Reproducible concurrency for NPTL based applications

Praveen Srinivasan
May 13th 2005
Master’s Project defense

Committee
Dr. Jerry James, Chair
Dr. Douglas Niehaus, Member
Dr. David Andrews, Member

University of Kansas
Presentation Outline

• Introduction
• Background
• Recording
• Replay
• Evaluation
• Conclusion
• Future work
Presentation Outline

• Introduction
• Background
• Recording
• Replay
• Evaluation
• Conclusion
• Future work
Introduction

• Debugging single-threaded programs
  • Program is repeatedly traced - GDB
  • Focus on specific parts of the program where the bug is
  • Generally known as “cyclic debugging”
  • Assumption – repeated executions are identical

• Debugging multi-threaded programs
  • Available features more suitable for cyclic debugging
  • Main problem – repeated executions not identical
  • Affected by several non-deterministic factors
  • Need faulty execution reproduction to identify bugs
Introduction

• Objectives
  • Identify execution path and reproduce faulty execution
  • Make program execution deterministic
    – cyclic debugging techniques can then be applied
• Focus on POSIX threads on uni-processor Linux systems
• Proposed Solution – Two phases
  • Recording
    – Log necessary data during experimental run
  • Replay
    – Reproduce execution within GDB
      – Use recorded data to set appropriate replay breakpoints
Presentation Outline

• Introduction
• Background
• Recording
• Replay
• Evaluation
• Conclusion
• Future work
Background

- POSIX thread Library
  - 1:1 thread library model
    - Each thread is mapped to a kernel process
    - Kernel takes care of scheduling
  - LinuxThreads – old implementation
  - NPTL
    - Latest implementation
    - Requires 2.6 kernel series
    - Faster Thread Creation/Destruction
    - Futexes
Background

• Pthreads
  • Individual PID, user and kernel mode stacks
  • Shared address space
  • Scheduled by kernel scheduler

• Threads created using clone system call

• TGID – Thread Group ID
  • TGID = Parent’s PID for Pthreads
  • TGID = PID for “real” processes

• TGID List
  • List of all threads created by a process
  • Used for group stop and exit
Background

• DataStreams Kernel Interface
  • Framework to collect status and performance related data from kernel

• Instrumentation points
  • Hierarchy
    – Family
    – Events, Counters and Histograms

• Device driver interface
  • Select subset of events, counters or histograms to be logged
Presentation Outline

• Introduction
• Background
• Recording
• Replay
• Evaluation
• Conclusion
• Future work
Recording

- Execution path – set of all instructions executed
- Factors affecting execution path
  - Scheduling decisions – non-deterministic
  - Signals – non-deterministic
  - Inputs (network, system and user) – variable
Recording

- Sample multi-threaded program – two threads

```
Program 3.1 A simple program to illustrate the importance of recording interleaving information.

1: #include <stdio.h>
2: #include <pthread.h>
3: 
4: int shared_var = 0;
5: 
6: int thread_func(void) {
7:   shared_var += 5;
8: }
9: 
10: int main() {
11: 
12:   pthread_t t1;
13: 
14:   pthread_create(t1, NULL, thread_func, null);
15: 
16:   shared_var += 3;
17: 
18:   printf("In main: shared_var is %d", shared_var);
19: 
20:   return 0;
21: }
```


Recording

- Execution Path 1
  - Output printed: 8

- Execution Path 2
  - Output printed: 3
Recording

• **Thread schedule**
  - (thread identifier, stop address) pair
  - Example: (main thread, 14)
    – Main resumed execution and stopped after line 14 in a schedule

• **Schedule Order (SO)** – an ordered set of thread schedules
  - Ordered by time
  - Example: { (main thread, 18), (thread 1, exit), (main thread, exit) }

• **SO uniquely identifies an execution path**
  - If inputs supplied to the program are the same
  - If the effects due to signals are reproduced
    – Addressed by Ram’s project
Recording

