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

DOI:
https://doi.org/10.1145/3377555.3377895

Citation for published version (APA):
Gorgovan, C., Callaghan, G., & Luján, M. (2020). Balancing Performance and Productivity for the Development of Dynamic Binary Instrumentation Tools - A Case Study on Arm Systems. In Proceedings of the 29th International Conference on Compiler Construction (CC ’20) Association for Computing Machinery. https://doi.org/10.1145/3377555.3377895

Published in:
Proceedings of the 29th International Conference on Compiler Construction (CC ’20)

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

Cosmin Gorgovan
Department of Computer Science
University of Manchester
United Kingdom
cosmin.gorgovan@manchester.ac.uk

Guillermo Callaghan
Department of Computer Science
University of Manchester
United Kingdom
guillermo.callaghan@manchester.ac.uk

Mikel Luján
Department of Computer Science
University of Manchester
United Kingdom
mikel.lujan@manchester.ac.uk

Abstract

Dynamic Binary Instrumentation (DBI) is a well-established approach for analysing the execution of applications at the level of machine code. DBI frameworks implement a runtime system capable of modifying running applications without access to their source code. These frameworks provide APIs used by DBI tools to plug in their specific *analysis* and *instrumentation* routines. However, the dynamic instrumentation needed by these DBI tools is either challenging to implement, and/or introduces a significant performance overhead.

An added complexity beyond the well studied scenario of x86 and x86-64, is that state-of-the-art Arm systems (i.e. Arm v8) introduced a distinct 64-bit execution mode with a new redesigned instruction set. Thus, Arm v8 is a computer architecture which contains three instruction sets. This further complicates the development of DBI tools which can work for both 32-bit Arm (includes the A32 and T32 instruction sets), and 64-bit (the A64 instruction set).

This paper presents the design of a novel DBI framework API that provides support both for portable (across A32, T32 and A64), and for native-code-level analysis and instrumentation, which can be intermixed freely. This API allows DBI tool developers to balance performance and productivity at a fine-grain level. The API is implemented on top of the MAMBO DBI system.

CCS Concepts • **Software and its engineering → Just-in-time compilers**; **Runtime environments**.

Keywords Dynamic Binary Instrumentation, Memory Error Checking, Online Cache Simulation, Arm

CC '20, February 22–23, 2020, San Diego, CA, USA

ACM Reference Format:
Cosmin Gorgovan, Guillermo Callaghan, and Mikel Luján. 2020. Balancing Performance and Productivity for the Development of Dynamic Binary Instrumentation Tools: A Case Study on Arm Systems. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20), February 22–23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3377555.3377895

1 Introduction

Dynamic Binary Instrumentation (DBI) frameworks provide APIs which are used by external tools to implement dynamic code analysis and instrumentation. For example, well-known tools for detecting memory errors rely on DBI frameworks such as Valgrind [18], Pin [13] and DynamoRIO [3]. Further examples of tools implemented using DBI frameworks include microarchitectural simulators [6, 14, 15], development tools such as memory error checkers [2, 18] and application analysis tools such as taint tracers [7]. The design of such an API is a three-way trade-off between programming ease, flexibility and performance. Furthermore, architectures supporting multiple instruction sets and ISA heterogeneity raise the question of how to implement portable instrumentation efficiently.

This paper presents the functionality required to implement efficiently common DBI tasks and to show a novel API which provides it in a portable manner, while also allowing low level access to the machine code, for detailed analysis or for generating high performance instrumentation. The proposed API is based on two layers: a low level layer allows code analysis and generation directly at the machine code level for full control. On top of the low level layer, a high level layer allows the development of portable instrumentation using: (1) a generic RISC-like instruction set for generating instrumentation, which is efficiently mapped by the runtime to any of the supported native instruction sets, (2) code analysis functions which abstract the decoding of application code, and (3) code generation helpers for a number of common DBI tasks, such as updating counters at runtime. The use of the two layers can be interleaved inside a single block of instrumentation, such that the boilerplate code (e.g.

pushing register values on the stack) can be generated using the high level, while specialised or performance-critical code can be implemented separately for each instruction set to take advantage of their specific features.

We have implemented the API on top of the existing Dynamic Binary Modification (DBM) system MAMBO [8], with support for AArch32 - both the A32 and T32 instruction sets - and AArch64 using the A64 instruction set. We did this by extending the existing API of MAMBO - which exposed to the plugins the same low level functionality used internally - by allowing the plugins to observe new execution events specifically selected to facilitate the implementation of instrumentation tasks, and with an abstracted and portable layer for code analysis and generation.

The contributions of this paper are:

- identifying the requirements for a general purpose DBI API;
- proposing 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;
- implementing this API on top of an open source DBM system; and
- implementing practical DBI tools (a memory error checker and an online cache simulator) using this API and evaluating their performance against similar tools.

In the evaluation, both DBI tools are compared for multi-threaded scaling and single threaded performance against state-of-the-art tools in Valgrind and DynamoRIO. The experiments demonstrate comparable or improved performance compared to state-of-the-art implementations.

2 Related Work

In 1994 Hollingsworth *et al.* [10] proposed a highly abstracted and partly ISA-independent dynamic instrumentation model, in which users define their instrumentation tools using a predefined set of *metrics* (e.g. processor usage, execution counters) that can be measured for predefined *resources* (such as processing cores). However, this is not a general-purpose approach because it is limited to the resources and metrics supported by the DBI system.

More recently, a number of other DBI systems have defined APIs related to the API that we propose. Valgrind [17] is a free software DBI framework for heavyweight instrumentation. Its design prioritises convenience and developer productivity over performance. In particular, Valgrind is notable for using a Disassemble-and-Resynthesise (D&R) code scanner, which lifts the native code to a single-static-assignment Intermediate Representation (IR) level. The IR is then used for code analysis and inserting instrumentation and is compiled back to native code at runtime. This approach allows implementing ISA-independent instrumentation, however the abstraction hides away the original instruction stream, which makes it challenging to implement certain types of DBI tools, such as microarchitectural simulators, which need access to the original code of the application. Furthermore, this approach does not allow developers to use explicitly ISA-specific features, which can often be useful for minimising the overhead of the instrumentation. Valgrind serialises all application threads to simplify its usage, however this comes at a high cost by forgoing performance improvements on multicore machines, as shown in our evaluation (Section 7).

