MAMBO - Dynamic Binary Instrumentation on ARM and RISC-V

HiPEAC 2024, Munich, 17 January 2024.

Joshua Lant,
John Alistair Kressel,
Igor Wodiany,
Konstantinos Iordanou,
Kyriakos Paraskevas, and
Mikel Luján.

University of Manchester
<firstname>.<lastname>@manchester.ac.uk

MANCHESTER 1824
The University of Manchester

Thanks!

EPSRC
Engineering and Physical Sciences Research Council

UK Research and Innovation

ROYAL ACADEMY OF ENGINEERING

arm

THE ROYAL SOCIETY

Tutorial Format

- Getting Started / Logistics
- Introduction to Dynamic Binary Modification and MAMBO (~25 min)
- Introduction to MAMBO’s plugin API (~30 min)
- Coffee break (30 min)
- MAMBO use case and example plugin (~30 min)
- Hands on MAMBO session (~60 min)

SET UP MAMBO NOW!

IT TAKES 30 - 60 MIN WITH QEMU!

https://tinyurl.com/mambo-docker-setup

MANCHESTER
1824
The University of Manchester

Introduction to Dynamic Binary Modification and MAMBO

What is DBM/DBI/DBT?

- **D**ynamic - Working at runtime.
- **B**inary - Natively compiled user-space code.
- **M**odification - Alteration of applications at runtime, at the level of native code.
- **I**nstrumentation – The measurement, analysis or modification of aspects of a computer program's operating behaviour or performance by means of inserting additional code such as timers, counters, or loggers.
- **T**ranslation- In this context, can be either the use of a DBM tool for translation from one binary format to another, or the method by which some DBM tools work; first lifting to an IR and resynthesizing / recompiling.

Uses of DBM/DBI/DBT tools

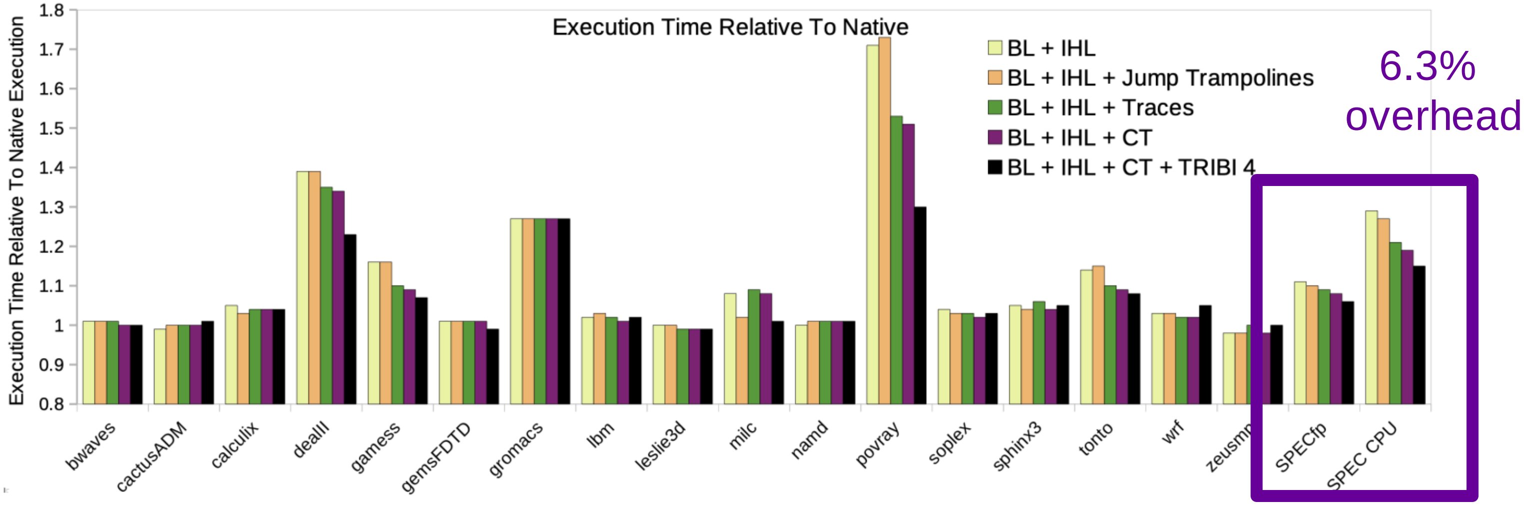
- Microarchitectural simulation
  - Sniper, ZSim, SimAcc (MAMBO)
- Cache simulation
  - Valgrind Cachegrind, DynamoRIO drcachesim, MAMBO cachesim
- Program analysis
  - Valgrind Callgrind
- Memory error detection / debugging
  - Valgrind Memcheck, Dr. Memory, MAMBO memcheck
- Dynamic binary translation
  - QEMU, Apple Rosetta, TANGO

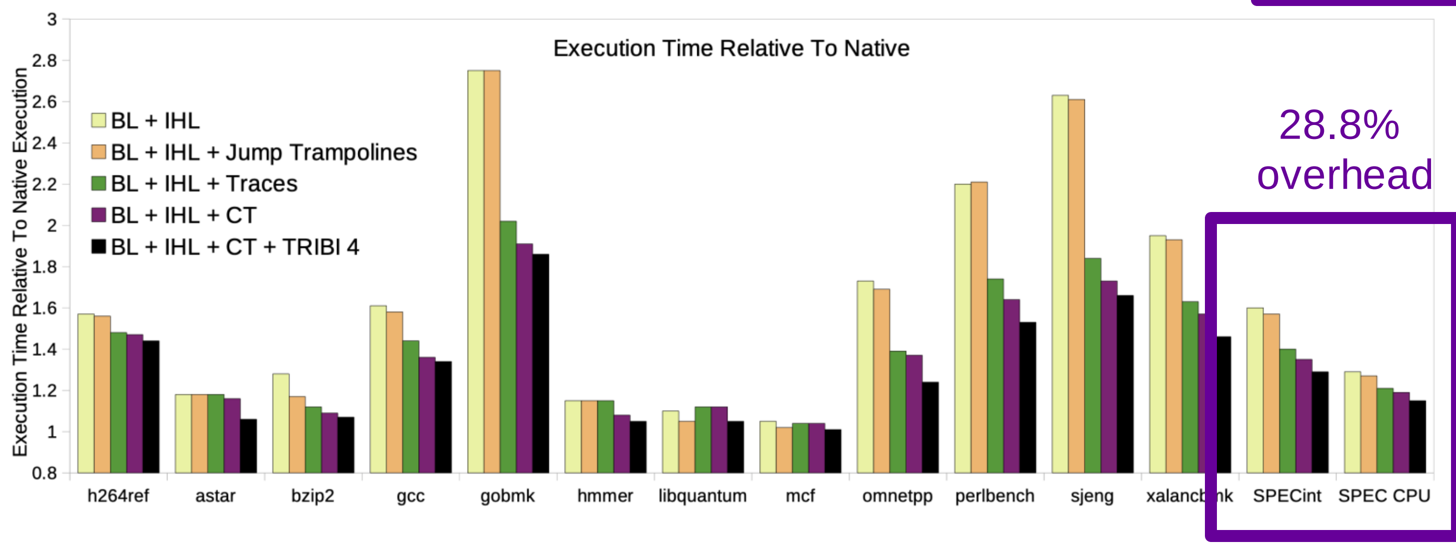
Why MAMBO?

- Optimized for ARM 32-bit, ARM 64-bit & RISC-V 64-bit
  - Low overhead
  - Only available DBM optimized for RISC-V
- Low complexity
  - Relatively small codebase ~20k LoC
- Simple plugin API for extension/tools
  - Architecture agnostic helper functions for portable plugins

MANCHESTER 1824
The University of Manchester

Why MAMBO on RISC-V?

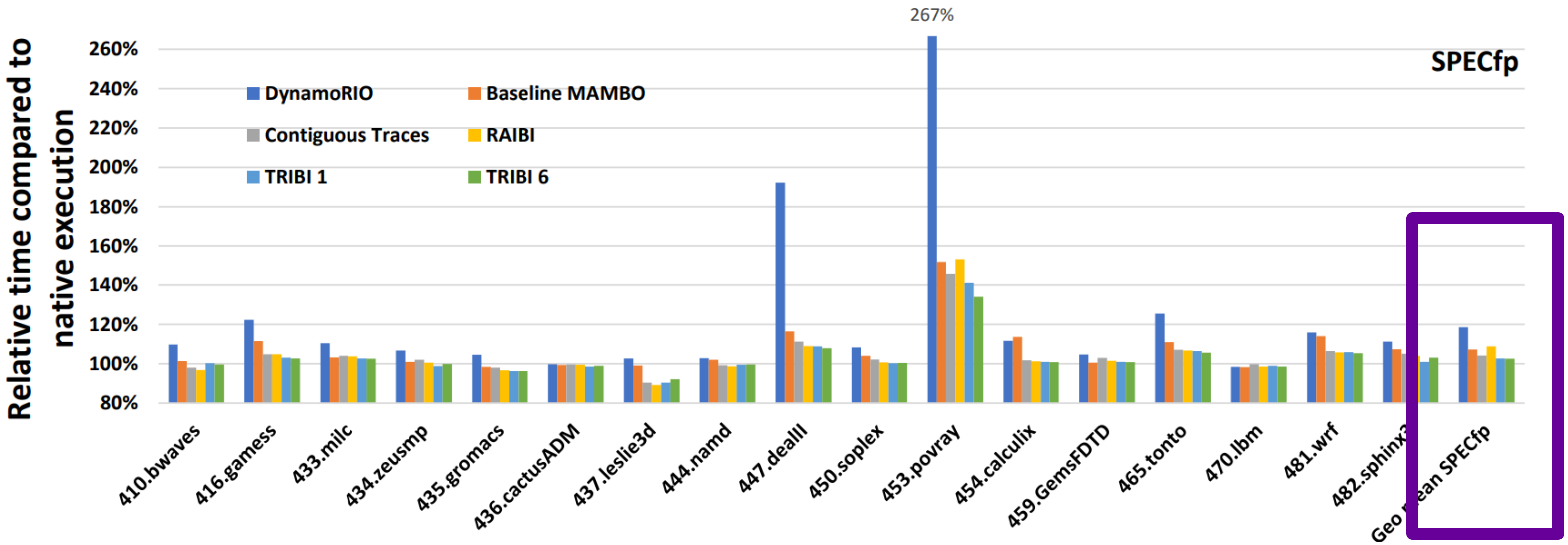


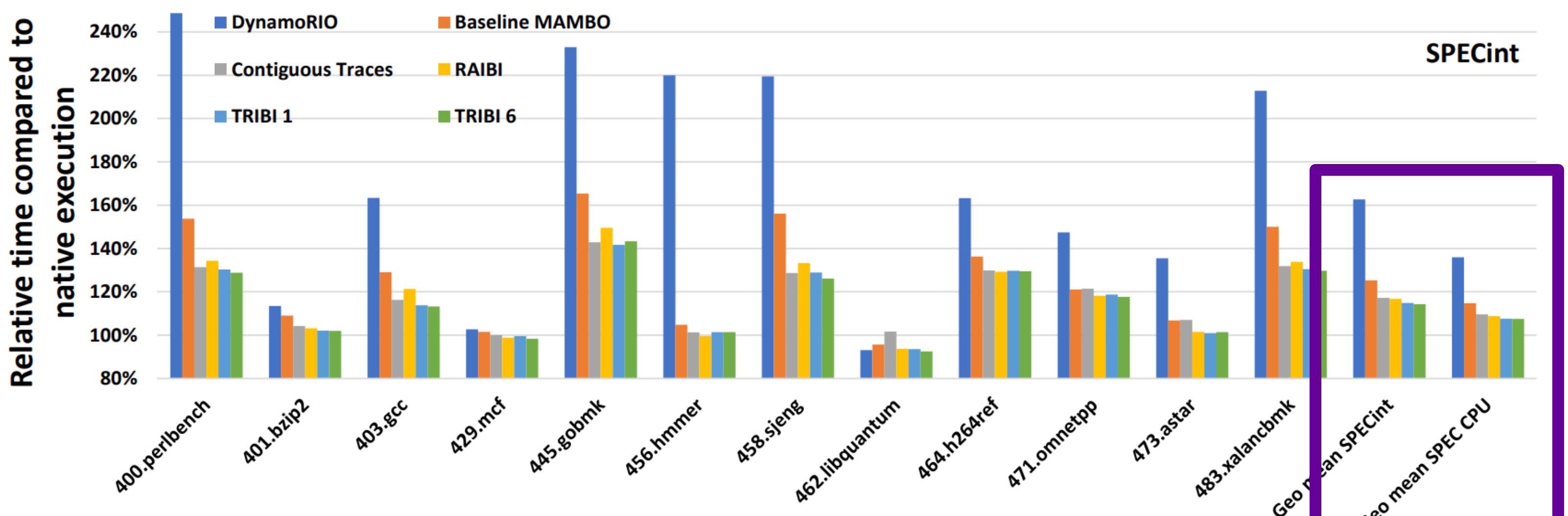


Slowdown relative to native execution for SPEC CPU2006 – RISC-V 64GC.

