FBT_ARM: A Software Dynamic Translator for the ARM Architecture

Edward David Moreno
DCOMP/PROCC
UFS – Federal University of Sergipe
Aracaju, Brazil
edwdavid@gmail.com

Felipe Oliveira Carvalho, Admilson R.L. Ribeiro
DCOMP/PROCC
UFS - Federal University of Sergipe
Aracaju, Brazil
felipekde@gmail.com, admilson@ufs.br

***Abstract*—Software dynamic translation is a technique that allows code modification and monitoring of program execution. This paper addresses some applications of software dynamic translation (SDT) and the porting of fastBT—a dynamic translator for the IA32 architecture—to the ARM architecture. The result is a dynamic translator called FBT_ARM that works for the ARM and IA32 architectures.**

Keywords-software dynamic translator; ARM architecture; IA32 architecture; fastBT; FBT_ARM

I. INTRODUCTION

Software Dynamic Translation (SDT) is a technique that allows code modification and monitoring of the execution of program instructions at runtime. In the last few years, products using dynamic binary translation have become popular in the areas of virtualization, instrumentation and emulation [8]. SDT can be applied in several forms: dynamic code optimization, hardware architecture simulation, system virtualization, instruction set translation, profilers, debuggers, security constraint checking at runtime, co-designed virtual machines etc.

In the implementation of a SDT system, a software layer acts as virtual machine that manages the execution examining and dynamically translating all or part of the instructions of a program before they get executed by the host CPU.

Software dynamic translators are often written for a single application and/or platform. Besides the lack of portability due to the single application and single architecture approaches, few of the translation or instrumentation systems are open which prevents research in the area making them hard to study requiring the reimplementation of complex and delicate systems. Most of the code of a SDT system depends on the target hardware architecture. Both the data structures and the translated code emission must be designed according to the instruction set architecture.

Unfortunately, robust, general-purpose instrumentation tools are not nearly as common in the embedded arena compared to IA32, for example [3].

The Pin dynamic software translation system [4] provided support for the ARM architecture [3], but the ARM support has been discontinued in early 2007 (a search on http://archive.org indicates that the last version of Pin for ARM was released in January 2007). With the growing relevance of the ARM architecture driven by its adoption on most of the Smartphones, tablets, several embedded systems, and even some network servers (where x86 is commonly used), it makes sense to develop tools that improves the understanding of the ARM architecture, software development and execution in this platform.

As a result of this work, we present a software dynamic translator for the ARM architecture called "FBT_ARM". The translator can be used for program instrumentation in embedded systems based on ARM processors. The designer of a software instrumentation product can provide his own translation table. Such table should contain the routines that must be executed while translating the instructions of the executable code from the original program being instrumented. Such tool can be used to analyze the behavior of a software during its development (searching bugs, performance analysis, exploring ARM extension ideas) or even to run the program in production environment over the dynamic translator (e.g., implementation of a security layer that prevents a program from executing certain functions on a server, execution of a binary that contains non-standard ARM instructions implemented in software, implementation of a software-based transactional memory system).

The rest of the paper is structured as follows. Section II presents main concepts about SDT systems, section III is dedicated to fastBT translator, which is the baseline of our SDT for ARM. Section IV presents the steps for porting fastBT to our FBT_ARM. Finally, section V presents the conclusions and some ideas for future works.

II. THEORY AND IMPLEMENTATION OF SOFTWARE DYNAMIC TRANSLATION SYSTEMS

SDT systems can be divided in two classes according to its implementation approach: based on intermediate representation and table-based. Many SDT systems translate machine code to be executed into an Intermediate Representation (IR) that can be executed by an interpreter or just-in-time compiler. This additional level of indirection simplifies the implementation of the translator that can then represent the state of the translated program (execution stack, registers) in software. Valgrind [5], Strata [7], Pin [4] and QEMU [2] are examples of SDT systems that use an IR (Intermediate Representation) for their translators. An advantage of an IR is that it allows software reuse.