DynamoRIO [3] is a free software low overhead DBI framework which provides a flexible instrumentation API. Its API provides facilities both for inserting inline instrumentation and for calling C/C++ functions. Its code generation API has some support for ISA-independent code analysis and generation, based on an IR specific to each architecture. Certain classes of IR instructions are only available on a subset of the supported platforms. On Arm, DynamoRIO uses thread-shared code caches, limiting the scalability of multi-threaded instrumentation, especially for tools which collect thread-private data as they are required to perform expensive runtime thread-level-storage lookups, e.g. memory error checking.

Pin [13] is a proprietary DBI framework developed by Intel for the x86 and x86-64 architectures. Its API has a narrow usage model: instrumentation is done at a high level of abstraction, with two usage models. The first model is to insert calls to functions implemented in C/C++, which can have high overhead compared to inline instrumentation. The second model is to use a *Buffering* API, for collecting runtime data in buffers which are periodically batch processed by C/C++ functions. The latter model is not suitable for DBI tools which need to process data in real-time, e.g. memory error checking. Pin++ [9] is a framework which may be used on top of Pin to improve the developer productivity using object-oriented design and template metaprogramming.

Dyninst [4, 19] is a free software framework for patching applications. As opposed to the other systems discussed in this section and to MAMBO, it is based on code patching, rather than DBM. Its modification system is not guaranteed to maintain correctness because of the code discovery problem: for non-trivial programs, it is impossible to accurately determine which locations contain code and which locations contain data ahead of time. Therefore it is susceptible to incomplete coverage, when code is misidentified as data and incorrect modification, when data is misidentified as code [11]. Additionally, static code patching cannot be used on Just-In-Time compiled and self-modifying code (e.g. JVM and V8 for Javascript).

Lui *et al.* [12] developed a workload analysis methodology which provides a unified abstraction layer on top of multiple DBI systems (and other types of application analysis tools). Their system provides separate implementations for each supported DBI system, using their native APIs.

We have considered implementing an API compatible with an existing DBI system for MAMBO. However, in practice the APIs of DBI systems are coupled with the internal implementation of such systems (e.g. the API of Valgrind works on IR because Valgrind uses a D&R approach). Therefore, attempting to reimplement an existing API for a different DBI system would constitute a larger engineering challenge. Furthermore, we could minimise the complexity of the API by refining its scope to the features required for the Arm architecture (i.e. an architecture with separate 32-bit and 64-bit execution modes and three separate but similar instruction sets).

3 API Design and High Level Overview

DBI consists of an analysis pass, in which the code of the application is inspected (e.g. to identify instructions of a specific type in the application) and an instrumentation pass, in which instrumentation code is inserted at specific locations in the application. A high level overview is shown in Figure 1: DBI frameworks work by *scanning* the code of the application and then passing it to the plugin. The plugin *analyses* and *instruments* the code, before returning control back to the DBI framework, which *translates* the code as necessary to maintain correct execution. Code is processed at the level of *basic blocks* (BBs) - single-entry and single-exit linear regions. To minimise overhead, the resulting instrumented code is stored in a *code cache*, from where it can subsequently execute directly. This processing and the execution of the instrumented code are interleaved at runtime.

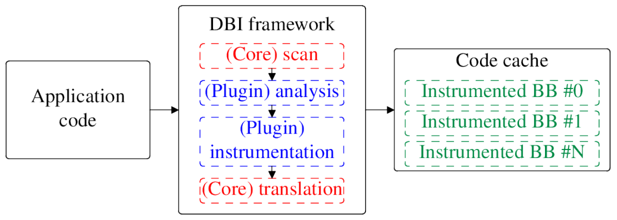

Figure 1. DBI overview.

The programming model is event-driven, with plugins registering callback functions (event handlers) for specific events. The API defines when these callbacks are executed in relation to the triggering event and to each other, with these constraints being designed to facilitate ease of use. Events are further discussed in Section 4.

The instrumented application, the runtime system, and the plugin execute in the same process and address space. Memory allocated in a plugin event handler can be accessed by the inserted instrumentation and vice versa. Shared buffers are the main communication mechanism between instrumentation and event handlers.

Plugins can analyse the code either by using architecture-independent decoding helpers, that are provided by the API for specific classes of instructions (e.g. branches, loads and stores) or by inspecting the instruction set-specific opcodes and, optionally, by decoding the full instructions using instruction decoding helpers.

Instrumentation is inserted at specific locations in the application, selected at the granularity of instructions, basic blocks or functions, using the relevant API event type. Instrumentation can be done at three levels of abstraction:

- native code, when the plugin explicitly selects each machine instruction to be emitted;
- generic RISC code, which automatically generates the corresponding native instruction for a number of common operations (e.g. add, multiply); or
- using helpers to generate common DBI tasks (e.g. atomically incrementing a counter).

4 API Events

Table 1 summarises the events provided by the API, with entries in bold marking the new events and descriptions in italics marking new or extended functionality for pre-existing events. Plugins can register callback functions for any combination of events. Events without a registered callback are ignored. *Thread*, *Exit* and *Virtual memory* are raised for specific system calls and therefore could be considered subsets of the *Syscall* events. However, they are either used ubiquitously (*Thread* and *Exit*) or involve intercepting and abstracting a larger number of system calls (*Virtual memory*).

The pre and post *Basic Block*, *Instruction* and *Function* events are raised as part of the code scanning process, which takes place when control in the application is passed to code which does not yet have a translation in the code cache. These events allow the plugins to analyse the application code and/or to insert their instrumentation.