Kressel et al. Evaluating the Impact of Optimizations for Dynamic Binary Modification on 64-bit RISC-V.

Why MAMBO on ARM?

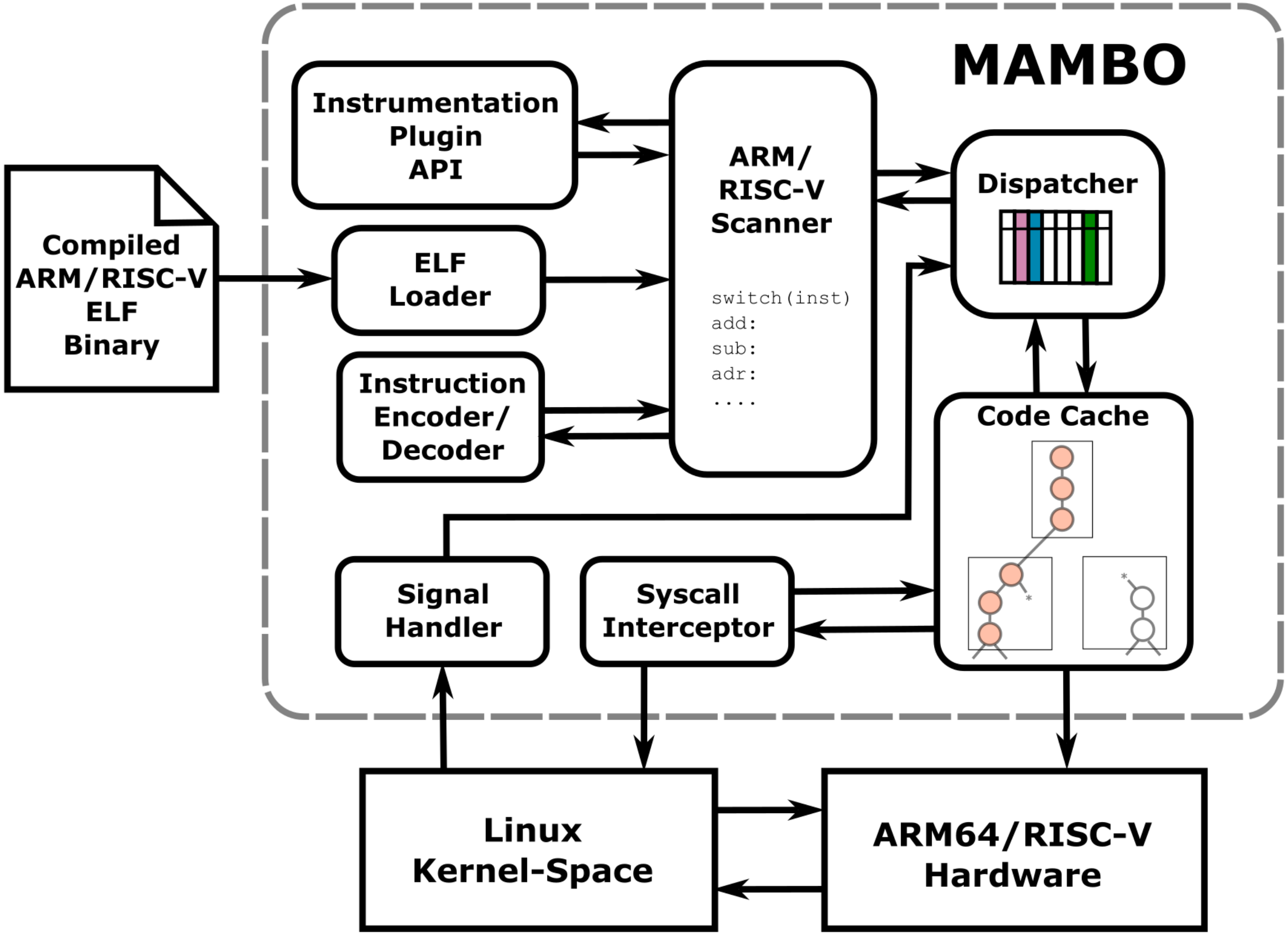




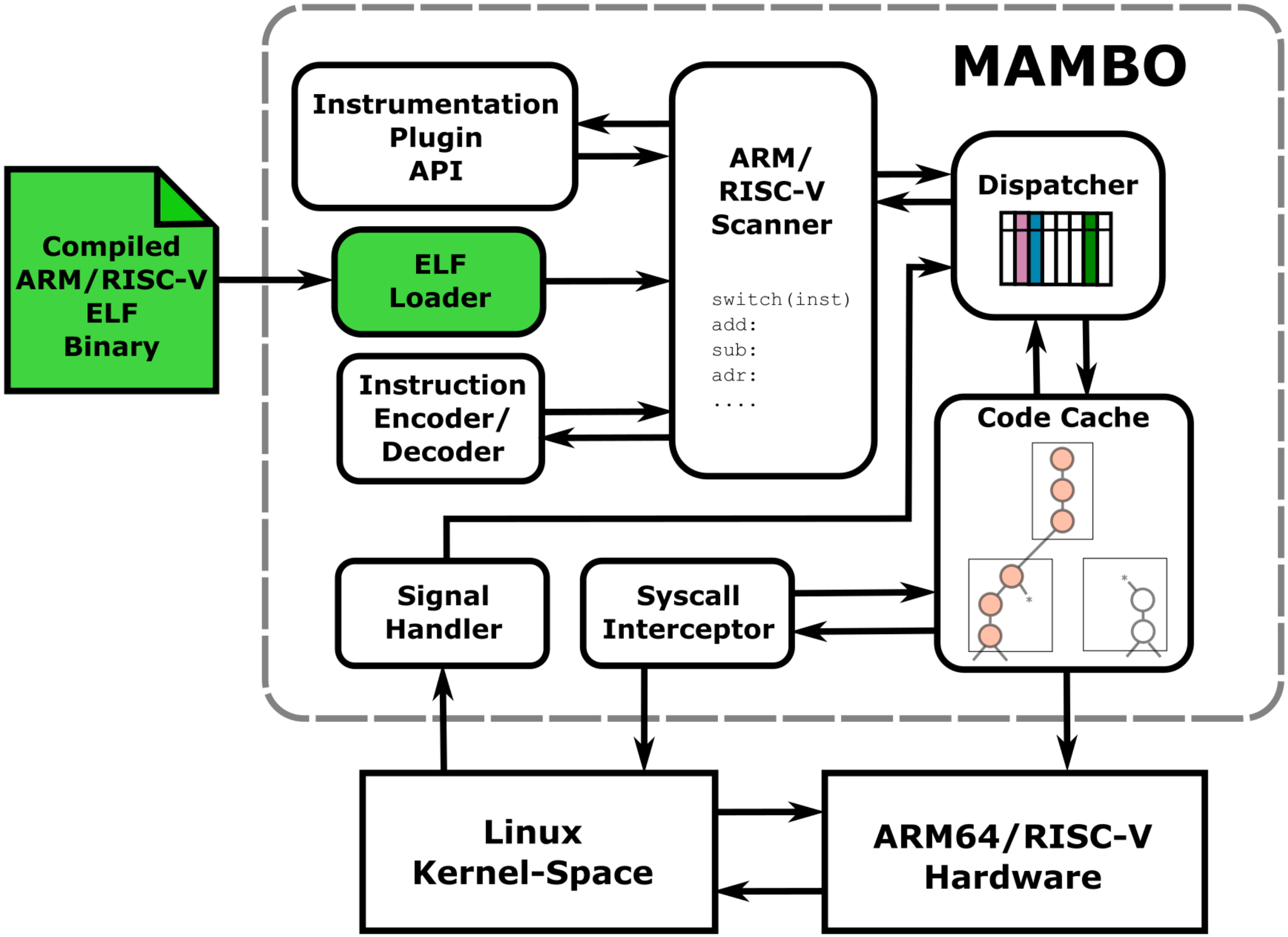
Slowdown relative to native execution for SPEC CPU2006 – ARM64 64.

Callaghan et al. Optimising dynamic binary modification across 64-bit Arm microarchitectures.

MAMBO Architecture

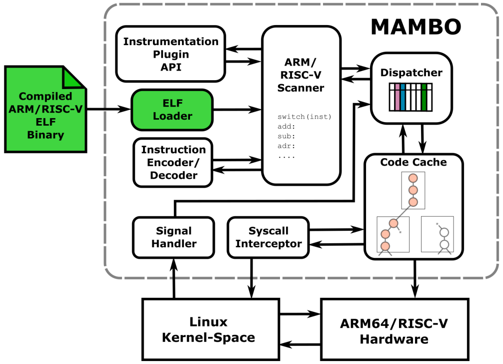


ELF Loading

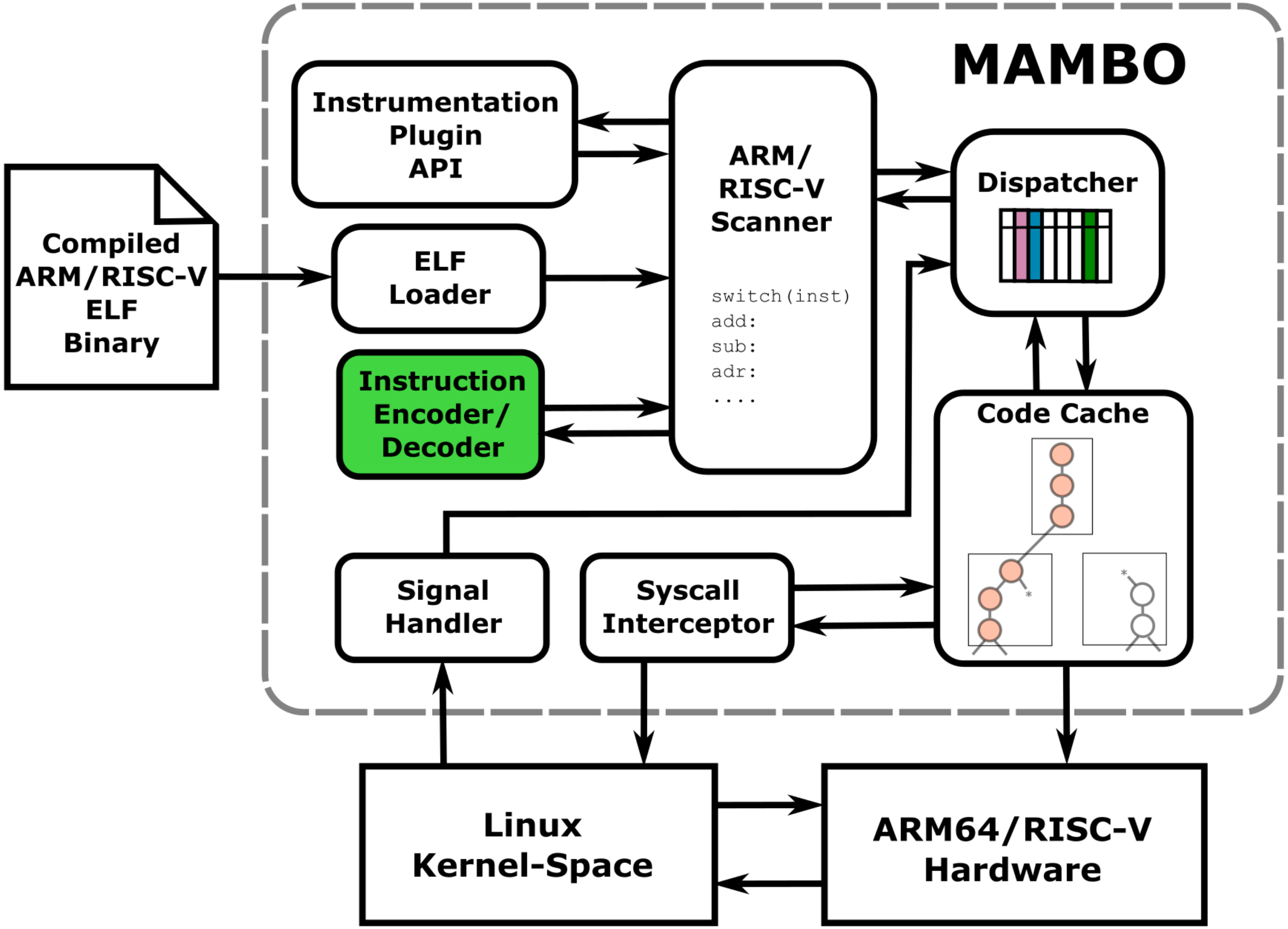

MAMBO
Instrumentation
Plugin
API
Compiled
ARM/RISC-V
ELF
Binary
ELF
Loader
Instruction
Encoder/
Decoder
ARM/
RISC-V
Scanner
switch(inst)
add:
sub:
adr:
....
Dispatcher
Code Cache
Signal
Handler
Syscall
Interceptor
Linux
Kernel-Space
ARM64/RISC-V
Hardware

ELF Loading

- Executable & Linking Format (ELF).
- Binary format for Linux and Unix-like operating systems.
  - Provides static information for linker to build executable.
  - Provides runtime information needed for the loader.
- MAMBO is loaded as a normal program by the Linux loader (ld).
- MAMBO then must act as the loader for the hosted application.
  - Uses libelf to parse ELF header of hosted program.
  - Sets up virtual address space for program to execute.
  - Parses program arguments.
  - Sets up initial stack frame.
  - Sets up global data.
- The readelf utility is useful for many tasks when developing MAMBO plugins.

