Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution

Tong Li\textsuperscript{1}, Carla S. Ellis\textsuperscript{1}, Alvin R. Lebeck\textsuperscript{1}, and Daniel J. Sorin\textsuperscript{2}

\textsuperscript{1}Department of Computer Science
Duke University
{tongli,carla,alvy}@cs.duke.edu

\textsuperscript{2}Dept. of Elec. and Comp. Engineering
Duke University
sorin@ee.duke.edu
Motivation

- Deadlock is a potential problem for all multithreaded programs.
- Existing detection techniques have limitations.
- Goals
  - Increase the types of deadlocks that can be detected.
  - Provide insights into cause of deadlock.
Limitations of Existing Techniques

- **Dynamic deadlock detection**
  - **Timeouts**
    - Inaccurate, no insight about cause of deadlock
  - **Wait-for-graphs (WFGs)**
    - General resource graphs with single-unit reusable resources
    - Often applied to lock-like resources

- **Static deadlock detection**
  - **Model checking**
    - Accurate, but state space too large
  - **RacerX (Engler and Ashcraft SOSP 2003)**
    - Practical, but only lock-like resources

- Both WFGs and RacerX consider only lock-like resources
Beyond Locks

- Need to handle non-lock-like (consumable) resources
- Why is it challenging?
  - Consumable resources have no owners
    - Pipes, synchronization semaphores, etc.
  - Any process could be a producer at some future time
    - Any process could write to a pipe or “up” a semaphore

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(sem) // block</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>V(sem)</td>
<td>V(sem)</td>
<td></td>
</tr>
</tbody>
</table>

- Knowing only the present state is not enough for identifying all dependences!
The Big Idea

- We need to look into the future
  - What would process X do if it were not blocked?
  - Would it unblock process Y in the future?

- If we can answer these questions, then we know how processes depend on each other

- Could use static tool, but state space explosion, variable aliasing, etc.

- We use dynamic scheme to look into the future
Introducing Pulse

- Speculatively unblock each blocked process
- Discover dependences by running ahead
- Construct general resource graph with consumable resources

Venn diagram of deadlocks detectable by static tools, WFG-based dynamic tools, and Pulse

- Pulse can detect deadlocks that the other tools cannot
Outline

- Motivation
- Overview of Pulse
- Design
- Implementation
- Evaluation
- Conclusion
Overview of Pulse

- Features: Dynamic, speculative execution, general resource graph
- Pulse runs as a daemon process
- Three modes
  - Nap: sleeps in kernel
  - Monitor: looks for long-sleeping processes/threads
  - Detection
    - Long-sleeping processes are potentially deadlocked
Detection Mode

- Identify events long-sleeping processes are waiting for
  - E.g., semaphore up: V(sem)

- Fork each process to create a speculative process

- Unblock speculative process
  - E.g., “up” the semaphore in its own address space

- Record events generated by speculative processes
  - E.g., all semaphore up operations

- Construct general resource graph and check for cycle
Example: Smokers Problem

- Three smokers, one agent
- Three ingredients: paper, tobacco, matches
- Each smoker has one ingredient, but needs two more
- Agent puts out two at a time
- One smoker gets them and signals agent to continue

<table>
<thead>
<tr>
<th>Smoker 1</th>
<th>Smoker 2</th>
<th>Smoker 3</th>
<th>Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>while (1) {</code></td>
<td><code>while (1) {</code></td>
<td><code>while (1) {</code></td>
<td><code>while (1) {</code></td>
</tr>
<tr>
<td>P(tobacco)</td>
<td>P(paper) // block</td>
<td>P(matches)</td>
<td>P(order) // block</td>
</tr>
<tr>
<td>P(paper) // block</td>
<td>P(matches)</td>
<td>P(tobacco) // block</td>
<td>V(one of tobacco, paper, matches at random)</td>
</tr>
<tr>
<td>V(order)</td>
<td>V(order)</td>
<td>V(order)</td>
<td>V(one of the three at random but not above)</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

- Semaphores for synchronization, not mutual exclusion
Constructing Process Nodes

- Enter detection mode after all blocked for a long time
- Construct a *process node* for each long-sleeping process

```plaintext
Smoker 1
while (1) {
    P(tobacco)
    P(paper) // block
    V(order)
}

Smoker 2
while (1) {
    P(paper) // block
    P(matches)
    V(order)
}

Smoker 3
while (1) {
    P(matches)
    P(tobacco) // block
    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)
}
```
Constructing Event Nodes

Construct an *event node* for the event each process is waiting for.

- **Smoker 1**
  ```
  while (1) {
    P(tobacco)
    P(paper) // block
    V(order)
  }
  ```

- **Smoker 2**
  ```
  while (1) {
    P(paper) // block
    P(matches)
    V(order)
  }
  ```

- **Smoker 3**
  ```
  while (1) {
    P(matches)
    P(tobacco) // block
    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)
  }
  ```
Constructing Request Edges

- Construct *request edge* from process node to event node

```plaintext
Smoker 1
while (1) {
  P(tobacco)
  P(paper) // block
  V(order)
}

Smoker 2
while (1) {
  P(paper) // block
  P(matches)
  V(order)
}

Smoker 3
while (1) {
  P(matches)
  P(tobacco) // block
  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)
}
```

Diagram:
- Smoker 1
- Smoker 2
- Smoker 3
- Agent

Events:
- V(paper)
- V(order)
- V(tobacco)

Arrow from Smoker 2 to Smoker 1: waiting for V(paper)
Constructing Producer Edges

- Speculatively execute processes ahead
- Smoker 1 produces \( V(\text{order}) \), agent produces \( V(\text{paper}) \)
- Construct *producer edge* from event to process node