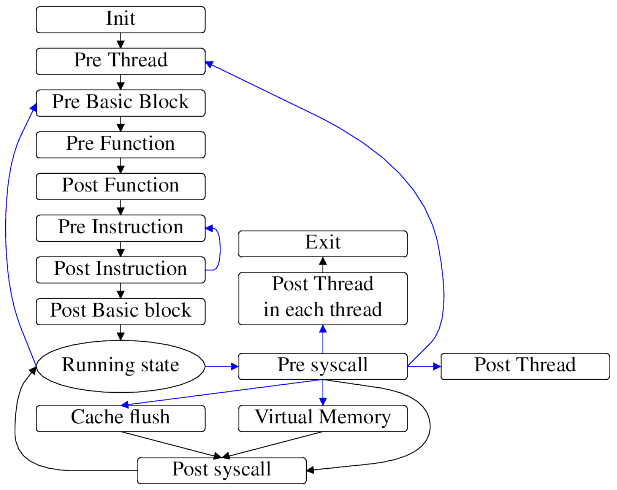

Figure 2. The order in which API events may be delivered.

Figure 2 shows the order in which events may be delivered to a plugin in each thread. Each box represents a type

Table 1. Summary of DBI events. The column *Inst.* marks the events which allow the insertion of instrumentation code.

Event name	Type	Inst.	Description	Purpose
Init	pre	N	runs before the application is loaded	allocating global resources, initialising global data, *shaping the address space for shadow memory*
Thread	pre	N	runs before each application thread is started	tracking active threads; allocating and initialising thread-private resources
	post	N	runs just before *any application thread exits* either via thread termination *or application exit*	tracking active threads; *aggregating and outputting data from thread-level analysis and instrumentation*; releasing thread-private resources
Exit	pre	N	runs just before the application exits	aggregating and outputting data from application-wide analysis and instrumentation
Syscall	pre	N	runs before a system call of the application is passed to the OS or internal DBI emulation routines	inspecting and/or modifying system call arguments; emulating system calls in the plugin without passing them to the DBI system or the OS
	post	N	runs after a system call has been executed, before passing the result to the application	inspecting or modifying the return value of system calls
Virtual memory	post	N	abstracts OS system calls which map, unmap or change the permissions of virtual memory	particularly useful for implementing shadow memory
Basic block	pre	Y	runs just before scanning a single-entry and single-exit code region	can generate basic block-level instrumentation
	post	Y	runs after scanning a single-entry and single-exit code region	can be used to backpatch instrumentation in the basic block based on information not available earlier (e.g. basic block size)
Instruction	pre	Y	runs before an instruction in the application is copied or translated to the code cache	inserting instruction-level instrumentation before the associated instruction has executed
	post	Y	runs after an instruction in the application was copied or translated to the code cache	inserting instruction-level instrumentation after the associated instruction has executed
Function	pre	Y	runs before the entry point of a specific function is translated to the code cache	inserting instrumentation which executes before specific functions; replacing specific functions
	post	Y	creates a wrapper around specific functions to intercept their return to the callee	inserting instrumentation which executes immediately after specific functions return
Code cache flush	pre	N	runs before the DBI code cache is flushed	allows plugins to release resources associated with specific parts of the code cache

of event, while the *Running state* represents the execution of the instrumented code. It is important to underline the difference between the callback functions, which execute according to the provided model, and the instrumentation that they might insert, which executes in the *Running state* only, interleaved with the application code. The arrows show the allowed ordering, with the blue arrows representing conditional transitions. For example, a *Virtual Memory* event is delivered between the pre and post *Syscall* events only if the application executes a system call related to virtual memory management.

Events of the same type are not nested, as shown in Figure 2. After a *Pre Basic Block* event, it is guaranteed that the matching *Post Basic Block* will be delivered before any other *Pre Basic Block* event. This property holds even if execution is interrupted by asynchronous events (e.g. by signals to the application) during the scanning of a basic block. It allows plugins to maintain persistent data across events in a simple thread-private structure. A common usage pattern is to initialise structures related to the analysis of a basic block using the *Pre Basic Block* event, update these structures in each *Pre Instruction* event and then process the results in the *Post Basic Block* event.

The interaction of application threads and API events can lead to subtle bugs or a challenging programming model if not carefully designed. In the proposed API, we have attempted to move as much of the burden of handling threads to the underlying runtime system and to simplify the model exposed to the end-user. Towards that end, the *Pre Thread* and *Post Thread* callbacks are guaranteed to execute for every single thread used by the application, including the initial thread and all threads active at the time the application exits. This model facilitates the development of multithread-scalable plugins by providing a convenient event to allocate and initialise thread-private resources in *Pre Thread*. Furthermore, by delivering the *Post Thread* event to all active threads at the time the application exits, it removes the burden of tracking application threads in each plugin. This event allows each instrumented thread to collate its results in a global data structure before the execution of the global *Exit* event callback. The thread-private analysis and instrumentation model is further supported by API functions for managing thread-private data (the `mambo_set_thread_plugin_data()` and `mambo_get_thread_plugin_data()` functions), facilities for atomic operations on variables, a thread-private code cache which allows the direct encoding of pointers to thread-private data in the instrumentation, and compatibility with standard POSIX mutexes.

5 Code Analysis and Generation

The main tasks of a typical DBI tool are to *analyse* the code of the application to determine if and where to insert its instrumentation and then to *generate* the appropriate instrumentation code. Our API provides several abstraction layers and supporting infrastructure to facilitate portable and efficient code analysis and generation.

5.1 Code Analysis

Generating instrumentation often requires the analysis of the application code. While ISA-specific decoding can be used directly, some classes of instructions are 1) commonly analysed and 2) non-trivial to decode due to the number or complexity of the available encodings, and should be abstracted by the API. For example, memory operations meet these criteria, by 1) being analysed for a wide range of DBI use cases (e.g. hardware simulation and application analysis) and 2) being available in most ISAs under a variety of encodings to support different size of transfers and addressing modes.

Load and store analysis can be divided into three problems: identifying the instruction, determining the data size and determining the base address of the accessed memory. The result of the first two can be established at code scanning time. Our API introduced the functions `mambo_is_load()`, `mambo_is_store()` and `mambo_get_ld_st_size()` to perform this analysis in a portable manner. The address of most loads and stores can only be determined at runtime and must be instrumented to obtain their address for each execution. In the API, the function `mambo_calc_ld_st_addr()` is provided for this purpose. It sets the address in a register chosen by the user and it can instrument any type of load or store, on any of the supported instruction sets.

