MANCHESTER
1824
The University of Manchester

Dynamic Binary Instrumentation and Modification with MAMBO

Cosmin Gorgovan
Guillermo Callaghan
Mikel Luján

School of Computer Science
University of Manchester

Outline

- Brief introduction to Dynamic Binary Modification (DBM)
  - and Dynamic Binary Instrumentation (DBI)
- High level overview of MAMBO and its API
- Coffee break (11:00 - 11:30)
- Building portable and high level instrumentation
- Native code analysis & instrumentation, advanced features
- Overview of the plugins distributed with MAMBO

Please feel free to interrupt for questions.

The University of Manchester

- Brief introduction to Dynamic Binary Modification (DBM)
  - and Dynamic Binary Instrumentation (DBI)
- High level overview of MAMBO and its API
- Building portable and high level instrumentation
- Native code analysis & instrumentation, advanced features
- Overview of the plugins distributed with MAMBO

Defining key terms

- Dynamic - at runtime
- Binary - at the level of native code
- Dynamic Binary Modification (DBM)
  - altering applications at runtime, at the native code level
- (Software) Instrumentation
  - *the transformation of a program into its own measurement tool*
- Dynamic Binary Instrumentation (DBI)
  - DBM, when the modification consists of adding instrumentation code
- DBM / DBI system
  - software runtime implementing DBM / DBI

MANCHESTER 1824
The University of Manchester

Example uses of DBM/DBI

- microarchitectural simulation
  - *Sniper Multi-Core Simulator, APTSim* (MAMBO-based)*
- cache simulation
  - *Valgrind Cachegrind, drcachesim, MAMBO cachesim*
- program analysis
  - *Valgrind Callgrind*
- memory error detection / debugging
  - *Valgrind Memcheck, Dr. Memory*
- dynamic binary translation
  - *QEMU, Apple Rosetta*

* *John Mawer, Oscar Palomar, Cosmin Gorgovan, Andy Nisbet, Will Toms, and Mikel Luján. 2017. The Potential of Dynamic Binary Modification and CPU-FPGA SoCs for Simulation. FCCM, 2017*

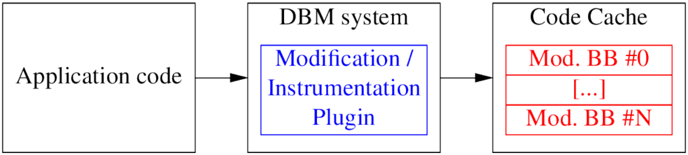
Working principles of DBM

The DBM system scans the application code and copies it to a software code cache:

- it transforms the code to maintain correctness & control
- organised in basic blocks
  - single-entry and single-exit regions
- all application code runs from the code cache
- it enables doing other modifications
  - by plugins via an API (in the case of MAMBO)
- think JIT (re)compilation for native code

The University of Manchester

The code cache



Key DBM concepts for plugin developers

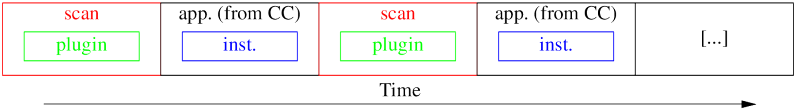
- the modified application, the plugin and the inserted instrumentations execute in the same process
  - can directly share memory buffers
  - instrumentation must preserve application context
    - register values, application data
- code is scanned at the granularity of basic blocks
- new basic blocks scanned as they are discovered
  - interleaved execution of modified code and code scanning

The University of Manchester

Execution timeline with DBM

Essential to understand the difference between:

- execution of the plugin at code scanning time
  - safe execution in the DBM system context
- execution of the inserted instrumentation within the application
  - shares and must preserve the context of the application



MANCHESTER 1824
The University of Manchester

ARM

Two distinct execution modes:

- 64-bit execution (AArch64)
  - with 1 instruction set (A64)
- 32-bit execution (AArch32)
  - 2 interworking instruction sets: ARM (A32) and Thumb (T32)

Load/store architecture

Registers:

- Stack Pointer, Program Counter
- General purpose
- FP/SIMD
- Special-purpose (condition flags, FP control and status, etc)