Other approach used in several projects is the Dynamic Binary Recompilation (DBR) [5]. It is similar to the use of IR in compiler projects where the front-ends of several high level programming languages deal with IR code generation that can be compiled by the backend of every supported hardware architecture. Reusing the backend allows

 ISBN: 978-1-61208-473-2

compilers of many languages to use most of the compilation optimizations from IR and apply them to all architectures supported by the backend, thus greatly reducing the effort of compiler creation.

A table-based software dynamic translator translates each instruction by executing specified functions from a table for each instruction. In general, this approach generates translators with better performance than IR-based translators. With the gain in performance comes a loss of flexibility, so many restrictions are imposed to the translation of instructions. Branch instructions, for example, should be treated especially so that translated code execution does not escape from translator control.

Finally, SDT can be used in different applications, especially as virtualization, instrumentation and emulation.

Virtualization: SDT is one of the approaches of virtualization of 32-bit x86 systems implemented by VMware [1] in all versions of VMware ESX until version 4.0. VMware ESX is an enterprise level product that provides computer virtualization at the kernel level. The translator used by VMWARE does not map instructions coming from target architecture to another. Instead, it translates the unrestricted x86 code to a subset of itself that can be safely executed. The translator particularly replaces privileged instructions with instruction sequences that perform the same privileged operations in the virtual machine instead of performing them in the physical machine.

Instrumentation: it is a technique that consists in the insertion of code in a program for the data collection and analysis of the instrumented program. One of the instrumentation techniques, Dynamic Binary Instrumentation (DBI), uses SDT to execute the instrumentation code at runtime. One example of the use of this technique is the Valgrind tool set [5].

Emulation: SDT systems are used to implement instruction set emulators. QEMU is an example of architecture emulator that allows, for example, the execution of ARM programs on x86 processors [2].

III. THE FASTBT DYNAMIC TRANSLATOR

The fastBT is a low overhead dynamic translator, it has a low memory footprint, is table-based and provides optimizations for all forms of dynamic control transfer instructions. fastBT presents a novel technique of translated target address prediction for dynamic control instructions combined with adaptive schemes to select the best configuration for each indirect control transfer. These optimizations lead to optimal translation depending on the instruction location in the program and not only in the class of the instruction [6].

The project and implementation of fastBT is neutral in relation to the processor architecture, but the available open source implementation is compatible only with IA-32 and Linux systems. The current implementation provides a table for the IA-32 architecture instructions and uses a thread-local cache for translated code [6]. Although it may increase memory usage, it avoids a complicated and error-prone lock scheme for the translation of multithreaded programs.

Besides that, fastBT authors say that in practice little code is shared between threads during the execution of programs, rendering the translated code cache redundancy even less of a problem.

The translation tables are generated from a high level description and are statically linked to the translator program during compilation. This is a flexibility that is not offered by many translators, see examples on Fig. 1 and Table I.

Finally, we made some experiments with fastBT and studied the performance. We used the programs from the Computer Language Benchmarks Game available in [10]. For most programs the overhead was small except for some cases where the overhead reached 400 % (`revcomp`) or even more than 23000 % (`knucleotide-4`).

IV. FBT_ARM: PORTING FASTBT TO ARM SYSTEMS

In this section we show the steps for porting fastBT to our FBT_ARM software, specific to ARM architectures. We show it using four subsections: How the translator works, the instructions table, the ARM instructions disassembler, implementations of simple calls, and finally, an example how the FBT_ARM works in a real program.

A. How the Translator Works

A program can be translated dynamically by preloading the `libfastbt.so` library before program execution. This `libfastbt.so` defines two symbols that will overwrite the symbols of the same name in the executable: `_init` and `_fini`. These two symbols are routines responsible for initialization and finalization of the program execution. Thus, this `libfastbt.so` defines `_init` with code that starts the dynamic translator hijacking control and starting the translation of the program.

