MadT: A Memory Access Detection Tool for Symbolic Memory Profiling

Marco Cesati*, Renato Mancuso†, Emiliano Betti‡, Marco Caccamo†
*University of Rome Tor Vergata, Italy, cesati@uniroma2.it
†University of Illinois at Urbana-Champaign, USA, {rmancus2,mcaccamo}@illinois.edu
‡Epigenesys s.r.l., Italy, betti@epigenesys.com

Abstract

Tools for memory access detection are widely used, playing an important role especially in real-time systems. For example, on multi-core platforms, the problem of co-scheduling CPU and memory resources with hard real-time constraints requires a deep understanding of the memory access patterns of the deployed taskset. While code execution flow can be analyzed by considering the control-flow graph and reasoning in terms of basic blocks, a similar approach cannot apply to data accesses.

In this paper, we propose MadT, a tool that uses a novel mechanism to perform memory access detection of general purpose applications. MadT does not perform binary instrumentation and always executes application code natively on the platform. Hence it can operate entirely in user-space without sand-boxing the task under analysis. Furthermore, MadT provides detailed symbolic information about the accessed memory structures, so it is able to translate the virtual addresses to their original symbolic variable names. Finally, it requires no modifications to application source code. The proposed methodology relies on existing OS-level capabilities. In this paper, we describe how MadT can be implemented on commercial hardware and we compare its performance with state-of-the-art software techniques for memory access detection.

I. Introduction

In real-time and, more generally, in cyber-physical systems (CPS), there is often the need for accurate workload characterization. In such systems, in fact, it is crucial to gather detailed insights about deployed applications from both a behavioral and a timing perspective. On one hand, understanding the execution flow of a task under analysis can be relatively easy when relying on control-flow information. On the other hand, however, accesses to data can follow non-linear patterns that are hard to analyze without extracting online execution traces.

The introduction of multi-core and many-core architectures results in increasingly less expensive computational resources. Consequently, in modern systems the memory resources represent the main performance bottleneck and the main source of execution time unpredictability. It follows that optimizations performed on task's data access patterns to all levels of the memory hierarchy can yield critical improvements on both average and worst-case performance. As typical example, consider a mixed-criticality system where safety-critical control tasks run in parallel with non-critical applications. If full knowledge about the memory access patterns for critical tasks is available, the system can be designed in a way to bound the amount of interference that critical tasks can suffer on the shared memory resources. In general, we envision that by making the extraction of memory access patterns practical on commodity platforms, memory-usage characteristic of tasks can be considered in order to perform memory and CPU co-scheduling on multi-core platforms. In fact, following this reasoning, several articles presented new solutions that aim to schedule accesses to main memory [1, 2, 3], or to optimize cache management [4, 5]. However, memory and CPU co-scheduling has received comparatively less attention with respect to classic multi-core CPU scheduling, also because extraction of memory traces from large applications and subsequent analysis is still unpractical.

In this article we propose MadT: a mechanism to perform memory access detection for a generic application that approaches from a novel perspective the problem of accurately characterizing applicative memory behavior. Specifically, we have designed and developed a methodology that is not based on binary or source code instrumentation, and that is able to trace memory accesses by executing instructions natively on the target CPU. As a result, the proposed technique only introduces tracing-related overhead upon performed memory accesses, while non-memory instructions execute with no overhead. In addition, MadT offers a number of benefits that are briefly summarized below.

First, the core functionality of MadT is neither binary instrumentation, nor emulation. Hence, it is easier to port on multiple architectures, as long as they feature broadly available minimal requirements as described in Section II. Second, it is able to detect all memory accesses, including those to dynamically allocated memory locations, providing also additional information about where the corresponding allocations were performed. Next, MadT exploits symbol resolution techniques to associate raw memory addresses to both source code objects (e.g., functions, variables) and memory regions (e.g., heap, bss). MadT does not require OS-level modifications or changes to the source code of the analyzed task to properly operate, while still executing entirely in user-space. Finally, it allows selective tracing: tracing can be selectively enabled and disabled to capture memory accesses only for a specific fragment of execution.[1] The performed evaluation suggests that MadT yields significant improvement in tracing computation-intensive workloads while achieving comparable runtime to binary instrumentation tools for memory-intensive benchmarks.

[1]This feature is the only one that requires source code instrumentation: just two lines to bound the portion of code one need to profile. On the other hand, tracing the whole program does not need any instrumentation.

The rest of the article is organized as follows. Section II provides an high-level description of the proposed technique. Next, Section III contains additional details about the implementation, while the results of extensive evaluation using a selection of real and synthetic benchmarks are reported in Section V. Finally, we review the related work in Section VI, and the article concludes in Section VII.

II. Methodology Overview

MadT is basically a shared library that is linked to the target program at compilation time. Whenever the target program is executed, the code in MadT's library sets up the execution context so that data memory accesses are detected and properly resolved. MadT is based on three basic components:

A) A *profiler* that detects memory accesses by natively executing the target program.
B) A *symbolic resolver* that finds the symbolic name associated with any numerical address used by the target program.
C) A *dynamic memory tracker* that intercepts the calls to the C library functions related to dynamic memory handling.

A. The profiler

The *profiler* is the component of MadT devoted to detect and record the data memory accesses performed by the target program. MadT does not emulate or instrument the machine code of the target program. Rather, MadT sets up the target process so that it executes natively and self-detects its memory accesses. This approach turns out to be particularly convenient for programs in which the number of machine instructions that load or store values in the memory cells is a small fraction of the total count of executed instructions.

The general idea is to set up the target process so that any access to data in a target's memory page shall result in a page fault. Consequently, the program receives a `SIGSEGV` signal, and the corresponding signal handler records the address of the machine instruction that tempted the access, the type of access (load or store), and the address of the memory cells being accessed. The mechanism for generating a page fault for any access consists of removing any access right from the memory pages containing the data of the target program.

