Balancing Performance and Productivity for the Development of Dynamic Binary Instrumentation Tools: A Case Study on Arm Systems

Cosmin Gorgovan, Guillermo Callaghan, Mikel Luján
Department of Computer Science
University of Manchester

Dynamic Binary Instrumentation (DBI)

- DBI is an approach for analysing the execution of applications at the level of machine code
- DBI frameworks
  - implement a runtime capable of modifying applications as they execute
  - provide APIs used by DBI tools to plug in their analysis and instrumentation routines
- Used for a wide range of applications such as:
  - development tools: memory error checkers, profilers
  - application analysis: taint tracers, debuggers
  - microarchitectural simulators

Contributions

- an API design which:
  - emphasises convenience and portability for the common building blocks of DBI
  - while allowing low level control over performance-critical or specialised instrumentation
- implemented the API on top of the open-source MAMBO system
- implemented a number of DBI tools using this system
  - and evaluated their performance against similar tools

Our API

Event-driven: plugins register handlers for events related to:

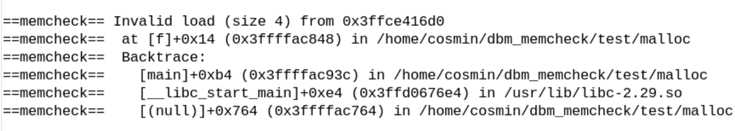
- code scanning
- execution of system calls
- function calls
- multithreading

Two layers:

- low level - operates directly on machine code
- high level - portable instrumentation
  - a RISC-like instruction set for generating instrumentation
  - code analysis functions which abstract the decoding of application code
  - code generation helpers for a number of common DBI tasks

M-memcheck

- memory error checker
- detects memory usage bugs:
  - out-of-bounds memory accesses
  - invalid frees
- similar functionality to Valgrind Memcheck and Dr. Memory
- implemented using our API
- representative of heavyweight DBI plugins
- `github.com/beehive-lab/mambo/tree/memcheck`

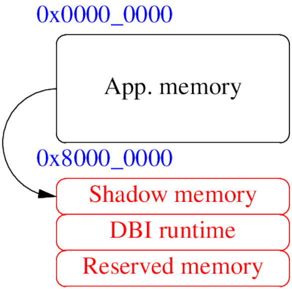


```
==memcheck== Invalid load (size 4) from 0x3ffce416d0
==memcheck==  at [f]+0x14 (0x3ffffac848) in /home/cosmin/dbm_memcheck/test/malloc
==memcheck==  Backtrace:
==memcheck==    [main]+0xb4 (0x3ffffac93c) in /home/cosmin/dbm_memcheck/test/malloc
==memcheck==    [__libc_start_main]+0xe4 (0x3ffd0676e4) in /usr/lib/libc-2.29.so
==memcheck==    [(null)]+0x764 (0x3ffffac764) in /home/cosmin/dbm_memcheck/test/malloc
```

Shadow memory

Tracks whether a memory location is valid:

- granularity of 1 byte
- updated when the application allocates and releases memory
- in the same address space as the application
- using the address space shaping capabilities of the API to reserve a contiguous shadow memory location for the entire address space of the application



Shadow memory layout on 32-bit architectures

Instrumenting memory accesses

Each memory access in the application is instrumented to

- load the corresponding values from the shadow memory
- check whether the whole accessed range is valid
- and print an error message + backtrace if not

```
mambo_register_pre_inst_cb(ctx, &memcheck_pre_inst_handler);
[...]
int memcheck_pre_inst_handler(mambo_context *ctx) {
  if (mambo_is_load_or_store(ctx)) {
    int access_size = mambo_get_ld_st_size(ctx);
    bool is_store = mambo_is_store(ctx);
    [...]
```

