A Fast, Cycle-accurate Space Processor Emulator Based on a Dynamic Binary Translator*

Jong-Wook CHOI¹⁾ and Byeong-Gyu NAM²⁾

¹⁾*Satellite Flight Software Department, Korea Aerospace Research Institute, Daejeon 305–806, Korea*

²⁾*Department of Computer Science and Engineering, Chungnam National University, Daejeon 305–764, Korea*

For many years, flight software (FSW) has been developed and verified using real hardware-based software test beds (STBs). Even though the STB presents the best real-time behavior of the FSW, it has a number of drawbacks such as very long delivery time for the target hardware, frequent changes in hardware configurations, and limited concurrent access to the STB. Therefore, software-based satellite simulators have been developed from the start of a project to overcome the restrictions and limitations imposed by the STB. They enable the development of FSW to progress in parallel with developing the target hardware from the initial phase of the project. In this approach, the processor emulator is the essential component of the satellite simulator, but its interpretation-based approach cannot meet the real-time requirements as the clock speed of the emulation target increases. In this paper, we discuss the design and development of a high-performance and cycle-accurate space processor emulator based on a just-in-time (JIT) dynamic binary translation (DBT) scheme. We also present a solution for self-modifying code emulation which is essential for satellite software but has been a limiting factor in the DBT scheme.

Key Words: Dynamic Binary Translation, Emulator, Simulator, Flight Software, ERC32, LEON2

Nomenclature

DBT: dynamic binary translation

BB: basic block

TB: translation block

TC: translation cache

EM: emulation manager

SPC: source program counter

TPC: target program counter

FSW: flight software

1. Introduction

The importance of on-board flight software (FSW) in satellite missions is growing depending on the complexity and diversity of the mission. For many years, FSW has been developed and verified using the software test bed (STB) environments based on real hardware at the Korea Aerospace Research Institute (KARI). During the development of FSW on the STB, we have experienced many annoying problems such as delayed delivery of target hardware, limited concurrent access to the STB from software engineers, and the time and cost to create multiple STBs for different satellites. Thus, software-based satellite simulators are now developed from the beginning of a satellite project to avoid these issues.

The software-based simulator enables the FSW to be developed concurrently with the satellite hardware from the initial phase of a project. It simulates the on-board computer (OBC), I/O modules, avionics and payloads. It can easily

be deployed and adapted when changing hardware configurations and supports debugging and testing of the FSW. Moreover, the FSW can be loaded into a simulator without any modifications and tested faster than when using real hardware.

The processor emulator is one of the essential parts in a satellite simulator, emulating guest processors such as the ERC32¹⁾ and LEON2/3-FT^{2,3)} (Fault Tolerant), the most dominant processors in space science. The prior emulator for the ERC32, *laysim-erc32*,⁴⁾ adopted an instruction-level interpretation method and allowed faster-than-real-time (FTRT) simulation due to the slow target CPU running at 20 MHz. The next processor emulator (i.e., *laysim-leon2*) for the LEON2-FT based on the interpretation method, cannot run in real-time due to the architectural complexity of LEON2-FT and higher clock speed of the processor, which can reach 100 MHz.

In this paper, we present a high-performance processor emulator, *laysim-dynamic binary translation (DBT)*, to support real-time simulations for the high-performance processor LEON2-FT. It supports cycle-accurate simulations for system-level simulation with a real-time operating system (RTOS) and embedded peripheral devices. It also provides a solution for the self-modifying code problem, which is a potential problem of the DBT.

The proposed *laysim-DBT* has the following three features. First, it improves the performance of the *laysim-DBT* through just-in-time (JIT) DBT by minimizing helper functions and maximizing translation block chaining. The helper functions are C routines called from the translation cache (TC) to support complicated target instructions. Translation block chaining links the translation blocks (TB) to each other instead of branching to the emulation manager

(EM). The DBT is an effective technique to achieve fast simulations of instructions in target machines such as the ERC32 and LEON2 on a host machine like the x86. The DBT translates blocks of guest code into host code during run time and stores them in a code cache. Once the guest code blocks are translated, they can run on a host platform using its native code, greatly improving performance over the interpretation method. Second, the laysim-DBT updates the number of cycles used by the basic blocks (BB) under a static cycle-approximate model for cycle accuracy. It checks every trap condition for precise trap handling and interrupt conditions for fast interrupt services. Finally, it supports the self-modifying code that needs to update the original guest code after a translation process, which is known to be a potential problem of the DBT.

We evaluated the performance of the laysim-DBT and compared the cycle accuracy with TSIM-ERC32,⁵⁾ laysim-erc32 and QEMU laysim-erc32⁶⁾ using various embedded benchmarks such as Dhrystone v2.1, Whetstone, Stanford, EEMBC CoreMark⁷⁾ and MiBench.⁸⁾ The laysim-DBT simulation proved to be 8.72 times faster than TSIM-ERC32 and 1.56 times faster than QEMU laysim-erc32 despite its cycle accuracy. It demonstrated a cycle accuracy of 95.49% with a $\pm 8\%$ error margin compared to TSIM-ERC32, which is a cycle-true emulator of the ERC32.

The rest of this paper is organized as follows. Section 2 introduces the motivation for introducing the new processor emulator and history of our processor simulators. In Section 3, our new processor emulator, laysim-DBT, is described in detail, and performance and cycle comparisons are presented in Section 4. Section 5 briefly introduces our simulator and shows its performance at the simulator level. Finally, we offer a conclusion in Section 6.