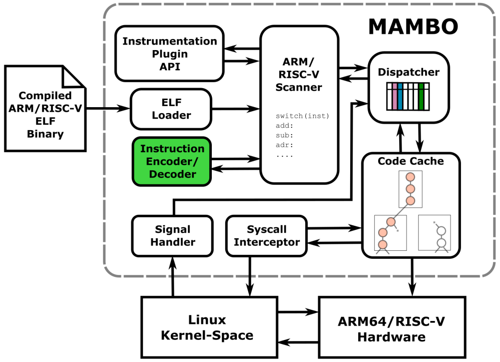


Instruction Encoder/Decoder

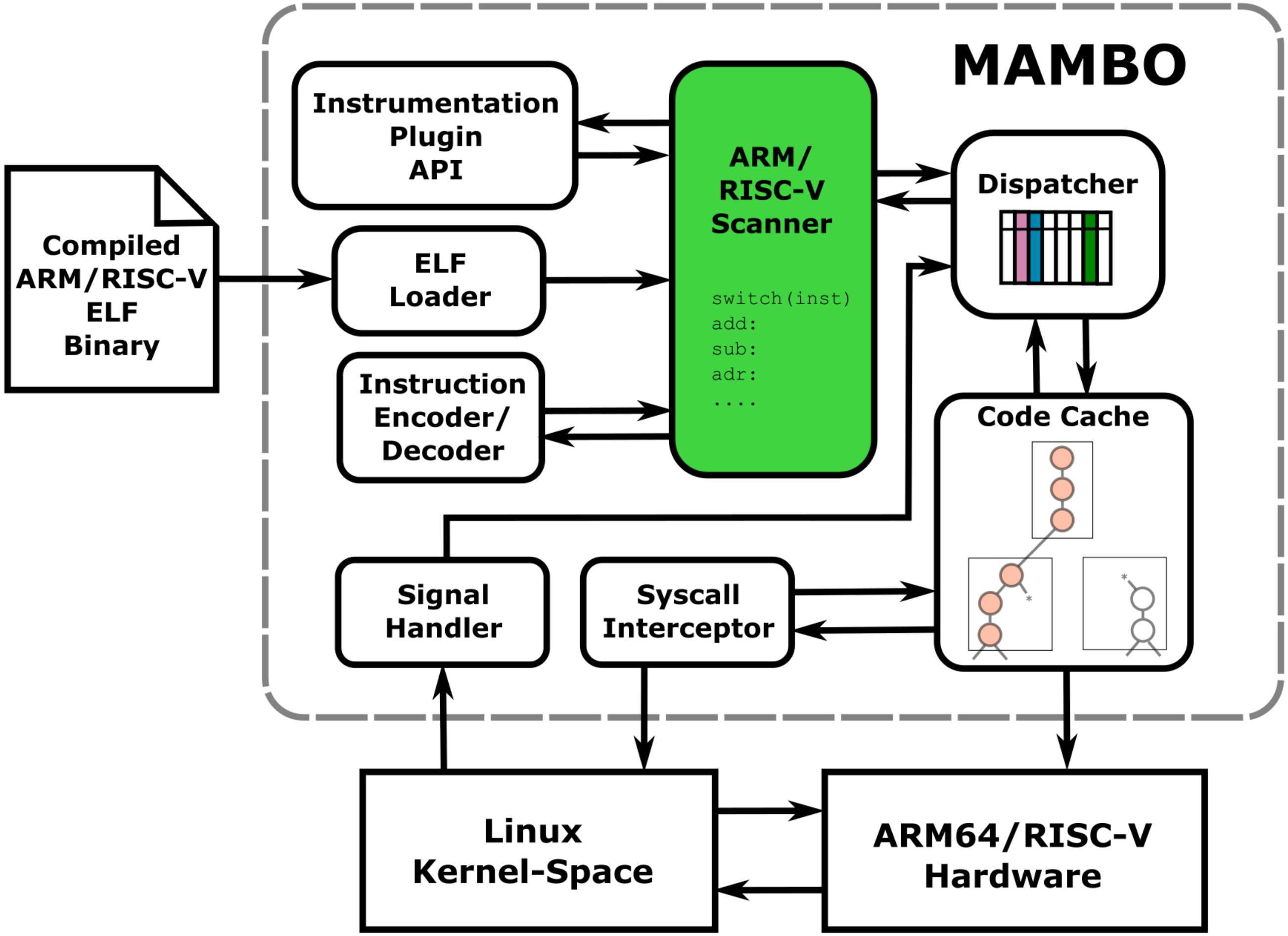


Instruction Encoder/Decoder

– Automatic generation of encoder/decoder from a text file with instructions specification within.
– Raw binary instructions decoded and fields separated for use by the scanner.
– The instructions are grouped into instruction types.
  - Scanner can then decode further, or...
  - User can decode fields with API for certain user plugin tasks.
– If adding a new instruction, this is where you would do it.



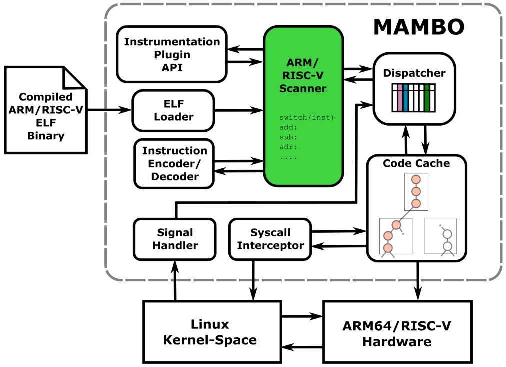
Basic Block Scanning



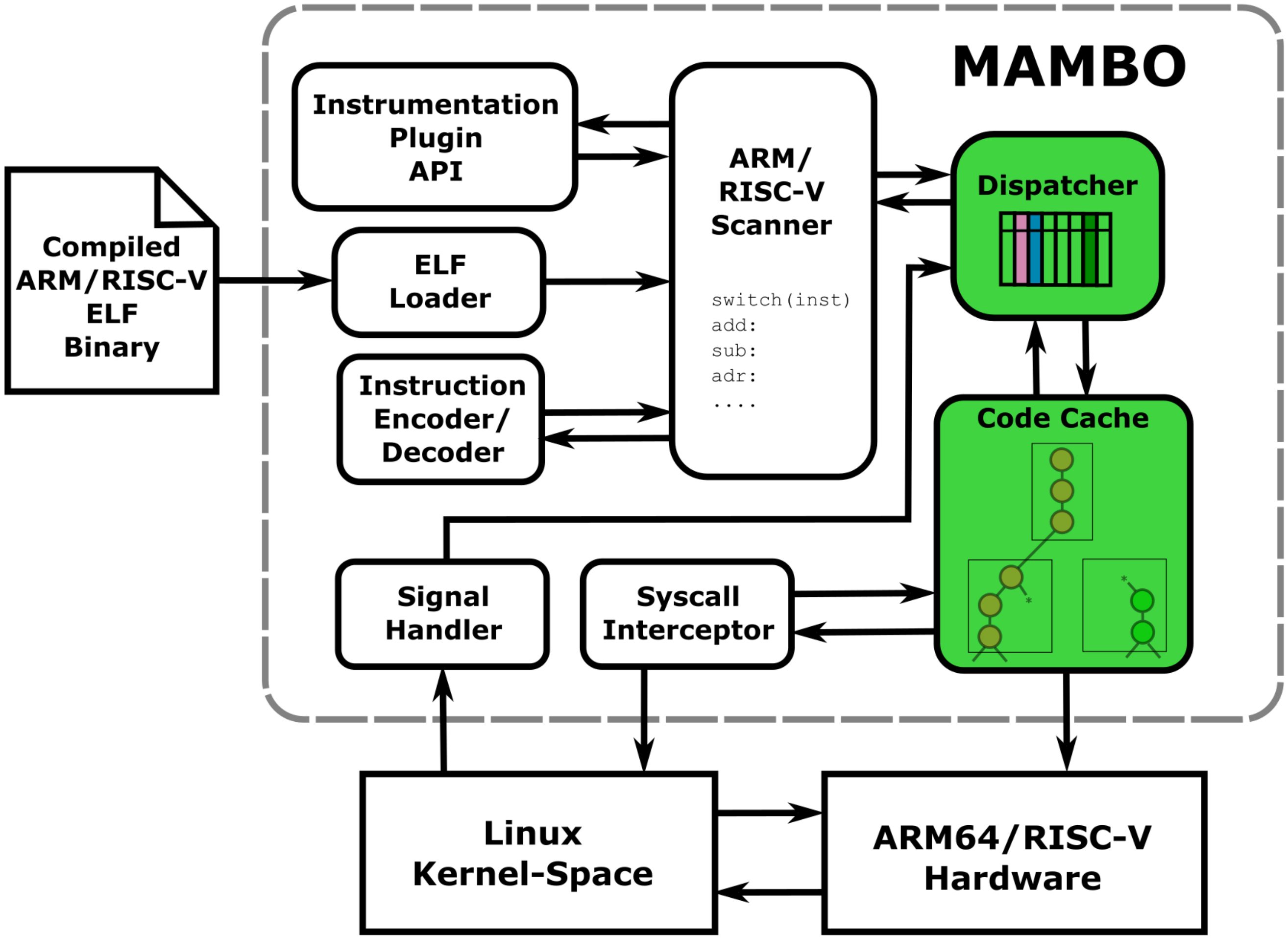
MANCHESTER 1824
The University of Manchester

Basic Block Scanning

- The scanner reads basic blocks of the program, which are then placed in the code cache for execution.
- Reads instructions in from the hosted program one by one.
- Modifies the instruction for execution under MAMBO if required.
  - Branching instructions.
  - Instructions which use PC relative addressing.
  - Special instructions such as system register access, or syscalls.
- If no modifications required, straight to the code-cache.
  - Copy & annotate.
  - As opposed to disassemble & resynthesise.
- After the full basic block is scanned, hand off execution to the dispatcher.



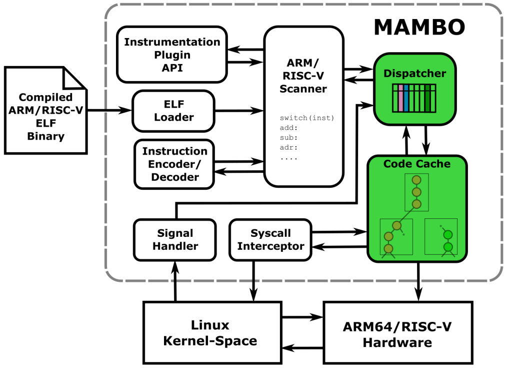
Dispatch and Code-Cache



MANCHESTER 1824
The University of Manchester

Dispatch and Code-Cache

- The code cache is used to help amortize cost of scanning (pareto principle).
- Basic blocks are scanned into the code cache single entry single exit
  - i.e. only branch taken is scanned in.
- Hash table maps scanned application addresses to code cache addresses.
- Translated blocks branch to hash lookup in dispatcher
  - Fast lookup, 15 instructions for indirect branch (assuming no collisions).
  - If hit, jump to translated address.
  - If not, go to dispatcher to prepare scanning new basic block.
- Thread private code cache. Share nothing.
  - Simple multi-threaded support.
  - Limits thread scalability.