Once the memory access has been recorded, the target program should resume and execute again the machine instruction that triggered the fault. This time the instruction must not cause a page fault, otherwise we would be in an infinite loop. Therefore, the handler of the `SIGSEGV` signal restores the proper access rights to the accessed memory pages. The problem is that any further access to the same pages performed by the following machine instructions would then go undetected. It is thus necessary to remove the access rights immediately after the single machine instruction that triggered the page fault has been successfully terminated. In order to achieve this goal, the page fault handler of the `SIGSEGV` signal modifies the hardware context of the target process saved on the stack so as to set the hardware "trap flag". When the trap flag is set, the CPU control unit generates a trap exception for each machine instruction that terminates. Right after the `SIGSEGV` handler terminated, the target's machine instruction that accessed the memory is executed again, this time without generating page faults. Then, the CPU control unit raises the trap exception. The OS kernel reacts to the trap exception by sending a `SIGTRAP` signal to the target process. The MadT's handler for the `SIGTRAP` signal removes the access rights from the pages, and modifies the hardware context of the target process saved on the stack so as to clear the "trap flag". When the target process resumes its normal execution, instruction trapping is disabled and all data pages are "protected" again.

MadT's profiler currently exploits a couple of features of the x86 architecture. However, we consider it to be quite portable. On one hand, any general-purpose CPU coupled with a Memory Management Unit (MMU) must support memory protections and page faults. On the other hand, the trapping instruction mechanism is present at least in any architecture based on a CISC ISA, because it makes much easier to implement debuggers and monitors. The trapping mechanism is also present in several RISC-based architectures: prominent examples are the various ARM architectures.[2]

Finally, observe that because the MadT profiler operates at the machine instruction level, it is not difficult to extend it so as to support target programs written in high-level languages different than C.

B. The symbolic resolver

One distinctive feature of MadT is that it can output the memory accesses in a symbolic form. Usually, each time a program is executed, the virtual memory addresses used by the process change. Basically there are two reasons for this behavior:

1) In each run, the kernel loads the segments of the shared libraries used by the process at different addresses. This mechanism is called "address space layout randomization" and is essentially aimed at making more difficult to exploit security flaws in the program. Although address space layout randomization can usually be disabled by the user, the effective load addresses of the shared libraries may still change between different kernel versions, dynamic loader versions, or shared libraries versions.
2) The virtual addresses returned by dynamic memory allocator procedures cannot be easily predicted in advance and may change at each run. The C standards explicitly mention that this behavior is allowed and expected.

[2]In a RISC CPU all machine instructions have opcodes of fixed length. Moreover, very few instructions access memory. Thus, even if the trapping mechanism it not available at the hardware level, it is not too hard to devise a mechanism that execute the very single instruction that caused the page fault, and then resumes normal execution.

Thus, a memory access detection tool that returns only a list of numerical addresses is not very practical, because the user faces the difficult task to relate those addresses with the corresponding function and variable names of the target program and of the shared libraries.

Instead, in MadT each recorded memory access yields one line in the output file that includes the symbolic names of the function, variable, or dynamically allocated block of the target program. Actually, MadT forges a new symbolic name for each dynamic memory block allocation (see below). In the other cases, MadT associates any virtual address with the corresponding symbol as known at compilation time or at dynamic loading time.

To fully enable the address resolution mechanism, the target program must be recompiled in such a way to include debugging information in the executable file. It is thus possible to extract from the executable file some tables that relate virtual addresses with symbolic names (even for the symbols that are "private" to the target program). These tables are read by MadT when the target program is launched. Moreover, MadT translates virtual addresses related to shared libraries functions and variables by invoking suitable helper functions of the dynamic loader. The dynamic loader is included in the memory space of any process being executed and deals with shared libraries loading and shared symbols resolving.

MadT's symbolic resolver is quite portable across different architectures and even across different operating systems. It relies on the GCC toolchain [6] and on the dynamic loader coupled with the "glibc" GNU C library [7], thus any architecture to which these programs have been ported to is basically supported by MadT's symbolic resolver.

Extending the symbolic resolver to support target programs written in high-level languages different than C is also feasible, as long as two main requirements are satisfied: a) it must be possible to instruct the compiler to insert debugging information in the target executable; and b) the target program must use the same system dynamic loader used by C programs. Typically, programs written in high-level languages supported by the GCC toolchain satisfy these requirements.

C. The dynamic memory tracker

Virtual addresses returned by dynamic memory allocators have no symbols associated with them, neither at compile time nor at run time. To overcome this limitation, MadT forges a new symbol for any dynamic memory allocation.

The name of any forged symbol has a structure that helps the user to recognize the kind of allocation and the exact position of the call to the allocator procedure in the target's source code. For instance, if the 14th dynamic memory allocation of the target program has been triggered by a call to the `malloc()` C library procedure, and the call is included in some function `funcI()` of the target program, then the forged symbol name would be something like "`<malloc0014@funcI+230>`". Here, "230" represents the offset in bytes of the call to `malloc()` inside `funcI()`. Observe that the role of the counter is to distinguish between different allocations performed by the same allocator call; for example, consider a loop iterated several times and containing an invocation to `malloc()`.

In order to forge new "dynamic" symbols to be associated with dynamically allocated memory blocks, MadT has to detect when the target process invokes a C library procedure that allocates memory for the process itself. This is simply done by defining a set of "wrapper" functions in MadT that are invoked in place of the "original" C library procedures. Currently MadT install wrappers for the C library functions `malloc()`, `calloc()`, `realloc()`, `free()`, `mmap()`, `mremap()`, and `munmap()`.

Any wrapper records internally the dynamic memory event so that "dynamic" symbols can be properly forged. The dynamic event is also written in the output file produced by MadT. The wrapper function also takes care of invoking the "original" C library procedure.

The wrapper mechanism used by MadT is rather portable because it is based on features provided in all modern operating systems. Basically, whenever a process invokes a given procedure defined in a shared library, the dynamic loader looks for a matching procedure name starting from the first loaded library. The trick, therefore, is to load MadT's shared library before any other shared library used by the target program. The dynamic memory tracker of MadT can thus be easily ported to any platform that allows the user to override the loading order of the shared libraries. Currently, MadT supports Linux systems, where the loading order of the shared libraries can be controlled, for example, by defining some environment variables.

The dynamic memory tracker can also be extended to support programs written in high-level languages different than C. For example, in order to support a C++ target program, new wrappers for C++ library procedures that handle dynamic memory, like the `new` operator, should be included in MadT's shared library.

D. Limitations

It is important to note that currently MadT does not trace memory accesses caused by instruction fetches. Although the same mechanism used to trace data accesses can also be applied to trace instruction fetches, the performance impairment would likely be very high. Moreover, from a practical standpoint, individually tracing instruction fetches is not necessary due to the inherently sequential nature of code execution flow. Conversely, an analysis based on basic blocks is sufficient and many tools have been described in literature to perform control-graph extraction either at runtime or offline, as we discuss in Section VI.