Example code modification

Translating optional hardware divide instructions in AArch32:

```
SDIV R2, R5, R6
```

to a call to the software division routine:

```
; R2 = __aeabi_idiv(R5, R6)
MOV R0, R5
MOV R1, R6
BL __aeabi_idiv
MOV R2, R0
```

Example code instrumentation

Inserting an execution counter (AArch32):

```
SDIV R2, R5, R6
```

```
; counter++
PUSH {R0, R1}
ADR R0, counter
LDR R1, [R0]
ADD R1, R1, #1
STR R1, [R0]
POP {R0, R1}
```

Introduction summary

- defined Dynamic Binary Instrumentation, Dynamic Binary Modification
- working principles of DBM
- the code cache
- ARM basics

Questions?

- Brief introduction to Dynamic Binary Modification (DBM)
  - and Dynamic Binary Instrumentation (DBI)
- High level overview of MAMBO and its API
- Building portable and high level instrumentation
- Native code analysis & instrumentation, advanced features
- Overview of the plugins distributed with MAMBO

MAMBO

- Fast DBM implementation for ARM (AArch32 and AArch64)
- Runs on GNU/Linux
- Open source, Apache 2.0 license
  - `https://github.com/beehive-lab/mambo`
- contributions are welcomed
  - bug reports & patches
  - sample plugins
  - feedback on the API
- *Cosmin Gorgovan, Amanieu d'Antras, and Mikel Luján. MAMBO: A Low-Overhead Dynamic Binary Modification Tool for ARM. TACO, Article 14 (April 2016)*

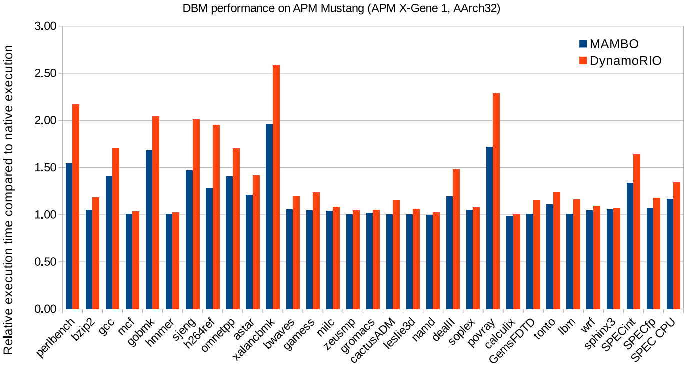
MANCHESTER
1824
The University of Manchester

Why MAMBO?

- small codebase: 16 kLoC (core) + 1.3kLoC (sample plugins)
- good compatibility with applications (and improving)
- allows analysis of app. code at the machine code level
  - useful for microarchitectural analysis and simulation
- the API allows trading off between performance, portability and ease of development
- good performance
  - the lowest base overhead among the DBM systems for ARM
  - allows performance scaling of multithreaded applications

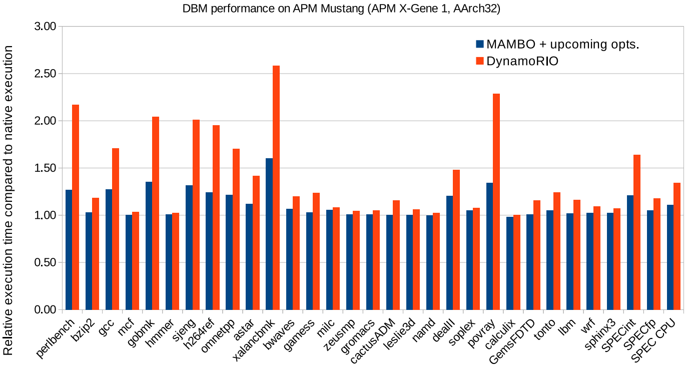
The University of Manchester

Why MAMBO? Low overhead



The University of Manchester

Why MAMBO? Low overhead [2]