Dispatch and Code-Cache

Original application code

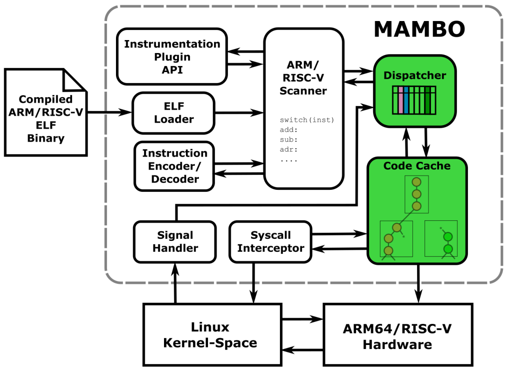
```
BB_X:
0x100    str x1, [x3, x0, lsl #3]
0x104    ldr x0, [x1, #0x10]!
0x108    cbnz x0, 0x80
```

Translated Basic Block in CC

```
T_BB_X:
0xff300    pop x0, x1
0xff304    str x1, [x3, x0, lsl #3]
0xff308    ldr x0, [x1, #0x10]!
0xff30C    cbnz x0, tpc_of(#Next
   BB)
0xff310    b tpc_of(#Next BB)
```

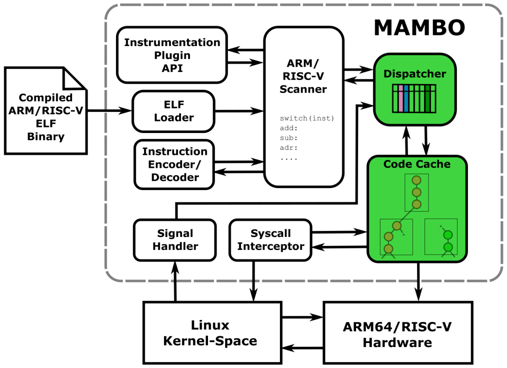
MAMBO dispatcher trampoline

```
tpc_of:
0xe0000    push x0, x1
…
Jumping to scanner or to the next T_BB
```

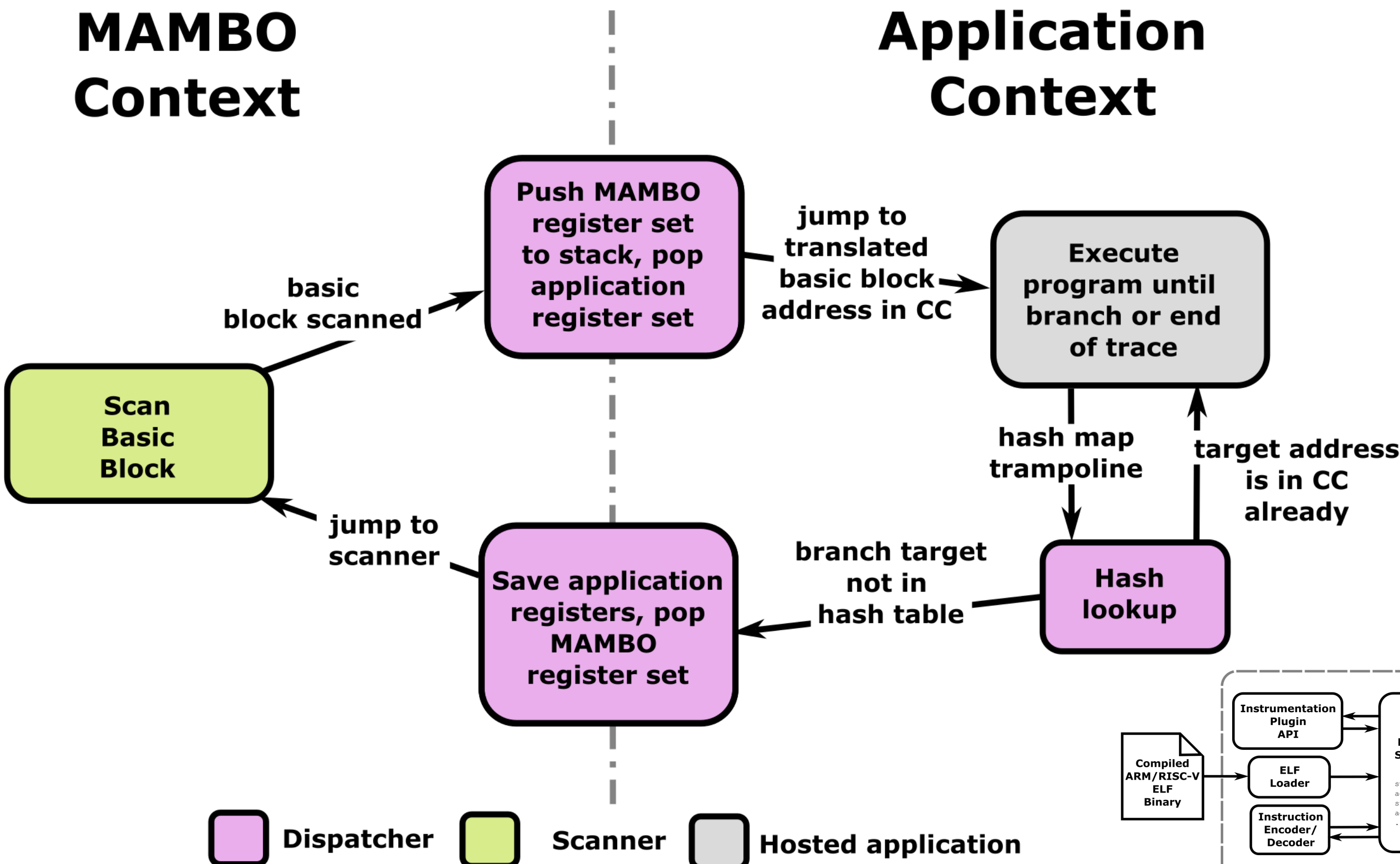


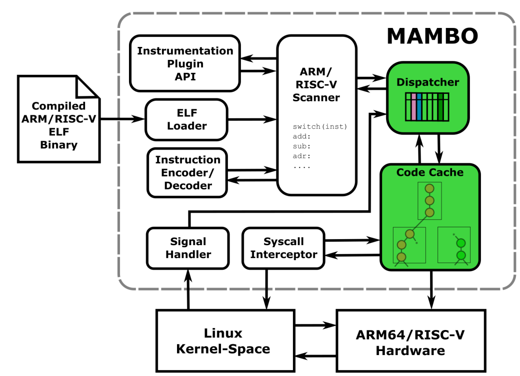
Dispatch and Code-Cache

- Direct branches.
  - Branch exits go to trampoline.
- Indirect branches.
  - Main source of overhead for DBM systems.
  - Optimization uses an inline hash lookup to determine if entry in CC.
- Dispatcher moves between MAMBO and application context.
  - Saves register state for current context.
  - Restores register state for next context.
  - Trampoline jumps to target address.
- *Traces* reduces this overhead by modifying branch targets for *hot* regions.

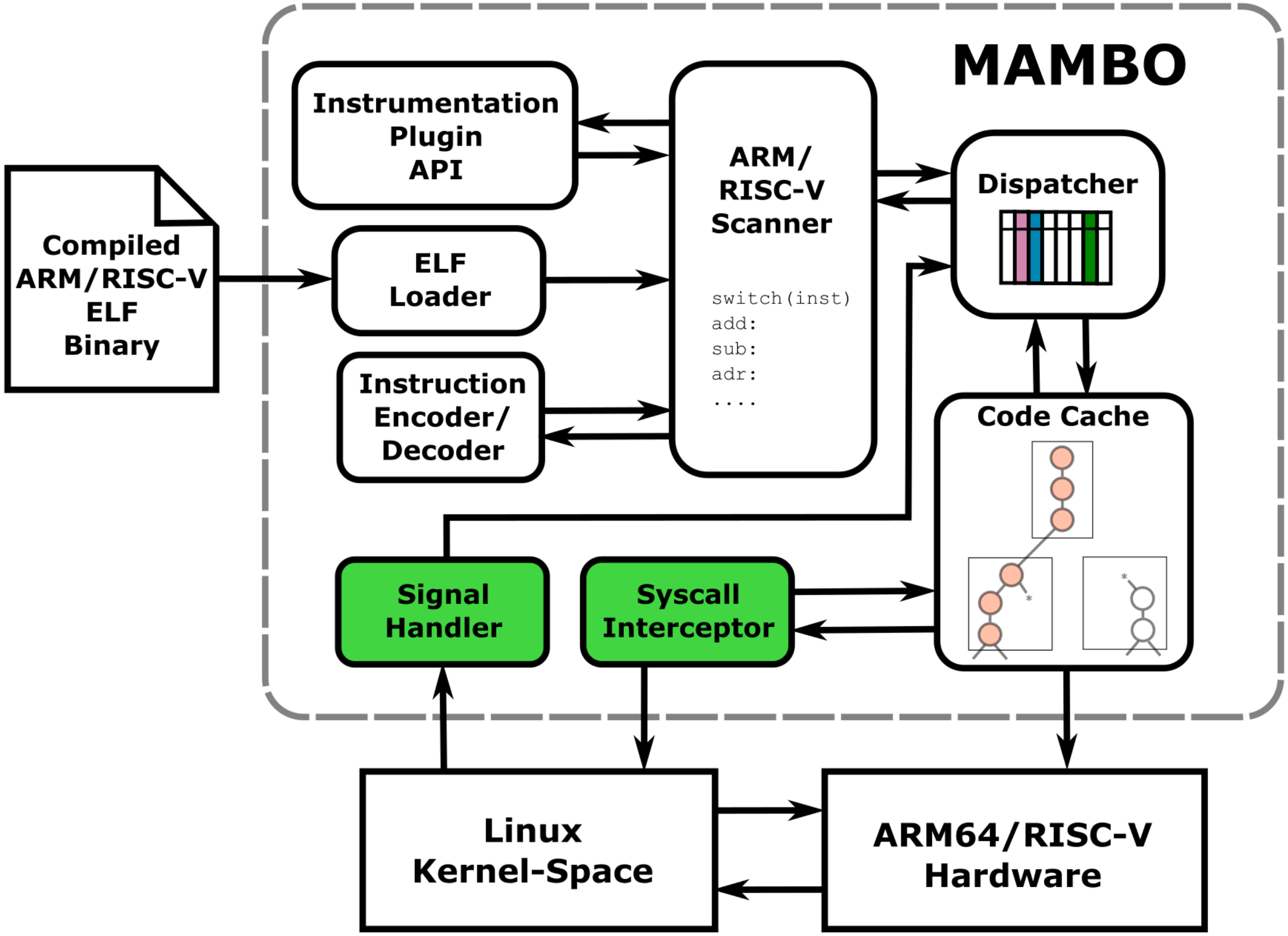


Dispatch and Code-Cache



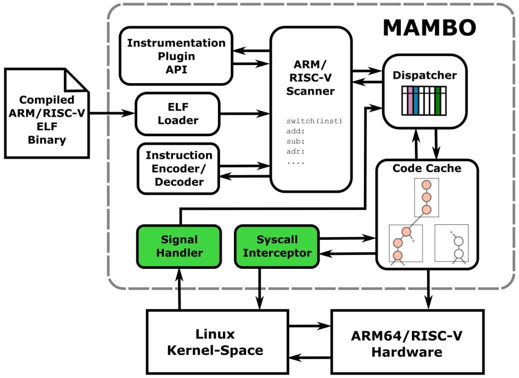


Kernel Interaction



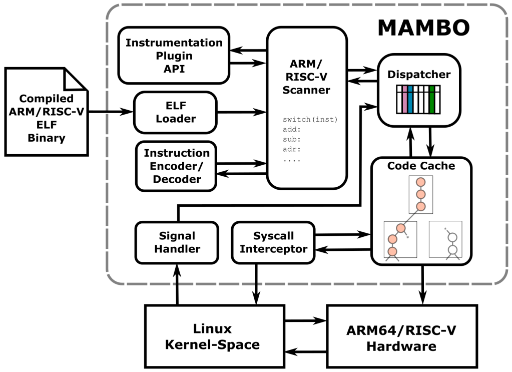
Kernel Interaction

- MAMBO sits in userspace between hosted application and the OS.
  - MAMBO cannot instrument the kernel.
- Some kernel interaction has consequences for MAMBO. So all kernel interaction from application must be wrapped and handled by MAMBO.
- Signals from kernel are passed to MAMBO. MAMBO handles passing signal to application.
  - MAMBO may need to modify stack to remove MAMBO data, allowing stack unwinding for example.
- Syscalls in application code become jumps back to MAMBO space.
  - Syscall not known until runtime.
  - Certain syscalls must be handled differently.
    - E.g. thread creation/destruction.