The code in `_fini` finishes the translator with an error message. This error message is a warning about the loss of control of execution by the translator. If the translator works correctly `_fini` should not be executed, for the translator, as the first step, creates a mapping from the code in `_fini` to a routine the finishes the translator with no error message. Thus, if the translator is translating the program code, the eventual branch to `_fini` will be redirected to the routine that finishes the translator without any error.

The sequence diagram in Figure 2 shows how program control is hijacked by the translator.

At first `fbt init` initializes the thread-local storage space and initializes the trampolines. Trampolines are small dynamically generated code blocks that are used when a branch to some specific address in the program is requested and some code must be executed before the branch.

For example, when translating an indirect branch instruction the translator should not simply copy the instruction with the same target in program code as the control of the program would be lost by the translator after the branch to an address in the untranslated original program. This is what happens instead: the branch instruction is translated as a branch to a `tld->unmanaged_code_trampoline`.

 ISBN: 978-1-61208-473-2

```
instr_description table_opcode_08[] = {

/*0x0*/ {0, "ADD",      ADD, OPND_REG_SHIFT_BY_IMM, "action_copy", Add_to_register"},
/*0x1*/ {0, "ADD",      ADD, OPND_REG_SHIFT_BY_REG, "action_copy", Add_to_register"},
/*0x2*/ {0, "ADD",      ADD, OPND_REG_SHIFT_BY_IMM, "action_copy", Add_to_register"},
/*0x3*/ {0, "ADD",      ADD, OPND_REG_SHIFT_BY_REG, "action_copy", Add_to_register"},
/*0x4*/ {0, "ADD",      ADD, OPND_REG_SHIFT_BY_IMM, "action_copy", Add_to_register"},
/*0x5*/ {0, "ADD",      ADD, OPND_REG_SHIFT_BY_REG, "action_copy", Add_to_register"},
/*0x6*/ {0, "ADD",      ADD, OPND_REG_SHIFT_BY_IMM, "action_copy", Add_to_register"},
/*0x7*/ {0, "ADD",      ADD, OPND_REG_SHIFT_BY_REG, "action_copy", Add_to_register"},
/*0x8*/ {0, "ADD",      ADD, OPND_REG_SHIFT_BY_IMM, "action_copy", Add_to_register"},
/*0x9*/ {0, "UMULL", UMULL, None "action_copy", Unsigned_long_multiply _(32x32_to_64);
/*0xa*/ {0, "ADD",      ADD, OPND_REG_SHIFT_BY_IMM, "action_copy", Add_to_register"},
/*0xb*/ {0, "STRH",     STRH, OPND_REG_OFFSET | OPND_INCR_OFFSET, "action_copy", Store"},
/*0xc*/ {0, "ADD",      ADD, OPND_REG_SHIFT_BY_IMM, "action_copy", Add_to_register"},
/*0xd*/ {0, "LDRD",     LDRD, OPND_REG_OFFSET | OPND_INCR_OFFSET, "action_copy", Load"},
/*0xe*/ {0, "ADD",      ADD, OPND_REG_SHIFT_BY_IMM, "action_copy", Add_to_register"},
/*0xf*/ {0, "STRD",     STRD, OPND_REG_OFFSET | OPND_INCR_OFFSET, "action_copy", Store"},
};
```

Figure 1. Example of the ARM Instructions Table

The code in this trampoline saves the execution context, calls a function that translates the code in the target of the indirect branch (or finds this code in the translated code cache), modifies `tld->ind_target` to point to the translated code, restores the execution context and finally executes a branch to `tld->ind_target`.

These small code blocks are called trampolines because they are the target of branches and quickly branch to another code region. After initialization, `fbt_start_transaction` gets executed. This function finds out the return address using `_builtin_return_address`. This return address is the address of the first instruction after the branch to `FBT_start_transaction` which is located in the beginning of the program since the call to `fbt_start_transaction` is one of the first things done by the program. It is from this address—`orig_begin`— that code translation starts with the call to `fbt_translate_noexecute`.