Currently, MadT does not trace single memory accesses to the stack. This is mainly because, similarly to instructions fetches, stack usage can be easily analyzed without performing tracing at runtime. In fact, stack boundaries are known at compile-time and can also be extracted by parsing the executable file. Nonetheless, MadT already detects memory block operations (e.g., `memcpy`, `memset`) that access the stack, and we plan to extend MadT to include stack single accesses detection in a future release of the tool.

Finally, MadT does not provide a component to analyze the collected trace. The main reason is that MadT is meant only to provide a method to detect memory accesses and to map them to the original variables defined by the programmer. Moreover, the output file produced by MadT can be easily converted in the format of other tools like *Valgrind* [8] and *OProfile* [9]. Therefore, any graphical analyzer suitable for the mentioned tools can also be used for MadT.

III. Implementation Details

The implementation of MadT relies on many different techniques, which will be discussed in this section.

A. Compilation of the target program

As a general rule, a target program that has to be profiled by MadT must be recompiled.[3] MadT depends on the GCC toolchain [6], because it makes use of a few peculiar extensions provided by the GCC compiler and loader.

No change to the source code of the target program is strictly required: MadT may start profiling right before executing the `main()` function, and it may stop right before terminating the process. However, the user may define a smaller portion of the target program execution to be profiled. Specifically, the `MadT__start()` and `MadT__stop()` functions can be added to the source code to, respectively, start and stop the profiler.

When compiling and linking the target executable, the user must specify some command line flags that instruct `gcc` to include debugging information, to link with the `libMadT.so` shared library, and to override the `main()` function of the target program with a specific MadT's function (this is used when initializing the library, see below).

Together with the target executable file, the user must also generate two files that include the contents of the relocation section and of the symbol table of the target program. This can be easily done by using the `readelf` utility program (included in the GNU Binutils [10] toolset).

B. Launching the target program

MadT collects accesses by monitoring a live execution of the target program. The target must be run on a Linux-based system, because MadT extracts some information about the target process from the Linux-specific `/proc/self/maps` virtual file.

In order to start profiling the target program, the user typically sets a few environment variables that control MadT's behavior. Next, the user launches the target program by passing the same command line arguments as in a normal execution. The memory accesses detected by MadT are written on a dedicated output file. By using some environment variables the user may force the profiler to start recording memory accesses right before executing the target's `main()` function. (It is always possible to start and stop the profiler by inserting calls to `MadT__start()` and `MadT__stop()` in the target source code.) The user may also decide the format of the addresses recorded in the output file: numerical virtual addresses, symbolic addresses, or both.

C. Library initialization

The MadT profiler is linked to the target program as a shared library. This means that at launch time the dynamic loader maps code and data of MadT in the memory space of the target process and ensures that specific library initialization functions are executed.

As explained in Section II, MadT defines wrappers for some of the procedures defined in the C library and possibly in other shared libraries used by the target program. Therefore, MadT must initialize some of its data structures as early as possible: after the other shared libraries have been loaded but before the initialization code of other shared libraries run. This can be achieved by (1) imposing the dynamic loader to load MadT's shared library before the other shared libraries, and (2) using a particular extension of the GCC C compiler that allows the programmer to define a given library function as a "constructor" procedure to be run before `main()`.

In particular, when MadT is being initialized, the code checks whether `libMadT.so` has been effectively loaded first. If not, MadT sets some environment variables that modifies the libraries loading order, and restart the whole target program. This is done in a wrapper function invoked in place of target's `main()`, because the target program has to be restarted by specifying the same command line arguments as the original execution, and these strings are made easily available only to `main()`.

The initialization code of MadT also reads from the files generated at compile time the dynamic relocation table and the symbol table of the target executable. It also reads from the `/proc/self/maps` virtual file, present in any Linux system, the list of virtual memory regions currently defined for the target process. Finally, the initialization code installs signal handlers for the `SIGSEGV` and `SIGTRAP` signals, and invokes the `main()` function of the target program.

[3] It is possibile to use MadT on programs that have not been specifically recompiled; in that case, however, the tool is much less flexible and accurate.

D. Collecting memory accesses

As explained in Section II, MadT forces a page fault exception whenever a machine instruction performs an operation that produces a memory access. In order to achieve this, MadT uses the `mprotect()` POSIX system call to remove all access rights to the set of pages containing the memory cells whose accesses are being profiled. In particular, when the profiler is being activated, MadT scans the list of all virtual memory regions of the target process, and it removes the access rights from the pages in the regions corresponding to:

- the data segment of the target program or shared libraries
- the bss segment of the target program or shared libraries
- the heap of the target process

Access rights are never removed for pages of the MadT library and of the dynamic loader, and for pages containing executable code and stack. Thus, accesses to those pages are not recorded.

When a page fault exception occurs, the Linux kernel determines whether the faulty address is included in a memory region whose access rights forbid the tempted access. As usual, in this case the kernel sends the `SIGSEGV` signal to the program. Since MadT has installed a signal handler for `SIGSEGV`, the kernel saves the hardware context of the target program in the User Mode stack and forces the invocation of the signal handler. The handler analyzes the hardware context saved on the stack and determines the faulty address that caused the fault, the address of the instruction that tried the memory access, and the type of access (load or store). The handler can thus record these data in a log buffer. It then invokes the `mprotect()` system call to restore the proper access rights to the pages including the faulty address. This is done because MadT must ensure that the machine instruction that is accessing memory is executed once again without generating a page fault. Finally, the page fault handler modifies the hardware context saved on the stack so as to activate the hardware tracing mechanism when normal execution resumes.

When the `SIGSEGV` handler terminates, the machine instruction that caused the page fault is executed again. This time the pages including the memory cells accessed by the instruction should have proper access rights, so that likely the memory access can be successfully completed. Right after the execution of the machine instruction the CPU's control unit raises a "trap" exception. The kernel thus raises a `SIGTRAP` signal and activates the corresponding handler defined in the `libMadT.so` library.

The trap handler removes any access rights to the pages of the virtual memory region including the address involved in the previous page fault event, Then, it modifies the hardware context of the target program saved on the stack so as to disable the hardware tracing mechanism. Finally, it checks whether the log buffer containing the events recorded by the profiler is nearly full; if so, it calls a procedures that empties the buffer by writing the events on the output file (see below).

When eventually the trap handler terminates, the normal execution flow of the process is resumed. Tracing is disabled, but pages to be profiled have no access rights, so that the profiler shall catch the next memory access.