The API also introduced portable analysis functions for branch instructions: the function `mambo_get_branch_type()` returns a bitwise field, which contains either the flag `BRANCH_NONE` or a combination of one or more flags indicating various ISA-independent properties of a branch, such as: `BRANCH_DIRECT`, `BRANCH_INDIRECT`, `BRANCH_CONDITIONAL`, `BRANCH_CALL`, `BRANCH_RETURN` and ISA-specific flags such as `BRANCH_TBZ` (Test bit and Branch if Zero). For direct branches, the API returns the target address using the `mambo_get_branch_target()` function, while for indirect branches, the target address at each execution can be obtained via instrumentation inserted by `mambo_calc_branch_target()`.

Another portable analysis function returns the input and output register sets used by each instruction, via the function `mambo_get_inst_regs([...], inset, outset)`. This function may be used to implement data flow and register liveness analysis.

5.2 Register Management

The API introduced an ISA-independent register naming scheme: the registers `reg0` to `reg10` map to the first 11 general purpose on each supported architecture (i.e. r0 to r11 on AArch32 and x0 to x11 on AArch64). This allows developers to explicitly use these registers in code which can be compiled for either platform. This set was selected as the intersection of general purpose registers without a dedicated function between the two architectures. In particular, the set includes all the general purpose registers used to pass arguments and return values following either the 32 or 64-bit Application Binary Interface (ABI). Furthermore, aliases are also defined for the *Program Counter*, *Stack Pointer* and *Frame Pointer* registers (`pc`, `sp` and `fp` respectively). Additionally, the register alias `es` is defined as the first *calleE-Saved* register according to each ABI, to allow plugins to better handle function calls in a portable manner. The addition and subtraction operations are defined on the register type, which allows plugins to select registers relative to a provided alias, for example `es+1`. The API further defines the full set of native registers (i.e. r0-r15 on AArch32 and x0-x31 on AArch64) for ISA-specific code generation. The `regset` type is also defined to represent sets of registers using a bitwise mask (for example the set of registers pushed or popped by an instruction). For each defined register, its mask is also defined as `m_[register name]`, for example `m_reg0` for `reg0`.

The API also provides register allocation and deallocation routines (`mambo_get_scratch_regs()` and `mambo_free_scratch_regs()`) which should be used when the instrumentation does not depend on using specific registers. This offloads spilling and restoring register values or register liveness analysis from the plugin developer. This functionality also allows the DBI system to allocate dead registers to the instrumentation, which can be used efficiently, without spilling and restoring their original values. Note that these register allocations are ephemeral and can only be used in a given contiguous block of instrumentation code - the DBI system checks for matching allocated and freed registers at runtime.

5.3 Code Generation

Code generation helpers use the register management infrastructure to abstract some the common building blocks of DBI. Table 2 summarises the types of code generation helpers introduced by the API. For example, the *reserve_branch* and *local_branch* class of helpers are used to insert short-range branches inside a block of instrumentation.

5.4 Function Names and Symbols

DBI operates at the machine code level, independently from the high level implementation language or the runtime of

Table 2. Description of the code generation helpers.

Type	Description
set_reg	generate a specific value in a given register, which can be as large as its native size
push & pop	push and pop sets of registers according to the ABI of the platform
branch	insert various types of conditional and unconditional branches to a given address
reserve_branch	reserve space for a branch to be inserted at a later time
safe_fcall	generate a function call to a specific address, preserving the value of caller-saved registers - allows calls to functions following the platform ABI to be inserted anywhere in application code
fcall	generate a function call to a specific address - for calling functions which preserve the register state internally
local_branch	generate a branch from a previously reserved space to the current position in the code cache
counter_incr	update counters in memory; optionally atomic; 64-bit version available for 32-bit machines
proc	generate 3 register data processing instructions (common arithmetic and bitwise instructions)
proc_i	generate 2 register + immediate data processing instructions

the instrumented application. However, executable files and libraries often include information about the mapping of constructs from the high level language (e.g. function names and variables) onto the executable image. On GNU/Linux, virtually all executables and dynamic libraries are distributed as ELF files, which provide this information using *symbols*. Symbols map from a name in the high level source code (a string) to a type (`FUNC` for functions, `OBJECT` for data), an address and a size. For some types of instrumentation it is required or convenient to have access to this high level information (e.g. function names are used by the memory error checker described in Section 6). Therefore, the API introduced the functionality for handling this information. However, it is important to clarify that this information is not necessarily present in every file and should be considered untrusted user input. The symbol-related functionality of the API is provided in a best-effort manner and plugins should gracefully handle missing information if possible.

The main function provided to plugins is `get_symbol_info_by_addr()`, which returns the symbol data associated with an address, if available. It can be used to obtain the name of a function and the file from which it was loaded, based on the address of any instruction which is part of that function. The `get_backtrace()` API function can obtain the call backtrace in a program, based on a frame pointer.

The API also allows instrumenting the entry and exit points of specific functions based on their name, providing function hooking and replacement functionality. This was implemented by associating each pair of pre and post *Function* event handlers with a function name. For example, to register the pre and post *Function* callbacks for `printf()`, a plugin would call:

```
mambo_register_function_cb(ctx, "printf",
      &pre_printf_handler, &post_printf_handler);
```

The *Function* event handlers are allowed to insert instrumentation and may use any of the available code generation helpers, similarly to the *Instruction* and *Basic Block* event handlers. Instrumentation inserted in the pre and post *Function* events executes before entry to the associated function and after return from the associated function, respectively. Furthermore, the *pre Function* instrumentation may pass data to the *post Function* instrumentation by creating a new stack frame, although functions which receive arguments on the stack (an uncommon case on Arm) need special handling; automating this handling is a topic of future work.

6 Case Study: Memory Error Checking

Using the API introduced in this paper, we have implemented a memory error checker, which we call `M-memcheck`. It instruments applications to detect two common classes of memory errors: *out-of-bounds memory accesses* and *invalid frees*.

We are interested in detecting out-of-bounds accesses inside mapped memory pages, which cannot be detected by the operating system via page mapping exceptions. To detect these types of errors, `M-memcheck` tracks the valid memory allocations at any given time and is able to check efficiently whether each memory access of the application is fully contained in one or more such allocations. This is achieved using shadow memory [5, 7, 16].