The University of Manchester

Virtual machine image

Portable QEMU virtual machine image (for non-ARM hosts):

https://github.com/beehive-lab/mambo-vm/releases

To use:

```
# install qemu-system-aarch64
# download mambo_vm.tar.gz
mkdir mambo-vm
cd mambo-vm
tar xf /path/to/mambo_vm.tar.gz
./start_vm.sh
# SSH alarm:alarm or root:root on localhost:5040
cd /home/alarm/mambo
# make sure to change the passwords if opening SSH to the network
```

Building MAMBO

- Recommended to build natively on ARM
- Build dependencies: git, libelf(-dev), ruby, GCC

```
git clone https://github.com/beehive-lab/mambo.git
cd mambo
git submodule init
git submodule update
make
```

- example plugins in `plugins/`
- more information at
  `https://github.com/beehive-lab/mambo`

MANCHESTER 1824
The University of Manchester

The MAMBO API

- Event-driven programming model
- Plugins typically handle:
  - Code analysis
  - Code generation, modification or instrumentation
  - Runtime event handling

MAMBO API events

- Generally allowing *pre*- and *post*- event hooks
  - *pre*- callbacks executed before MAMBO handles the event
  - *post*- callbacks executed after MAMBO has handled the event

- Categorised in:
  - runtime events - e.g. execution of system calls by the application
  - code scanning events - for code analysis and instrumentation

MANCHESTER 1824
The University of Manchester

Runtime events

- Init constructor - runs before the application is launched
  - for allocating resources and initialising the plugin
- Thread - on thread creation and termination
  - tracking the application's threads
  - enables low overhead thread-private instrumentation
- Exit - on application exit
  - for collating data and outputing results
- Syscall - application system call wrapper
  - observing and modifying the system calls executed by the application

Code analysis and instrumentation - BBs

Events:

pre BB -> *pre instruction* -> *post instruction* -> [...] -> **post BB**

- on scanning a new basic block
- *pre basic block*
  - after code cache space has been allocated
  - before any application instructions have been processed
  - use to initialise basic block-level analysis
- *post basic block*
  - after the end of the basic block was found
  - all application instructions in the BB have been processed
  - instrument based on basic block-level analysis

MANCHESTER 1824
The University of Manchester

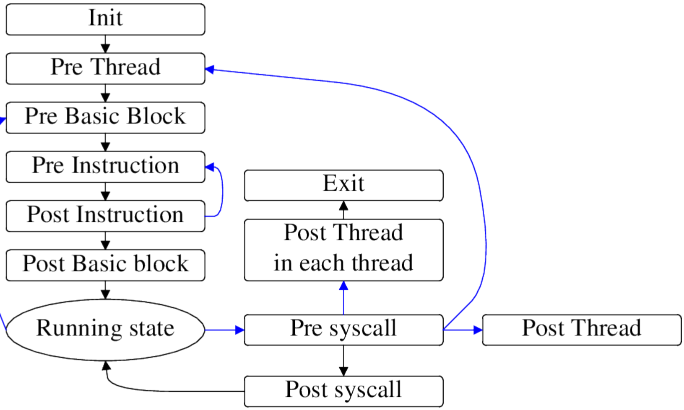
Code analysis and instrumentation - Insts.

pre BB -> **pre instruction** -> **post instruction** -> [...] -> *post BB*

- on scanning a new instruction in a basic block
- *pre instruction*
  - after an application instruction has been decoded
  - before the instruction has been copied to the code cache (CC)
  - use to analyse the instructions
  - use to insert instrumentation before an instruction
- *post instruction*
  - after an application instruction has been copied to the CC
  - can be used to analyse the instructions
  - use to insert instrumentation after an instruction

MANCHESTER 1824
The University of Manchester

Order of MAMBO API events



MANCHESTER 1824
The University of Manchester

Building and registering a MAMBO plugin

Only static linking is supported at the moment.

1. Add the source files to the makefile:

```
PLUGINS+=plugins/file.c plugins/file.S[...]
```

2. Write a constructor function:

```
__attribute__((constructor)) void plugin_init_fn() {
  mambo_context *ctx = mambo_register_plugin();
  assert(ctx != NULL);
}
```

The University of Manchester

Registering event callbacks

```
int plugin_pre_thread(mambo_context *ctx) {
  // [...]
}
```

```
__attribute__((constructor)) void plugin_init_fn() {
  mambo_context *ctx = mambo_register_plugin();
  assert(ctx != NULL);

  mambo_register_pre_thread_cb(ctx, &plugin_pre_thread);

}
```

MAMBO overview summary

- downloading and building MAMBO
- the event-driven programming model
- writing a plugin's init constructor
- building a plugin

Questions?