2. Emulator/Simulator for Space Processors

The processor in an OBC is responsible for performing the FSW, which controls the satellite and accomplishes missions to be loaded and executed. It is specially designed to be operated in a space environment. The RAD750, based on IBM's PowerPC750, is used as the OBC by the National Aeronautics and Space Administration (NASA). In the case of the European Space Agency (ESA), the ERC32 processor (SPARC v7 with FPU and embedded peripherals) has been used as main processor. Additionally, the LEON2/3 processor based on SPARC v8 has been developed and used for next-generation satellites. Currently, satellites being developed by KARI use the ERC32 and LEON2-FT processors in their OBCs.

2.1. Space processor emulator

The methods of emulating the processor can be categorized into two major ways: interpretation and dynamic binary translation. Interpretation is a widely used method for cross-platform program execution. It fetches an instruction from guest executable codes, decodes it to the host platform such as a x86 machine and then executes it. So it has a large overhead for every converting instruction, and it is very hard to

meet the real time performance when guest system is running on a high speed clock. But this method is relatively easy to implement and provide cycle-true emulation of the guest platform. DBTs such as QEMU,⁹⁾ CrossBit¹⁰⁾ and ARCSIM¹¹⁾ take a different approach. Blocks of guest instructions are compiled to host instructions JIT as they are encountered and stored in a TC. When the same block is encountered again, the precompiled block is retrieved from the TC and executed. This enables a remarkable performance enhancement of approximately 5- to 10-fold compared to the interpretation emulator. However, DBTs do not support any mechanism for instruction cycles.

Most ERC32/LEON2 processor emulators have been developed and used by ESA and partner companies, whereas only TSIM-ERC32/LEON2¹²⁾ (Aeroflex Gaisler) is sold as a commercial version. But the interpretation-based LEON2 processor emulator cannot meet the real-time simulation speed because of the high performance of the LEON2 processor, which has a performance rating that is 3–5 times higher than the ERC32. The TSIM emulator, which uses the interpretation method, peaks at approximately 30 DMIPS¹²⁾ (Dhrystone MIPS), while the Atmel AT697F^{2)/} LEON2-FT processor can deliver 86 DMIPS. Our AT697F development board also shows the same results. To resolve the performance bottleneck, ESA and Aeroflex Gaisler developed the hardware-based LEON2 emulator, TSIM-HW,¹²⁾ and the University of Coimbra undertook a study for ESA to investigate the dynamic translation of the LeonVM.¹³⁾ SciSys UK adapted QEMU to allow it to be used to emulate the ERC32 and LEON2 processors^{14,15)} with instruction cycle counting, and used it for the GALILEO Constellation Simulator. Couverture/GNATemulator,¹⁶⁾ which is an extension of the QEMU for ERC32/LEON2, is used at Ada-Core for software development.

2.2. History of our emulator/simulator development

Our first KOMPSAT-FSS¹⁷⁾ (flight software simulator for Korea multi-purpose satellites) used the TSIM-ERC32 as the processor emulator, and VASI ASIC, various controllers and memories were implemented in external interfaces of the TSIM-ERC32. When the FSW accesses external registers or memories, TSIM-ERC32 calls the external emulation function of the KOMPSAT-FSS according to access address and type. However, it is difficult to modify the emulator core and add functions to the TSIM-ERC32 for simulating a number of FPGA/ASIC in the OBC when developing the the KOMPSAT-FSS using a TSIM-ERC32. We were also met with a variety of difficulties when trying to establish debugging interfaces for monitoring registers and variables in real time. We needed to remove the constraints and dependencies of the TSIM for domestic satellite development.

After construction of the first KOMPSAT-FSS, we developed a new ERC32 processor emulator, laysim-erc32, to resolve those problems and be a substitute for the TSIM-ERC32. The laysim-erc32 is a graphic user interface (GUI)-based, cycle-true emulator based on interpretation and includes the embedded source-level debugger. We integrated the laysim-erc32 into the second KOMPSAT-FSS,

and it was used as the formal simulator for FSW development and as the ground operation simulator. The performance of the laysim-erc32 was slightly lower compared to the TSIM-ERC32 due to the overload of GUI processing, but it provided significantly better environments for the FSW developers.

We developed another ERC32/LEON3 processor emulator, QEMU laysim-erc32/leon3, based on QEMU to fulfill the real-time simulation of a high-performance processor. QEMU laysim-erc32/leon3 has a simulation speed that is 8.64 times faster than the TSIM using the Dhrystone benchmark. The high performance of the QEMU laysim-erc32 will be a big advantage to guarantee the performance of satellite simulators. However the QEMU laysim-erc32/leon3 is not a cycle-accurate model, rather a functional emulator, because it relies on a host clock without instruction cycle counting. Therefore, it cannot emulate cycle accuracy, which leads to issues with the guest processor clock and I/O timing. So, it is inappropriate to run FSW, which has time-constrained attributes.

After our experience with processor emulators and simulators, we decided to develop a new processor emulator based on the DBT from scratch to overcome all shortcomings of prior emulators; performance limitations of interpretation emulators and the absence of cycle counting of QEMUs.

3. The Design and Development of laysim-DBT

Although DBT is an attractive technology, the technology itself is very complex and difficult to implement, requiring a lot of manpower and material resources.¹⁰⁾ Nevertheless, the

laysim-DBT is inevitable required to develop, test and verify FSW, and it can resolve all drawbacks of prior emulators.

The aims of the laysim-DBT are to achieve high performance, be cycle accurate, and resolve self-modifying problems. The first strategy for improving performance is to use DBT. The DBT engine generates a host x86 TB from the guest ERC32 BB without intermediate representation (IR) on run time. To enhance the performance of the DBT engine, it minimizes helper functions, which are C routines called from the TC for emulating complicated target instructions to avoid function call overload. Additionally, it maximizes TB chaining without self-chaining by jumping to the next TB directly and increasing speed of processing, which avoids switching back to the EM. Furthermore, of course it uses multi-threading for other avionics equipment simulators.