6.1 Shadow Memory

Each byte of virtual memory (VM) available to the application is shadowed by a bit in memory which determines whether it is part of a valid allocation. This is achieved by reserving the top half of the VM range. On AArch32 machines running Linux, addresses above 3 GiB are reserved by the kernel, therefore the available VM size is reduced by only 1 GiB compared to uninstrumented execution, allowing most applications to execute correctly. On 64-bit architectures, the VM range is large enough that halving it is not expected to raise issues for most applications. We reserve half of the VM range as opposed to only 1/8 (the size required for the shadow memory itself) to simplify the runtime calculation of the shadow memory address, as shown in Listing 1, where `address` is the address of the checked memory access (dynamic value), `MSB` is the bitwise mask for the most significant bit of virtual addresses on the platform, `access_size` is the size of the memory operation in bytes (known at instrumentation time), `shadow_addr` is the shadow memory address to load (byte-addressed) and `shadow_mask` is the bitwise mask to apply to the value loaded from the shadow memory to select the relevant shadow bits before checking their validity.

```
shadow_addr = ((address & (~MSB)) >> 3) | MSB;
shadow_mask = ((1 << access_size) - 1) << (address & 7);
```

Listing 1. Calculating the shadow memory address corresponding to VM addresses

The VM is shaped by reserving its top half in the *Init* event of the instrumentation API. To facilitate the implementation of shadow memory, the API guarantees that the *Init* callback executes before either of the following operations: 1) the loading of the application and 2) allocating the stack used by the initial thread of the application, therefore ensuring that both the image of the executable and any of its allocations are constrained by the VM shaping done in the *Init* event handler.

Table 3 shows an example VM layout for a dynamically linked *Hello world* application running on AArch64 under `M-memcheck`. Entries 8, 10 and 11 are created by the OS at the time MAMBO is loaded. Then, `M-memcheck` maps entries 1, 6, 7, 9 and 12 in its *Init* handler, as part of its shadow memory implementation. Afterwards, MAMBO loads the application, dynamic linker and allocates a stack for the application (entries 5, 4, and 3, respectively), which are constrained by the VM shaping done by `M-memcheck`. At runtime, the standard library is also loaded (entry 2).

Table 3. Example VM layout for a dynamically linked `hello_world` executable under `M-memcheck` (AArch64). The mappings in bold belong to the shadow memory / VM shaping implementation.

ID	Mapping	VM range
1	**reserved (alias)**	`0x3fb7fff000 - 0x3fb8001000`
2	libc.so	`0x3ffdfaf000 - 0x3ffe0f5000`
3	[stack]	`0x3ffe10a000 - 0x3ffe122000`
4	ld.so	`0x3ffff7e000 - 0x3ffffac000`
5	hello_world	`0x3ffffac000 - 0x3ffffbe000`
6	**shadow memory**	`0x4000000000 - 0x4800000000`
7	**reserved**	`0x4800000000 - 0x7000000000`
8	MAMBO	`0x7000000000 - 0x70000c2000`
9	**reserved**	`0x70000c2000 - 0x7fb7fff000`
10	[vvar]	`0x7fb7fff000 - 0x7fb8000000`
11	[vdso]	`0x7fb8000000 - 0x7fb8001000`
12	**reserved**	`0x7fb8001000 - 0x8000000000`

6.2 Function Hooks

The shadow memory is updated whenever 1) a heap management function (e.g. malloc or free) executes successfully or 2) the application directly executes a memory allocation system call (e.g. mmap, munmap). However, the heap management functions use the same system calls internally to obtain or release memory from/to the OS. Hence, it is necessary to distinguish between executions of these system calls from heap management functions and directly by the application. In `M-memcheck`, this was implemented using the *Function* event to insert instrumentation hooks before and after the heap management functions, thus: before entry to such a function, a thread-private counter is incremented by one and on return from such a function, the same thread-private counter is decremented by one. This thread-private counter is initialised with value 0 in the *pre Thread* event handler. Therefore, a non-zero value of this counter indicates that execution in the associated thread is within a heap management function. We use this property to ignore the memory allocation systems calls executed by the heap management.

6.3 Instrumentation and Error Reporting

Each memory load or store is instrumented with inline code which loads the corresponding shadow memory bits and then calls an error reporting function if any are marked as invalid. This instrumentation is inserted using the *pre-Instruction* event, which allows us to check whether the current instruction is a load or store using the `mambo_is_load_or_store()` analysis helper. The instrumentation code which loads and checks the shadow bits is generated by the `check_shadow_mem()` function, based on the size of the transfer and using the data processing and branch code generation helpers. This instrumentation sets the value of reg1 to zero for a valid access (the common case) or a non-zero value otherwise.

When an invalid access is detected by the inline instrumentation, the C function `memcheck_error()` is called with the address, size, type (load/store) of access and the frame pointer as arguments. It prints this information in human-readable format to *stderr*.

7 Evaluation

7.1 Experimental Setup

The evaluation was done on two machines: an AppliedMicro *Merlin* system with an 8-core X-Gene2 SoC and an NVIDIA Jetson TX1 system with a 4-core (Cortex-A57) Tegra X1 SoC. The *Merlin* system has 32 GiB RAM and runs Debian 9 with Linux 4.9.0, while the *TX1* system has 4 GiB RAM and runs Ubuntu 16.04 LTS with Linux 3.10.

The workload consists of the PARSEC 3.0 benchmark suite [1] compiled with GCC 6.3.0 using the *native* input set and the *pthreads* configuration, except for the *freqmine* benchmark which does not have a pthreads implementation and was built using the *openmp* configuration. To analyse multi-threaded scaling, the number of threads is varied between one and the number of cores on each system. The benchmarks *canneal* and *raytrace* do not build on either AArch32 or AArch64. The benchmark *ferret* contains a bug which causes it to perform an out-of-bounds access, detected both by Valgrind Memcheck and `M-memcheck`, and crash nondeterministically. Therefore it is not included in this evaluation.