Eventually, when `fbt_translate_noexecute` returns a pointer to the translated code block, the return address of the call to `fbt_start_transaction` at the top of the stack is replaced by the pointer to the translated code. Thus, `fbt_start_transaction` does not return execution to the untranslated program code but to the translated code.

The `fbt_translate_noexecute` has a loop that iterates over the program instructions from `orig_begin`, calls `fbt_disasm_instr` for each instruction and executes an action found in the instructions table (see Fig.1 and Table I to generate the translated code equivalent to that instruction. Besides that, `action_copy`, `action_branch`, and `action_branch_and_link`, and others return a value that indicated whether the block translation should be interrupted. Branch instructions (B, BL. …) for example, interrupt block translation.

Once it happens, a trampoline is added at the end of the translated code block. This trampoline is responsible for starting the translation or execution of the next translation block. When execution reaches the end of the translated code block, control returns to the translator or to the next translated instruction.

B. The Instructions Table

To understand the instructions (which operation, arguments …) and decide how to translate each instruction, the translator queries a table with information about each instruction.

This table is generated from many other tables with a higher level description of the instructions. FBT_ARM supports only the 32-bit ARM instruction set which makes the table-based instruction decoding simpler.

The high level tables in `arm_table_generator/arm_opcode.map.h` are built from the observation that it is possible to define what each instruction is about from the [27:20] and [7:4] bits.

For example, when the 32 bits of an ARM instruction follow the 0x 08 1 format it is already possible to assume that it is an ADD and that the second operand is left-shifted by a length specified in a register. 0x 08 1 is the bi-nary enconding of addf<c>g <Rd>, <Rn>, <Rm>, lsl <Rs> in ARM assembly language.

There is a high level table with 16 entries for each configuration of the [27:20] bits. The index of each entry is the configuration of the [7:4] bits. Fig. 2 shows the table with information about the instructions where the [27:20] bits are 0x80.

These tables are analyzed by the ARM table generator to automate the generation of a table with 4096 (2^{12}) entries. The index used to query this table is the concatenation of the 8 [27:20] bits with the 4 [7:4] bits.