Main Points for MAMBO

- MAMBO and the hosted application share the same process
  - Thus share the same address space.
  - System registers and execution time are shared between MAMBO and the app.
  - MAMBO is not designed to secure itself against malicious activity from the application it is translating.
- Execution in the process swaps between MAMBO space (scanning & plugin callbacks) and actual application execution.
- MAMBO and the app are in userspace. No analysis in kernel space. Kernel interaction by app is wrapped and handled by MAMBO.

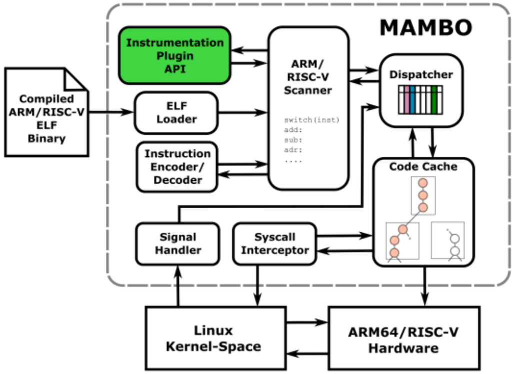


MAMBO Repo

```
./mambo
    ├── api/
    ├── arch/
    |    └── ...
    ├── docker/
    ├── docs/
    |    └── ...
    ├── elf/
    ├── pie/
    ├── plugins/
    |    └── ...
    └── test/
         └── ...
    | dbm.c/h
    | plugins.h
    | signals.c
    | syscalls.c/h
    | traces.c
    | utils.S/h
```

The University of Manchester

Introduction to MAMBO plugin API



MAMBO Plugins

- MAMBO Plugins use an event driven programming model
- Examples for the typical use case for plugins:
  - Code analysis
    - Building CFG
  - Code generation
    - Insertion of new functionality
  - Code modification
    - Reimplementation of library functions
    - Soft implementations of hardware instructions
  - Code instrumentation
    - Performance counters
    - Metrics on code hotspots
  - Runtime event handling
    - Tracking thread creation/destruction

Event Driven Programming Model

- User defined functions are registered as callbacks upon specific events.
- Internally MAMBO will execute callback functions when the event is encountered.
- Two categories of events:
  - Hosted application runtime events (need to be specially handled by MAMBO, e.g. system calls).
  - MAMBO scan-time events (most typically used for analysis and general instrumentation).
- Provides a simple way for a user to instrument/analyse their application.
  - Fine grained instrumentation (per-instruction).
  - Instrumentation on higher level abstractions, e.g:
    - thread creation/destruction.
    - calls to specific function symbols etc.

MAMBO API Event Flow

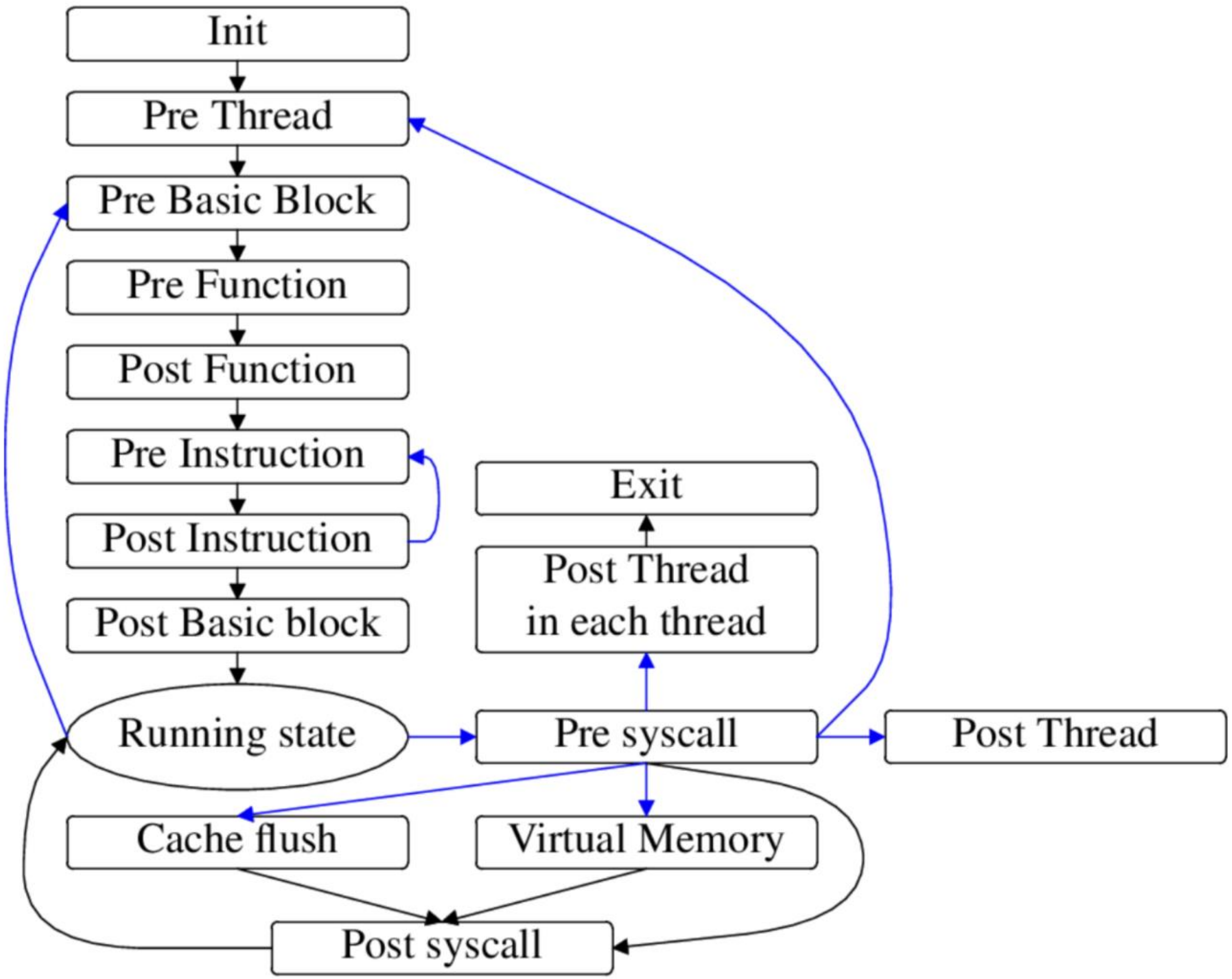


Image taken from: Gorgovan et al. Balancing Performance and Productivity for the Development of Dynamic Binary Instrumentation Tools: A Case Study on Arm Systems, p.3.

The University of Manchester

MAMBO API

- MAMBO's API consists of different types of functions for users:
  - Constructor function.
  - Callback registering functions.
  - Code analysis functions:
    - Get branch types.
    - Get addresses and fields.
  - Code instrumentation functions:
    - Set registers.
    - Perform memory operations (load/store).
    - Call user defined functions (C/ASM).
    - Insert other instructions.
  - Data structure helpers:
    - Hash table support.
    - Get/set private MAMBO data.

Constructor Function

- Used to register the plugin in MAMBO.
  - Required before main MAMBO program runs (hence compiler attribute).
  - Callbacks should also be registered in here...
  - Allocation of the *context* for the plugin should happen here. You will need:

```
_attribute__((constructor)) void <plugin name>() {
    mambo_context * ctx = mambo_register_plugin();
    ...
    ...
}
```

`_attribute__((constructor))`

The constructor attribute causes the function to be called automatically before execution enters main(). […] Functions with these attributes are useful for initializing data that will be used implicitly during the execution of the program.

Source: https://gcc.gnu.org/onlinedocs/gcc-4.7.0/gcc/Function-Attributes.html

MAMBO Context

```
mambo_context * ctx = mambo_register_plugin();
```

– Context provides necessary data structures which the user needs to analyse/instrument the code. Some useful fields are:

```
ctx->code.read_address // The untranslated application address of an instruction.
ctx->code.write_p      // The current code cache address to place the next instruction.
ctx->code.inst         // The enum of the decoded instruction.

// Thread private data can be stored and retrieved with these functions.
void * mambo_get_thread_plugin_data(mambo_context *ctx);
int mambo_set_thread_plugin_data(mambo_context *ctx, void *data);
```

Callback Registering Functions

– Used to call your own functions when a given event happens at scan/runtime (see plugin_support.h):

```
// Call <user function> before every instruction is scanned at SCAN-TIME.
int mambo_register_pre_inst_cb(mambo_context *ctx, &<user function> );

// Call <user function> before every basic block is SCAN-TIME.
int mambo_register_pre_basic_block_cb(mambo_context *ctx,
    &<user function> );

// Call <user function> after every thread is destroyed at RUNTIME.
int mambo_register_post_thread_cb(mambo_context *ctx, &<user function> );

// Call <user function> before a syscall is executed at RUNTIME.
int mambo_register_pre_syscall_cb(mambo_context *ctx, &<user function> );
```

Callback Registering Functions

– Used to call your own functions when a given event happens at scan/runtime (see plugin_support.h):

```
// Call <user fn pre> before and <user fn post> after a function with
symbol fn_name is scanned at SCAN-TIME.
int mambo_register_function_cb(
    mambo_context *ctx,
    char *fn_name,
    & <user fn pre> ,
    & <user fn post> ,
    int max_args);
```

Code Analysis Functions

– Used to help analyse code that triggered the callback functions (see plugin_support.h).

```
// 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);
```

Code Analysis Functions

– Used to help analyse code that triggered the callback functions (see plugin_support.h).

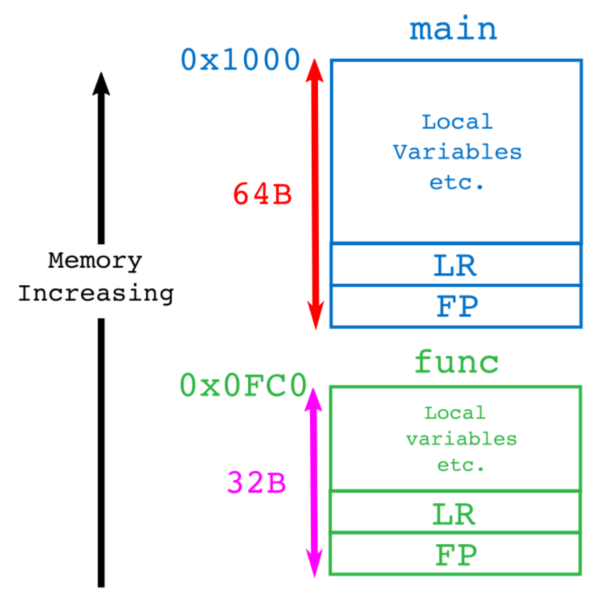
```
int plugin_pre_inst_handler(mambo_context *ctx) {
    mambo_branch_type type = mambo_get_branch_type(ctx);
    if (type & BRANCH_RETURN) {
        // Procedure return instruction
    } else if (type & BRANCH_DIRECT) {
        // Direct branch instruction }
    else if (type & BRANCH_INDIRECT) {
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

AARCH64 Assembly / ABI Recap

- Register file comprised of 32 registers (X0-X31).
  - Stack pointer (SP) = X31.
  - Link Register (LR) = X30.
  - Frame Pointer (FP) = X29.
- Arguments to/return values from function calls placed in X0-X7.
- Caller saved X9-X15.
- Callee saved X19-X28.
- Function prologue
  - Push LR & FP, first shifting stack pointer accordingly
- Function epilogue
  - Pop LR & FP, unwinding stack frame

```
main:
     stp  x29, x30, [sp, #-64]!
     mov  x29, sp
     ...
     bl  func
     ...

func:
     stp  x29, x30, [sp, #-32]
     ...
     ldp  x29, x30, [sp], #32
     ret
```



Code Instrumentation Functions

- Used to insert new code into the instruction stream of the basic block in the code cache.
- Some are safe (preserves stack and register state).
- Some are unsafe (you need to preserve stack/register integrity yourself).
- See helpers.h

```
void emit_counter64_incr(mambo_context *ctx, void *counter,
    unsigned incr);
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_fcall(mambo_context *ctx, void *function_ptr);
int  emit_safe_fcall(mambo_context *ctx, void *function_ptr,
    int argno);
int  emit_indirect_branch_by_spc(mambo_context *ctx, enum reg reg);
```

Hash Map Helper

- There is a hash map helper that can be used when instrumenting, to quickly store data and associate with particular code regions etc.
- Useful in the context of binary instrumentation, since lookup at runtime should be fast.