E. Translating virtual addresses to symbol names

When MadT initializes, it reads from some files generated at compile time the dynamic relocation table and the symbol table of the target executable. The symbol names found in these tables are inserted in a red-black balanced binary tree indexed by the starting address of the symbol. MadT also reads from the `/proc/self/maps` virtual file the list of virtual memory regions currently defined for the target process, and insert each memory region in another red-black balanced binary tree indexed by the starting address of the region. Each node of the red-black tree is augmented with the size of the corresponding memory region, thus the binary tree allows MadT to quickly discover the memory region containing a given address.

The log buffer containing the events recorded by the profiler has limited size, thus it must be periodically flushed by writing its contents into the output file. Each record in the log buffer includes the virtual address of the machine instruction that triggered the fault, as well as the address of the memory cell being accessed. The record also includes a reference to the virtual memory region containing the page being accessed, and the type of access (i.e., load or store).

When flushing the log buffer, MadT translates the virtual addresses into meaningful symbol names and offsets relative to the symbols. In order to translate a raw virtual address into meaningful symbol name and offset, MadT looks up the raw address in the red-black balanced binary tree including all canonical symbols of the target program. If a match is found, it outputs the symbol name and the difference between the raw address and the "start" value associated with the symbol. If no match is found in the symbol tree, MadT invokes the `dladdr()` function of the dynamic loader to resolve the address at run-time.[4] If `dladdr()` returns a valid symbol, it is inserted in the symbol red-black tree. Otherwise, if MadT does not succeed in finding a symbol associated with the raw address, it outputs the name of the memory region including the raw address, as well as the relative offset with respect to the start of the region.

F. Dynamic memory handling

The mechanism just described for translating virtual addresses into symbol names does not work when the addresses belong to memory blocks that have been dynamically allocated by means of C library procedures like `malloc()` or `mmap()`. Therefore, MadT tracks dynamic memory events so as to write meaningful information in the output file when these memory blocks are accessed.

[4]The `dladdr()` function is a non-POSIX extension of the "Glibc" library.

The key idea is to establish proper wrappers for any C library function that handles dynamic memory. Currently MadT knows about the following functions: `malloc()`, `calloc()`, `realloc()`, `free()`, `mmap()`, `mremap()`, and `munmap()`. It is easy to extend MadT and add wrappers for other library functions, if required.

Once started, any wrapper routine typically checks if the profiler is active; if so, the wrapper disables profiling by restoring the access rights of the target memory pages. Then, the wrapper invokes the original procedure. Next, it records in the log buffer some data about the dynamic memory event: the address returned by the library function and the block size, in case of memory allocation or re-allocation; the address of the block being freed, in case of release. Finally, the wrapper removes the access rights to the target pages if the profiler was active when it started.

For performance reasons, symbols related to dynamic memory events are not forged when the events occur, but later when the corresponding entries stored in the internal buffer are written to the output file. Some dynamic memory events also modify the list of virtual memory regions of the process; in this case, MadT also updates the internal data structure that keeps track of those regions.

G. Interpreting profiler's output

Each line in the output file produced by MadT starts with a character that encodes the type of the recorded event. The type encoding is summarized in Table I. The output file may include "raw" lines and "symbolic" lines. The second character in any line is "`#`" for "raw" lines, or "`$`" for "symbolic" lines.

TABLE I. TYPES OF MEMORY EVENTS IN THE OUTPUT FILE

`L`	memory access of type "load"
`S`	memory access of type "store"
`Y`	block memory "copy" (`memcpy()`-like operation)
`W`	block memory "store" (`memset()`-like operation)
`G`	block memory "fetch" (`write()`-like operation)
`M`	dynamic memory allocation — `malloc()`
`C`	dynamic memory allocation — `calloc()`
`P`	dynamic memory allocation — `mmap()`
`R`	dynamic memory reallocation — `realloc()`
`E`	dynamic memory reallocation — `mremap()`
`F`	dynamic memory release — `free()`
`U`	dynamic memory release — `munmap()`

"Raw" lines show the numerical virtual addresses collected by the profiler. These lines may also include a string identifying the virtual memory region that includes the memory cells being accessed.

"Symbolic" lines show the addresses with the format "*`symbol_name`*`+`*`offset`*": "`symbol_name`" is the symbolic name associated with the address, and "`offset`" is the difference between the address and the "start" value associated with the symbol. These lines may also include sizes (memory block lengths), and strings that identify the virtual memory region including the memory cells being accessed.

Symbol names forged by MadT to track dynamic memory blocks can be easily recognized because they always start with the character "<": as this character cannot be used in valid C symbol names, there can be no confusion. The format of these symbol names is shown in Table II. As already explained, MadT keeps a counter for the number of dynamic memory allocations. The current counter value is included inside the symbol name, as it helps in recognizing different memory allocations triggered by the same source code instruction.

TABLE II. SYMBOL NAME FORMAT FOR DYNAMIC MEMORY EVENTS

`<malloc`*`number`*`@`*`symbol`*`+`*`offset`*`>`	`malloc()`
`<calloc`*`number`*`@`*`symbol`*`+`*`offset`*`>`	`calloc()`
`<realloc`*`number`*`@`*`symbol`*`+`*`offset`*`>`	`realloc()`
`<freed:`*`number`*`@`*`symbol`*`+`*`offset`*`>`	`free()`
`<memmap`*`number`*`@`*`symbol`*`+`*`offset`*`>`	`mmap()`
`<mremap`*`number`*`@`*`symbol`*`+`*`offset`*`>`	`mremap()`
`<unmap:`*`number`*`@`*`symbol`*`+`*`offset`*`>`	`munmap()`

Moreover, the allocation counter helps in recognizing accesses to memory blocks that have already been released. When MadT detects an invocation to `free()` or `munmap()`, it updates the name of the corresponding symbol by overwriting the first seven characters with the strings "`<freed:`" or "`<unmap:`". The rest of the symbol name, however, is left unchanged, thus it is still possible to identify the dynamic memory block. Accesses containing symbol names starting with "`<freed:`" or "`<unmap:`" should never appear in the output file, unless the target program is accessing a dynamic memory block that has been previously released. Usually, this is a target program bug.

IV. MADT USE CASE

In order to provide a description of the capabilities of our memory access detection tool, we analyze in detail a trace fragment obtained by the execution of a benchmark program.