The second strategy for cycle accuracy is to calculate the BB instruction cycles with a static cycle-approximate model considering a conditional branch and trap case during every translation. After execution of the TB, it updates execution cycles, and checks every trap condition for precise trap supporting and interrupt conditions for fast interrupt handling.

The last strategy for self-modifying code is to use shadow memory, which reflects the translation status to resolve the side effect of using *VirtualProtect()* in Windows.

3.1. Architecture of laysim-DBT

The laysim-DBT was developed using the GNU compiler 4.6.2 and GTK 2.22.1 library for the GUI, so it can be built and executed on Windows and Linux platforms without any modification. laysim-DBT can be divided into several parts as depicted in Fig. 1.

Fig. 1. The architecture of the laysim-DBT.

The EM controls all operations of processor emulation. The DBT module is responsible for generating native codes from guest codes. During execution of a TB in the TC, the TB can directly access guest registers and memory on behalf of state mapping. The helper functions called from the TB support complicated guest instructions and can communicate with the simulator via I/O access. The cycle module maintains the simulation time and reflects instruction execution time to it. It also handles the timer devices of the processor and invokes timer interrupts on expiration. The time event module services timed-events in the event table. The interrupt module checks interrupt conditions and manipulates the interrupt controller of the processor. All synchronous and asynchronous traps (interrupts) are handled by the trap module. The code cache holds the blocks of code translated by the DBT engine, and the source program counter (SPC)-to-target program counter (TPC) map table provides a way of indexing into the code cache implemented as a hash table. The shadow memory has the status of translating guest memory to check self-modifying conditions.

3.2. Emulation manager

The laysim-DBT does not use an interpretation mode, but rather generates native host codes directly using the DBT engine like the QEMU. Advanced control is provided by the EM, which is part of the runtime support. The workflow of the EM is as follow:

- 1) Get the SPC of the first instruction in the next BB.
- 2) Calculate hash index using the SPC and check whether its corresponding TPC exists in the SPC-to-TPC map table.
- 3) If a TPC exists in the map table, it indicates that the BB has already been translated as the TB stored in the TC, which then branches to the TPC and executes the TB; otherwise, jump to 4).
- 4) If a TPC is missed in the map table, then
 - a) Update the map table
 - b) Generate Prologue
 - c) Invoke the translator to generate a TB from the BB, and store it to the TC
 - d) Generate Epilogue
 - e) Branch to the TPC and execute the TB
- 5) After execution of the TB, time operation is processed to update and service the timed events.
- 6) Then the pending interrupt and trap are serviced by Interrupt/Trap modules.
- 7) To speed up execution, chaining operation for TBs is performed using a machine jump instruction.
- 8) Finally, it handles the retranslation condition, which flushes the TC, map table and shadow memory to resolve the self-modifying problem.

The EM executes steps 1)–8) repeatedly as shown in Fig. 2.

3.3. State mapping

State mapping refers to mapping the register and memory of the guest to the host address space. In the host address space, guest registers can be mapped to the host registers or register context block in memory, where the guest code

Fig. 2. Workflow of emulation manager.

Fig. 3. State mapping for guest processor.

and data are mapped into the host memory. For register state mapping, if the number of host registers is larger than the guest, it becomes an easier to implement by mapping all of the guest registers into the host registers for the duration of emulation. But if the number of host registers is insufficient, then frequently used guest registers are mapped to host registers and the unmapped registers must be mapped to a register context block in the host's memory.

In our system, the ERC32/LEON2-FT is the guest and x86 is the host. The SPARC instruction set architecture (ISA) has more registers than the x86 ISA, so the guest registers must be mapped to the register context block in the host. The *edx* register of the x86 is used to point to the guest register context block, which is allocated in the host memory, and other x86 registers are used by the DBT engine. During execution of the load/store instructions, the guest memory address can be accessed by the combination of host register, *ebx* register, which holds the base address of the guest memory, and *eax* register, which has the offset. Figure 3 shows the guest register context block for the ERC32, which consists of IU registers, control registers, FPU register, ERC32 system registers, and variables of DBT for operation. This context block is used in the TB for execution, and all emulator modules share it.

3.4. DBT

The DBT engine of QEMU and CrossBit uses two-phase translation: in the first phase, it translates one BB of guest code into machine-independent IR, then it translates IR to

the specific host TB while optimizing and profiling the IR. This strategy is for a multi-platform DBT engine and can apply various optimization techniques, but translation overload is inevitable.¹⁸⁾ For the laysim-DBT, it directly generates the host TB from guest BB without IR to support faster translation and easy optimization. Our evaluation result shows the time of translation is really a small portion (0.132%) considering all execution time.

3.4.1. Construct BB

The DBT engine translates one block of guest code at a time, the natural unit for translation is a BB. BBs are determined by the actual flow of a program as it is executed. It always begins when the instruction is executed immediately after a branch or jump, and ends with a delay instruction at the next branch or jump except annulled branch instructions.

Figure 4 shows an example of constructing BBs. On first execution, if argument *a* is less than *b*, then the DBT engine constructs three BBs by execution flow and translates them to three TBs. On second execution, if argument *a* is bigger than *b*, then the DBT engine constructs two new BBs although BB #5 is a subset of BB #3 because of the code discovery.