```c
int mambo_ht_init(mambo_ht_t *ht, size_t initial_size,
    int index_shift, int fill_factor, bool allow_resize);

int mambo_ht_add_nolock(mambo_ht_t *ht, uintptr_t key,
    uintptr_t value);

int mambo_ht_add(mambo_ht_t *ht, uintptr_t key, uintptr_t value);

int mambo_ht_get_nolock(mambo_ht_t *ht, uintptr_t key,
    uintptr_t *value);

int mambo_ht_get(mambo_ht_t *ht, uintptr_t key, uintptr_t *value);
```

Plugin API Scan-Time vs. Runtime

– Important to remember. Most common mistake when first writing plugins:

C:

```c
uint64_t run_many_times(uint64_t num) {
  return num * num;
}
```

ARM64:

```
run_many_times:
0x8000 sub sp, sp, #16
0x8004 str x0, [sp, 8]
0x8008 ldr x0, [sp, 8]
0x800C mul x0, x0, x0
0x8010 add sp, sp, 16
0x8014 ret
```

Plugin API Scan-Time vs. Runtime

```
char* message = "We are here\n";

int pre_inst_callback(mambo_context ctx*){
  if(ctx->code.read_address == 0x8000) {
        printf(message);
  }
  return 0;
}
```

```
run_many_times:
0xFC7000 sub sp, sp, #16
0xFC7004 str x0, [sp, 8]
0xFC7008 ldr x0, [sp, 8]
0xFC700C mul x0, x0, x0
0xFC7010 add sp, sp, 16
0xFC7014 ret
```

```
Output:
We are here!
```

```
char* message = "We are here!\n";

int pre_inst_callback(mambo_context ctx*) {
  if(ctx->code.read_address == 0x8000) {
    emit_push(ctx, (1 << 0));
    emit_set_reg(ctx, reg0, message);
    emit_safe_fcall(ctx, my_print_fn, 1);
    emit_pop(ctx, (1 << 0));
  }
return 0;
}
```

```
run_many_times:
0xFC7000 str x0, [sp,#8]
0xFC7014 mov x0, &message
0xFC7018 bl my_print_fn
0xFC7000 ldr x0, [sp,#-8]
0xFC7004 sub sp, sp, #16
0xFC7008 str x0, [sp, 8]
0xFC700C ldr x0, [sp, 8]
0xFC7010 mul x0, x0, x0
0xFC701c add sp, sp, 16
0xFC7020 ret
```

```
Output:
We are here!
We are here!
We are here!
We are here!
...
```

Coffee Break (30 min)

SET UP MAMBO!

IT TAKES 30 - 60 MIN WITH QEMU!

http://tinyurl.com/mambo-docker-setup

MAMBO use-case and example plugin

The following code example can be found at:

https://github.com/beehive-lab/mambo/blob/master/plugins/

Example Plugin: Branch Counter

1. Build the constructor and add required data structures.

```
#ifdef PLUGINS_NEW
...

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

  mambo_register_pre_inst_cb(ctx, &branch_count_pre_inst_handler);
  mambo_register_pre_thread_cb(ctx, &branch_count_pre_thread_handler);
  mambo_register_post_thread_cb(ctx, &branch_count_post_thread_handler);
  mambo_register_exit_cb(ctx, &branch_count_exit_handler);

  setlocale(LC_NUMERIC, "");
}

...
#endif
```

Example Plugin: Branch Counter

1. Build the constructor and add required data structures.

```
#ifdef PLUGINS_NEW
...

struct br_count {
  uint64_t direct_branch_count;
  uint64_t indirect_branch_count;
  uint64_t return_branch_count;
};

struct br_count global_counters;

...
#endif
```

Example Plugin: Branch Counter

2. Create pre_thread handler to set up initial state.

```
#ifdef PLUGINS_NEW
...

int branch_count_pre_thread_handler(mambo_context *ctx) {
  struct br_count *counters = mambo_alloc(ctx, sizeof(struct br_count));
  assert(counters != NULL);
  mambo_set_thread_plugin_data(ctx, counters);

  counters->direct_branch_count = 0;
  counters->indirect_branch_count = 0;
  counters->return_branch_count = 0;
}

...
#endif
```

Example Plugin: Branch Counter

3. Create pre_inst handler which has the core function of the plugin.

```
#ifdef PLUGINS_NEW
...

int branch_count_pre_inst_handler(mambo_context *ctx) {
  struct br_count *counters = mambo_get_thread_plugin_data(ctx);
  uint64_t *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);
  }
}
...
#endif
```

Example Plugin: Branch Counter

4. Create post_thread handler which clears up after program execution.

```
#ifdef PLUGINS_NEW
...

int branch_count_post_thread_handler(mambo_context *ctx) {
  struct br_count *counters = mambo_get_thread_plugin_data(ctx);

  fprintf(stderr, "Thread: %d\n", mambo_get_thread_id(ctx));
  print_counters(counters);
  atomic_increment_u64(&global_counters.direct_branch_count,
                       counters->direct_branch_count);
  atomic_increment_u64(&global_counters.indirect_branch_count,
                       counters->indirect_branch_count);
  atomic_increment_u64(&global_counters.return_branch_count,
                       counters->return_branch_count);
  mambo_free(ctx, counters);
}

...
#endif
```

Example Plugin: Branch Counter

The University of Manchester

5. Create exit handler which displays results before termination.

```
#ifdef PLUGINS_NEW
...

void print_counters(struct br_count *counters) {
  fprintf(stderr, "  direct branches: %'" PRIu64 "\n",
   counters->direct_branch_count);
  fprintf(stderr, "  indirect branches: %'" PRIu64 "\n",
   counters->indirect_branch_count);
  fprintf(stderr, "  returns: %'" PRIu64 "\n",
   counters->return_branch_count);
}

int branch_count_exit_handler(mambo_context *ctx) {
  fprintf(stderr, "Total:\n");
  print_counters(&global_counters);
}

...
#endif
```

lantj@lantj: /mnt/josh_stuff/mambo_tutorial/mambo ⌥⌘1

lantj@softwood: ~ (-zsh) ⌘1 | ...ambo_tutorial/mambo (ssh) ⌘2 | ...ddc: ~/riscv (com.docker.cli) ⌘3 | tmux ⌘4

