An Empirical Investigation of Strategies for Debugging Multithreaded Programs

Scott D. Fleming
Department of Computer Science
Michigan State University
East Lansing, Michigan, USA 48824
sdf@cse.msu.edu

COPSE PROJECT  http://www.cse.msu.edu/sens/copse/
Collaborators: R. E. Kurt Stirewalt (MSU), Eileen Kraemer (UGA), Laura K. Dillon (MSU), and Shaohua Xie (UGA)
Problem: Debugging Multithreaded Programs

Multithreaded Program: Program with multiple concurrently-executing threads of control
  - Shared memory
  - Nondeterministic scheduler

Problem: Difficult to build correctly

Problem: Difficult to debug

Goal: Understand the strategies and practices that successful programmers use to debug multithreaded programs

Goal: Improve effectiveness of strategies
Background: Debugging

Debugging process: Iterative generation, verification, and refinement of hypotheses (Araki, Furukawa, & Cheng, 1991)

Two core debugging techniques:
  - Cyclic debugging (LeBlanc & Mellor-Crummey, 1987)
  - Data and control dependence analysis (Weiser, 1982)

Benefits:
  - Aid in verification of hypotheses
  - Reduce space of plausible hypotheses

Problem: Techniques ineffective for concurrent software
  - Difficult to reproduce failures
  - Complex control flow
Related Work: Debugging Concurrent Softw.

Cyclic debugging:
- Parallel Debugger (e.g., Brindle et al. (1989); Godefroid (1997))
- Replay (e.g., LeBlanc & Mellor-Crummey (1987); Carver & Tai (1991))

Data and control dependence analysis:
- Slicing (e.g., Cheng (1993); Krinke (2003); Zhao (1999))

Other related work: Fault detection
- Dynamic error detectors (e.g., O'Callahan & Choi (2003); Savage et al. (1997))
- Testing (e.g., Biberstein et al. (2003); Taylor, Levine, & Kelly (1992))
- Model checking (e.g., Clarke, Emerson, & Sistla (1986); Holzmann (1997))
- Proof techniques (e.g., Andrews (1991); Owicki & Gries (1976))

Problem: Methods and tools not widely used in practice (Lu et al., 2008)
Related Work: Empirical Studies of Programmers

Studies of strategies and techniques used during

- Debugging (e.g., Vessey (1985); Weiser (1982))
- Program comprehension (e.g., LaToza et al. (2007); Littman et al. (1987); von Mayrhauser & Vans (1996))
- Software maintenance tasks (e.g., Ko et al. (2005); Robillard et al. (2004))

Studies of practices associated with success on such tasks (e.g., Ko et al. (2005); LaToza et al. (2007); Littman et al. (1987); Robillard et al. (2004); Vessey (1985))

Studies of parallel programmer productivity (Hochstein et al., 2005)

No prior studies address debugging concurrent software
Overview

**Thesis:** During debugging, successful programmers
- Use strategies for verifying hypotheses that do not require reproducing failures or understanding global control flow
- Systematically manage an increased number of plausible hypotheses

**Method:** Empirical investigation of programmers performing a debugging task

**Result:** Hypothesis verification via *failure-trace modeling* strategy (Fleming et al., 2008a)

**Result:** Hypothesis management via *breadth-first fault diagnosis* (Fleming et al., 2008b)

**Discussion and future work**
Overview

Thesis: During debugging, successful programmers
- Use strategies for verifying hypotheses that do not require reproducing failures or understanding global control flow
- Systematically manage an increased number of plausible hypotheses

Method: Empirical investigation of programmers performing a debugging task

Result: Hypothesis verification via failure-trace modeling strategy (Fleming et al., 2008a)

Result: Hypothesis management via breadth-first fault diagnosis (Fleming et al., 2008b)

Discussion and future work
Background: Think-Aloud Method

(ERICSSON & SIMON, 1993)

Well-accepted and rigorous empirical method used to
- Obtain a theory of the cognitive processes used during an activity
- Test the validity of a proposed theory

**Step 1:** Observe participants performing an activity
- Ask participants to “think aloud”
- Collect transcripts (or protocols) of speech and actions