The complexity of the memory error checker and of the online cache simulator made it impractical to port and adapt our implementation to each DBM system. Therefore, we evaluated our implementation for MAMBO against the equivalent tools available for DynamoRIO and Valgrind, while trying to match the same functionality. However, we also ported a set of lightweight plugins across the three DBM systems, which will allow for a direct performance comparison.

7.2 Lightweight Plugins

To enable a direct performance comparison between DBM systems and their APIs, we developed a set of specifications for relatively simple plugins, which are straightforward to implement using the architecture-independent / portable functionality of various DBM APIs. The plugins insert low and medium sparsity instrumentation - on average one or fewer blocks of instrumentation per basic block - for common tasks in the area of DBI-based performance analysis.

We implemented them for our API (*MAMBO*), for MAMBO using only the pre-existing low level API (*MAMBO-lowlevel*) - to show the effect on productivity of our API, for DynamoRIO and Valgrind. The general architecture and as much of the codebase as possible is shared between our three implementations, however we use the recommended approach of doing various tasks under each API. For example, correct multithreaded execution of the dynamic counters we insert is achieved in various ways: under MAMBO, we maintain thread-private counters and add the results to a global value when threads terminate; under Valgrind, execution is serialised by the system, removing the burden of synchronisation from the developer; and under DynamoRIO we use the option for generating synchronised accesses provided by the dynamic counter API. The three plugins are: *bbstat*, *xorcount* and *brcount*. The plugin *bbstat* prints the number of basic blocks it has observed at code scanning time and the distribution of their size in bytes, when the application terminates. It consists only of an application exit handler and a lightweight code analysis phase; it does not insert any instrumentation. This plugin is intended to determine the base overhead of a DBM system when a plugin is enabled and receiving callbacks, without actually modifying the code of the application.

The plugin *xorcount* inserts a dynamic execution counter for each XOR instruction in the application and prints the total per-application value when the application exits. The aim of this plugin is to measure the overhead when inserting sparse instrumentation (an average of less than one block of instrumentation per basic block for most applications).

The plugin *brcount* inserts a dynamic execution counter for each branch instruction in the application, maintaining separate counters for direct branches, indirect branches, and returns. It prints the values of the three counters when the application exits. This plugin is intended to measure the overhead for medium density instrumentation (around 1 block of instrumentation for each basic block).

Table 4. Geometric mean slowdown relative to native execution on PARSEC 3.0 for the lightweight plugins.

		AArch32		AArch64		
	DBM system	Merlin	TX1	Merlin	TX1	LoC
bbstat	MAMBO	1.12	1.20	1.15	1.23	58
	MAMBO-lowlevel	1.12	1.20	1.15	1.23	108
	DynamoRIO	1.35	1.44	1.29	1.44	46
	Valgrind	3.93	4.95	5.19	6.49	55
xorcount	MAMBO	1.15	1.25	1.16	1.25	76
	MAMBO-lowlevel	1.15	1.25	1.16	1.25	219
	DynamoRIO	1.38	1.45	1.32	1.45	34
	Valgrind	4.09	5.07	5.28	6.60	68
brcount	MAMBO	1.65	1.82	1.60	1.82	75
	MAMBO-lowlevel	1.65	1.82	1.60	1.82	438
	DynamoRIO	1.95	2.17	1.74	2.00	73
	Valgrind	4.44	5.65	6.07	7.10	94

The counters used by the *xorcount* and *brcount* plugins should be 1) 64-bit wide, to prevent overflowing in longer running applications and 2) thread-safe. For MAMBO, we used the `emit_counter64_incr()` API function and thread-private counters. The Valgrind API is based on inserting calls to C functions, therefore we implemented our counters using the `uint64_t` data type. DynamoRIO provides the `drx_insert_counter_update()` function to insert dynamic execution counters in an efficient and portable manner. However, we have encountered the following limitations in the version under test: DynamoRIO silently inserted 32-bit counters in 32-bit applications, ignoring the `DRX_COUNTER_64BIT` flag; and the atomic update option `DRX_COUNTER_LOCK` is not supported on Arm (neither on AArch32, nor on AArch64). Therefore, these two DynamoRIO plugins fail to fully implement the intended behaviour at this time and their performance may degrade when the missing API features are implemented.

Table 4 shows the geometric mean slowdown introduced by the plugins running in each of the three DBM systems, compared to native execution, and the number of lines of code (excluding comments and empty lines) used to implement them. In terms of performance, MAMBO starts with a lower overhead compared to the other DBM systems and the API we have developed allows us to maintain this advantage as denser instrumentation is inserted.

The line count is significantly reduced by using our API compared to the pre-existing MAMBO API. The line count for the *MAMBO* implementation is generally similar to that of the equivalent DynamoRIO and Valgrind implementations, with two exceptions: the implementation of *xorcount* for DynamoRIO is significantly shorter than those for MAMBO and Valgrind, because DynamoRIO translates all types of native XOR instructions into a single XOR instruction in its IR, which can then be trivially identified using a single comparison. On the other hand, the IR of Valgrind contains multiple types of XOR instructions, which therefore require more elaborate rules to decode and identify. Similarly, the MAMBO API does not provide an architecture-independent abstraction for identifying XOR instructions and therefore we had to write ISA-specific decoding rules. The second exception is that the implementation of *brcount* for Valgrind required significantly more code to implement compared to the MAMBO and DynamoRIO implementations. This is caused by the Valgrind API only exposing the application code to the plugin after lifting it to an ISA-independent IR. Therefore, to analyse the branch instructions in the native code we had to reverse engineer the translation done by Valgrind and implement more complex analysis to map from the IR back to the native code. In addition to being more complex, this plugin is also dependent on a specific version of Valgrind - future versions might change the translation of branch instructions and break the way we map from the IR to the native instructions.

