Supporting the Neon and VFP Instruction Sets in an LLVM-based Binary Translator

Yu-Chuan Guo, Wuu Yang, Jiunn-Yeu Chen
Computer Science Department
National Chiao-Tung University
Hsinchu, Taiwan, R.O.C.
Emails: qoo12345654321@gmail.com, {wuuyang, jiunnyeu}@cs.nctu.edu.tw

Jenq-Kuen Lee
Computer Science Department
National Tsing-Hua University
Hsinchu, Taiwan, R.O.C.
Email: jklee@cs.nthu.edu.tw

Abstract—**Binary translation attempts to emulate one instruction set with another on the same or different platforms. This important technique is widely used in instruction-set-architecture migration, binary instrucmentation, dynamic optimizations, software security, and fast arhitecture simulation. Vector and floating-point instructions are widely used in many applications, including multimedia, graphics, and gaming. Though these instructions are usually simulated with software in a binary translator, it is important to support them in such a way that the host SIMD (single instruction multiple data) and floating-point hardware is efficiently used in the translation process. We report our design and implementation of the emulation of ARM Neon and VFP (vector floating point emulation) instructions in the MC2LLVM (machine-to-low-level-virtual-machine) binary translator. Our approach can take full advantage of the vector and floating-point functional units, if present, of the host machine. The experimental results show that code generated by MC2LLVM with the Neon and VFP extensions achieves an average speedup of 1.174x in SPEC 2006 benchmark suites compared to code generated by MC2LLVM without the Neon and VFP extensions.**

Keywords–binary translation; cloud computing; LLVM; floating-point instruction; Neon, VFP; vector instruction; virtualization.

I. Introduction

Binary translation [14] attempts to emulate one instruction set with another on the same or different platforms. The important technique is widely used in instruction-set-architecture migration [4][5][13][18], binary instrucmentation [6], dynamic optimizations [2][11], software security, and fast arhitecture simulation [15], [17]. The Neon and VFP coprocessors [1] are extensions to the ARM architecture. They are designed for applications with SIMD and floating-point instructions to meet the growing demand of computing power in embedded systems, such as multimedia, 2D/3D graphics, and gaming.

Existing binary translators, such as QEMU (quick emulator), support vector and floating-point instructions with naive software simulation (using scalar, integer, and shift operations) [3]. The result is poor performance. In this paper, we attempt to make use of the vector and floating-point hardware on the host platform, if present, to execute the Neon and VFP instructions. Furthermore, all the related exceptions and flags in the floating-point environment are taken care of properly. Our system is built on top of MC2LLVM, a retargetable static/dynamic/hybrid binary translator developed in our lab in the past few years. The resulting performance is the key issue in our design.

We leverage the existing LLVM backend (low-level virtual machine) to build a high-performance and retargetable binary translator. In this research, the emulation of Neon and VFP architectures is layered on the top of an LLVM backend.

To be fully compliant with Neon and VFP instruction set architectures, we need to know the details of the machine features [1] in the Neon and VFP architectures, including the flush-to-zero mode, the default NaN (Not a Number) mode, and the floating-point exceptions. These details can affect the behaviors of the Neon and VFP instructions. Our implementation must faithfully mimic these features. We also propose new methods to detect floating-point exceptions if the IR layer of a binary translator does not provide the relevant information.

To achieve a high degree of reliability, we also developed a verification framework for testing all the emulated instructions in Neon and VFP in order to gain confidence in the correctness of our approaches. Verification is performed automatically.

The remainder of this paper is organized as follows: Section 2 lists the related work and background knowledge, including Neon and VFP architectures, and describes the related terminology. In Section 3, we describe the implementation details. Section 4 illustrates the verification of the implementation. Section 5 discusses the experimental results. Section 6 concludes this paper.

II. Background and Related Works

In this research, we attempt to translate the Neon and VFP instructions into LLVM IR. In this chapter, we will introduce background knowledge and related works about the Neon and VFP architectures, including their machine features related to binary translation.