Thread 1

Thread 2

User Mode

Thread 1

Kernel Mode

Thread 2
Recording

- Transition due to interrupt
  - Instruction at transition address was already executed
  - Breakpoint should be set at resumption address

- Transition due to system call
  - Instruction at transition address is a system call
    - transition should not be allowed for most system calls (especially sys_futex)
    - breakpoint should be placed at transition address
  - Exceptions: clone, exit
    - effects should actually take place
    - breakpoint should be placed at resumption address
Recording

- Record both during context switch
- Sample SO
  - CS <Thread 1, T11, R11>
  - CS <Thread 2, T22, R22>
  - CS <Thread 1, T12, R12>
- Resumption address
  - User-space return address - found in kernel stack
- Transition address
  - Address of the previous instruction – but length can vary
  - Manual lookup required
- Record only return addresses
Recording

- Resumption point – not determined by return address alone
  - (address, count) pair
- Basic blocks – set of instructions with single point of entry and exit with no branches in between
  - GCC’s block profiling feature can be used
- Final SO
  - CS <thread 1, (R11, basic block count)>
  - CS <thread 2, (R22, basic block count)>
  - CS <thread 1, (R12, basic block count)>
Recording

- Need to identify pthread processes in kernel
  - Log context switch events of pthread processes alone
- New variable `pthread_flag` in `task_struct`
- Set `pthread_flag` during clone
  - New `clone` flag in both GLIBC and kernel
- Set `pthread_flag` for main thread when it creates first thread
Recording

• Maintaining basic block count
  • Function \texttt{__bb_trace_func} called for every basic block entry
  • New function \texttt{__bb_new_trace_func} increments basic block count
  • Work done by Satya at ITTC

• Modifications
  • New variable \texttt{bb\_count} in \texttt{pthread\_struct}
  • New functions: \texttt{pthread\_incr\_bb\_count} and \texttt{pthread\_get\_bb\_count}
  • \texttt{__bb\_new\_trace\_func} now calls \texttt{pthread\_incr\_bb\_count}

• Accessing basic block count during context switch
  • New variable \texttt{bb\_count\_addr} in \texttt{task\_struct}
  • Update \texttt{bb\_count\_addr} during \texttt{set\_thread\_area} for main thread
  • Update \texttt{bb\_count\_addr} during \texttt{clone} for other threads
Recording

- **Return address**
  - Stored in trap frame
  - Trap frame stored in kernel stack

- **Kernel stack**
  - \texttt{THREAD\_SIZE}: stack size
  - Stored from higher to lower addresses
  - \texttt{task\_struct} stored in bottom

- **Trap frame can be obtained using**
  \[
  (\text{THREAD\_SIZE} + \text{current} \rightarrow \text{thread\_info}) - 1
  \]
Recording

- Virtual system calls
  - Added to 2.6.x kernels for performance improvement
  - Kernel page `vsyscall` mapped to all user processes
- Problem: return address in trap frame points to `SYSENTER_RETURN` in `vsyscall`
  - Breakpoint cannot be set
- Solution: user stack
  - Follow frame pointer address in trap frame
Recording

Sample SO

```xml
<ENTITY number="1" tag="2540" ... time_std="0" type="EVENT">
  <EVENT name="PTHREAD_SWITCH_FROM" family="SCHEDULER" id="66">
    <EXTRA_DATA format="base64">
      TIALQNDg/7846/+AwAAAA==
    </EXTRA_DATA>
    <EXTRA_DATA format="custom">
      from_eip(hex)=400b804c
      bb_count=3
    </EXTRA_DATA>
  </EVENT>
</ENTITY>

<ENTITY number="2" tag="2541" ... time_std="0" type="EVENT">
  <EVENT name="PTHREAD_SWITCH_FROM" family="SCHEDULER" id="66">
    <EXTRA_DATA format="base64">
      GEcRQMqkkAcG5JAzQA==
    </EXTRA_DATA>
    <EXTRA_DATA format="custom">
      from_eip(hex)=40114718
      bb_count=205
    </EXTRA_DATA>
  </EVENT>
</ENTITY>
```
Presentation Outline