3.4.2. DBTs of laysim-DBT

The EM triggers the DBT engine to generate a TB from a BB on miss after checking the map table with the TPC. Figure 5 shows DBT operation with a BB from the ERC32 RAM start-up code. First the DBT engine generates the prologue to be called from the EM using macros automatically and stores it to the TB. Then it starts to decode BB instructions, translates them to the TB, and calculates the execution cycles of the BB based on the static cycle model during run time. The address of the ERC32 register context block is allocated to the *edx* register of the x86, and the *eax*, *ebx* and *ecx* registers are used for emulation. In the case of guest *jmp* instruction, it may cause a trap on the mis-aligned 32 bit address. Therefore, the DBT engine generates the logic for checking the memory alignment, which sets the trap number, updates the total execution cycles and returns to the EM under trap conditions. Finally, the epilogue is created to update the PC/NPC of the guest. It is then finished with increments of instruction cycle counts, and by returning to the EM. In view of cycle counting, a BB increases 5 cycles during normal operation, but it increases three cycles under trap conditions.

All translated codes are stored in the TC as illustrated in Fig. 6. The TB consists of 28 native x86 instructions with the prologue and epilogue. The delay *nop* instruction, followed by *jmp* of the BB is not translated, but increases the number of the instruction cycles. If the instruction followed by the branch is not a delay *nop* instruction, it is also translated. During execution of the TB, if a trap occurs due to a mis-aligned address, control is transferred to the EM by returns and then the EM handles the trap using the trap module. This enables precise trap handling by the DBT.

3.4.3. Fast interrupt and precise trap handling

The SPARC architecture supports three types of traps; synchronous, floating-point and asynchronous. Synchronous

<pre> int max (int a, int b) { if (a > b) return a; else if (a == b) return -1; else return b; } </pre>		1 st Exe. (< <i>a</i> > <i>b</i>)	2 nd Exe. (> <i>a</i> > <i>b</i>)
2001b18: save %sp, -112, %sp	2001b1c: st %i0, [%fp + 0x44]	Basic Block #1	
2001b20: st %i1, [%fp + 0x48]	2001b24: ld [%fp + 0x44], %i1		
2001b28: ld [%fp + 0x48], %i0	2001b2c: cmp %i1, %i0		
2001b30: ble 2001b48	2001b34: nop		
2001b38: ld [%fp + 0x44], %i0	2001b3c: st %i0, [%fp + -12]		
2001b40: b 2001b74	2001b44: nop	Basic Block #2	Basic Block #4
2001b48: ld [%fp + 0x44], %i1	2001b4c: ld [%fp + 0x48], %i0		
2001b50: cmp %i1, %i0	2001b54: bne 2001b6c		
2001b58: nop	2001b5c: mov -1, %o0		
2001b60: st %o0, [%fp + -12]	2001b64: b 2001b74		
2001b68: nop	2001b6c: ld [%fp + 0x48], %i0	Basic Block #3	Basic Block #5
2001b70: st %i0, [%fp + -12]	2001b74: ld [%fp + -12], %i0		
2001b78: nop	2001b7c: ret		
2001b80: restore			

Fig. 4. Example of constructing basic blocks.

<pre> [ERC32 Basic block] 02000000 a0100000 clr %i0 02000004 29000004 sethi %hi(0x2001000), %i4 02000008 81c52000 jmp %i4 + 0x0 0200000c 01000000 nop </pre>	
<pre> [laysim-DBT engine] #----- prologue ----- x86_emit_push_r32(ebp) x86_emit_mov_r32_r32(esp, ebp) x86_emit_mov_imm32_edx((guint32)(&ERC32)) #----- or ----- x86_emit_mov_m32_r32(edx, eax, (4 * op3_rs1)) x86_emit_mov_m32_r32(edx, ebx, (4 * op3_rs2)) x86_emit_or_r32_r32(ebx, eax) x86_emit_mov_r32_m32(eax, edx, (4 * op3_rd)) #----- sethi ----- x86_emit_mov_imm32_eax((INST.inst_f2_sethi.imm22 << 10)) x86_emit_mov_r32_m32(eax, edx, (4 * INST.inst_f3_i0.rd)) #----- jmp ----- x86_emit_write_imm32_erc32_pc(pERC32->erc32.pc) x86_emit_write_imm32_erc32_npc(pERC32->erc32 npc) x86_emit_mov_m32_r32(edx, eax, (4 * op3_rs1)) x86_emit_mov_m32_ebx(operand) x86_emit_add_r32_r32(ebx, eax) x86_emit_test_imm8_al(0x3) x86_emit_jz_with_offset(22) x86_emit_write_erc32_trap_with(ERC32_TRAP_07_MEM_NOT_ALIGNED) x86_emit_increas_icnt((inst_cnt+1)) x86_emit_return x86_emit_mov_imm32_ebx(pERC32->erc32.pc) x86_emit_mov_r32_m32(ebx, edx, (4 * op3_rd)) x86_emit_write_eax_erc32_npc #----- epilogue ----- x86_emit_read_erc32_npc_eax x86_emit_write_eax_erc32_pc x86_emit_add_imm8_r32(4, eax) x86_emit_write_eax_erc32_npc x86_emit_increas_icnt(inst_cnt) x86_emit_return </pre>	

Fig. 5. Example translation operation by the DBT engine.

traps are caused by hardware responding to a particular instruction or by a *Ticc* instruction, and they occur during the instruction that caused them. Floating-point traps caused by FP instruction occur before the instruction is completed. Asynchronous traps (interrupt) occur when an external event interrupts the processor such as timers, UART and various controllers. The trap module handles all trap operations according to the SPARC trap mechanism.