MC2LLVM (Machine Code To LLVM) is a process-level binary translation system based on LLVM developed in our lab in the past few years. It adopts the approach of the modern compiler techniques which separate the translation process into a frontend and a backend. The frontend translates the guest binary into LLVM IR and then uses the existing LLVM backend to generate the host binary from LLVM IR.

Figure 1 shows the flow of dynamic binary translation in MC2LLVM. The emulation manager maintains and manipulates the progress of the emulation, such as handling the control transfer between translated basic blocks in code cache and invoking the translator to translate a target basic block that has not yet been translated. In this research, we will focus on the translator module, which is also employed in the static and hybrid modes.

QEMU [3] is an open source binary translation system which supports full system emulation and, unlike MC2LLVM, always runs in the DBT mode. QEMU has its own IR, known as Tiny Code Generator (TCG) [3], to implement a two-stage

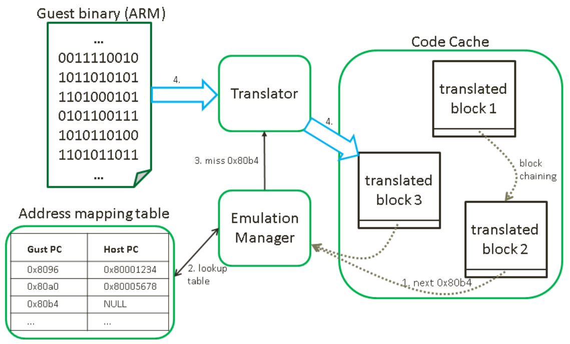


Figure 1. Structure of MC2LLVM.

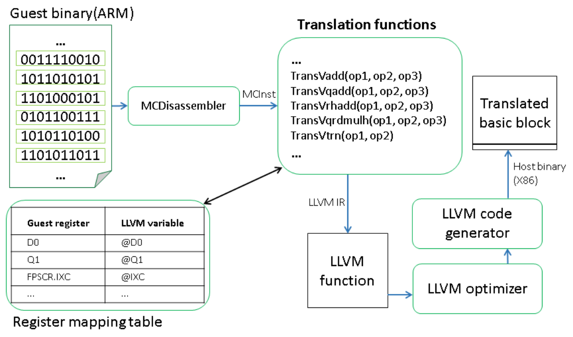


Figure 2. Translation from guest binary to host binary.

translation. It is able to emulate several ISAs, such as x86, PowerPC, ARM, and Sparc, etc.

Neon is a SIMD (Single Instruction Multiple Data) processor integrated into the ARM chip. Neon provides a 64/128-bit SIMD instruction set that provides acceleration for multimedia and signal processing applications, such as compressed video decoding, image processing, 2D/3D graphics, sound synthesis, etc.

VFP is a vector floating-point processor integrated as a part of the ARM chip. VFP provides a single-precision and double-precision floating-point instruction set that is fully compliant with IEEE 754 [9]. (Neon is not fully compliant with IEEE 754.)

Neon and VFP share the same extension register bank, which is distinct from the ARM core register bank.

The Neon and VFP extensions have a shared register space for system registers. Only the system register known as Floating-Point Status and Control Register (FPSCR) in this space is accessible in the application programs. FPSCR contains the arithmetic status flags as well as the bit fields for controlling the floating-point unit. Table I shows the bit fields of the FPSCR register.

The DN bit controls the default NaN (not-a-number) mode, which affects the behavior of the floating-point operations involving one or more NaNs. The NaN processing follows the IEEE 754 standard if the DN bit is 0. A floating-point operation involving one or more NaNs returns the default NaN if the DN bit is 1.

