `futex` as an argument. Thus, in `futex` (before the caller is about to sleep), we add code to record this memory address, which uniquely identifies the resource on which the caller thread is blocked. The condition to unblock the caller, however, depends on the context in which the `futex` system call is invoked, and thus requires modifications to the NPTL library. Note that the applications using the library do not need to be modified.

For the NPTL function that acquires a mutex lock, (`pthread_mutex_lock`), we pass <*equal-to*, 0> as two extra arguments to the `futex` system call. They signify that the condition to unblock the caller thread is when the mutex value is zero (i.e., the mutex is free).

For the NPTL function that does a semaphore down operation (`sem_wait`), we modify it to pass <*greater-than*, 0> as two extra arguments to the `futex` system call. They signify that the condition to unblock the caller is when the semaphore value is greater than zero.

**Write.** Linux's `write` system call is a generic interface to a wide range of file systems. Our implementation currently considers only writes to pipes, which is implemented in the `pipe_writev` function. For a blocking pipe write, the caller process blocks if the pipe buffer is full. We add code in function `pipe_writev` (before the caller is about to sleep) to record the address of the pipe's inode structure, which uniquely identifies the pipe resource. The condition under which the caller unblocks is when another process reads data from the pipe. We use the pair <*and*, POLLOUT | POLLWRNORM> to encode this condition. The fields POLLOUT and POLLWRNORM are kernel-defined bit masks, denoting that writing now becomes unblocked. As we will see in Section 4.3, we also record similar bit masks in system calls that can unblock the write to denote the events produced by those calls. The *and* operator here helps match the blocked process with the process that can potentially unblock it: if the logical *and* of POLLOUT | POLLWRNORM and the bit mask produced by another process is true, then that process can unblock this process.

**Poll.** The `poll` system call takes a list of file descriptors and events as arguments. If none of the events has occurred for any of the file descriptors, the caller blocks, waiting for one of these events to occur. The events are represented by bit masks, similar to the ones we discussed above. We add code in the `poll` system call (before the caller is about to sleep) to construct a resource list. Each resource is a file descriptor, denoted by the address of its inode structure. We also record a condition list. Each condition is denoted by a <*op*, *val*> pair, where *op* equals *and*, similar to what we do for pipe writes. The *val* field is simply the event bit mask that the caller passed to the `poll` system call.

Based on the information recorded by the modified system calls, Pulse can identify what events a sleeping process is waiting for and create the event nodes for them. The three modified system calls allow us to demonstrate the ability of Pulse in this paper. In general, the same approach can be applied to modify other blocking system calls.

## 4.3 Constructing edges

As we discussed in Section 3.3, Pulse can construct request edges at the same time when it constructs the process and event nodes. Thus, in this section, we focus on constructing producer edges via speculative execution. We explain our implementation by following the flow of speculative execution in time.

**Creating speculative processes.** After Pulse identifies the long sleeping processes, it forks a speculative process for each of them. The Unix fork mechanism allows a speculative process to run in its own address space, not affecting the state of its parent. The difficulty, however, is that the existing fork implementation assumes its caller process to be the same as the one to be forked. What we need, instead, is a function that can fork an arbitrary given process. Our first attempt was to write such a function by modifying Linux's existing `do_fork` function such that it took one more argument: a `task_struct` pointer to the process to be forked. Soon we found that this approach would require us to rewrite many existing kernel functions. First, they all needed to take this one extra argument. Second, they all needed to be modified to operate on data structures of the specified process instead of those of the caller process. To avoid this tedious work, we designed an in-kernel fork mechanism that allows us to use the existing `do_fork` function with only slight modifications.

Figure 5 illustrates our in-kernel fork design. To fork process *P*, Pulse first switches *P* in (thus switching itself out), but to a predefined function. Within this function, Process *P* calls `do_fork` to create a copy of itself, *P'*. Finally, it switches the Pulse process back in, which then resumes execution from where it was left off. Similar to ordinary processes, the speculative process *P'* participates in Linux's normal scheduling.