The EM checks the pending interrupts in the interrupt pending register of the processor for fast interrupt handling after every TB execution. These interrupts come from embedded processor peripherals, or external devices in the simulator. These interrupt events, except timers, are registered as timed events in the event table by the time event module. The interrupt, which is recorded in the event table, can be delayed

[x86 Translation block in TC]			
#-----			prologue ---
55	push	%ebp	
89e5	mov	%esp,%ebp	
bae0b00101	mov	\$0x101b0e0,%edx	; ERC32 register addr -> %edx
#-----			or -----
8b4200	mov	0x0(%edx),%eax	; ERC32.g0 -> %eax
8b5a00	mov	0x0(%edx),%ebx	; ERC32.g0 -> %ebx
0948	or	%ebx,%eax	; %ebx OR %eax -> %eax
894240	mov	%eax,0x40(%edx)	; %eax -> ERC32.l0
#-----			sethi -----
b800100002	mov	\$0x2001000,%eax	; 0x2001000 -> %eax
894250	mov	%eax,0x50(%edx)	; %eax -> ERC32.l4
#-----			jmp -----
c78200200000000002	movl	\$0x2000000,0x280(%edx)	; PC -> ERC32.PC for trap
c78200200000000002	movl	\$0x200000c,0x284(%edx)	; nPC -> ERC32.nPC for trap
8b4250	mov	0x50(%edx),%eax	; ERC32.l4 -> %eax
bb00000000	mov	0x0,%ebx	; 0x00000000 -> %ebx
01d8	add	%ebx,%eax	; %ebx ADD %eax -> %eax
a803	test	\$0x3,%al	; Check 32bit memory aligned
7416	je	0x16	; Jump to normal Jmpl op. below
c782dc03000007000000	movl	\$0x7,0x3dc(%edx)	; 0x00000007 -> ERC32.trap
8182cc03000003000000	addl	\$0x3,0x3cc(%edx)	; Add 3 to ERC32.icnt
c9	leave		; Return to EM on trap con.
c3	ret		
898284020000	mov	%eax,0x284(%edx)	; %eax -> ERC32.nPC
#-----			epilogue -----
80b284020000	mov	0x284(%edx),%eax	; Read ERC32.nPC
898280020000	mov	%eax,0x280(%edx)	; ERC32.nPC -> ERC32.PC
83c004	add	\$0x4,%eax	; Add 0x4 to %eax
898284020000	mov	%eax,0x284(%edx)	; %eax -> ERC32.nPC
8182cc03000005000000	addl	\$0x5,0x3cc(%edx)	; Add 5 to ERC32.icnt
c9	leave		; Return to EM on normal
c3	ret		

Fig. 6. Example translation block in the TC.

[LD instruction using helper function]			
...			
x86_emit_add_r32_r32(ebx, eax)			; %eax <- %ebx ADD %eax
...			
x86_emit_mov_imm32_ebx(&ERC32.erc32.regs[INST.inst_f3_i0.rd])			; %ebx <- r[rd] address
x86_emit_push_imm8(WORD_ACCESS)			; 3rd argument: WORD access
x86_emit_push_r32(ebx)			; 2nd argument: r[rd] address
x86_emit_push_r32(eax)			; 1st argument: read address
x86_emit_call_function(ERC32_mem_read)			; call helper function
x86_emit_mov_imm32_edx(&ERC32)			; %edx <- ERC32 register addr.
...			
[LD instruction without helper function for RAM access]			
...			
x86_emit_add_r32_r32(ebx, eax)			; %eax <- %ebx ADD %eax
...			
x86_emit_load_ram_addr_ecx			; %ecx <- %eax-RAM start addr.
x86_emit_cmp_imm32_ecx(0x005FFFFFFF)			; check %ecx is within RAM area (6MB)
x86_emit_ja_with_offset(18)			; If %ecx is out of RAM, then jump to call
x86_emit_and_imm32_eax(0x007FFFFFFC)			; %eax <- %eax AND 0x7FFFFFFC
x86_emit_mov_imm32_ebx(&ERC32->ram)			; %ebx <- addr of ERC32.RAM
x86_emit_load_mem_eax_with_ebx			; %eax <- [%ebx] (content)
x86_emit_mov_r32_m32(eax,edx,(4*INST.inst_f3_i0.rd))			; r[rd] <- %eax
x86_emit_jmp_with_offset(45)			; jump to next operation
...			
x86_emit_mov_imm32_ebx(&ERC32.erc32.regs[INST.inst_f3_i0.rd])			
			; %ebx <- r[rd] address
x86_emit_push_imm8(WORD_ACCESS)			; 3rd argument: WORD access
x86_emit_push_r32(ebx)			; 2nd argument: r[rd] address
x86_emit_push_r32(eax)			; 1st argument: read address
x86_emit_call_function(ERC32_mem_read)			; call helper function
x86_emit_mov_imm32_edx(&ERC32)			; %edx <- ERC32 register addr.
...			

Fig. 7. RAM direct access for LD instruction.

until the end of TB execution if the event time is the start of the TB. However, this delayed time can be negligible because the interrupt is not associated with any particular instruction, and there is an acceptable response latency before control is transferred to the interrupt handler.

Precise trap handling is the ability of the guest processor to identify which instruction caused a fault during execution of the TB. To resolve the trap mechanism, the PC/nPC of the guest where the trap occurred must be recognized by the EM. In the laysim-DBT, whenever it translates a trappable instruction, it updates the PC/nPC at the start of each instruction, which is illustrated at the start of the *jmpl* instruction in Fig. 6. If a trap occurs during execution of the TB, control is transferred to the EM after updating current cycle counts without executing remaining instructions in order to not change the state of the guest.

