On Debugging Real-Time Applications

Frank Mueller and David Whalley

Department of Computer Science
Florida State University
Tallahassee, FL 32304-4019

e-mail:

mueller@cs.fsu.edu
whalley@cs.fsu.edu

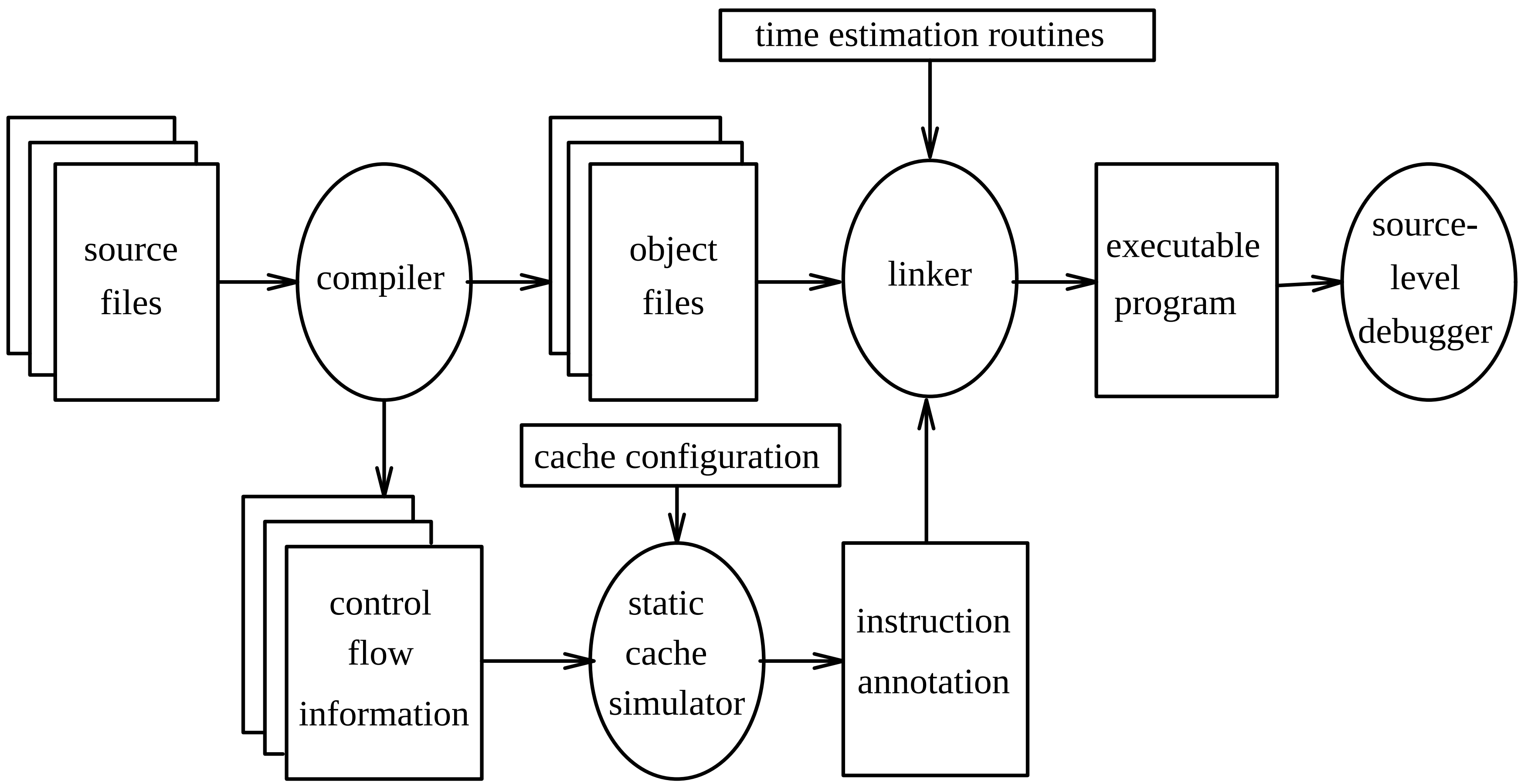
Overview

- debugging part of development cycle (up to 50% of time)
- few debuggers address real-time issues:
  - deadline monitoring
  - time distortion due to interference of debugging
- new debugging environment:
  - cache simulation to keep track of elapsed cycles
  - displays elapsed execution time (cycles)
  - works without changing debugger
  - slows down execution by factor of 1-4
  - much faster than hardware simulators
  - to find missed deadlines
  - to locate time-consuming code portions

Problems with Real-Time Debugging

- time distortion:
  - interference: breakpoints, debugger kernel traps, caching
  - replace real time with virtual time
  - external events simulated
- deadline monitoring and task tuning
- uni- vs. multi-processor

Overview of Debugging Environment



Real-Time Debugging Environment

- elapsed time tracing:
  - query during debugging (any breakpoint)
  - calculate based on cache hit and misses simulated so far
  - can translate number of cycles into seconds
- debugging optimized code:
  - allows realistic timing simulation
  - restricts debugging to basic blocks (breakpoints)
  - can also display most variables in registers
  - sometimes inconsistent values (due to optimizations)

Sample Session with dbx

```
(dbx) stop at 43
(dbx) stop at 123 if elapsed_cycles() > 4000000
(dbx) display elapsed_cycles()
[...]
stopped in four at line 43
   43     mmax=2;
elapsed_cycles() = 29413
(dbx) next
stopped in four at line 44
   44     while(n>mmax) {
elapsed_cycles() = 29428
[...]
  123       four(tdata,nn,isign);
elapsed_cycles() = 4015629
[...]
elapsed cycles() = 4095351
execution completed, exit code is 1
```

- break after 4 million cycles
- display elapsed time
- breakpoints do not affect virtual time keeping