• Introduction
• Background
• Recording
• Replay
• Evaluation
• Conclusion
• Future work
Replay

- Thread debugging support in GDB
  - Builds thread list internally
  - Thread create and death events: \textit{threadnum}

- Thread debugging commands
  - Thread specific breakpoints
    - \texttt{break address thread threadnum}
    - All threads are stopped when any thread hits breakpoint
  - Switch context to desired thread
    - \texttt{thread threadnum}
  - Scheduler locking
    - \texttt{set schedlock on}
    - \texttt{continue} command resumes only current thread (thread that has context)
Replay

• PIDs in SO are invalid – needs mapping
  • Post-processing filter in DataStreams
  • Map PIDs in thread creation order: similar to threadnum

• Transition addresses have to be found
  • Only for some system calls
  • Manual lookup in objdump output
  • Can be input to GDB using syscall_address_file command

• Clever Insight
  • Nested breakpoints
  • Ability to attach TCL scripts to breakpoint
Replay

- Run *inferior* in scheduler locked mode
  - Only one thread runs at a time
  - Context can be switched using “thread threadnum” command
- Use SO to insert replay breakpoints
  - at return addresses: SO
  - transition addresses: mapping file
Replay

Program 4.1 Pseudo code for Automatic Breakpoint insertion

1: Set breakpoint at main and start the inferior
2: Wait for the inferior to report an event
3: If Event = Thread Exit
4:    Add threadnum to exited thread list
5:    Goto 10
6: Else If Event = Replay Breakpoint
7:    Goto 10
8: EndIf
9: Continue Inferior
10: Read next schedule event from schedule order file
11: If threadnum is in exited thread list
12:    Goto 10
13: EndIf
14: Switch to threadnum
15: If breakpoint return address has a corresponding transition address
16:    Set Next Replay Breakpoint at Transition address
17: Else
18:    Set Next Replay Breakpoint at Return address
19: EndIf
20: Goto 9
Replay

• Sample command group executed when a replay breakpoint is hit

Program 4.2 A sample command group that is executed upon hitting a replay breakpoint

1: info threads
2: thread 1
3: break *1074174393 thread 1 if pthread_get_bb_count() == 68
4: commands
5: disable_last_breakpoint_hit
6: set_next_replay_breakpoint
7: end
8: continue
Replay

• GDB features can still be used
• Experimental interleaving
  • Automatic breakpoint insertion can be controlled
  • Control returns to user after a replay breakpoint is hit
  • New interleaving can be created
Presentation Outline

- Introduction
- Background
- Recording
- Replay
- Evaluation
- Conclusion
- Future work
Evaluation

• Recording framework
  • Testing basic blocks
  • Testing return addresses

• Replay framework
  • Tested with programs that had data races: different results for different executions
  • Save experimental execution result
  • Replay in GDB produced experimental execution result
Presentation Outline

- Introduction
- Background
- Recording
- Replay
- Evaluation
- Conclusion
- Future work
Conclusion

• A framework to record execution path of NPTL based applications
  • GCC and GLIBC sources modified to support basic block count
  • GLIBC and kernel sources modified
    – to identify pthreads in kernel
    – to retrieve basic block count during context switch

• A replay framework to reproduce user-mode execution path of a program
  • Automatic breakpoint insertion feature in GDB
Presentation Outline

- Introduction
- Background
- Recording
- Replay
- Evaluation
- Conclusion
- Future work
Future Work

• Finding transition addresses automatically
  • Transition address required only for some system calls - Wrap them to generate a SYSCALL event

• Using *Progenitor* to separate events of multiple NPTL applications executing at same time

• Modify basic block maintenance using edge profiler
  • Newer versions of GCC use edge profiler
  • Use new GIMPLE intermediate language of GCC 4.0 to do tree walking/modification
Thank you
Questions?