When MadT is used to observe the memory access pattern of a given target program, the produced output is a collection of trace messages corresponding to a set of specified events following the encoding in Table I. As can be noted, the produced output trace features a very high event granularity, going down to the single memory access (loads and stores). We first show how the details about the collected events can be aggregated to produce a high-level view of the memory access pattern of the target program; next, we demonstrate how a one-to-one correspondence can be established between collected trace events and original source code.

For the analysis carried out in this section, we consider the "disparity" benchmark from the San Diego Vision Benchmark Suite [11]. This benchmark computes the 3-dimensional depth of a scenery provided in input as a pair of images taken from slightly different positions (stereoscopic view). Table III contains a summary of the number of events observed during the execution of the considered benchmark. The benchmark is executed with in input a stereoscopic image in Common Intermediate Format (CIF), i.e., having a resolution of 352x288.

TABLE III. SUMMARY OF MEMORY EVENTS FOR DISPARITY BENCHMARK

Event	Occurrences
Number of process memory region	30
Number of performed stores	2540679
Number of performed loads	5491371
Number of performed block-copy	0
Number of performed block-store	608288
Number of performed block-fetch	4

One way to read the memory trace produced by MadT is to use a visual approach. Figure 1 has been generated directly by post-processing the obtained memory trace for the disparity benchmark. This figure plots the virtual memory space of the analyzed process. Lower memory regions correspond to the top of the figure, while the bottom of the figure represents higher memory addresses. Gray horizontal lines denote the end of a given memory region (text, heap, ...) and the beginning of the next. Memory pages are ordered by starting address and each one is represented with a circle inside the corresponding memory region. Memory pages inside regions are ordered from left to right and from top to bottom. We use a color intensity code to show the access frequency of each page in the observed memory region (the darker, the more accesses). Gray and white pages represent respectively, memory pages that were not traced, and for which no access was performed. Subsequent non-traced regions or regions where no accesses were observed have been aggregated.

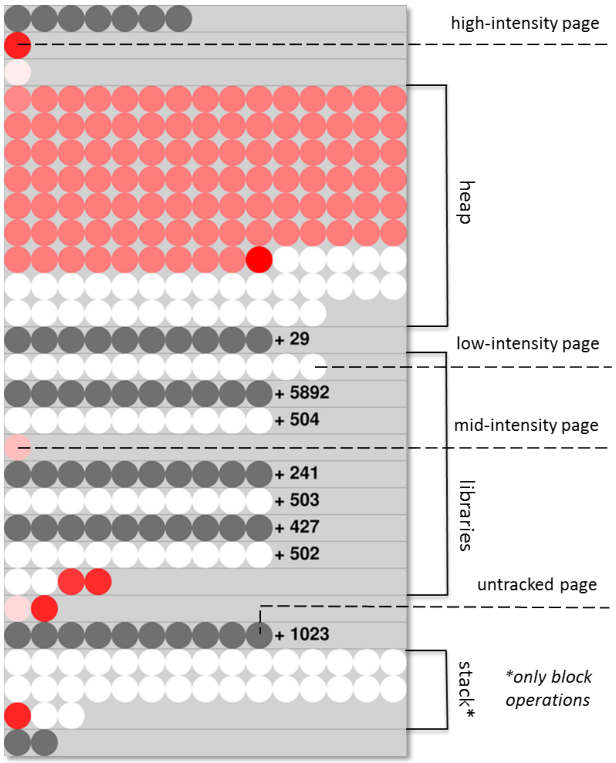

Fig. 1. Graphic view of disparity benchmark memory profile

From the arrangement of accesses in the profile, some features of the benchmark are evident. First, the benchmark makes a heavy use of the heap section. Specifically, 99 pages marked in light red experience an uniform access frequency. These pages correspond to the buffer containing the stereoscopic image in input. The number of accesses recorded on the pages of the input buffer is on the order of 10^4. One page of the heap, marked in darker red right after the input buffer, instead, experiences a number of accesses that is three orders of magnitude higher. As confirmed by source code analysis, this page stores the progressively

Listing 1. Body of `findDisparity()` function

```
#define subsref(a,i,j) a->data[(i) * a->width + (j)]

void findDisparity(F2D* retSAD, F2D* minSAD,
   I2D* retDisp, int level, int nr, int nc) {
  int i, j, a, b;
  for(i=0; i<nr; i++) {
    for(j=0; j<nc; j++) {
      a = subsref(retSAD,i,j);
      b = subsref(minSAD,i,j);
      if (a<b) {
        subsref(minSAD,i,j) = a;
        subsref(retDisp,i,j) = level;
      }
    }
  }
}
```

Listing 2. Fragment of symbolic `findDisparity` memory trace

```
/* Dynamic allocation of minSAD */
M$7:<m06@fMallocHandle+45>,528
/* Dynamic allocation of retDisp */
M$8:<m07@iMallocHandle+45>,528
/* Dynamic allocation of retSAD */
M$14:<m13@fMallocHandle+45>,528
            ...
/* Loop 0: i = 0, j = 0 */
L$0:<m13@fMallocHandle+45>+0,[heap],findDisparity+55
L$1:<m13@fMallocHandle+45>+8,[heap],findDisparity+73
L$2:<m06@fMallocHandle+45>+0,[heap],findDisparity+90
L$3:<m06@fMallocHandle+45>+8,[heap],findDisparity+108
S$4:<m06@fMallocHandle+45>+8,[heap],findDisparity+127
L$5:<m07@fMallocHandle+45>+0,[heap],findDisparity+134
S$6:<m07@fMallocHandle+45>+8,[heap],findDisparity+147
            ...
/* Loop 0: i = 0, j = 1 */
L$8:<m13@fMallocHandle+45>+0,[heap],findDisparity+55
L$9:<m13@fMallocHandle+45>+12,[heap],findDisparity+73
L$10:<m06@fMallocHandle+45>+0,[heap],findDisparity+90
            ...
/* Loop 0: i = 9, j = 12 */
L$1032:<m13@fMallocHandle+45>+0,[heap],findDisparity+55
L$1033:<m13@fMallocHandle+45>+524,[heap],findDisparity+73
L$1034:<m06@fMallocHandle+45>+0,[heap],findDisparity+90
            ...
```

built disparity map. Hence the large number of accesses. The frequently accessed pages at the top of the memory addressing space represent the data (bss, rodata) region. Moreover, the frequently accessed pages depicted in the second half of the figure belong to dynamic linker (`libdl.so`), C library, and stack. However, only stack pages accessed through block operations are highlighted in the figure, while no text pages are highlighted. This is because, as discussed earlier, they are not being traced by MadT in the current setup. In this example we show aggregate data, however it is possible to do the same analysis filtering a subset of the collected records, for instance selecting just read or write accesses.