To implement this design, we wrote a simple context switch function, similar to Linux's context switch function. To fork a process *P*, Pulse calls our context switch function, which switches process *P* in by saving the memory and register state of the Pulse process and loading the state of process *P*. With a normal context switch, process *P* would then resume its execution from the

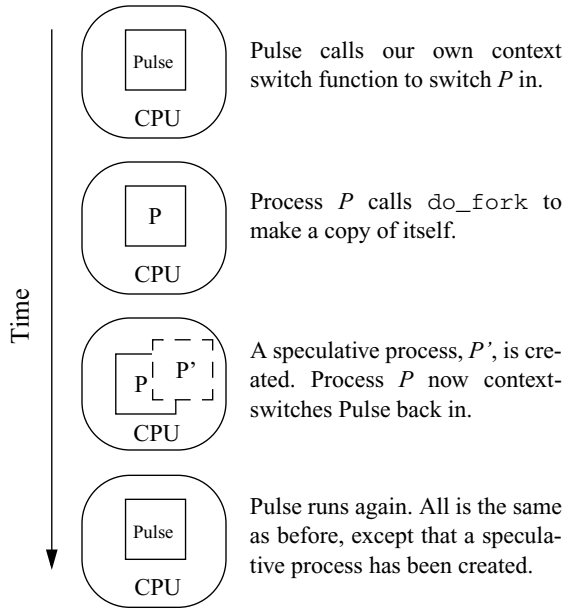


Figure 5: Illustration of in-kernel fork.

point where it was suspended previously. However, in our context switch function, we force it to enter an `in_kernel_fork` function that we added to the kernel. Within this function, process *P* calls `do_fork` to create a speculative process *P'*. It then calls our context switch function to switch the Pulse process in and switch itself out such that it goes back to sleep again.

Certain fields, such as the instruction and stack pointers, in the `task_struct` of process *P* may be changed by the `do_fork` call. Thus, after being switched back in, Pulse restores these fields. Our goal is to ensure that the parent process stays exactly the same before and after the creation of a speculative (child) process. Thus, we modify `do_fork` such that it does not link a speculative process to the children list of its parent. The speculative process does keep a pointer back to its parent, because it needs to know to which sleeping process it corresponds. To allow the speculative process to exit independently of its parent, we reset its parent to be the init process in the `do_exit` function. In `do_fork`, we also initialize the speculative process such that it does not send any signal to its parent when it exits. As such, the speculative process is completely independent of its parent and does not affect the parent in any way.

**Starting speculative processes.** After a speculative process is created, it participates in normal process scheduling. The normal fork semantics would make the speculative process resume the code in which the parent was suspended. Thus, the speculative process would resume the system call that blocked the parent process previously. However, allowing the speculative process to

execute in the kernel freely may cause changes to important kernel structures (e.g., the semaphore protecting a pipe's inode). Such changes should never occur because they could affect the normal execution of other processes, violating the safety property. To solve this problem, we force all speculative processes to execute a common function, `ret_from_spec_fork`, when they are scheduled to run the first time after creation. This is achieved by setting the initial program counter (EIP) value of each speculative process to be the address of this common function when the process is created.

In `ret_from_spec_fork`, a speculative process first creates the event awaited by its parent as follows. The speculative process checks what events its parent is waiting for by looking up the resource and condition lists maintained for the parent. For example, if the parent is blocked in a `futex` system call, then it must be waiting for the data at a user-space address (*resource*) to become equal to or greater than (*op*) a certain value (*val*). If *op* is *equal-to*, the speculative process writes *val* to the address identified by *resource* within its own address space. If *op* is *greater-than*, it writes *val* + 1 to that address. In fact, for the latter case, any value greater than *val* would be fine since the parent is not waiting for a specific value. However, it is possible that different values may have different meanings. For example, a semaphore's value often represents how many processes are allowed to enter a critical section simultaneously. Thus, our choice of setting the value to *val* + 1 is only heuristic; the parent process may indeed expect a different value at the given address, although it did not explicitly say so when it invoked the `futex` system call.

If the parent process is blocked in a pipe `write` or `poll` system call, the speculative process is not allowed to create the events awaited by the parent. This is because creating the events requires doing I/O on a pipe, which can affect normal processes that access the pipe.

Regardless of whether the events are created or not, the `ret_from_spec_fork` function then forces the speculative process to exit the system call in which its parent is blocked. This is done by jumping to Linux's `syscall_exit` routine and returning the value that represents success for the corresponding system call. Thus, after returning to the user-level code, the application program will have the illusion that the blocking system call has returned successfully. The speculative process then runs ahead in the program.

**Recording events.** During execution, a speculative process records all of the events that can awaken a sleeping process. Since our implementation considers only three blocking system calls, we only modify the counterpart system calls of these three calls. Similar to recording events for which a sleeping process waits, we record the events that a speculative process produces as resource and condition lists too. For the system calls we modify, these lists happen to both have only one element.