The FZ bit controls the flush-to-zero mode, which affects the behavior of the floating-point operations. If the FZ bit is 0, the behavior of a floating-point operation in VFP follows the IEEE 754 standard. If the FZ bit is 1, the flush-to-zero mode is enabled. In the flush-to-zero mode, denormalized numbers (in the IEEE 754 standard [9], denormalized or denormal numbers are very small numbers whose exponent fields are 0. They are used to fill the gap between zero and the minimum normalized number) will be flushed 0. The flush-to-zero mode also changes the criteria for the floating-point exceptions to occur (described in a later section). Neon always uses the flush-to-zero mode, regardless of the value of the FZ bit.

IDC, IXC, UFC, OFC, DZC, and IOC are input denormal, inexact, underflow, overflow, division by zero, and invalid operation cumulative exception flags, respectively. These flags show abnormalities during floating-point operations. A *cumulative* exception bit is set to 1 when the corresponding floating-point exception occurs. However, it is not reset to 0 when the corresponding exception does not occur automatically. These flags are usually used in applications with high safety requirements.

III. Design and Implementation

The implementation of the emulation of the Neon and VFP extensions, including the Neon and VFP registers, instructions, and machine features, is discussed in detail in this section.

Figure 2 shows the translation flow of MC2LLVM. The translator uses LLVM MCDisassembler [10] to disassemble binary instructions into MCInst (the IR of MCDisassembler). MCInst is translated by the functions we provided for each guest instruction into LLVM IR. The LLVM IR is organized as LLVM functions. The LLVM optimizer performs target-independent optimizations before code generation.

A. *Flush-to-Zero Mode Emulation*

The flush-to-zero mode [1] is a special processing mode that replaces denormalized operands, intermediate results, and final results with zero while reserving the sign bit. It is used to avoid handling denormalized numbers, thus saving execution time. Because supports for denormalized numbers increase the complexity in hardware design, floating-point units often save hardware cost by simply delegating supports for denormalized numbers to software.

The flush-to-zero mode is not compliant with IEEE 754. It also changes the behavior of a floating-point operation and the criteria for the floating-point exceptions in three ways: (1) In the flush-to-zero mode, a floating-point operation can cause the input denormal exception, which is not included in IEEE 754. (2) The inexact exception would not be raised when a result is rounded to zero or flushed to zero. (3) If the result of an operation is rounded to zero or flushed to zero, the underflow exception would be raised.

There are two operations in the flush-to-zero mode: flush-input-to-zero and flush-output-to-zero. Each floating-point operation must go through the preprocessing in LLVM IR in which denormalized numbers are detected and replaced with zero before a floating-point operation is performed. This is the flush-input-to-zero operation, shown in Figure 3.

According to the definition of the flush-to-zero mode, if the intermediate result of a floating-point operation that is

TABLE I. Floating-Point Status and Control Register format

bit(s)	31	30	29	28	27	26	25	24	
meaning	N	Z	C	V	QC	AHP	DN	FZ	
bit(s)	23:22	21:8	7	6:5	4	3	2	1	0
meaning	RMode	reserved	IDC	reserved	IXC	UFC	OFC	DZC	IOC

```
1.  procedure FP32_FlushInputToZero
(operand)
2.     if (IsDenormal(operand))
3.         SetIDC(); // input
denormal exception
4.         flush operand;
5.
6.  function IsDenormal(op)
7.     if (abs(op) < 0x00800000 &&
abs(op) > 0)
8.         return TRUE;
9.     return FALSE;
```

Figure 3. Algorithm for flush-input-to-zero.

```
1.  procedure FP32Mul_FlushOutputToZero(op1,
op2, result)
2.     if (isDenormal(result))
3.         SetUFC(); // underflow exception
4.         flush result;
5.     else if (isZero(result) &&
!isZero(op1) && !isZero(op2))
6.         SetUFC(); // underflow exception
7.     else if (isMinNorm(result))
8.         multiply op1's and op2s
fractions with leading 1 to A;
9.         if( A has 23 consecutive 1s
starting at the most significant 1)
10.            SetUFC();
11.            flush result;
```

Figure 4. Algorithm for flush-output-to-zero for single-precision floating-point multiplcation.