A. MadT output analysis

Another way to exploit the output produced by the MadT tool is to relate captured accesses (or fragments of them) to portions of source code. This allows the user to extract accurate information about the memory access pattern of a given code fragment in order, for example, to deduce data locality and memory-intensity. Considering the previously discussed disparity benchmark, we have isolated a single function used to find the disparity between two raster images by shifting them over each other. The function is called `findDisparity()`, and its C code is reported in Listing 1.

As can be observed, the function performs a nested loop over `nr` and `nc`, which represent the number of rows and columns of the image. For simplicity, we now analyze an execution of the function when a low resolution image is used in input to the benchmark. Specifically, we use a 10×13 image, so that `findDisparity` will be called with `nr`=10 and `nc`=13. In the considered invocation of the function, the parameter `level` is set to 0.

We now show that it is possible to relate a fragment of captured memory accesses with the execution flow of the benchmark. Three of the 130 iterations of nested loop are reported in Listing 2; specifically, they are the first, the second, and the last iteration. In order to follow the example, moreover, we report at the top of Listing 2 the dynamic memory allocations for the three structures used in the function (`retSAD`, `minSAD` and `retDisp`), as captured by MadT.[5] For all the involved structures, the field `data` has a offset of 8 bytes. Finally, for all the considered iterations, the condition `(a<b)` is always satisfied.

[5]The names <mallocnumber@symbol+offset> have been shorted in <mnumber@symbol+offset> in order to save space.

It can be noted that the pieces of information that appear in the entries of the trace contain: (1) the location where `malloc()` was performed (e.g., `fMallocHandle()` for allocation of floats); (2) in which memory region the access falls, e.g., `[heap]`; (3) the offset in the function of the instruction performing the access, in this case `findDisparity+`*`off`*.

The first memory operation in the function expands in an operation of the form: `a = retSAD->data[0]`. Hence two loads (`L`) are generated at offset 0 and 8 from `m13`, for `retSAD` and `retSAD->data` respectively. The next two loads are produced in an analogous way for `minSAD` and `minSAD->data`. Since the condition `(a<b)` holds, the symbol `minSAD` is accessed again in order to store the value of `a` at the memory location of `minSAD->data`. Analogously, with the last statement of the loop, the value of `level` is stored in `retDisp->data`, originating another pair of load and store operations.

The second iteration of the nested loop is identical to the first one, except for the value of `j` that has been incremented to 1. Thus, only the final addresses for the expressions of the type `subsref(buffer, i, j)` will change. Given the value of `j`=1, `subsref(buffer, i, j)` will expand in `buffer->data[1]`. Each element in the `data` arrays occupies 4 bytes. Thereby, accesses to `retSAD`, `minSAD` and `retDisp` will produce operations on data at offsets 12 from the respective structure.

In the final iteration of the inner loop, the values of `i` and `j` are 9 and 12 respectively. This means that the first operation will be of the form `retSAD->data[9 * 13 + 12]`, i.e., `retSAD->data[129]`. Following the pointer arithmetic, this will result in an access to `retSAD->data + 516` = `retSAD + 524`. A similar reasoning allows deriving the rest of the accesses in the iteration.

V. Evaluation

In this section, we compare the performance of MadT with respect to existing tools that are able to perform complete memory tracing of applications. Specifically, we focus our comparison on two well established memory analysis tools, namely the Intel Pin tool [12] and the Valgrind Lackey tool [13].

Pin is a proprietary software component developed by Intel, which is free for non-commercial use, and implements a dynamic binary instrumentation framework for the IA-32 and x86-64 architectures. It provides a set of APIs that abstract the underlying ISA, supporting the creation of tools for dynamic program analysis. Pin performs its instrumentation at run-time on the compiled binary files without requiring recompilation. Its instrumentation framework also provide basic access to symbols and debug information.

In the context of this work, we focus on one of the sub-tools developed using the Pin APIs: the Pinatrace tool. This tool performs instrumentation only of instructions that read or write memory, producing in output the complete trace. Each entry in the trace reports three pieces of information: (1) address of memory instruction; (2) type of operation (read or write); (3) address of accessed memory location.

Similarly, Valgrind is a open-source binary instrumentation framework on top of which several dynamic analysis tools have been developed for code graph analysis and heap/stack profiling. The Valgrind framework is being actively developed and, unlike the Pin tool, features support for a large number of architectures, including IA-32, x86-64, PowerPC, ARM, and MIPS. One of the tools included in the Valgrind suite, called Lackey, performs a complete tracing of memory accesses and instruction fetches when executed with the option `--trace-mem=yes`. The style of the output is quite minimal, since per each memory access only two pieces of information are recorded: (1) access type (read, write or instruction fetch) and (2) address of the accessed memory location. However, each record corresponding to a data memory access appears in the trace immediately after the memory access corresponding to the fetch of the instruction that performed the data access.

We have executed the experiments on a set of real benchmarks that include applications from the MiBench suite [14] and the San Diego Vision benchmark suite [11]. The selected applications include object tracking, texture creation, jpeg compression/decompression, heavy mathematical processing, and sorting. We also tested the behavior of the considered tracing tools on simple synthetic benchmarks that stress single library functions that access memory in blocks and/or modify the memory layout of the application: `calloc`, `realloc`, `read`, `fwrite` and `mmap`. In addition, `mandelbrot` represents a computation intensive benchmark. The experiments have been performed on a server-grade machine featuring a Intel Xeon E5-2640 CPU operating at a frequency of 2.5 GHz, with 15 MiB of last-level cache and 16 GiB of DRAM. On this platform, the tools have been tested using a Linux 3.2 kernel and the GCC 4.7.2 compiler.

Figure 2 shows the comparison among run-times for a selection of benchmarks from the considered suites. The execution times have been normalized with respect to the MadT runtimes. Each bar shows the percentage of slow down (positive range) or runtime reduction (negative range) of Pinatrace and Lackey compared to MadT. As can be observed, in 5 benchmarks (`stringsearch`, `qsort`, `math`, `disparity`, and `localization`) MadT achieves significant performance benefits over both Pinatrace and Valgrind. In these examples, in fact, the slowdown introduced by Pinatrace and Lackey goes from 90% up to almost 800% compared to MadT. In general, this trend is visible with benchmarks that are mostly computation-intensive, while MadT does not performe so well with memory intensive applications (i.e., `texture_synth`). Intuitively, this is because on one hand the handling time for the sequence of exceptions (`SIGSEGV` + `SIGTRAP`) is higher than instrumenting the single memory instruction, while on the other hand instructions that do not access memory are always executed natively with zero overhead on the CPU.