```
lantj@lantj:~$ 
```

```
[0] 0:bash  1:bash  2:bash  3:vim  4:bash- 5:bash*                   "lantj" 15:27 11-Jan-24
```

The University of Manchester

SimAcc: A Configurable Simulator for customised accelerators on SoCs with FPGAs.

SimAcc: A Configurable Cycle-Accurate Simulator for Customized Accelerators on CPU-FPGAs SoCs
https://ieeexplore.ieee.org/document/8735519

Search Wikipedia
Search

Multi-tool

Article Talk

From Wikipedia, the free encyclopedia

A **multi-tool** (or **multitool**) is a hand tool that combines several individual functions in a single unit. The smallest are credit-card or key sized units designed for carrying in a wallet or on a keyring, but others are designed to be carried in a trouser pocket or belt-mounted pouch.

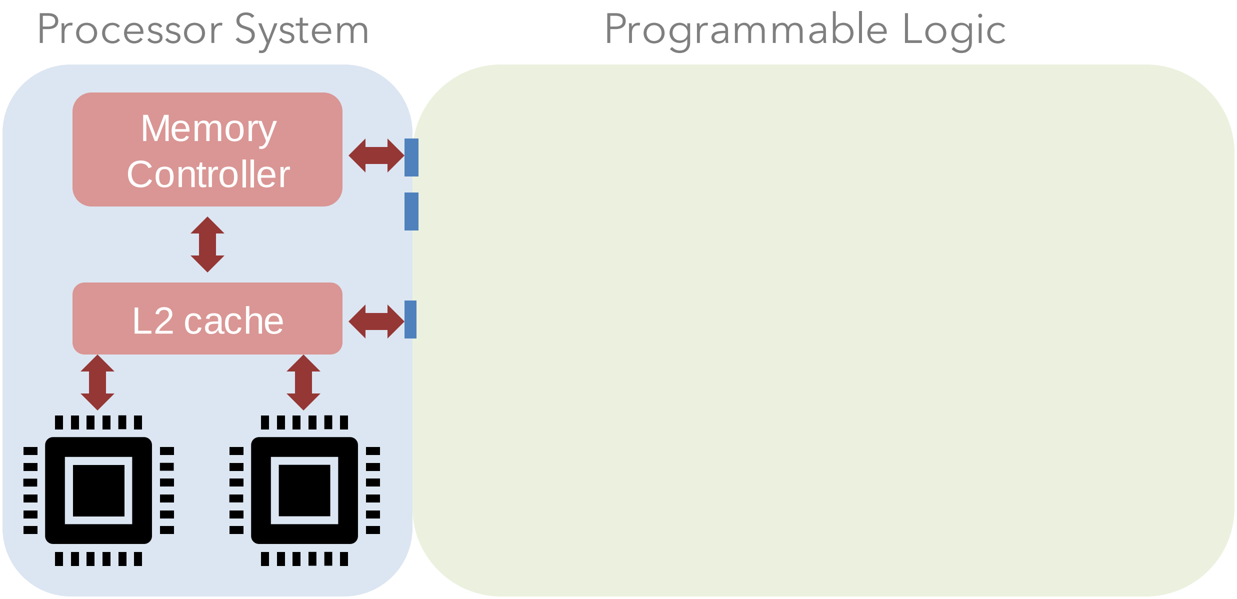
Motivation.

A trend for System-on-Chips is to include application specific accelerators on the die. However:

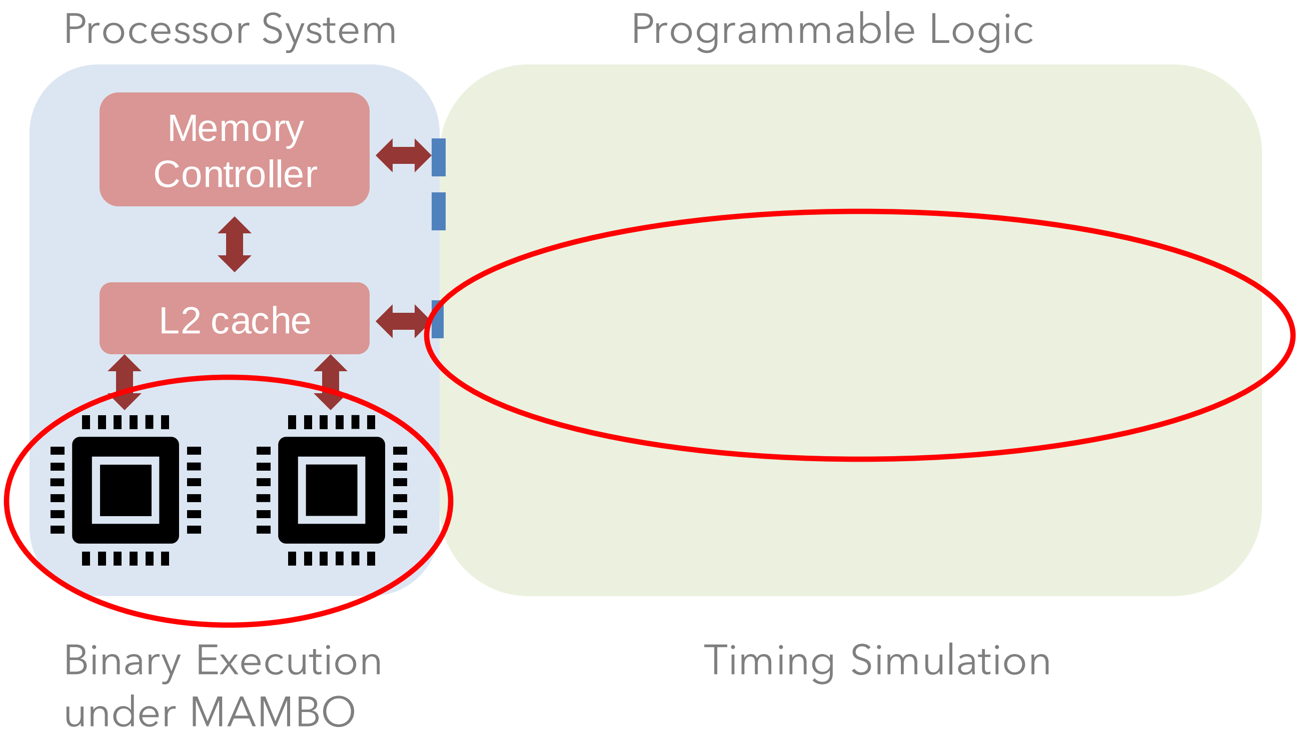
- Could we design a simulation infrastructure for **fast** computer architecture simulation and **prototyping of accelerator IP** on SoCs with FPGAs?

- How do **hardware accelerators interact** with the processors of a system and what is the **impact** on overall performance?

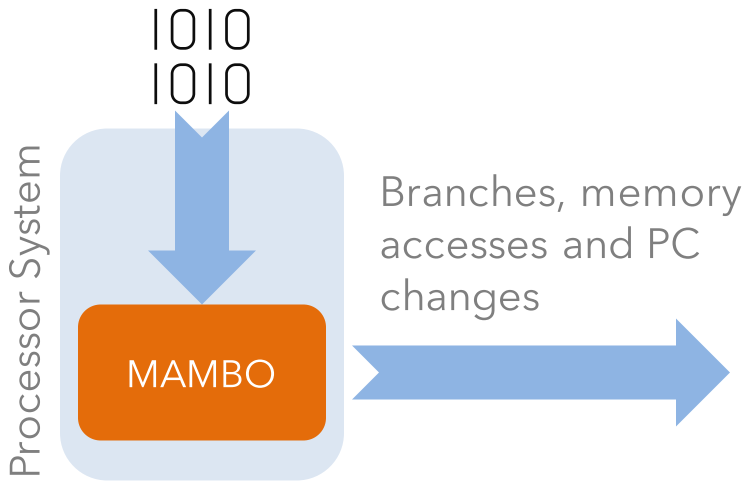
Simulation Infrastructure.



Simulation Infrastructure.

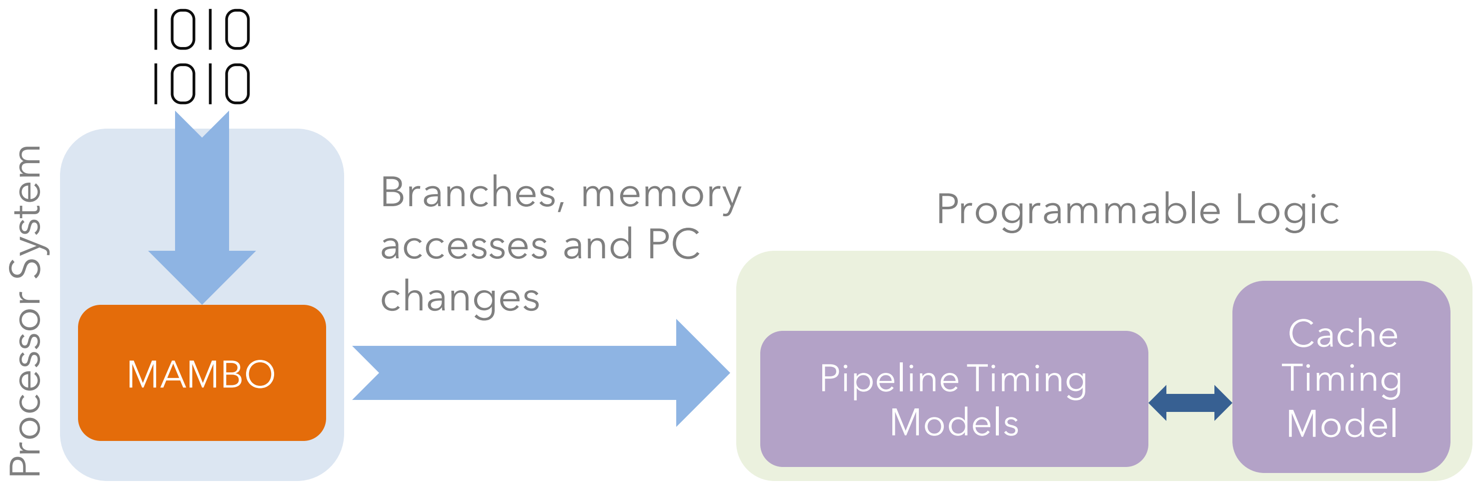


Simulation Infrastructure.



MAMBO plugin consists of a set of callbacks (are executed at various points of the program execution) and drives the hardware models on the Programmable Logic Part of the host platform.

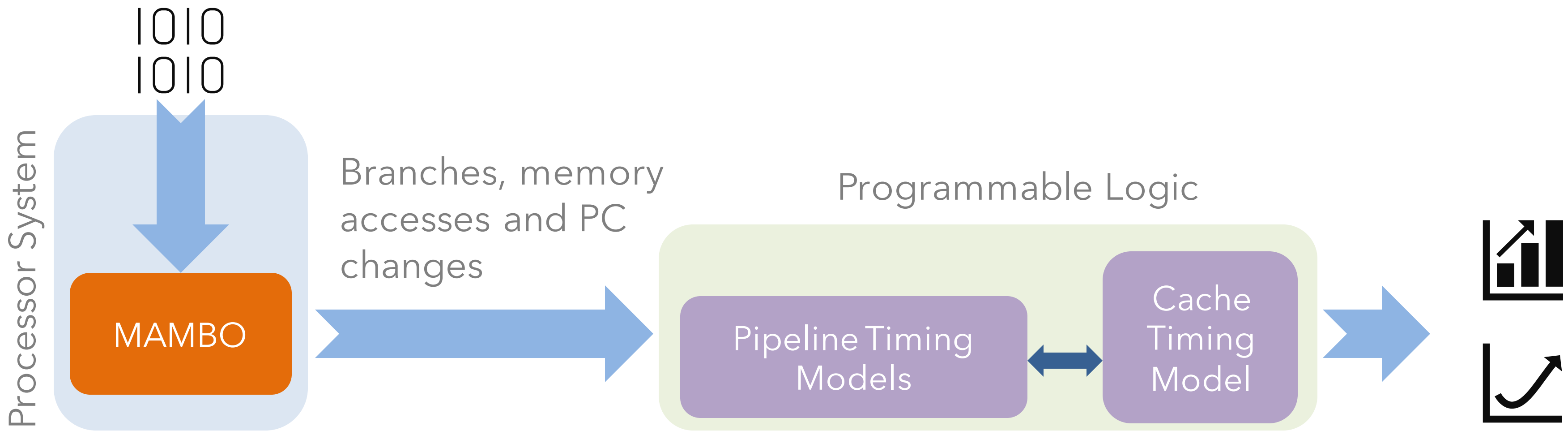
Simulation Infrastructure.



- Pipeline Model (inspired by Arm Cortex-A9)
    - Re-order buffer, register renaming module, branch prediction models, Branch Target Buffer (BTB), return address stack (RAS), and LD/ST queue.

- Cache System Model
    - It gathers statistics about the behaviour of a cache system. The model stores only address tags and states for cache lines.

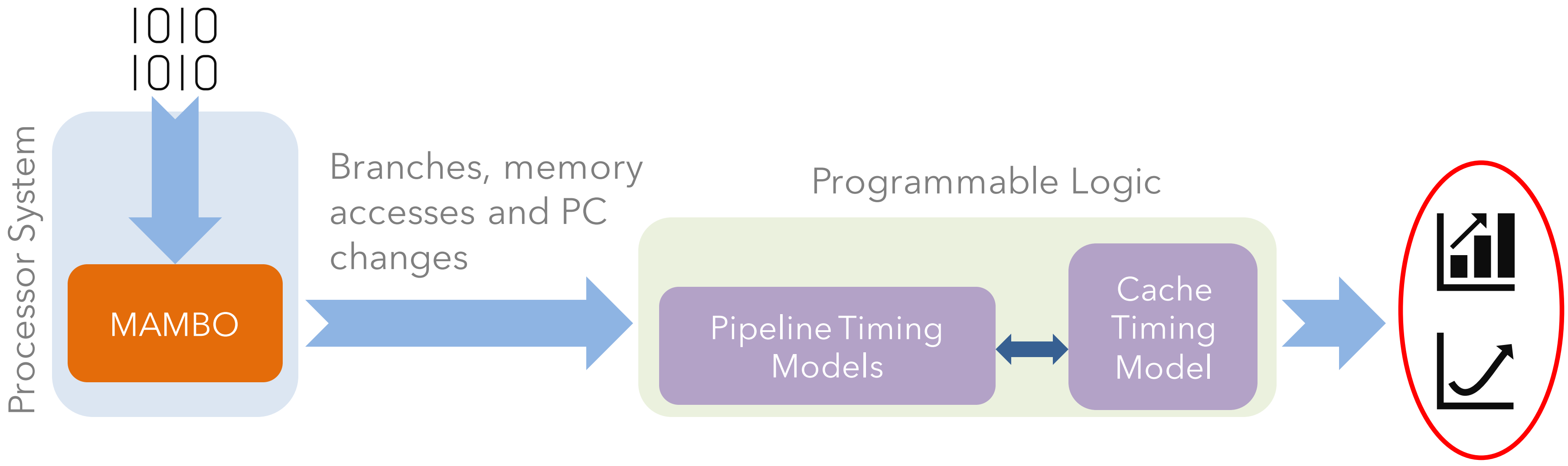
Simulation Infrastructure.



- Pipeline Model (inspired by Arm Cortex-A9)
  - Re-order buffer, register renaming module, branch prediction models, Branch Target Buffer (BTB), return address stack (RAS), and LD/ST queue.