- deadline missed after 4+ million cycles (points to area where it's misses)
- program terminated even later
- could set breakpoint at inner nesting level next to localize missed deadline

Performance Overhead

unopt. code	opt. code with time estimates			
	1kB	2kB	4kB	8kB
1.8	7.8	4.5	3.0	2.1

- implemented on SPARC
- modified VPO
- used dbx under SunOS 4.1.3
- verified correctness of instruction cache simulation by comparing with trace-driven simulation
- 11 test programs
- unopt. about 1.8 times slower than opt. code
- instr. opt. about 2.1 to 7.8 times slower than opt. code
- instr. opt. about 1 to 4 times slower than common unopt. code
- cache size influences overhead (due to conflicts)

Future Work

- external event table
- `pragma zero`$_{t}imetoinsertnon$ – $intrusivedebuggingcodeextendtodatacaching, pipelining$
- to be used with Pthreads real-time kernel on SPARC VME board
- profiling with timing information

Related Work

- debugging capabilities typically very restricted
- Remedy: interface, suspend all processors on breakpoint
- DCT: special hardware (bus monitoring), non-intrusive
- RED, ART: software instrumentation, remote debugging
- DARTS: (1) program trace (2) debug time-stamped trace
- hardware simulators: very slow

- debugging remote, indirect (off-line), not within program
- Remedy:
- DCT: Distributed computing testbed
- RED: Remotely executed debugger
- ART: instrumentation permanent part of programs
- DARTS: Debug assistant for RTSs, no data queries

Summary

- developed new debugging enhancement for real-time
- replace real time with virtual time
- keep track of virtual time by instruction cache simulation
- feasible through static cache simulation
- debug unoptimized or *optimized* code
- display time repeatedly at breakpoints
- find missed deadlines through conditional breakpoints
- locate time-consuming code, tune it
- tuning may be used to make schedule feasible
- performance overhead of factor 1 to 4 over unoptimized code