Due to space constraints, the summary for the results obtained on the complete set of benchmarks is reported in Table IV. As can be seen from the table, visible benefits are obtained on synthetic benchmarks that heavily use C library functions to operate on blocks of memory. This is because MadT is able to intercept library calls that perform block operations and treat their accesses in an aggregate way, instead of tracing their behavior instruction by instruction. Moreover, as discussed above, MadT behaves very well on computation intensive applications such as the Mandelbrot benchmark. Here MadT is able to provide

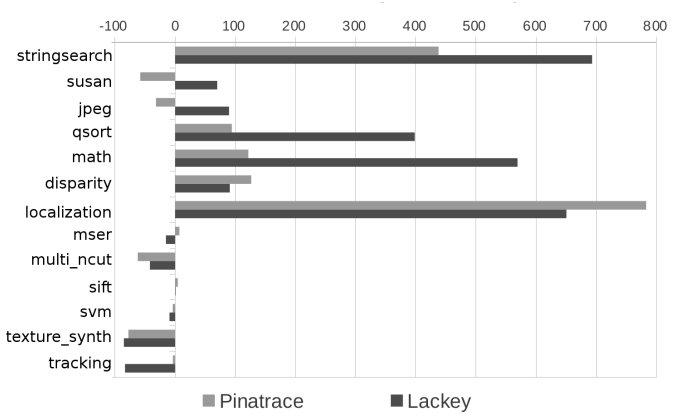

Fig. 2. Comparison on memory trace extraction time.

TABLE IV. SUMMARY OF BENCHMARKS NORMALIZED RUNTIMES

	No Tracing	MadT	Pinatrace	Lackey
Synthetic Benchmarks				
calloc	1	50	1520	2040
realloc	1	1	1410	1900
read	1	1	1270	1730
mmap	1	10	1170	1530
fwrite	1	20	2320	2980
mandelbrot	1	1	2	4949
MiBench Benchmarks				
stringsearch	1	290	1560	2300
susan	1	61520	25840	104490
jpeg	1	6527	4448	12378
qsort	1	15140	29410	75450
math	1	1021	2264	6832
San Diego Vision Benchmarks				
disparity	1	2744	6216	5237
localization	1	587	5190	4410
mser	1	1907	2044	1617
multi_ncut	1	12732	4815	7413
sift	1	20940	21842	21123
svm	1	17221	16548	15593
texture_synth	1	24855	5534	3654
tracking	1	19720	18966	3299

performance that match the native execution because this benchmark performs a limited number of memory accesses, while intensely executing mathematical operations on few CPU registers.

Next, we consider a subset of benchmarks from the MiBench suite that reflect common workloads in automotive and industrial domain [14]. The measurements for this set of benchmarks is included in second section of Table IV. It is easy to observe that MadT always achieves better or comparable performance with respect to Pinatrace and Lackey tools. As in the case of basic benchmarks, Valgrind runs in about the same order of magnitude of the Pin tool. This reflects the fact that both the tools use the same approach of performing binary instrumentation. The slight disadvantage of Valgrind seems to be related to optimization problems, since it is designed to support a broad set of architectures.

Finally, the last section of Table IV reports the achieved performance on the set of computer vision benchmarks. These benchmarks accept as input a series of sensor data or images, read them into internal buffers and compute their output locally by working on dynamically allocated memory. As a result, they are memory-intensive, hence significant slowdown is introduced by all the considered memory tracing tools. From the obtained results, it can be concluded that MadT does not improve on binary instrumentation tools when normal instruction execution is closely interleaved with frequent memory accesses. Intuitively, it can be concluded that, from a pure performance standpoint, not much advantage can be achieved by executing non-memory instructions natively and triggering memory protection traps upon memory accesses. In fact, the penalty of handling frequent segmentation faults can come close to the slowdown introduced by adding an intermediate binary translation layer.

We argue however that when comparable performance are achieved, the proposed MadT still represents the most valid memory-tracing option for system designer or third-party tools. In fact, MadT provides two main advantages that can allow post-processing of the produced trace to be simpler and more detailed. (1) Unlike Pinatrace and Lackey, MadT performs symbol resolution on observed memory accesses. This way, it is always able to provide the memory region in which the access has been performed and in most of the cases the name of the variable it is related to. Furthermore, (2) because no sandboxing is performed on the executed task, no additional memory regions are added to the task at analysis time, apart from the minimal number of regions required to link the MadT library. This makes significantly easier to infer the memory behavior of the task executing natively (i.e., outside the tracing environment) from the collected trace. Conversely, both Pin and Valgrind significantly rearrange the memory region layout of the analyzed task by almost doubling the total count of regions.

VI. Related Work

The problem of determining the memory access pattern of a task under analysis for workload characterization purposes has been approached from several perspectives, resulting in solutions with different advantages and trade-offs.

A first approach is to understand the structure of the task and reason offline about its behavior. A body of work in this direction uses abstract interpretation [15, 16] to determine the possible sequences of memory references across the possible execution paths of a given task.

Symbolic execution [17] represents another technique that has been largely used to determine possible execution paths in the control flow of a task, and thus the set of possible memory references [18, 19].

A third approach is based on collecting accurate memory traces from live execution of the target program. To the best of our knowledge, two main methods have been developed. The first one is to rely on dedicated hardware modules. However, hardware tracers are classified as industrial-grade instruments, thus they are normally expensive (tens of thousands of dollars). Moreover, they cannot be used on high-end platforms (e.g., Intel i7 and Xeon CPUs), which do not expose suitable hardware debugging interfaces. Finally, as it can be difficult to trace a selected portion of the target program execution, the users may face the difficult task to analyze huge trace logs to find the interesting information.

The second method to collect accurate memory traces is to perform instrumentation of the task code, triggering the tracing routines when instructions that perform memory accesses are encountered. Tools that allow code instrumentation are usually classified in two main categories: source-level and binary-level. When instrumentation is performed on the binary code, additional instructions are dynamically added to a compiled program. This can be done before the code starts its execution, as in QPT [20] and Pin [12], or while the program is running, as done by Valgrind [8].