3.4.4. Enhance performance of the DBT engine

To improve the performance of the DBT engine, there are many optimization schemes,¹⁹⁾ such as TB chaining, super block, and code optimization based on dynamic profiling. We adopt two methods for enhancing the performance of the DBT engine: minimizing helper functions and maximizing TB chaining.

Minimizing the helper functions removes the overload of functions called from the TB, which is saving all of the registers to the stack, pushing the function arguments to the stack, incrementing the stack pointer, and restoring the registers from the stack. We applied this method to memory read/write instructions such as *ld*, *st* and *swap*, and integer condition code (icc) modification instructions such as *addcc*, *subcc* and *xnorcc*, and so on. After all of the helper functions of the laysim-DBT are minimized into only eight functions for the floating point unit (FPU) status check, FPU exception check, *wrpsr*, *restore*, *rett* and *save* instructions, and I/O read/write operations.

When the read/write instruction is decoded, the DBT engine adds the check logic for RAM address and directly accesses the RAM instead of calling the memory read/write

helper functions at the time of RAM access, as shown in Fig. 7. This scheme improves the performance of the DBT engine by 5.9–9.2% compared to the baseline, which does not apply any enhancement schemes.

The icc modification instructions update condition codes in the processor status register (PSR) according to execution results, and these condition codes are tested by conditional branch instructions such as *ble*, *bl* and *bgu*. The DBT engine uses x86 status flags for reflecting the SPARC icc, because the SPARC condition codes (i.e., carry (C), zero (Z), negative (N) and overflow (O)) are equivalent to the x86 EFLAGS (i.e., carry (CF), zero (ZF), sign (SF) and overflow (OF)). During translation of icc modification instructions, the DBT engine adds the setcc instructions (i.e., *setc*, *sets*, *sete* and *seto*) of the x86 for updating the SPARC condition codes. This method also helps to simplify the checking condition code logic of the conditional branch instructions. This scheme improves the performance of the DBT engine by 19.7–24.5% for IU operations and 18.6–24.1% for FPU operations with RAM direct access as compared to the baseline.

Another strategy for improving performance is to maximize TB chaining. The DBT engine continuously fetches new TBs by querying the TC or translating a BB from the guest. Although cache querying is less expensive than translating the same block over and over again, it is still slow. The TB chaining links TB directly to another TB instead of branching to the EM at the end of execution to further speed up performance. The EM checks to see if the successor BB was already translated after completion of execution. If it has not yet been translated, then the EM patches the epilogue to jump to the next TB body followed by the prologue to be allocated by the translator. If the successor BB was already translated, then the EM retrieves the TPC of the successor block from the map table and sets up a link to the successor block by overwriting the epilogue with a direct jump to the successor BB. The EM does not allow self-chaining to prevent the internal loop by checking self-flagging in the map table. This method also greatly improves the performance

about 29.6–43.9% for IU operations and 13.9–20.8% for FPU operations.

3.4.5. Self-modifying code

Self-modifying code means the modification of an original guest code that was already translated to the TB during translation by itself. This poses potential problems when the DBT is used, because the code actually being executed is a translated code, not the original guest code. When this happens, the TB in the TC no longer corresponds with the modified guest code. To resolve the self-modifying code problem, many DBT systems^{9,10,19)} set the original guest code region as write-protected by the runtime using system calls such as the *mprotect()* of Linux or *VirtualProtect()* of Windows. Any attempt to write a page in the protection area will be trapped and a signal is sent to the EM. Consequently, all of the TCs, or the TC corresponding to the modified page, will be flushed, and the EM will start retranslation.

But we have experienced the side effect of *VirtualProtect()/mprotect()* when using linear TC; it is possible to change the protection attributes per page when the minimum size is 4 KB in the x86 architecture. Our guest, RTOS, is VxWorks 5.4, where the code and data of the kernel are mixed in the task control block (TCB) area. Any change of data in the TCB by VxWorks is misleading, invoking retranslation as recognition of the self-modifying code. This problem is overcome by using the shadow memory, which reflects corresponding memory that has been translated without using system call. During translation, instruction fetch sets the corresponding address of the shadow memory. If a RAM write instruction is decoded, the DBT engine adds the check logic to determine whether or not write address has already been translated using the shadow memory. If a self-modifying case is detected, then the EM will initialize the TC, map table and shadow memory, and start retranslation. Although this mechanism can solve the problem of self-modifying code regardless of the host operating system, it leads to the degression of performance. In the laysim-DBT, it shows less than -2.72% loss of performance for IU operations and -1.165% for FPU operations. However, this situation can be disregarded considering the performance enhancement discussed in Section 3.4.4.

4. Experimental Result of laysim-DBT

In this section, we evaluate the performance of the laysim-DBT and check the cycle accuracy using various embedded benchmarks. We choose Dhrystone v2.1, Whetstone, Stanford, EEMBC CoreMark and MiBench. All benchmarks were compiled for ERC32 using LECCS 1.2.2²⁰⁾ based on GCC 3.2.3 and for LEON2 using BCC 1.0.43²¹⁾ based on GCC 4.4.2. All measurements were performed using an Intel Core i7 CPU 870, processing speed of 2.93 GHz, quad-core with hyper-threading enabled on Windows 7 Enterprise x86 64 bit computer except for QEMU laysim-erc32, which was done on a Fedora 14 Linux system.

First, we measured the performance of real processors and compared the execution times of TSIM-ERC32, laysim-

Fig. 8. Performance comparison with real processors.