A process blocked in the `futex` system call can unblock only if another process calls `futex` with a FUTEX_WAKE argument. Thus the `futex` system call is the counterpart of itself. We add code in `futex` such that, when called by a speculative process to perform a wakeup, it records this event. We modify the corresponding NPTL library functions, such as mutex unlock (`pthread_mutex_unlock`) and semaphore up (`sem_post`), such that they pass the necessary information to allow `futex` to fill in the values for the *resource*, *op*, and *val* fields, which characterize the wakeup event produced by the speculative process. For example, if the speculative process invokes `futex` from `pthread_mutex_unlock`, it records that this process is producing an event, that is, the memory location of the mutex (*resource*) now obtains a new value (*val*). The *op* field is set to be the same as what is used for the counterpart system call (in this case, *equal-to*).

A process blocked in a pipe write system call can be unblocked if another process reads the pipe. Thus we add code in the `read` system call to record the read event. When called by a speculative process to read a pipe, the `read` system call records *resource* to be the address of the pipe's inode structure, *op* to be the same as what is used for the corresponding blocking pipe write call, i.e., *and*, and *val* to be POLLOUT | POLLWRNORM, which are exactly the bit masks for which the blocking write call is waiting.

For the `poll` system call, we modify two counterpart system calls: `write` and `writev`. For both of them, we record *resource* to be the address of the pipe's inode structure, *op* to be *and*, and *val* to be POLLIN | POLLRDNORM, which are kernel-defined macros representing that the pipe has data available to read.

To ensure safety, all of these system calls return immediately with a success code after they record the produced events. Thus, a speculative process calling these system calls does not really perform the wakeup and read/write operations that these calls normally do. Similar to Fraser and Chang [6], we modify all potentially unsafe system calls such that a speculative process returns immediately when entering these calls. In this way, any state change made by a speculative process is contained within itself, not affecting any other process.

**Constructing producer edges.** A speculative process terminates according to the conditions in Section 3.3. We limit the lifetime of a speculative process to be less

than one second. After all speculative processes terminate, Pulse matches their produced events against those awaited by the sleeping processes. If two events have the same *resource* and *op* values, then Pulse applies the operation represented by *op* on the *val* fields of the two events. For example, if *op* is *and*, then Pulse does a logical *and* on the two *val* fields. A result of true indicates that the speculative process produces an event for which the sleeping process is waiting. Thus Pulse constructs a producer edge from the node that represents the event to the node that represents the parent process of the speculative process.

## 4.4 Summary

The major components of our implementation are the in-kernel fork and modification of the blocking system calls and their counterpart calls. The in-kernel fork implementation is the most involved part in our coding; however, our code is small and highly efficient, consisting of only 94 lines of C code, 47 lines of inline assembly, and 7 lines of assembly. Modifying the system calls is easy since it simply involves identifying the corresponding resources and conditions, and recording them in a per-process structure (~160 bytes). For this paper, we have modified only three blocking system calls and their counterpart calls. The same methodology, however, can be easily applied to modify other system calls.

# 5 Evaluation

We apply Pulse against deadlocked solutions to the classical dining-philosophers and smokers problems, and a well-known deadlock scenario in the Apache web server version 2.0.49. This section describes how Pulse works for these different deadlock cases and evaluates its overhead. The evaluation is performed on an IBM xSeries 445 eServer with eight Intel Xeon 2.8 GHz processors and 32 GB memory. We configure Pulse to transition from nap mode to monitor mode with a default value of every five minutes, unless otherwise mentioned. We set the event buffers of speculative processes to store at most ten events, and every speculative process exists in the system for at most one second. Our results show that Pulse can detect deadlocks caused by incorrect ordering of lock acquisitions, as well as deadlocks involving synchronization semaphores and pipes, which, to the best of our knowledge, no existing tools can detect. Furthermore, Pulse generates no false positives and negatives of deadlock throughout our experiments.

```
while (1) {
    think();
    lock(fork[i]);                  // take left fork
    lock(fork[(i + 1) % 5];         // take right fork
    eat();
    unlock(fork[i]);                // put left fork
    unlock(fork[(i + 1) % 5]);      // put right fork
}
```

Figure 6: The code of philosopher i.