Conversely, in the source-level approach, instrumentation is performed at compile time. Existing automatic source-level instrumentation is done either by performing source-to-source translation (e.g., ROSE [21]), or by introducing additional specific compilation logic [22, 23].

In this work, we propose a novel technique to perform memory access detection relying on memory protection mechanisms that sets itself apart from what we have described so far. MadT is aimed at demonstrating how it is possible to acquire accurate memory traces by introducing controlled memory faults and trap exceptions. In this way, we (1) do not rely on advanced debugging hardware capabilities, (2) do not need to instrument any of the instructions in the observed binary, (3) do not necessarily need to instrument the source code: in fact, we can directly perform memory tracing of a binary executable by dynamically linking a shared library containing MadT's code.

Actually, detecting memory accesses through controlled page faults has been proposed many times and it is considered a fundamental idea, for instance in garbage collection algorithms and consistency/replication protocols for distributed shared memory [24, 25]. However, collecting accurate traces of all accesses to a given set of memory pages is a different thing. In fact, garbage collection and consistency/replication protocols only require to determine that an access has been performed, which is a task significantly simpler than detecting all accesses.

A previous proposal for an accurate tracing mechanism based on page faults is the IF (Interpretation Fault) library [26], aimed at detecting memory accesses on the SPARC architecture. Actually, two alternative ideas were discussed: the first one was that, after the page fault, the machine code following the instruction that triggered the fault is modified so as to perform a jump to a recovery procedure, which removes again the access rights of the page and restores the original code of the program. The other idea was to rely on a hardware feature of the SPARC processors, namely using the special `npc` register to implement a branch to the recovery procedure. Although MadT is similar in some aspects to the IF library, it is also quite different because: (1) it does not rely on machine code instrumentation; (2) it exploits a CPU hardware feature that is much more common than the delayed branch register of the SPARC processors, namely the trapping (or single-stepping) mechanism that automatically generates an exception after each machine instruction; (3) MadT translates each virtual address in a symbolic form, mapping it to its original variable from the source code, allowing the user to easily understand the real memory access pattern of the target program.

VII. Conclusion

Understanding the memory footprint of an application is often fundamental for the analysis and design of real-time systems. In this paper, we present a novel technique for memory access detection and propose the implementation of a proof-of-concept tool, namely MadT, that is capable of recording memory accesses performed by an application in a fully symbolic form. In addition, our tool is capable of accurately tracking dynamic memory allocations and releases, forging when necessary new symbol names that simplify the identification of accesses to the dynamic memory.

By using MadT the user can accurately relate all memory accesses, including those to dynamically allocated blocks, to the source code of the target application. Compared to other profiling tools, MadT shows similar performances, and in some cases it is significantly faster. Furthermore, MadT does not require modifications to the source code of the application or of the operating system.

Acknowledgement. We gratefully thank Alessandra Cocozza, who worked on a preliminary version of MadT.

ACKNOWLEDGMENT

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1302563, and CNS-1219064. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

We gratefully thank Alessandra Cocozza, who worked on a preliminary version of MadT.

REFERENCES

[1] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. In *Real-Time Systems*. Springer, 2011.

[2] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for COTS-based embedded systems. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 269–279, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4344-4.

[3] E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha. Real-time i/o management system with cots peripherals. *Computers, IEEE Transactions on*, 62 (1):45–58, Jan 2013. ISSN 0018-9340. doi: 10.1109/TC.2011.202.

[4] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS'13, pages 45–54, Philadelphia, PA, USA, April 2013. IEEE Computer Society.

[5] R. Wang, Y. Gao, and G. Zhang. Real time cache performance analyzing for multi-core parallel programs. In *Cloud and Service Computing (CSC), 2013 International Conference on*, pages 16–23, Nov 2013.

[6] Free Software Foundation, Inc. GCC, the GNU compiler collection. `https://gcc.gnu.org/`, 1988–2014.

[7] Free Software Foundation, Inc. The GNU C library (glibc). `http://www.gnu.org/software/libc/`, 1992–2014.

[8] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250746. URL `http://doi.acm.org/10.1145/1250734.1250746`.

[9] W. Cohen. Tuning programs with OProfile. *Wide Open Magazine*, pages 53–62, 2004.

[10] Free Software Foundation, Inc. GNU Binutils. `http://www.gnu.org/software/binutils/`, 1988–2013.

[11] S.K. Venkata, I. Ahn, Donghwan Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M.B. Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 55–64, October 2009.

[12] CK Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of Programming Language Design and Implementation*, PLDI'05, pages 190–200, June 2005. URL `http://www.pintool.org/`.

[13] N. Nethercote. Lackey – a Valgrind tool. `http://www.valgrind.org/docs/manual/lk-manual.html`, 2012.

[14] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001. URL `http://www.eecs.umich.edu/mibench/`.

[15] P. Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 138–156, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-41635-8.

[16] C. Cullmann. Cache persistence analysis: Theory and practice. *ACM Trans. Embed. Comput. Syst.*, 12(1s):40:1–40:25, March 2013. ISSN 1539-9087.

[17] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782.

[18] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[19] M. von Detten. Towards systematic, comprehensive trace generation for behavioral pattern detection through symbolic execution. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 17–20, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0849-6.

[20] J.R. Larus. Efficient program tracing. *Computer*, 26(5):52 –61, May 1993. ISSN 0018-9162.

[21] Q. Sun and H. Tian. The ROSE source-to-source compiler infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT*, Galveston Island, Texas, USA, October 2011.

[22] Q. Sun and H. Tian. A flexible automatic source-level instrumentation framework for dynamic program analysis. In *Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on*, pages 401 –404, July 2011.

[23] Xiaofeng G., M. Laurenzano, B. Simon, and A. Snavely. Reducing overheads for acquiring dynamic memory traces. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pages 46–55, Oct 2005.

[24] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proc. of the SIGPLAN'88 Conf. on Programming Language Design and Implementation*, pages 11–20, Atlanta GA, USA, June 1988. ACM.

[25] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 96–107, New York, NY, USA, 1991. ACM. ISBN 0-89791-380-9. doi: 10.1145/106972.106984. URL `http://doi.acm.org/10.1145/106972.106984`.

[26] D.R. Edelson. Fault interpretation: fine-grain monitoring of page accesses. Technical report, University of California at Santa Cruz, 1992.