Fig. 9. Improved integer performance tracing of the laysim-DBT.

erc32, QEMU laysim-erc32 and laysim-DBT (baseline) using Dhrystone, as shown in Fig. 8. Only QEMU laysim-erc32 and laysim-DBT (baseline) were able to meet the real-time performance of the LEON2-FT processor. QEMU laysim-erc32 was 1.64 times faster than the laysim-DBT (baseline), which did not have any enhancement schemes.

After that, we measured the execution time of the laysim-DBT by applying the performance enhancement techniques and self-modifying code detection. Figure 9 shows the trace of Dhrystone integer performance compiled with optimization level 0–3. After applying RAM direct access, performance improves about 4.8–9.2%. Removing the helper functions of the icc instructions increases speed about 19.8–20.7% compared to the former result. The TB chaining boosts performance substantially, about 29.6–43.9%. The detection of self-modifying code slows it down slightly, less than -2.72% . After all, the final performance of the laysim-DBT is 11.23 times faster than TSIM-ERC32 and 1.43 times faster than QEMU laysim-erc32 on Dhrystone.

The comparison of floating-point performance using Whetstone benchmarking, as shown in Fig. 10, indicates a performance enhancement of over 39.9% compared to the baseline. The final performance of the laysim-DBT shows that it is 6.54 times faster than TSIM-ERC32 and 1.71 times faster than QEMU laysim-erc32 on Whetstone (double-precision). Unlike the results for integer performance, the baseline of laysim-DBT performance is higher than that of QEMU laysim-erc32. This is because QEMU uses many helper functions for floating-point operation from the math

library, whereas laysim-DBT does not use a math library and generates x87 instructions using the DBT engine for floating-point instructions.

Figure 11 shows a comparison of relative performance for all benchmarks as a reference using TSIM-ERC32. The laysim-DBT consistently outperforms QEMU laysim-erc32, although it provides cycle accurate emulation by cycle counting. Finally, on average the laysim-DBT is 8.62 times faster than TSIM-ERC32 and 1.56 times faster than QEMU laysim-erc32 for all cases.

Next, we compare the total cycles computed using a laysim-DBT-adopted static cycle-approximation model to that of the TSIM-ERC32, which is a cycle-true emulator of the ERC32 processor, and that of the laysim-erc32, which is our cycle-true emulator, whereas the QEMU laysim-erc32 does not support instruction cycles. The cycle accuracy of the laysim-DBT is 95.49% with a $\pm 8\%$ error margin as shown in Table 1.

The cycle accuracy of the laysim-DBT satisfies the time attribution of our FSW. The deviation in cycle accuracy is due to the static cycle-approximate model, although a more detailed cycle counting method such as pipeline lock-up, annul or register dependency may improve the cycle accuracy. However, this would lead to laysim-DBT performance loss.

We also evaluated the execution of RTOSs such as

VxWorks 5.4, RTEMS and eCOS. All RTOSs worked correctly under the laysim-DBT, and their self-modifying operation, which is common in a RTOS, was properly treated without causing any additional loss in DBT performance.

5. Performance of KOMPSAT-FSS with laysim-DBT

Integrating the laysim-DBT with our KOMPSAT-FSS, as shown in Fig. 12, can be done easily as compared to the laysim-erc32 owing to using the same external interfaces. KOMPSAT-FSS simulates all features of the OBC, I/O modules, power control and distribution unit, sensors and actuators, and payloads. It can directly load and run the FSW without any modification, and process uplinked command and downlink telemetries as low-rate, high-rate or playback mode through the virtual RF interfaces. It also provides an error injection function, variable trace function, and thermal control functions, which are difficult to test on real hardware.

For FSW development and testing, it provides a virtual network connected to Tornado IDE, which is the framework of FSW development. If target server of the Tornado is connected to KOMPSAT-FSS, the FSW can be downloaded dynamically, debugged, executed, and monitored and checked

Table 1. Cycle accuracy comparison.

Benchmarks	Total cycles and cycle accuracy		
	TSIM-ERC32	laysim-erc32	laysim-DBT
Dhrystone	633,639,362	633,639,223	604,838,274 (99.99%) (95.45%)
Whetstone SP	2,092,189,460	2,064,362,894	2,261,798,805 (98.67%) (108.11%)
Whetstone DP	2,443,432,999	2,450,234,938	2,626,116,296 (100.28%) (107.48%)
Stanford	37,118,286	36,898,680	35,504,199 (99.41%) (95.65%)
CoreMark	286,125,763	286,125,297	276,845,452 (99.99%) (96.76%)
MiBench.qsort	103,742,892	94,299,659	104,063,427 (90.89%) (100.31%)
MiBench.dijkstra	399,444,134	399,442,953	381,148,771 (99.99%) (95.42%)
MiBench.basicmath	292,667,095	320,161,268	281,889,706 (109.39%) (96.32%)

Fig. 10. Improved floating-point performance tracing.

Fig. 11. Overall relative performance comparison.

Fig. 12. Overall KOMPSAT flight software simulator.

Table 2. Performance at simulator level.

	Processor emulator of KOMPSAT-FSS with		
	TSIM-ERC32	laysim-erc32	laysim-DBT
Telemetry downlink rate	2 TLM/s	2 TLM/s	28 TLM/s

for time-based FSW behavior. All of the test activities for the FSW, such as unit testing, integration testing and verification testing can be performed under KOMPSAT-FSS, because all of these development environments are identical to those of real hardware thanks to the processor emulator.