MANCHESTER 1824
The University of Manchester

- Brief introduction to Dynamic Binary Modification (DBM)
  - and Dynamic Binary Instrumentation (DBI)
- High level overview of MAMBO and its API
- Building portable and high level instrumentation
- Native code analysis & instrumentation, advanced features
- Overview of the plugins distributed with MAMBO

Register names

- AArch32: `r0-r15, sp, lr, pc`
- AArch64: `x0-x31, sp, lr, pc`
- portable: `reg0-reg12`
  - mapped to `r0-r12`/`x0-x12`
- `es` - first calleE-Saved register
- also defined: `m_reg_name` for `(1 « reg_name)`
  - e.g. `m_reg0 = (1 « reg0)`

The University of Manchester

Portable code analysis

Some of the portable code analysis helpers:

```
// is the inst a branch? is it direct, indirect, etc?
mambo_branch_type mambo_get_branch_type(mambo_context *ctx);
// is it a conditionally executed instruction?
mambo_cond mambo_get_cond(mambo_context *ctx);
bool mambo_is_load(mambo_context *ctx);
bool mambo_is_store(mambo_context *ctx);
// if load or store, what size of data does it access?
int mambo_get_ld_st_size(mambo_context *ctx);
// what's the size of the instruction word? 2 / 4 bytes for Thumb
int mambo_get_inst_len(mambo_context *ctx);
```

The University of Manchester

Portable branch analysis

ISA-independent way to determine if an instruction is a branch

- and what type of branch

```
int plugin_pre_inst_handler(mambo_context *ctx) {
  mambo_branch_type type = mambo_get_branch_type(ctx);
  if (type & BRANCH_RETURN) {
    // Procedure return instruction
  } else if (type & BRANCH_DIRECT) {
    // Direct branch instruction
  } else if (type & BRANCH_INDIRECT) {
    // Indirect branch instruction
  }
}
```

The University of Manchester

Portable load / store analysis

```
int plugin_pre_inst_handler(mambo_context *ctx) {
  bool is_load = mambo_is_load(ctx);
  bool is_store = mambo_is_store(ctx);
  if (is_load || is_store) {
    int size = mambo_get_ld_st_size(ctx);

    // [...]
  }
}
```

The University of Manchester

Calls to C functions

The API allows the safe insertion of calls to C functions:

```
int plugin_pre_inst_handler(mambo_context *ctx) {
  // Calling a function with no arguments
  int ret = emit_safe_fcall(ctx, &function_with_no_arguments, 0);

  // or for a function taking arguments
  emit_push(ctx, m_reg0 | m_reg1);
  emit_set_reg(ctx, reg0, ARGUMENT0);
  emit_set_reg(ctx, reg1, ARGUMENT1);
  emit_safe_fcall(ctx, &function_with_2_arguments, 2);
  emit_pop(ctx, m_reg0 | m_reg1);
}
```

MANCHESTER 1824
The University of Manchester

Calls to assembly functions

- Potentially more efficient than calls to C functions
- but have to manually preserve all application state

```
int plugin_pre_inst_handler(mambo_context *ctx) {
  emit_push(ctx, m_reg0 | m_reg1 | m_lr);
  emit_set_reg(ctx, reg0, ARGUMENT0);
  emit_set_reg(ctx, reg1, ARGUMENT1);
  emit_fcall(ctx, &asm_function);
  emit_pop(ctx, m_reg0 | m_reg1 | m_lr);
}
```

Calls to assembly functions [continued]

The called assembly instruction

```
// Only registers 0 and 1 can be safely modified
// Must be implemented for each ISA
asm_function:
  #ifdef __arm__
    ADD r0, r0, r1
    BX LR
  #elif __aarch64__
    ADD X0, X0, X1
    RET
  #else
    #error
  #endif
```