**Step 2:** Analyze transcripts

**Benefit:** Reveals goals and intentions of participant
Program to Correct

**Function:** Simulates an e-business server that accepts and processes requests from remote clients

**Architecture:** Based on Reactor pattern (Schmidt et al., 2000)

**Seeded fault:** Improper use of synchronization may cause a thread to try to pull from empty request queue

**Realistic:** Failure intermittent

- Takes about 5 minutes with a *stress tester* at a specific setting
- Sometimes the server needs to be restarted before the failure occurs
The Think-Aloud Study

Materials:

- Pretest
- Workstation equipped with development and video-capture software
- Source code for faulty program
- Stress-tester
- Posttest

Participants: 15 students enrolled in a graduate-level formal-methods course at Michigan State University

Procedure:

- Prior to study: Concurrency tools lecture and pretest
- Study session (3 hours): Task followed by posttest
Data Collection and Analysis

Data collection: Ran 15 sessions
- Prompter present during sessions, but not PIs
- Collected screen video and audio of participant speaking

Qualitative analysis: PIs and research assistants viewed videos to identify:
- Strategies
- Attributes that reflect practices of participants

Coding process:
- At least two researchers looked at each video
- Extensive discussion and refinement

Quantitative analysis: Looked for statistically-significant relationships between strategies and success
Overview

**Thesis:** During debugging, successful programmers
- Use strategies for verifying hypotheses that do not require reproducing failures or understanding global control flow
- Systematically manage an increased number of plausible hypotheses

**Method:** Empirical investigation of programmers performing a debugging task

**Result:** Hypothesis verification via failure-trace modeling strategy (Fleming et al., 2008a)

**Result:** Hypothesis management via breadth-first fault diagnosis (Fleming et al., 2008b)

Discussion and future work
Result: Hypothesis Verification Strategy

Programmer goal: Verify hypotheses about cause of failure

Problem: Core verification techniques ineffective because
- Failures difficult to reproduce
- Complex global control flow

Successful programmers verify hypotheses without reproducing failures or understanding global control flow

Observed strategy: Failure-trace modeling
Failure-Trace Modeling Strategy

Form of hypothesis: Program may enter an error state characterized by X

Process:
(1) Construct a candidate error suffix
(2) Verify that the error suffix is feasible

Error suffix: Models part of a hypothetical execution trace
- Seldom begins in the initial state of the program
- Elides many details
- Ends in a clear error state

Feasible error suffix: Consistent with an actual execution
- Initial state of error suffix is reachable program state
- Actions and state transitions are consistent with code
Indicators of Failure-Trace Modeling

Verbalization or drawing that

- Supposes the existence of two or more threads and one or more shared objects,
- Represents a sequence of actions by and among these entities, and
- Leads to a clear error state

Example: “Handler thread A calls retrieve_request, but the queue is empty so it waits on the condition variable. Then, the listener thread calls submit_request, adds a request to the queue, and signals the condition variable, which causes A to transition to the ready state...”
Strength of Model Articulation

Observation: Wide variation in quality (i.e., clarity and distinctness) among the models articulated by participants

Sample indicators of low quality:
- Actors with undefined roles (e.g., “some thread calls X”)
- Omissions of synchronization-relevant operations, such as calls to `acquire` and `release` on mutexes

Attributes of strongly-articulated model:
- Distinct actors and objects with well-defined synchronization states
- Describes how actions by the actors cause the objects and other actors to transition among these synchronization states
- Drawn or verbalized in sufficient detail to enable formulation of a well-formed UML sequence diagram
Quantitative Results

9 participants created a strongly articulated model
4 participants correctly modeled the failure

Participants who diagnosed the fault with confidence tended to
- Create a strongly articulated model \( (p < 0.01) \)
- Correctly model the failure \( (p < 0.05) \)

Participants who fixed the fault tended to
- Create a strongly articulated model \( (p < 0.05) \)
- Correctly model the failure \( (p < 0.05) \)

25% of participants who fixed the fault introduced a new fault
Overview

**Thesis:** During debugging, successful programmers
- Use strategies for verifying hypotheses that do not require reproducing failures or understanding global control flow
- Systematically manage an increased number of plausible hypotheses

**Method:** Empirical investigation of programmers performing a debugging task

**Result:** Hypothesis verification via failure-trace modeling strategy (Fleming et al., 2008a)

**Result:** Hypothesis management via breadth-first fault diagnosis (Fleming et al., 2008b)

Discussion and future work
Result: Hypothesis Management

Programmer goal: Efficiently diagnose fault

Problem: Lack of effective techniques for narrowing the space of plausible hypotheses