## 5.1 The dining-philosophers problem

Figure 6 shows one incorrect solution to the dining-philosophers problem. We implement the locks as NPTL mutexes. The lock routine is implemented using the `pthread_mutex_lock` function and unlock using `pthread_mutex_unlock`. In Figure 6, suppose that all five philosophers take their left forks simultaneously. Then they will all block on their right forks, and there will be a deadlock. We choose this problem as an example of deadlock caused by incorrect use of mutual exclusion locks. All existing dynamic and static tools target this type of deadlock.

When Pulse enters the detection mode, it identifies that each philosopher process is waiting for an address (corresponding to its right fork) to contain a value zero (corresponding to the release of the right fork). It then forks a speculative process for each philosopher. The speculative processes first unlock their right forks within their own address spaces, and then execute ahead. During execution, they discover that they produce the unlock events that could unblock their left neighbors. Finally, Pulse constructs the resource graph shown in Figure 7. This graph contains a cycle, indicating the existence of a deadlock.

## 5.2 The smokers problem

The smokers problem is a classic example of using semaphores for synchronization, instead of mutual exclusion. The static deadlock detection tool RacerX [5] specifically ignores checking deadlocks involving such semaphores. With speculative execution, Pulse is able to detect such deadlocks.

Figure 8 shows a solution to the smokers problem that can deadlock. The processes share four binary semaphores: tobacco, paper, matches, and order. The semaphores are implemented using the NPTL library. A P operation is implemented using the `sem_wait` func-

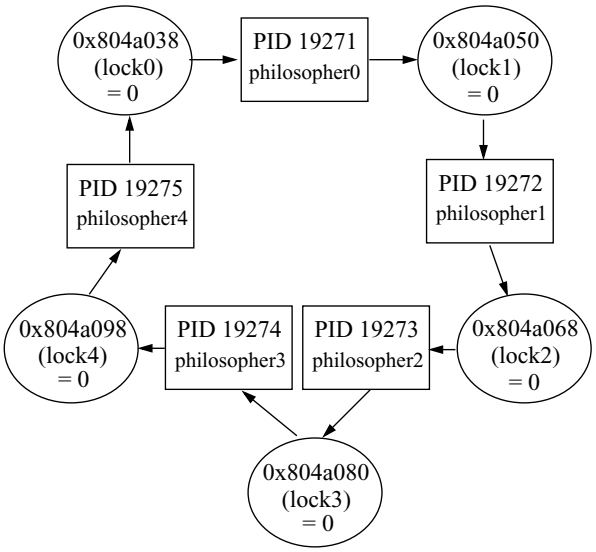


Figure 7: Resource graph for the dining-philosophers problem. The hex numbers are virtual addresses corresponding to the locks.

```
smoker 1              smoker 2              smoker 3
while (1) {           while (1) {           while (1) {
  P(tobacco)            P(paper) // block     P(matches)
  P(paper) // block     P(matches)            P(tobacco) // block
  V(order)              V(order)              V(order)
}                     }                     }

    agent
    while (1) {
      P(order) // block
      V(one of tobacco, paper, matches at random)
      V(one of the three at random but not above)
    }
```

Figure 8: A deadlocked solution to the smokers problem.

tion, and V is implemented using `sem_post`. In Figure 8, suppose that the agent releases tobacco and matches, smoker 1 grabs the tobacco, and smoker 3 grabs the matches. Then smoker 1 will block in P(paper), waiting for the address corresponding to paper to have a value greater than zero. Similarly, smoker 2 will block in P(paper), smoker 3 in P(tobacco), and the agent in P(order). Thus all processes will deadlock.

By speculatively unblocking smoker 1, Pulse observes an event corresponding to V(order), which matches the event awaited by the agent. Thus Pulse constructs a producer edge from an event node representing V(order) to smoker 1's node. Then the speculative process executes P(tobacco) again. Tobacco is not available any more, so this speculative process blocks again. For smoker 2, after the speculative process is unblocked from P(paper), it blocks immediately in P(matches), and

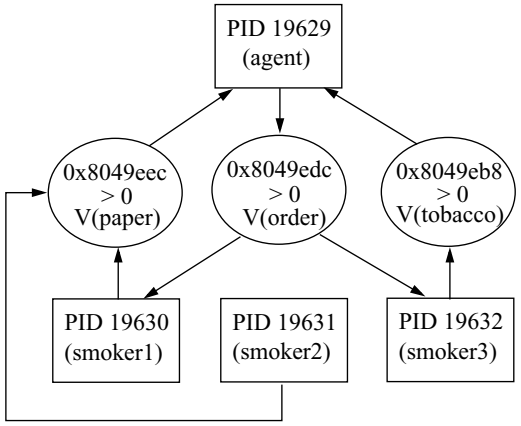


Figure 9: Resource graph for the smoker's problem. The hex numbers are virtual addresses corresponding to the semaphores.

thus it does not produce any event that could unblock another process. So the process node of smoker 2 has no incoming producer edges. Smoker 3 executes similarly to smoker 1.