The University of Manchester

Basic inline instrumentation

A few portable code generation helpers are provided:

```
void emit_push(mambo_context *ctx, uint32_t regs);
void emit_pop(mambo_context *ctx, uint32_t regs);
void emit_set_reg(mambo_context *ctx, enum reg reg, uintptr_t value);
void emit_mov(mambo_context *ctx, enum reg rd, enum reg rn);
int emit_add_sub_i(mambo_context *ctx, int rd, int rn, int offset);
int emit_add_sub(mambo_context *ctx, int rd, int rn, int rm);

void emit_counter64_incr(mambo_context *ctx, void *counter, unsigned incr);
int mambo_calc_ld_st_addr(mambo_context *ctx, enum reg reg);
```

MANCHESTER 1824
The University of Manchester

Portable inline execution counters

```
void *counter = NULL;
mambo_branch_type type = mambo_get_branch_type(ctx);
if (type & BRANCH_RETURN) {
  counter = &counters->return_branch_count;
} else if (type & BRANCH_DIRECT) {
  counter = &counters->direct_branch_count;
} else if (type & BRANCH_INDIRECT) {
  counter = &counters->indirect_branch_count;
}
if (counter != NULL) {
  emit_counter64_incr(ctx, counter, 1);
}
```

Summary of portable and high level instrumentation

- Helpers for portable instruction decoding
- Helpers to calling C / assembly functions
- Helpers for generating inline instrumentation

Questions?

MANCHESTER 1824
The University of Manchester

- Brief introduction to Dynamic Binary Modification (DBM)
  - and Dynamic Binary Instrumentation (DBI)
- High level overview of MAMBO and its API
- Building portable and high level instrumentation
- Native code analysis & instrumentation, advanced features
- Overview of the plugins distributed with MAMBO

MANCHESTER 1824
The University of Manchester

Native code analysis

differentiate between AArch32 and AArch64

- use `#ifdef __arm__` and `#ifdef __aarch64__`

for AArch32, differentiate between ARM and Thumb code:

```
if (mambo_get_inst_type(ctx) == THUMB_INST) {
} else if (mambo_get_inst_type(ctx) == ARM_INST) {
}
```

get the instruction:

```
// enum in pie/pie-[ARCH]-decoder.h
int mambo_get_inst(mambo_context *ctx);
```

The University of Manchester

Native code analysis [continued]

To decode individual instructions:

- use the decoding helpers defined in pie/pie-[ARCH]-field-decoder.h

```
if (inst_set == ARM_INST) {
  if (inst == ARM_SDIV) {
    uint32_t rd, rn, rm;
    arm_sdiv_decode_fields(mambo_get_source_addr(ctx), &rd, &rn, &rm);
  }
}
```

The University of Manchester

Native code instrumentation

- code generation helpers defined in `api/emit_[ARCH].h`
- can be used together with portable code generation

```
// emit_counter64_incr(mambo_context *ctx, void *counter, unsigned incr);
emit_push(ctx, m_reg0);
emit_set_reg(ctx, r0, (uintptr_t)counter);
if(mambo_get_inst_type(ctx) == THUMB_INST) {
  emit_thumb_vfp_vpush(ctx, 1, 0, 0, 4);
  emit_thumb_vfp_vldr_dp(ctx, 1, r0, 0, 1, 0);
  emit_thumb_neon_vmovi(ctx, 0, 0, 0, 0, 0, incr >> 7, incr >> 4, incr);
  emit_thumb_neon_vshr(ctx, 1, 0, 0, 0, 0, 0, 1, 32);
  emit_thumb_neon_vadd_i(ctx, 3, 0, 0, 0, 0, 1, 0, 0);
  emit_thumb_vfp_vstr_dp(ctx, 1, 0, r0, 0, 0);
  emit_thumb_vfp_vpop(ctx, 1, 0, 0, 4);
}
emit_pop(ctx, m_reg0);
```