7.3 Memory Error Checking

We have also measured the performance of Valgrind Memcheck [18] (version 3.13.0) and Dr. Memory [2] (commit bcb36073a2c from July 2017). At the time this evaluation was done, Dr. Memory was not available for AArch64. On AArch32, Dr. Memory crashed when running the benchmarks *bodytrack*, *facesim*, *streamcluster*, *vips* and *x264*. For Valgrind Memcheck we used the arguments `–leak-check=no` `–undef-value-errors=no` `–track-origins=no` `–keep-stacktraces=none` `–show-mismatched-frees=no` and for the Dr. Memory we used the argument `-light` to disable detection for classes of errors not yet implemented in `M-memcheck`, such as use of undefined values. Note that we have not reviewed the code of these third party tools to determine whether disabling detection for these classes of errors disables all related instrumentation to obtain the maximum performance improvement compared to the default configuration.

The benchmark *dedup* attempts to memory map a large file (672 MiB), which fails on AArch32 under `M-memcheck` on our systems, therefore was not included in the evaluation. This is caused by the fragmentation introduced by the shadow memory implementation; enough virtual memory is available, however it is not available in a single contiguous region due to address space layout randomisation. As a topic of future work, we believe this problem can be avoided by allowing the DBI system to allocate its heap in the regions of memory above 2 GiB that are reserved but not otherwise used by `M-memcheck`.

Table 5. Geometric mean slowdown relative to native execution on PARSEC 3.0 for memory error checkers.

		AArch32		AArch64	
Threads	DBI framework	Merlin	TX1	Merlin	TX1
1	MAMBO	4.5	4.5	4.2	4.4
	Valgrind	9.6	11.3	10.5	12.7
2	MAMBO	4.5	4.5	4.4	4.4
	Valgrind	18.9	22.0	20.5	24.0
4	MAMBO	4.6	4.5	4.3	4.4
	Valgrind	41.8	40.1	43.2	41.3
8	MAMBO	4.9	N/A	5.1	N/A
	Valgrind	86.8	N/A	77.2	N/A

The geometric mean slowdown introduced by Valgrind Memcheck and `M-memcheck` is shown in Table 5. A comparable geometric mean overhead could not be calculated for Dr. Memory because it fails to run the full set of benchmarks. We note that instrumentation built using Valgrind cannot scale for multithreaded applications because the runtime system serialises all threads: the slowdown introduced by Valgrind Memcheck increases from 9.6 - 12.7 for one thread up to 77.2 - 86.8 for 8 threads. On the other hand, the slowdown introduced by `M-memcheck` does not change significantly (4.2 - 4.6) between 1 and 4 threads, because the instrumentation does not introduce a significant number of additional synchronisation points between threads. When the number of threads is increased to 8, the slowdown increases slightly (to 4.9 and 5.1 respectively), likely due to the additional pressure on shared on-chip resources such as the last level cache. Nevertheless, these results show good multithreaded scalability potential for this type of instrumentation.

`M-memcheck` also introduces a significantly lower slowdown for single threaded execution compared to Valgrind memcheck. This is achieved both by using the low level code generation layer of the API to insert efficient instrumentation and because the DBI system itself (with no instrumentation) has lower overhead.

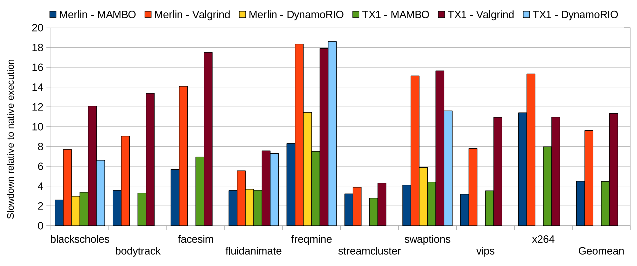

Figure 3. Slowdown relative to native execution for memory error checking - AArch32 (1 thread).

Figures 3 and 4 show the slowdown for each benchmark on AArch32 for 1 and 4 threads, respectively. Each set of results is labelled in the following format: [system] - [DBI framework]. In all cases, the MAMBO implementation is

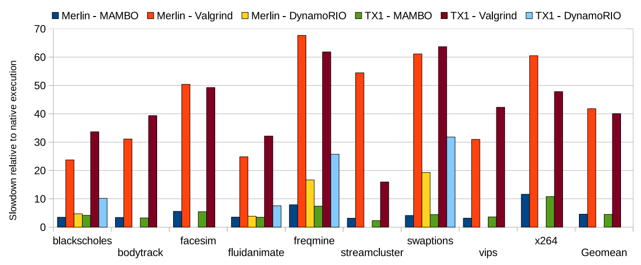

Figure 4. Slowdown relative to native execution for memory error checking - AArch32 (4 threads).

faster than the Valgrind and DynamoRIO implementations. The slowdown varies significantly between benchmarks, depending on the density of executed memory load and store instructions and the locality of accesses. The highest slowdown for `M-memcheck` is on the *x264* benchmark, because it continuously creates new threads which only execute for a few seconds. The thread-private instrumentation model requires instantiating new L1 cache models for each new thread and instrumenting each private copy of the code cache, however the performance of `M-memcheck` remains competitive.

7.4 Online Cache Simulation

We have implemented an online cache simulator (i.e. the workload and the simulation execute at the same time), named `M-cachesim`, using the plugin API proposed in this paper. It is configured to simulate a fairly typical Arm SoC cache hierarchy: 32 KiB 2-way associative L1 data and instructions caches with Least Recently Used (LRU) replacement policy and a globally shared 2 MiB 16-way associative L2 cache with random replacement policy. All cache lines are 64 bytes. For comparison, we have also measured the performance of Valgrind Cachegrind [15] (version 3.13.0) using similar settings.

Note that online cache simulators spend a significant part of their execution time running the cache model, as opposed to the instrumented application. On workloads with a high density of memory accesses, we have observed up to 56% of the execution time being taken by the cache model of `M-cachesim`. The performance of the cache model itself is a major contributor to the overall performance of the cache simulator and the results presented in this section reflect the aggregate performance of the DBI and cache model. `M-cachesim` uses a different cache model than Cachegrind, which we have designed to 1) allow more flexibility in configuring the simulated caches and the simulated cache topology, 2) accurately simulate accesses which span multiple cache lines and 3) more accurately simulate the cache subsystems implemented on existing physical Arm-based SoCs.