Error reporting

Called by the instrumentation for invalid memory accesses

- prints the location that was accessed
- the location of the instruction performing the invalid access
- symbol information for the function containing it
- and a backtrace if it's available

```
void memcheck_print_error(void *addr, void *pc, stack_frame_t *frame) {
  [...]
  int ret = get_symbol_info_by_addr(pc, &symbol, &symbol_base, &file);
  printf("\n==memcheck== Invalid access (size %d) at %p\n", size, addr);
  printf("==memcheck==  at [%s]+%p (%p) in %s\n",
         symbol, pc - symbol_base, pc, file);
  [...]
```

Instrumenting function calls

Standard library functions that allocate or free memory

- instrumented to update the shadow memory
- invalid accesses in these functions and their callees ignored
  - they access the heap metadata - outside the valid application allocations

```
// instrumenting void *malloc(size_t size);
mambo_register_function_cb(ctx, "malloc", &memcheck_malloc_pre,
                                          &memcheck_malloc_post, 1);
memcheck_malloc_pre() {// save size = reg0}
memcheck_malloc_post() {fcall(&memcheck_alloc_hook, retvalue, size);}
memcheck_alloc_hook(void *start, size_t size) {
  int ret = mambo_ht_add(&allocs, (uintptr_t)start, (uintptr_t)size);
  assert(ret == 0);
  memcheck_mark_valid(start, size);
}
```

Multithreading

- multithread-scalable plugins
- Events
  - Pre Thread - convenient event to allocate and initialise thread-private resources
  - Post Thread - all active threads at the time the application exits
- Removes the burden of tracking application threads in each plugin.

Evaluation

Benchmark: PARSEC 3.0 benchmark suite w/ *native* input set

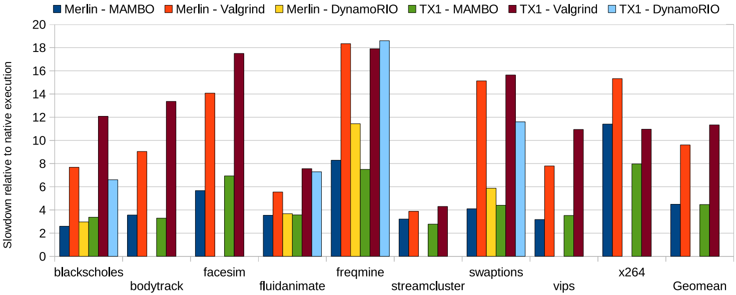
Platforms:

- Merlin - 8-core X-Gene2 SoC
- TX1 - 4-core (Cortex-A57) Tegra X1 SoC

Memory error checkers:

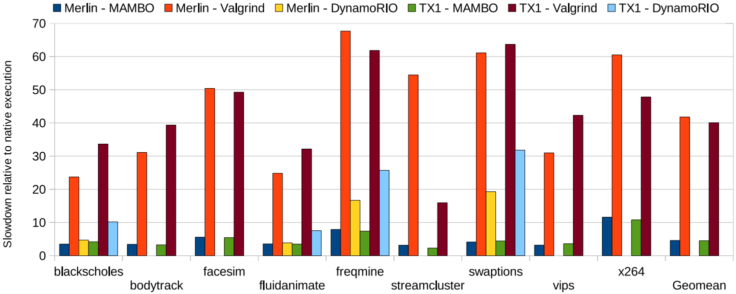
- M-memcheck
- Valgrind Memcheck 3.13.0
- Dr. Memory (bcb36073a2c) implemented using DynamoRIO
  - crashed on some of the benchmarks

Evaluation - single threaded



Geometric mean slowdown relative to native execution - 1 thread

Evaluation - multithreaded



Geometric mean slowdown relative to native execution, 4 threads

Summary

- API for a DBI framework (open-source MAMBO system)
  `github.com/beehive-lab/mambo/tree/memcheck`
- Portability and flexibility
  - across A32, T32 and A64 ISAs (Armv8)
- Two layers:
  - Low level - fine-grain
  - High level - portable
- Tools
  - M-memcheck - Memory Error Checking
  - M-cachesim - Online Cache Simulation