Successful programmers systematically manage hypotheses

Observed approach: *Breadth-first fault diagnosis*
Breadth-First Fault Diagnosis (Vessey, 1985)

Approach to debugging that involves
- Pursuing multiple lines of reasoning
- Deliberately investing intellectual resources into competing hypotheses as to the cause of the failure

Contrast with *depth-first fault diagnosis*, which involves
- Investigating hypotheses as they are formed
- Ignoring competing hypotheses
Space of Hypotheses

Elaborated a space of plausible hypotheses by creating a fault tree (Leveson et al., 1991)
- Each node represents a hypothetical event
- Child nodes represent the possible causes of their parents
- The root represents the failure

Extended standard syntax with annotations for representing
- Multiple interacting threads
- Ordering constraints on events
Indicators of Breadth-First Fault Diagnosis

**Hypothesis discovery:** Verbally describes an event in the tree
- Example: “Maybe the call to accept returned 0.”

**Hypothesis verification:** Attempt to determine how an event in the tree is enabled
- May be explicitly verbalized or inferred from behavior
- Example: Examining the definition of accept after stating the above hypothesis

**Breadth-first fault diagnosis:**
- Considered multiple competing hypotheses
- Attempted to verify a sufficiently large number of hypotheses down to depth 2 in the fault tree
Quantitative Results

10 participants used breadth-first fault diagnosis

Breadth-first participants were significantly more successful at

- Fixing the fault ($p < 0.05$)
- Fixing the fault without introducing new errors ($p < 0.01$)

Breadth-first participants failed to verify many plausible hypotheses at and below depth 2

Hypotheses that required only sequential reasoning were likely to be verified (77.5% chance)

Hypotheses that required reasoning about concurrent behavior were far less likely to be verified (45.45% chance)
Overview

**Thesis:** During debugging, successful programmers
- Use strategies for verifying hypotheses that do not require reproducing failures or understanding global control flow
- Systematically manage an increased number of plausible hypotheses

**Method:** Empirical investigation of programmers performing a debugging task

**Result:** Hypothesis verification via failure-trace modeling strategy (Fleming et al., 2008a)

**Result:** Hypothesis management via breadth-first fault diagnosis (Fleming et al., 2008b)

Discussion and future work
Other Findings

All participants attempted cyclic debugging using diagnostic print statements
- Many expressed understanding the limitations of the technique

All participants that used cyclic debugging without also using failure-trace modeling were unsuccessful

10 participants used a systematic comprehension strategy (Littman et al., 1987)
- Known to facilitate dependence analysis

Participants who used systematic comprehension were not significantly more successful than those who didn't
Threats to Validity

Type of error

Realism of

- Program
- Task

Composition of participant pool

- Students in formal-methods course
- All male

Motivation of participants
Recommendations

For education:
- Curricula should include more experience with debugging of multi-threaded software
- Teach importance of failure-trace modeling

For tool design:
- Automated construction of sequence diagrams
- Automated hypothesis management/generation using fault trees
Other Work

Synchronization contracts for object-oriented languages (Szumo) (Fleming et al., 2006)

Using formal models to objectively judge the quality of multithreaded programs in empirical studies (Dillon et al., 2008)

Assessing the benefits of various behavioral modeling notations for comprehending concurrent programs (e.g., UML sequence diagrams and state machines) (Xie et al., 2008)
Future Work

Investigate the benefits of *externalizing* representations of behavioral models for failure-trace modeling

Investigate the benefits of fault-tree editing tools for hypothesis management

Replicate study using different types of participants, synchronization faults, and concurrent programs

Develop a task model of concurrent software maintenance (Fleming, Stirewalt, & Kraemer, 2007b)

Investigate how modularization techniques (e.g., Szumo) affect maintainability using an approach based on *program families* (Fleming, Stirewalt, & Dillon, 2007a)
Acknowledgments

This material is based in part upon work supported by LogicBlox, Inc. and the National Science Foundation under Grant Numbers CCF-0702667 and IIS-0308063.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of LogicBlox, Inc. or the National Science Foundation.

Licenses for Camtasia provided by TechSmith, Inc.

Thank You!