- Cache System Model
  - It gathers statistics about the behaviour of a cache system. The model stores only address tags and states for cache lines.

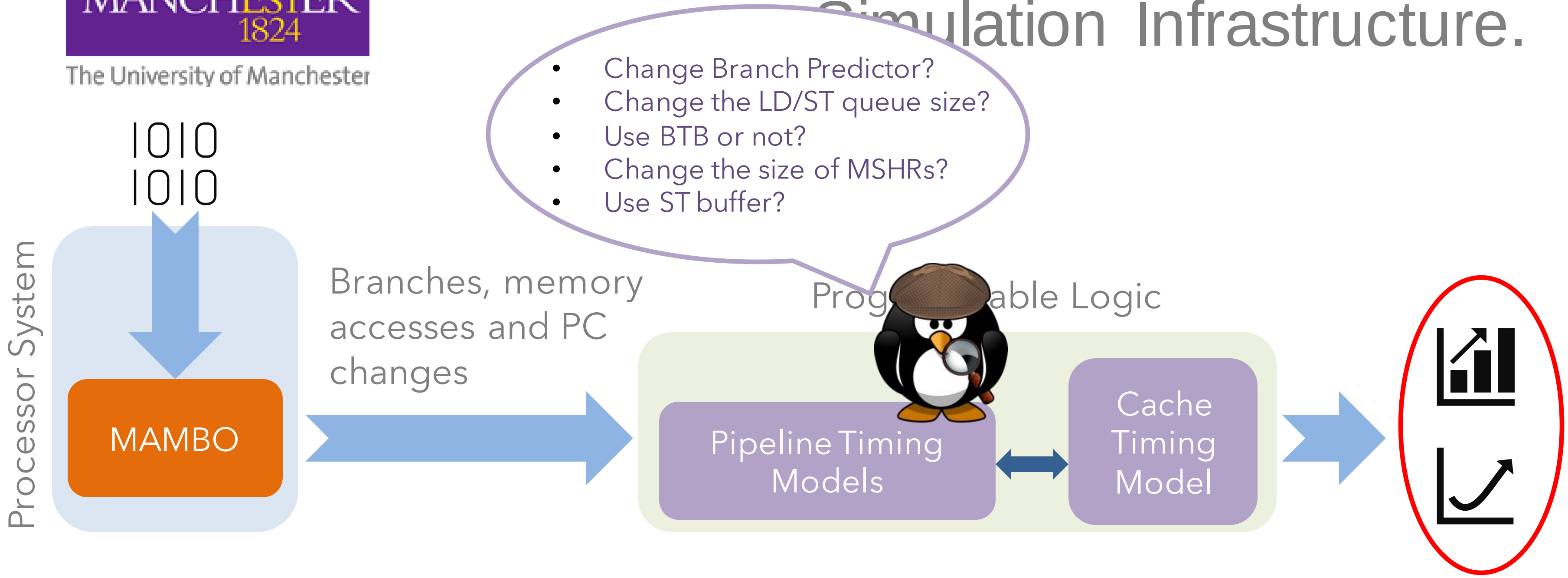
Simulation Infrastructure.



- Pipeline Model (inspired by Arm Cortex-A9)
  - Re-order buffer, register renaming module, branch prediction models, Branch Target Buffer (BTB), return address stack (RAS), and LD/ST queue.

- Cache System Model
  - It gathers statistics about the behaviour of a cache system. The model stores only address tags and states for cache lines.

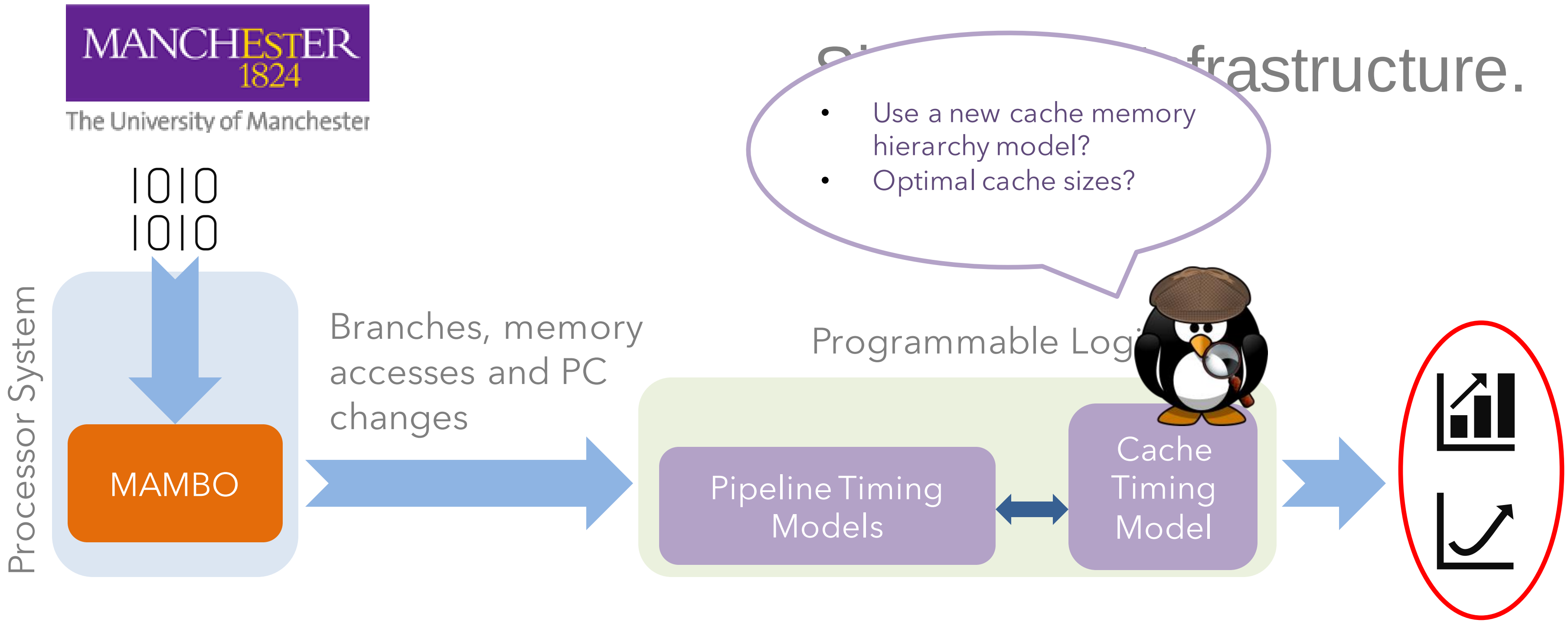
Simulation Infrastructure.



- Pipeline Model (inspired by Arm Cortex-A9)
  - Re-order buffer, register renaming module, branch prediction models, Branch Target Buffer (BTB), return address stack (RAS), and LD/ST queue.

- Cache System Model
  - It gathers statistics about the behaviour of a cache system. The model stores only address tags and states for cache lines.

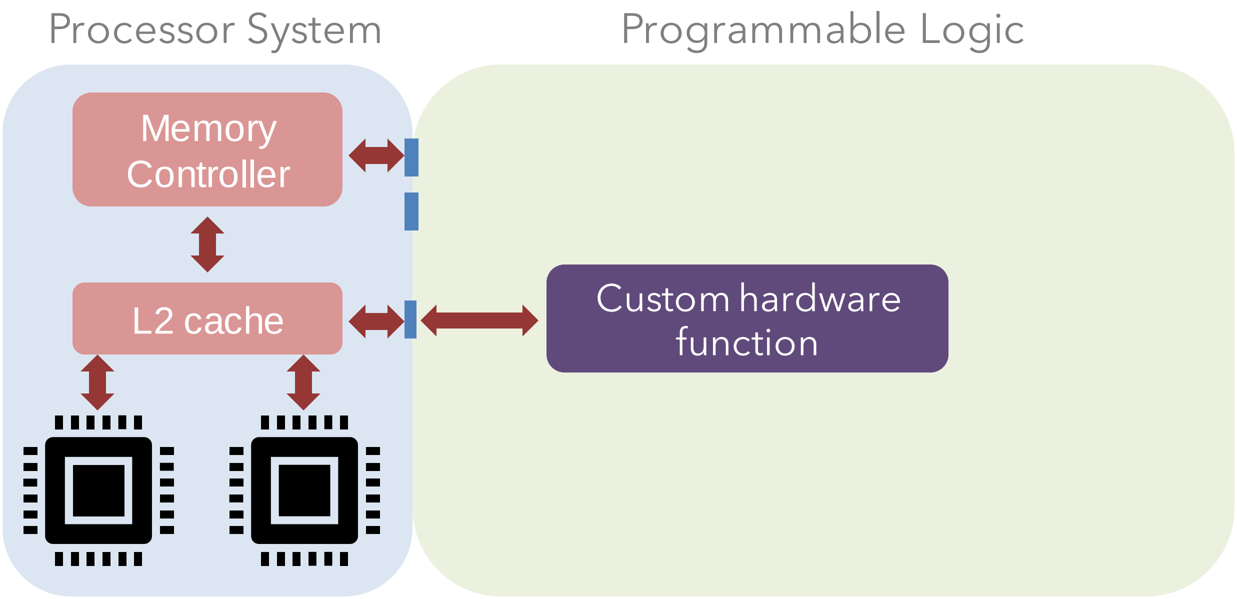
frastructure.



- Pipeline Model (inspired by Arm Cortex-A9)
  - Re-order buffer, register renaming module, branch prediction models, Branch Target Buffer (BTB), return address stack (RAS), and LD/ST queue.

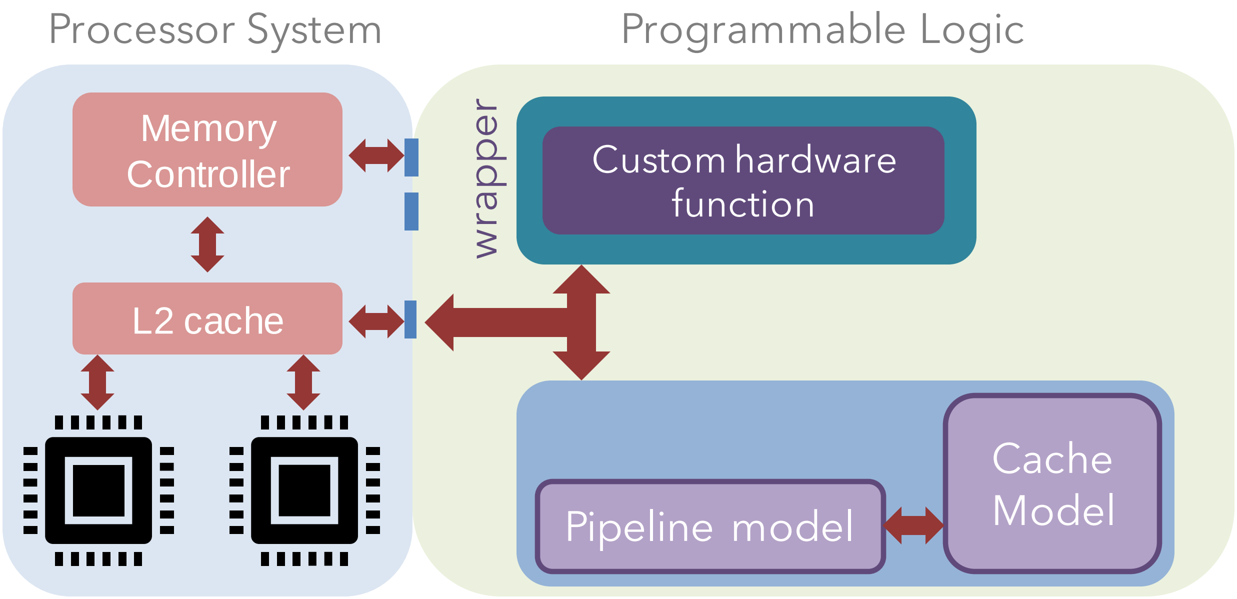
- Cache System Model
  - It gathers statistics about the behaviour of a cache system. The model stores only address tags and states for cache lines.

Simulation of Accelerated Apps.

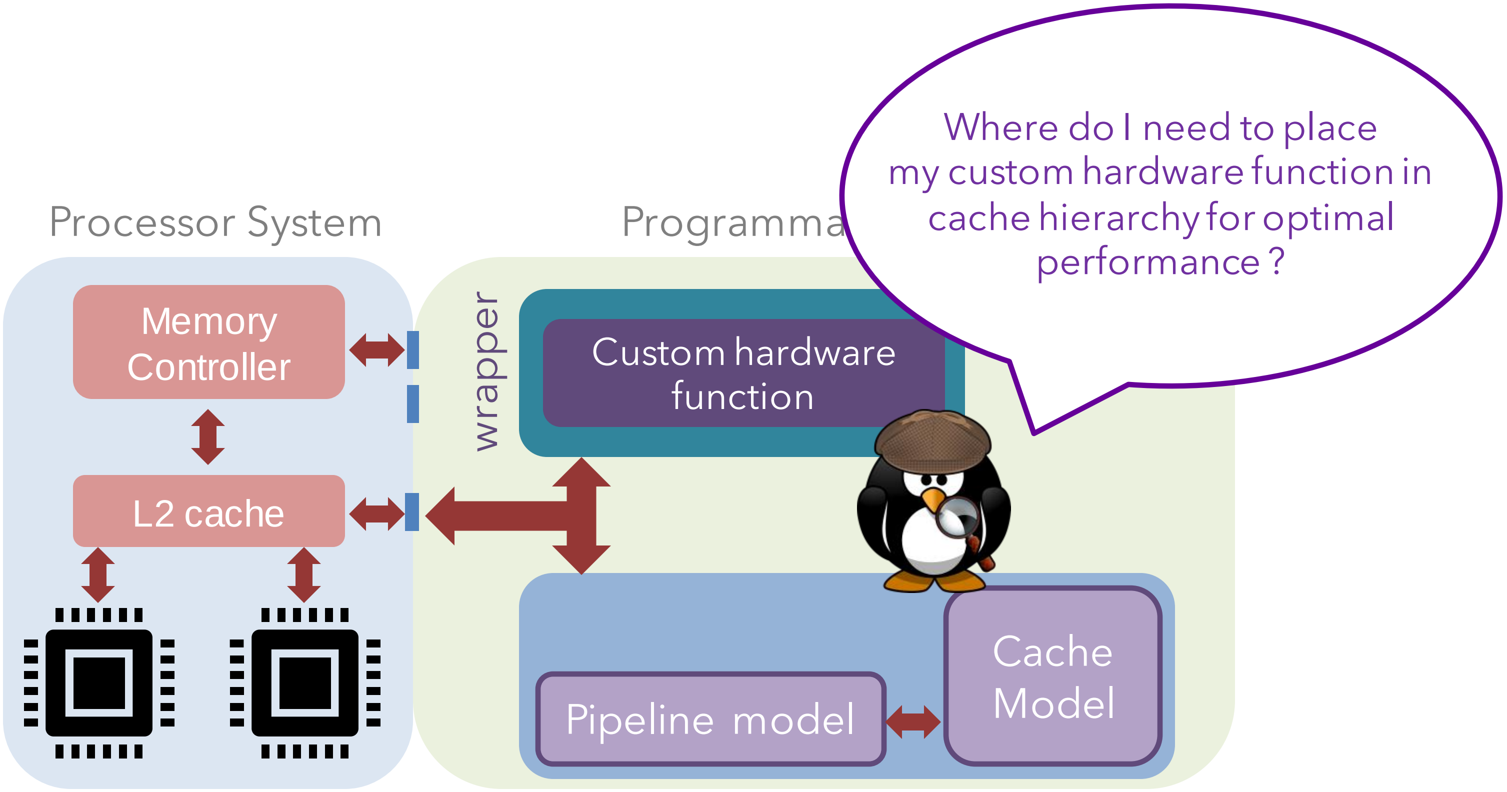


The University of Manchester

Simulation of Accelerated Apps.



Simulation of Accelerated Apps.



Hands On MAMBO session

Hands on MAMBO Session

Lab script:

Docs/ in MAMBO Repository
http://tinyurl.com/mambo-tutorial

Follow the instructions in the document

4 Main Exercises + Introduction + Extras

IMPORTANT!

PLEASE ASK US QUESTIONS!

MAMBO Roadmap

- Foster an open-source community
  - Collaborations/Contributions welcome
- Improve Documentation
- More tools
  - data race detector
  - call graph
- Keep up with Arm and RISC-V
- Current research projects
  - fast architectural simulation
  - cybersecurity
  - binary lifting

Thanks to the Digital Secure by Design programme

End of the tutorial

MAMBO Tutorial at HiPEAC 2024 - Feedback Form