 ISBN: 978-1-61208-473-2

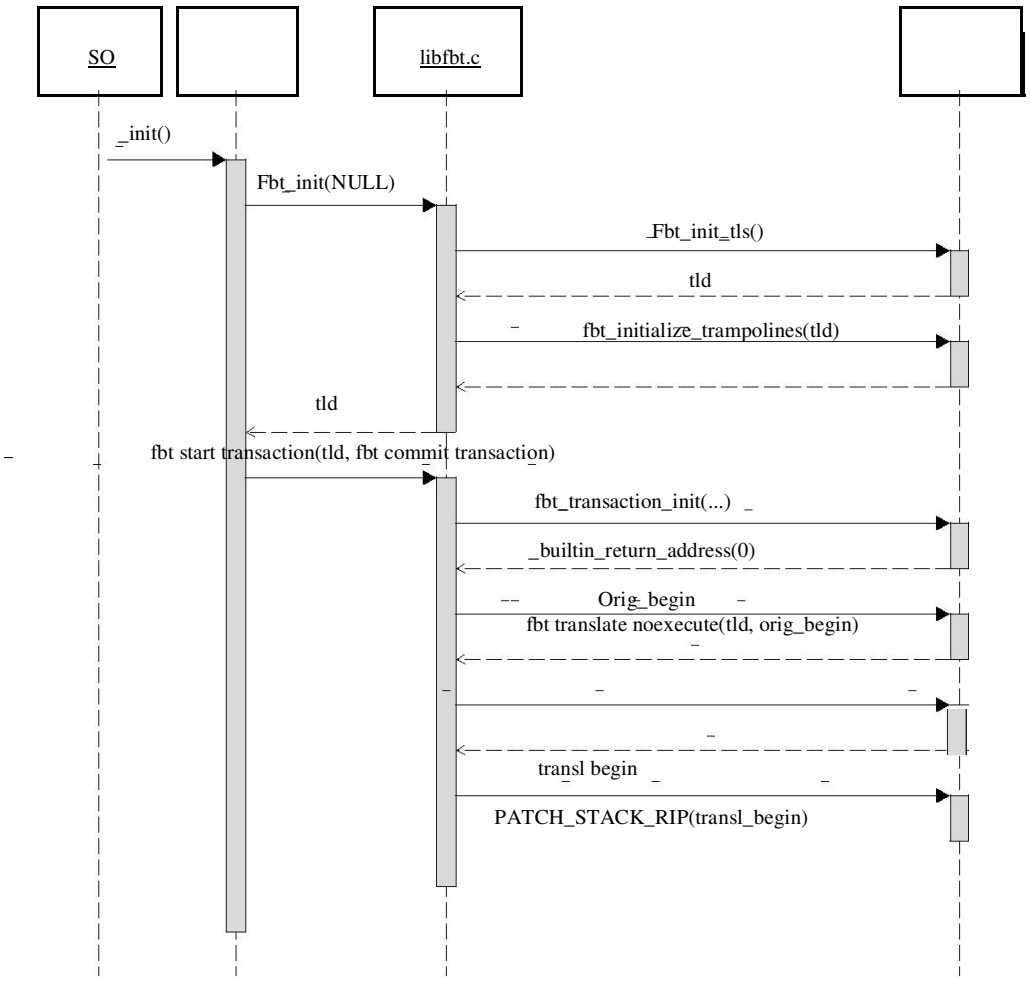

Figure 2. fastBT sequence diagram

TABLE I. COMPARISON OF THE FBT ARM DISASSEMBLER AND OBJDUMP DISASSEMBLER

0x	FBT_ARM	FBT_ARM (--sugar)	objdump -d
e52de004	str lr, [sp, #-4]!	push flrg	push flrg ; (str lr, [sp, #-4]!)
e92d4008	stmdb sp!, fr3, lrg	push fr3, lrg	push fr3, lrg
e59fe004	ldr lr, [pc, #4]	ldr lr, [pc, #4]	ldr lr, [pc, #4] ; 83d4
e8bd8008	ldmia sp!, fr3, pcg	pop fr3, pcg	pop fr3, pcg
eb00002c	bl 8474	bl 8474	bl 8474
012fff1e	bxeq lr	bxeq lr	bxeq lr
e12fff33	blx r3	blx r3	blx r3
e1b010a1	movs r1, r1, lsr #1	lsrs r1, r1, lsr #1	lsrs r1, r1, #1
e1a0c06c	mov ip, ip, rrx	rrx ip, ip	rrx ip, ip
01b0c0a0	movseq ip, r0, r0, lsr #1	lsrseq ip, r0, #1	lsrseq ip, r0, #1
e2844001	add r4, r4, #1 ; 0x1	add r4, r4, #1	add r4, r4, #1
e0a11a04	adc r1, r1, r4, lsl #20	adc r1, r1, r4, lsl #20	adc r1, r1, r4, lsl #20

 ISBN: 978-1-61208-473-2

The tables with the high level description of the ARM instructions for FBT_ARM were created after reading the ARM architecture reference manual.

C. The ARM Instructions Disassembler

To test the instruction table and understand what information is necessary in the tables to decode the instructions, we developed an ARM disassembler (`src/arm/fbt_disassemble.c`).

Table 1 shows the output produced by the FBT-ARM disassembler and the output of the `objdump -d` command that comes from the GNU `binutils` software package. When the *--sugar* flag is passed, FBT_ARM translates some instructions to pseudo-instructions if appropriate. Load instructions like LDR, and LDMIA are translated to the POP pseudo-instruction when the memory address is the register storing the pointer to the base of the stack—SP—and the operands configuration makes the instruction semantically equivalent to the popping from the stack. Similarly, store instructions can be translated to PUSH; and MOVs with shifted-operands can be translated as pseudo shift instructions.