<table>
<thead>
<tr>
<th>Smoker 1</th>
<th>Smoker 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (1) {</td>
<td>while (1) {</td>
</tr>
<tr>
<td>\hspace{1em} P(\text{tobacco}) &amp; block \text{V(\text{order})}</td>
<td>\hspace{1em} P(\text{paper}) // block \text{P(\text{matches})} \text{V(\text{order})}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Smoker 3</th>
<th>Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (1) {</td>
<td>while (1) {</td>
</tr>
<tr>
<td>\hspace{1em} P(\text{matches})&amp; block \text{P(\text{tobacco})// block \text{V(\text{order})}</td>
<td>\hspace{1em} P(\text{order}) // block \text{V(\text{one of tobacco, paper, matches at random})} \text{V(\text{one of the three at random but not above})}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
A cycle indicates potential deadlock
Processes: represented by PIDs
Events: \((\text{resource}, \text{condition}) \rightarrow (\text{semaphore address}, > 0)\)
Design Issues – Constructing Nodes

- Process nodes
  - Those processes asleep for a long time

- Event nodes
  - Need to know the events for which a process is waiting
  - Modify all blocking system calls to record the events
  - Modified calls record events in a per-process structure
Design Issues – Constructing Edges

- **Request edges**
  - Constructed together with event nodes

- **Producer edges**
  - Need to know what events a process can produce
  - Modify all counterpart system calls (calls that unblock the blocking ones)
  - Record events in an event buffer until the speculative process terminates (normal exit, full buffer, timeout)

![Diagram of process interaction with event nodes and system calls](image-url)
Safe Speculation

- Cannot change state of any other process
  - No change to memory state of other processes
  - No writes to file system (including I/O devices)
  - No signals to other processes

Solution:
- Similar to Fraser and Chang USENIX’03
- Fork with copy-on-write enabled
- Modify unsafe system calls (e.g., write, kill)
  - Speculative processes record the events they produce
  - Then return immediately
Limitations of Pulse

- **False positives**
  - Speculation may run unrealistic program paths
  - May have wrong cycles if resources are not consumable
  - For resources that are not single-unit reusable, a cycle is only necessary but not sufficient

- **False negatives**
  - Speculative processes miss relevant events
    - Programmer forgot V(sem)
    - Speculation not long enough
    - Event buffer full
    - Unrealistic program paths
  - Self-breaking mechanisms with timeouts
Outline

- Motivation
- Overview of Pulse
- Design
- Implementation
- Evaluation
- Conclusion
Implementation

- Linux kernel 2.6.8.1
- Modified three blocking system calls
  - futex, write (to pipe), and poll
- Modified four counterpart system calls
  - futex, read, and write/writev
- Our approach can be applied easily to modify other syscalls
- Forking an arbitrary process: fork($P$)
  - Existing fork copies the caller process
  - Adding a process argument to existing fork doesn’t work
  - We use existing fork with only slight modifications
Forking Blocked Processes

1. To fork process $P$, first switch $P$ in using our own context-switch function

2. $P$ calls the usual fork routine to create speculative process $P'$

3. $P'$ fakes the awaited event, calls syscall_exit with success, and resumes $P'$'s program

4. Finally, $P$ switches the Pulse process back in and then $P$ goes back to sleep
Evaluation

- All experiments on an 8-processor IBM x445 eServer
- Fork was the most involved part in coding
  - But only one-time effort
  - Code is small and efficient
    - 94 lines of C, 47 lines of inline assembly, 7 lines assembly
- Three deadlock benchmarks
  - Smokers Problem (discussed earlier)
  - Dining-philosophers Problem
  - Apache 2.0.49
Dining Philosophers Problem

- Deadlock if all philosophers take left forks at same time

**Philosopher** $i$

```c
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
}
```

- All existing tools target this type of deadlock
Dining Philosophers Problem

- Hex numbers are virtual addresses of lock variables
- Squares: processes, circles: events, edges: dependences

- PID 19271 philo0
  - 0x804a038 (lock0) = 0
  - 0x804a050 (lock1) = 0

- PID 19272 philo1
  - 0x804a068 (lock2) = 0

- PID 19273 philo2
  - 0x804a080 (lock3) = 0

- PID 19274 philo3

- PID 19275 philo4
  - 0x804a098 (lock4) = 0
Apache Deadlock

- Apache 2.0.49 with prefork Multi-Processing Module (MPM)
- Two-process deadlock:
  - A CGI script’s process blocks when writing to stderr pipe
  - An httpd process blocks when reading from stdout pipe
  - Each can be unblocked only by the other
- Not detectable by WFGs and RacerX
- Pulse successfully detects it
- Hex numbers are addresses of pipe inode structures

PID 31042 (CGI script) → 0xeee3b9380 stderr pipe read

0xeee3b9500 stdout pipe write

PID 31036 (httpd)
Performance Overhead

- Overhead of the modified system calls
  - Average slowdown per call: futex 0.2%, write 0.9%, poll 1%

- Overhead of periodic checking
  - Nap to monitor, and back to nap (5-min check interval): ~0.3 seconds for 2000 processes
  - Apache Bench (1-min interval): throughput difference < 0.2% w/ and w/o Pulse

- Overhead of deadlock detection
  - Less than 3 seconds from detection to finish
Conclusion

- Deadlock is potential problem for all multithreaded programs
- Existing detection tools focus on lock-like resources
- Pulse: dynamic, speculation, general resource graph
- Can detect deadlocks with non-lock-like resources
  - E.g., synchronization semaphores, pipes
- Linux implementation

Evaluation
- Dining-philosophers, smokers, Apache
- Negligible performance overhead