produced *before* rounding satisfies the following condition:

$$(1) \quad 0 < abs(result) < +MinNorm$$

where $MinNorm$ is the minimum normalized number of the destination precision, that intermediateg result is flushed to zero before the rest of the floating-point operation. However, an LLVM floating-point operation can only produce a result *after* rounding. The intermediate result produced after the floating-point operation but before rounding is invisible. Therefore, the flush-input-to-zero operation may produce an incorrect result if the intermediate result has already been rounded to $MinNorm$ or to zero. Hence the flush-output-to-zero operation, shown in Figure 4, is introduced to flush the intermediate and final results of a single-precision multiplication to zero.

If both operands of a floating-point multiplication are not zero but the result is zero, the result must have been rounded to zero. Obviously, the multiplication causes an underflow exception. Only when the intermediate result that is produced before rounding satisfies the following condition:

$$(2) \quad +MaxDenorm < abs(result) < +MinNorm$$

where $MaxDenorm$ is the maximum denormalized number (that is, the exponent field is all 0s and the fraction field is all 1s) of the destination precision, the result could be rounded to $MaxDenorm$ or $MinNorm$. In this case the after-rounding result may be a normalized number but it should be flushed to zero. We can reproduce the before-rounding intermediate result by (integer-)multiplying the two operands' fractions. If the intermediate result has 23 (if single precision) consecutive 1s starting from the most significant 1, that is, condition (2) above, the result should be flushed to zero and an underflow exception should be raised.

Different floating-point instructions come with different emulations of the flush-output-to-zero operation. For example, for floating-point additions, it is not necessary to consider the two special cases (rounding to zero or to $MinNorm$) because a floating-point addition is always exact (i.e., without loss of precision) when the result is a denormalized number.

B. Floating-Point Exception Emulation

Floating-point exceptions are emulated with two methods. The first method is by checking the exception flags in the underlying hardware (based on C++11 Standard Library) and the second method is by employing additional test code. The second method is complete and more efficient than the first. We will explain both methods in this section.

Although LLVM supports many floating-point operations, it does not provide any information related to floating-point exceptions. The developers of LLVM may consider that accessing the floating-point environment is unlikely to happen and supporting them would diminish the performance because the implicit data dependencies that might occur in the floating-point environment. For example, setting floating-point exception flags or trapping the floating-point exceptions for further processing may unnecessarily restrict the LLVM optimizer and make the LLVM optimizer more complex. So we have to detect the floating-point exceptions with additional code.

The first method for detecting floating-point exceptions is to check the exception flags in the underlying hardware with the `fetestexcept` function in the C++11 Standard Library. Because the exception flags in the underlying hardware may be changed inadvertently by the emulation manager, translator, or other modules in MC2LLVM or LLVM during execution, it is necessary to clear the exception flags with the `feclearexcept` function before executing every floating-point operation in order to avoid these outside disturbances.

The first method comes with several drawbacks. First, MC2LLVM adopting this method becomes much slower than QEMU. We used a benchmark that repeats single-precision addition one billion times. The resulting execution time is shown in Figure 5. Furthermore, we wrote two new functions in x86

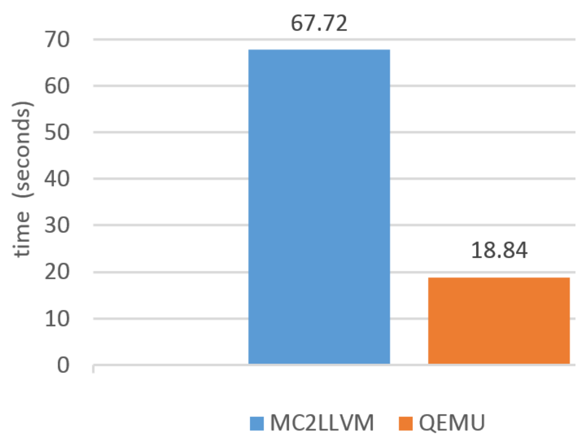


Figure 5. Execution time of single-precision floating-point addition. MC2LLVM adopts the first method for emulating floating-point exceptions.

assembly to replace `fetestexcept` and `feclearexcept` to make emulation of floating-point operation even faster; they are still a little (about 0.5 seconds) slower than QEMU evaluated by the benchmark. Second, the LLVM optimizer may reorder the instructions, which causes unexpected results.