D. Implementation of System Calls

To avoid `libc` as a dependency and allow the interception of system calls we had to implement Linux syscalls in Fbt. The ARM ABI standard defines how system calls should be implemented.

The source file `src/arm/fbt_syscalls_impl.h` has C preprocessor macros and the im-plementation of many syscalls using inline assembly. Besides being used in the translator, this syscalls implementation is also used to implement I/O functions (e.g. `fllwrite`, `fllprintf` ...) and a low-level memory allocator (`fbt_lalloc`, `fbt_mem_free` ...) which are used by our FBT_ARM.

E. Translating a Simple Program

Fig. 3 shows the ARM code of the program that will be translated using FBT_ARM. This program sums two 1-digit numbers passed as arguments from the command line (`./prog 3 7`) and terminates with an exit code equals to the sum of the two numbers. It is a simple example without branches consisting of a single translation block. Fig. 4 shows the output of the program of Fig. 3. It is important to observe that the exit code (the *$? shell variable*) is indeed the sum of the two arguments (see Fig. 4).

Fig. 5 shows fragments of the debug output (`debug.txt`) produced by FBT_ARM during the dynamic execution of the program. The debug output fragment produced by FBT_ARM shows the translation of each instruction until SWI, that when found by the translator, concludes the translation of the block (`closing TU upon request`, `invoking translation function on 0x000088a4`).

It is after the translation of the block that execution control is passed to the translated code (`starting transaction at 0xb5cd6000 (orig. addr: 0x0000885c`)).

The original program code starts at `0x0000885c` and the translated code which is the one executed by the processor can be found at the `0xb5cd6000` address.

```
bl fbt_start_transaction
        // int a = argv[1][0] - '0';

ldr r3, [fp, #-20]     // r3 = argv (argv stored in the stack)
add r3, r3, #4         // r3 = argv + 1
ldr r3, [r3]           // r3 = *(argv + 1) or r3 = argv[1]
ldrb r3, [r3]          // r3 = *(argv[1]) or r3 = argv[1][0]
sub r3, r3, #48        // r3 = argv[1][0] - '0'
str r3, [fp, #-8]      // stores a in the stack

        // int b = argv[2][0] - '0';
ldr r3, [fp, #-20]     // r3 = argv (argv stored in the stack)
add r3, r3, #8          // r3 = argv + 2
ldr r3, [r3]            // r3 *(argv + 2) or r3 = argv[2]
ldrb r3, [r3]           // r3 * (argv[2]) or r3 = argv[2][0]
sub r3, r3, #48         // r3 argv[2][0] - '0'
str r3, [fp, #-12]      // stores b in the stack

        // a + b
ldr  r2, [fp, #-8]     // r2= a
ldr  r3, [fp, #-12]   // r3= b
add r3, r2, r3       // r3 = a + b

        // exit(a + b)
mov r0, r3          // first argument (a + b)
mov r7, #1          // SYS_exit (syscall code)
swi  0              // request syscall handling by the kernel

bl     fbt_commit_transaction
```

Figure 3. Program code to be dynamically translated

```
$ ./prog 2 3
Starting BT
Stopping BT $
echo $?
5
$ ./prog 8 7
Starting BT
Stopping BT $
echo $?
15
```

Figure 4. Dynamic execution of a simple program

V. CONCLUSIONS

To make this work possible it was necessary to analyze software dynamic translation solutions and the implementation techniques they use. The main target of the analysis was fastBT whose code was extended to support the ARM architecture. FBT_ARM has tables describing 32-bit instructions of the ARMv6 architecture and necessary routines for instruction decoding. To demonstrate how the tables can be used, we have implemented a disassembler that produces the output similar to the output of production disassemblers (`objdump -d`). Although FBT-ARM is not capable of translating full programs, all the infrastructure of fastBT was ported and works on ARM processors.