Suppose that after the agent is speculatively unblocked, it executes V(tobacco) and V(paper). Then Pulse will discover producer edges from both the event nodes, corresponding to V(tobacco) and V(paper), to the agent. Figure 9 shows the final graph we obtain. This graph contains two cycles, agent → V(order) → smoker 1 → V(paper) → agent, and agent → V(order) → smoker 3 → V(tobacco) → agent, indicating the existence of deadlock.

## 5.3 Apache web server deadlock

We have reproduced a well-known deadlock situation in Apache 2.0.49 with the prefork Multi-Processing Module (MPM). A full description of this bug (number 22030) can be found in the Apache bug database (http://nagoya.apache.org/bugzilla/). This bug is widely referred to as the Apache oversized stderr buffer denial-of-service vulnerability by security companies like Symantec [12]. In the extreme case, this bug can cause an entire web site to stop functioning.

To reproduce the deadlock effect of this bug, we create a Perl CGI script, which first writes 4097 bytes to stderr and then some arbitrary data to stdout. When a client requests this CGI script from a remote browser, a deadlock happens on the server side. This deadlock involves two processes: the CGI script's process and the httpd process that handles the CGI request. The reason for this deadlock is because Apache redirects stderr and stdout of the CGI script to two pipes. The pipe buffer size in Linux is 4096 bytes; a blocking write to a pipe

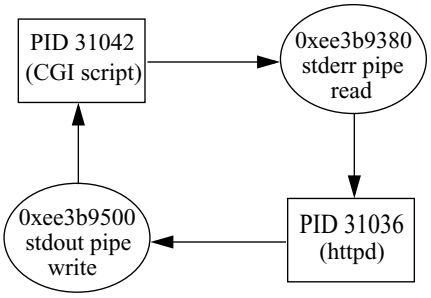


Figure 10: Resource graph for the Apache deadlock. The hex numbers are addresses of the corresponding pipe inode structures. We label the events in words although Pulse encodes them abstractly.

blocks if the write data exceeds 4096 bytes. Thus when the CGI script writes 4097 bytes to stderr, it blocks in the `write` system call, waiting for the httpd process to read data out of the pipe. Meanwhile, the httpd process is blocked in the `poll` system call, waiting for the CGI script to write data to the stdout pipe.

No existing techniques can detect this deadlock; with Pulse, we successfully detect it. Figure 10 shows the resource graph constructed by Pulse. This graph contains a cycle, indicating the existence of deadlock. It is worth noting that Apache has a timeout mechanism that allows it to fail the CGI request after about five minutes, thus breaking the deadlock. However, Apache provides no debugging information for the deadlock; it even has no idea that a deadlock has occurred. When this deadlock is triggered simultaneously from multiple sites, it can effectively become a denial-of-service attack.

## 5.4 Performance Overhead

We evaluate three aspects of Pulse's performance overhead: overhead of the modified system calls, overhead of periodic checking, and overhead of deadlock detection using speculative execution.

**System call overhead.** We write a microbenchmark for each of the three blocking system calls we modify. Our goal is to measure the overhead that our modified system calls introduce to correct, non-deadlocked applications. Since the additional code in our modified system calls that counterpart the blocking calls is executed only by speculative processes, we do not measure the overhead of these counterpart calls.

Our microbenchmark for the `futex` system call repeatedly invokes `futex` for a total of one million times. The arguments passed to `futex` are chosen such that the microbenchmark executes the new code we add, but it does not block (thus allowing the microbenchmark to repeatedly invoke the system call). We run the microbenchmark using the modified and unmodified `futex` call, and compare the average time taken per call. We run similar microbenchmarks for `write` and `poll`. Compared to the unmodified versions, the slowdowns of our modified system calls are: 0.2% for `futex`, 0.9% for `write`, and 1% for `poll`. These slowdowns are small, and most importantly, they occur mostly when a process is about to block.