Figures 5 and 6 show the slowdown for each benchmark on AArch64 for 1 and 4 threads, respectively. The single thread performance is generally similar between the two

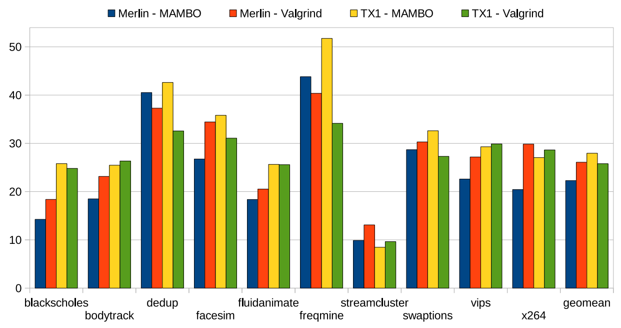

Figure 5. Slowdown relative to native execution for online cache simulation - AArch64 (1 thread).

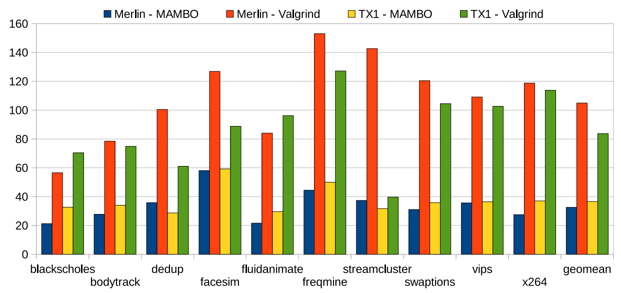

Figure 6. Slowdown relative to native execution for online cache simulation - AArch64 (4 threads).

implementations, however `M-cachesim` has unusually high overhead on the Merlin system for the *dedup* and *freqmine* benchmarks. A preliminary investigation indicates that this is caused by the cache model and not by the instrumentation, however further work is needed. On 4 threads, the improved scalability of `M-cachesim` over Cachegrind allows it to achieve better performance on all benchmarks.

8 Summary and Conclusions

This paper has presented an API for a DBI framework, which we have implemented on top of MAMBO [8]. This system was implemented for Arm v8, which uses three instruction sets (A32, T32 and A64). The API was designed to offer 1) portability of the DBI tool, and 2) improved flexibility, e.g. by allowing DBI tool developers to choose at a fine-grain level between a detailed level of code analysis and instrumentation, or a simplified and portable level. This approach allows DBI tool developers to improve their productivity by writing portable code most of the time, while still allowing low level analysis of the native code and native code instrumentation.

These features have been demonstrated using two examples DBI tools: a memory error checker (`M-memcheck`) and an online cache simulator (`M-cachesim`), which have been shown to have comparable or improved performance compared to state-of-the-art implementations in DynamoRIO and Valgrind. Both DBI tools have shown significantly improved multithreaded scaling compared to state-of-the-art tools. For example, from 1 to 8 threads, the geometric mean overhead of `M-memcheck` on AArch32 increased by only 8%, while the overhead of Valgrind Memcheck increased by 804% due to thread serialisation. Furthermore, the single thread performance of `M-memcheck`, which uses a direct mapping shadow memory, is also significantly improved; its geometric mean slowdown is between 2.1x and 2.9x lower than that of Valgrind Memcheck. The single thread performance of `M-cachesim` is similar to that of the fastest state-of-the-art cache simulator, despite using a more complex cache model.

Acknowledgments

This work is partially supported by an Arm/EPSRC iCASE PhD scholarship (Guillermo Callaghan), and EPSRC Rain Hub EP/R026084/1, and LAMBDA EP/N035127/1 projects. Mikel Luján is supported by an Arm/RAEng Research Chair award and is a Royal Society Wolfson Fellow.

References

[1] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[2] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization.* IEEE Computer Society, 213–223.

[3] Derek Lane Bruening. 2004. *Efficient, transparent, and comprehensive runtime code manipulation.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[4] Bryan Buck and Jeffrey K. Hollingsworth. 2000. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.* 14, 4 (Nov. 2000), 317–329. https://doi.org/10.1177/109434200001400404

[5] Michael Burrows, Stephen N Freund, and Janet L Wiener. 2003. Run-time type checking for binary programs. In *International Conference on Compiler Construction.* Springer, 90–105.

[6] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization (TACO)*, Article 5 (2014), 23 pages. https://doi.org/10.1145/2629677

[7] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. 2006. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Computers and Communications, 2006. ISCC'06. Proceedings. 11th IEEE Symposium on.* IEEE, 749–754.

[8] Cosmin Gorgovan, Amanieu d'Antras, and Mikel Luján. 2016. MAMBO: A Low-Overhead Dynamic Binary Modification Tool for ARM. *ACM Trans. Archit. Code Optim.* 13, 1, Article 14 (April 2016), 26 pages. https://doi.org/10.1145/2896451

[9] James H Hill and Dennis C Feiock. 2014. Pin++: an object-oriented framework for writing pintools. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 133–141.

[10] Jeffrey K Hollingsworth, Barton Paul Miller, and Jon Cargille. 1994. Dynamic program instrumentation for scalable performance tools. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the.* IEEE, 841–850.

[11] R. Nigel Horspool and Nenad Marovac. 1980. An approach to the problem of detranslation of computer programs. *Comput. J.* 23, 3 (1980), 223–229.

[12] M. Lui, K. Sangaiah, M. Hempstead, and B. Taskin. 2018. Towards Cross-Framework Workload Analysis via Flexible Event-Driven Interfaces. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).*

[13] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm Sigplan Notices*, Vol. 40. ACM, 190–200.

[14] 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. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on.* IEEE, 144–151.

[15] Nicholas Nethercote. 2004. *Dynamic Binary Analysis and Instrumentation.* Ph.D. Dissertation.

[16] Nicholas Nethercote and Julian Seward. 2007. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments.* ACM, 65–74.

[17] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, Vol. 42. ACM, 89–100.

[18] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision.. In *USENIX Annual Technical Conference, General Track.* 17–30.

[19] William R Williams, Xiaozhu Meng, Benjamin Welton, and Barton P Miller. 2015. Dyninst and MRNet: Foundational Infrastructure for Parallel Tools. (2015).