 ISBN: 978-1-61208-473-2

```
fbt_start_transaction(commit_function = 0x0000dcb8) {

 translate_noexecute(*tld=0xb6f2e000, *orig_address=0x0000885c) {
fbt_ccache_find(*tld=0xb6f2e000, *orig_address=0x0000885c) {

}-> 0x00000000

tld->ts.transl_instr: 0xb5cd6000 fbt_ccache_add_entry(*tld=0xb6f2e000, *orig_address=0x0000885c, *transl_address=0xb5cd6000)
{}
fbt_disasm_instr(*ts=0xb6f2e458) { Disassembling 0xe51b3014 } translating a 'ldr'
action_copy(*addr=0x0000885c, *transl_addr=0xb5cd6000) {}-> NEUTRAL
fbt_disasm_instr(*ts=0xb6f2e458) { Disassembling 0xef000000 } translating a 'swi'
action_copy(*addr=0x000088a0, *transl_addr=0xb5cd6000) {

        Encountered an interrupt - closing TU with some glue code
}-> CLOSE_GLUE

closing TU upon request, invoking translation function on 0 x000088a4
allocated trampolines: 0xb5ccf000, target: 0x000088a4, origin: 0 xb5cd6004
}-> 0xb5cd6000, next_tu=0x000088a4 (len: 0)
starting transaction at 0xb5cd6000 (orig. addr: 0x0000885c)

}
```

Figure 5. Debug output produced by FBT_ARM

FBT_ARM is open source and available in [9]. Since it is a table-based translator which often means better performance than those based on intermediate representation, there are many possibilities of use for FBT_ARM. Its implementation can be extended in the development of several tools that benefit from software dynamic translation like memory profilers, general program analysis tools, secure execution environments, etc.

For future works, we would like to suggest: (i) finish the implementation of our FBT_ARM, adding the translation to ARM´s control instructions, and (ii) execute a final version in real programs and benchmarks, and (iii) compare the performance of FBT_ARM to other systems.

ACKNOWLEDGMENT

This research work received financial support from CNPq, CAPES, FAPITEC (Brazilian Government Institutions for Science and Technology).

REFERENCES

[1] Agesen, O. Software and hardware techniques for x86 virtualization. 2009. Electronic Publication, www.vmware.com/files/pdf/software_hardware_tech_x86_virt.pdf. Visited in April 20, 2016.

[2] Bellard, F. Qemu, a fast and portable dynamic translator. In Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005.

[3] Hazelwood, K. and Klauser, A. A dynamic binary instrumentation engine for the arm architecture. In ACM Proc. of the 2006 Intl. Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), p. 261–270, USA, 2006.

[4] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. SIGPLAN Not., 40(6):190–200, 2005.

[5] Nethercote, N. and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Notes, 42(6):89–100, 2007.

[6] Payer, M. and Gross, T. R. Generating low-overhead dynamic binary translators. In Proceedings of the 3rd Annual Haifa Experimental Systems Conference, SYSTOR '10, pages 22:1–22:14, New York, NY, USA. ACM, 2010.

[7] Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J. W., and Soffa, M. L. Retargetable and reconfigurable software dynamic translation. In IEEE Proc. of the Intl. Symposium on Code Generation and Optimization (CGO), p. 36–47, USA, 2003.

[8] Wirth, M. Simple pluggable binary translator library in user space. Laboratory for Software Technology, ETH Zurich, 2008. http://www.nebelwelt.net/publications/students/07hs-wirth-fastBT.pdf. Semester thesis. Visited in April 12, 2016.

[9] Moreno, E.D. and Carvalho, F. Electronic Publication: Code of FBT_ARM, available at https://github.com/philix/Fbt, 2015. Accessed in April 30, 2016.

[10] Computer Language Benchmarks Game, Available at https://github.com/philix/c_bench, 2015. Accessed in April 25, 2016.

 ISBN: 978-1-61208-473-2