**Periodic checking overhead.** We evaluate the performance overhead that Pulse introduces to correct, non-deadlocked applications. This overhead comes from Pulse's periodic activities of transitioning from nap to monitor mode and checking if it needs to enter the detection mode. For correct applications, Pulse does not enter the detection mode—after the periodic checking, it returns back to the nap mode. Our experiments show that, for a system with 100 threads, Pulse takes 0.29 seconds to transition from nap to monitor mode, scan all the threads to determine if it needs to enter the detection mode, and finally transition back to the nap mode. This time stays almost constant as we create more threads in the system, and only increases to 0.32 seconds when the system has a total of 2000 threads.

To measure how much performance overhead Pulse introduces to normal applications over time, we run Apache Bench from a remote machine to stress test our server running Apache 2.0.49. We set Apache Bench to perform a total of five million HTTP requests and send 1000 simultaneous requests at any time (thus keeping the server busy). The entire test takes about 30 minutes to complete. During this period, without Pulse running, Apache Bench obtains throughput of 2689 requests per second. We then run Pulse in the background, and set it to periodically transition from nap to monitor mode every one minute. Apache Bench now obtains throughput of 2684 requests per second, which is almost the same as the throughput obtained without Pulse running. These results show that Pulse has negligible impact on the performance of applications that do not deadlock.

**Deadlock detection overhead.** We measure the time Pulse takes to detect a deadlock, which is the duration from the time Pulse enters the detection mode to the time it finishes the detection and prints out the results. We obtain the following results: it takes Pulse 2.1 seconds to detect the deadlock in the dining-philosophers problem, 1.7 seconds in the smokers problem, and 1.5 seconds in the Apache web server. We also run these three benchmarks together such that they all deadlock at the same time. We see that Pulse can construct a general resource graph that has three subgraphs, each corresponding to one of the deadlock scenarios, and the entire detection only takes three seconds.

## 6 Conclusion

Deadlock can occur in any concurrent system and is often difficult to debug. Existing deadlock detection techniques are either impractical for large software systems or over-simplified in their assumptions about deadlock-sensitive resources. In this paper, we propose Pulse, a novel operating system mechanism that dynamically detects deadlock in user applications.

Pulse runs as a system daemon. Periodically, it identifies long sleeping processes and the events they are waiting for. For each of these processes, Pulse forks a speculative process, which executes ahead in its parent's program. Speculative execution enables Pulse to discover dependences among the sleeping processes. Based on this information, it constructs a general resource graph. If the graph contains cycles, Pulse outputs that deadlock exists. It also prints out the entire graph to help application developers identify causes of the deadlock.

Our evaluation demonstrates that Pulse can detect various types of deadlock, including those involving consumable resources, which no existing tool can detect. Our results show that Pulse can detect deadlock quickly and that it introduces little performance overhead to normal applications that do not deadlock. For application developers, Pulse can be viewed as another tool to add to the deadlock detection toolbox. When Pulse and the existing tools are used together, they can provide the best coverage of deadlocks.

## Acknowledgments

We thank the anonymous reviewers for their comments and suggestions on the early draft of this paper. We thank David Becker and Jaidev Patwardhan for their help with the experiments. This work is supported in part by the US National Science Foundation (CCR-0312561, EIA-9972879, CCR-0204367, CCR-0208920, and CCR-0309164), Intel, IBM, Microsoft and the Duke University Graduate School. Sorin is supported by a Warren Faculty Scholarship.

## References

[1] Ruediger R. Asche. Putting DLDETECT to Work. MSDN Library Technical Articles, Microsoft Corporation, January 1994.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, pages 164–177, October 2003.

[3] Fay Chang and Garth A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, pages 1–14, February 1999.

[4] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Puasuareanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 1999.

[5] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlock. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, pages 237–252, October 2003.

[6] Keir Fraser and Fay Chang. Operating System I/O Speculation: How Two Invocations Are Faster Than One. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 325–338, June 2003.

[7] Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of The 24th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 174–186, January 1997.

[8] Klaus Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *Proceedings of the 7th SPIN Workshop*, pages 245–264, August 2000.

[9] Richard C. Holt. Some Deadlock Properties of Computer Systems. *ACM Computing Surveys*, 4(3):179–196, September 1972.

[10] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[11] Mukesh Singhal. Deadlock Detection in Distributed Systems. *IEEE Computer*, 22(11):37–48, November 1989.

[12] Symantec. *Symantec NetRecon™ 3.6 Security Update 6 Release Notes*. Symantec Corporation, August 2003.

[13] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Journal of Automated Software Engineering*, 10(2):203–232, April 2003.

[14] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 181–194, December 2002.