The University of Manchester

Branching in generated code

Branching to a target address:

```
int emit_branch(mambo_context *ctx, void *target);
int emit_branch_cond(mambo_context *ctx, void *target, mambo_cond cond);
int emit_branch_cbz(mambo_context *ctx, void *target, enum reg reg);
int emit_branch_cbnz(mambo_context *ctx, void *target, enum reg reg);
```

Branching in generated code [2]

```
  ADDEQ R0, R0, R1
```

Conditionally instrumenting the instruction above:

```
  BNE skip ; emit_branch_cond(ctx, pc+8, NE);
  ; instrumentation code here
  PUSH {R0, R1, LR}
  ; [...]
  POP {R0, R1, LR}
skip:
  ADDEQ R0, R0, R1
```

Branching in generated code [3]

Reserved branch slots & branching to the current address:

```
mambo_cond cond = mambo_get_cond(ctx);
mambo_branch skip_br;
if (cond != AL) {
  ret = mambo_reserve_branch(ctx, &skip_br);
  assert(ret == 0);
}
// instrumentation code here
emit_push(ctx, m_reg0 | m_reg1 | m_lr);
// [..]
emit_pop(ctx, m_reg0 | m_reg1 | m_lr);
if (cond != AL) {
  ret = emit_local_branch_cond(ctx, &skip_br, invert_cond(cond));
  assert(ret == 0);
}
```

Branching in generated code [4]

Branching to the current address:

```
int emit_local_branch_cond(mambo_context *ctx,
                           mambo_branch *br, mambo_cond cond);
int emit_local_branch(mambo_context *ctx, mambo_branch *br);
int emit_local_fcall(mambo_context *ctx, mambo_branch *br);
int emit_local_branch_cbz(mambo_context *ctx,
                          mambo_branch *br, enum reg reg);
int emit_local_branch_cbnz(mambo_context *ctx,
                           mambo_branch *br, enum reg reg);
```

MANCHESTER 1824
The University of Manchester

Upcoming features

- symbol handling
  - instrumenting function entry and return by name
  - obtaining symbol information by address
  - application backtraces
- shadow memory
  - maintain metadata for each byte of app. memory
  - enables apps. such as memory error checking and taint tracking

MANCHESTER 1824
The University of Manchester

Summary of advanced features

- raw machine code analysis
  - conditional compilation, `mambo_get_inst_type(ctx)`
  - instruction decoding, `pie/pie-[ARCH]-field-decoder.h`
- generating native code instrumentation
  - helpers in `api/emit_[ARCH].h`
- branching in the generated instrumentation
- upcoming symbol handling and shadow memory support

- Brief introduction to Dynamic Binary Modification (DBM)
  - and Dynamic Binary Instrumentation (DBI)
- High level overview of MAMBO and its API
- Building portable and high level instrumentation
- Native code analysis & instrumentation, advanced features
- Overview of the plugins distributed with MAMBO

MANCHESTER 1824
The University of Manchester

mtrace - data memory access tracing

`mambo/plugins/mtrace.*`

- outputs a trace of all **data** memory accesses
- change from text to binary output for improved performance

API use:

- *thread* and *instruction* events
- thread-level instrumentation
- portable code analysis & instrumentation
- calls to assembly & C functions

cachesim - online cache simulator

`mambo/plugins/cachesim/`

- simulates a configurable cache hierachy (**data** & **instruction**)
- configuration templates for simulating Cortex-A*

API use:

- *thread*, *instruction*, *exit* and *basic block* events
- thread-level instrumentation
- portable code analysis & instrumentation
- calls to assembly & C functions

soft_div - software emulation for HW divide

`mambo/plugins/soft_div.c`

- hardware divide instructions optional in AArch32
- translates them to calls to sw emulation routines

API use:

- *instruction* event
- native code analysis & generation
- calls to the C library helper functions
- demonstrates replacing (as opposed to instrumenting) code

Upcoming - memcheck: memory error checker

- significantly lower overhead than Valgrind Memcheck or Dr. Memory
- release date TBD

API use:

- **VM**, **function**, *thread* and *instruction* events
- both portable and native code analysis & generation
- calls to assembly and C functions
- uses shadow memory

MANCHESTER 1824
The University of Manchester

Questions?

Thanks for attending

`https://github.com/beehive-lab/mambo`