The performance of KOMPSAT-FSS can determine the rate of downlink telemetry, which is only transmitted every second in real hardware after the end of FSW operation. Table 2 shows a comparison of performance at the simulator level. KOMPSAT-FSS with TSIM-ERC32 and laysim-erc32, which use an interpretation method, can transmit two telemetries per every second, which means they can accelerate the speed of simulation twice at the most. On the other hand, KOMPSAT-FSS with laysim-DBT can downlink 28 telemetries per second, which is 14 times faster than the previous results.

This increase in simulator speed is highly beneficial for software engineers when testing and verifying the FSW, and ground operations to schedule missions and predict the long-term results.

6. Conclusion

This paper presents the design and development of a high-performance and cycle-accurate space processor emulator for satellite simulators, and suggests a solution for the self-modifying problem. High performance is achieved using dynamic binary translation with enhancement schemes by minimizing helper functions and maximizing translation block chaining without self-chaining. For the cycle accuracy, the

laysim-DBT updates the number of elapsed cycles of the basic block using a static cycle-approximation model. It considers the difference in cycle counts during branch hit/miss and trapping, and supports precise handling of trapping and fast interruption. The use of shadow memory solves the problem of self-modification with acceptable execution latency, regardless of the host operating system.

A performance evaluation shows that the laysim-DBT meets the real-time performance of a LEON2-FT processor, and that is consistently outperforms the QEMU laysim-erc32. Especially, KOMPSAT-FSS integrated with the laysim-DBT can accelerate simulation speed to 14 times that of the previous simulators. The cycle accuracy of laysim-DBT is 95.49% with a $\pm 8\%$ error margin, which satisfies the time attribution of our flight software.

Currently, KOMPSAT-FSS packaged with the laysim-DBT is being used for all lifecycles of flight software development, maintenance of flight software after launch, and ground operations.

Byeong-Gyu Nam is a professor in the Department of Computer Science and Engineering of the Chungnam National University, Korea. He can be contacted at: bgnam@cnu.ac.kr

References

- 1) Atmel: TSC695F SPARC 32-bit Space Processor User Manual, 2003.
- 2) Atmel: Rad-Hard 32 bit SPARC v8 Processor User Manual, 2011.
- 3) Aeroflex Gaisler: LEON3-FT Fault-Tolerant Processor, <http://www.gaisler.com/>
- 4) Choi, J. W., Shin, H. K., Lee, J. S., and Cheon, Y. J.: Virtualized System Development Based on ERC32 Processor for Satellite Simulator, *J. Korea Soc. Space Technol.*, **6**, 1 (2011), pp. 50–56.
- 5) Aeroflex Gaisler: TSIM Simulator User's Manual Version 1.3.8, 2006.
- 6) Choi, J. W. and Nam, B. G.: Development of High Performance Space Processor Emulator Based on QEMU—Open Source Dynamic Translator, Control, Automation and Systems (ICCAS), 12th International Conference on IEEE, 2012, pp. 300–304.
- 7) The Embedded Microprocessor Benchmark Consortium: CoreMark,

- <http://www.eembc.org/>
- 8) Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite, Workload Characterization, WWC-4, IEEE International Workshop, 2001, pp. 3–14.
 - 9) Fabrice Bellard: QEMU, a Fast and Portable Dynamic Translator, USENIX Annual Technical Conference, 2005, pp. 41–46.
 - 10) Yindong, Y., Haibing, G., Erzhou, Z., Hongbo, Y., and Bo, L.: CrossBit: A Multi-Sources and Multi-Targets DBT, The First International Conference on Cloud Computing, GRIDs, and Virtualization, 2011, pp. 41–47.
 - 11) Topham, N. and Jones, D.: High Speed CPU Simulation using JIT Binary Translation, Proceedings of the Annual Workshop on Modelling Benchmarking and Simulation (MOBS), Vol. 7, 2007.
 - 12) Hanbic, S. and Gaisler, J.: AT697F/LEON2-FT Processor Software Tools, Proceedings of Data System in Aerospace (DASIA), 2007.
 - 13) Marques, P., Feiteirinha, J., Pureza, L., and Lindman, N.: LeonVM: Using Dynamic Translation for Developing High-Speed Space Processor Emulators, Proceedings of the 9th International Workshop on Simulation for European Space Programmes (SESP), 2006.
 - 14) Alastair, P., Paul, R., and Sean, M.: QERx—A High Performance Emulator for Software Validation and Simulations, Proceedings of Data System in Aerospace (DASIA), 2009.
 - 15) Bruno, C., Alastair, P., and Palu, R.: QERx—a Faster than Real-Time Emulator for Space Processors, Proceedings of Data System in Aerospace (DASIA), 2012.
 - 16) Matteo, B., Cyrille, C., and Tristan, G.: Couverture—an Innovative Open Framework for Coverage Analysis of Safety Critical Applications, 14th International Conference on Reliable Software Technologies, Ada Europe, 2009.
 - 17) Choi, J. W., Lee, J. S., Yang, S. E., and Cheon, Y. J.: Development of Software-based Spacecraft Simulator for LEO Flight Software Development & Verification, Proceedings of Asia-Pacific International Symposium on Aerospace Technology, Vol. 1, 2010, pp. 489–493.
 - 18) Souza, M., Nicacio, D., and Araujo, G.: ISAMAP—Instruction Mapping Driven by Dynamic Binary Translation, 3rd Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT), 2010.
 - 19) Smith, J. E. and Ravi, N.: *Virtual Machines—Versatile Platforms for Systems and Processes*, Elsevier, 2005.
 - 20) Aeroflex Gaisler: LECCS User's Manual Version 1.2.2, 2003.
 - 21) Aeroflex Gaisler: BCC—Bare-C Cross-Compiler User's Manual Version 1.0.43, 2013.

K. Yamada
Associate Editor