The second method is to use additional code in LLVM IR to detect floating-point exceptions based on the operands and results of a floating-point operation. Normally, floating-point exceptions are detected at various instants during the execution of a floating-point operation. For example, the occurrence of the underflow and inexact exceptions is determined based on the before-rounding intermediate values of a floating-point operation. However, we cannot obtain any intermediate values from an LLVM floating-point operation. We could reproduce the intermediate values with software but this would incur high overhead. Alternatively, we use some tricks which we will discuss in details later for better efficiency, for example, employing other floating-point operations to test the occurrence of floating-point exceptions and bypassing useless detection of floating-point exceptions. We will use floating-point additions and multiplications to illustrate this approach. Other floating-point operations (`vcvt`, `vdiv`, `vcge`, `vcmpe`, etc.) are handled similarly. MC2LLVM adopting this second method is twice as fast as QEMU according to our benchmarks (for floating-point additions) mentioned earlier. Therefore, we decide to use this second method in our binary translator. In what follows we will discuss the emulation of each floating-point exceptions.

- Emulation of the Invalid Operation Exceptions
- Emulation of the Division by Zero Exceptions
- Emulation of the Overflow Exceptions
- Emulation of the Input Denormal Exceptions
- Emulation of the Inexact Exceptions
- Emulation of the Underflow Exceptions

Figure 6 summaries the emulation of a floating-point operation. Note that not all floating-point operations will go through all the steps shown in the figure. The emulation of floating-point operations involves many details and is error-prone for implementation. Therefore, a good verification technique is required for every implementation.

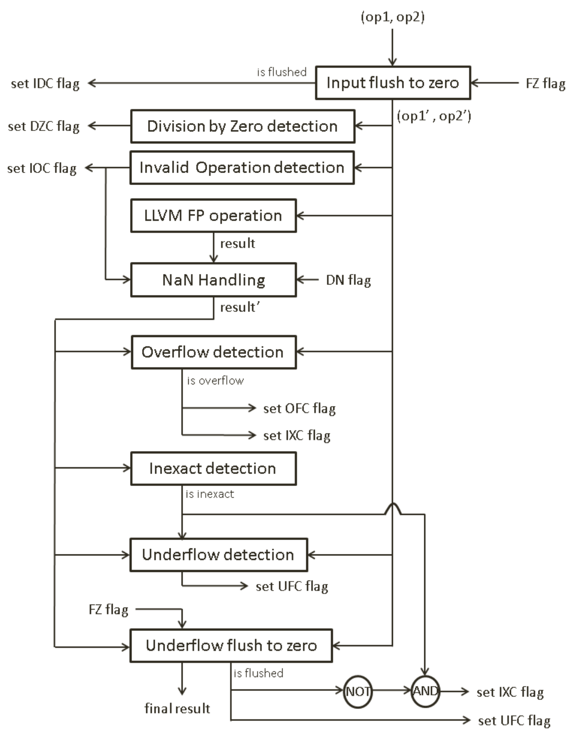


Figure 6. Emulation of a floating-point operation.

IV. Verfication of the Bianry Translator

Testing is a crucial building block in order to achieve a high degree of reliability. It is difficult to identify the mistranslated instructions generated by a binary translator because there are so many translation functions in a binary translator and, due to the complex interdependencies inherent in a floating-point operation, it is almost impossible to determine whether an instruction is functionally equivalent to its translated instructions directly.

Mistranslated instructions may produce wrong values, or fail to raise status flags such as the cumulative exception flags (`IXC`, `UFC`, etc.) and the cumulative saturation flag in FPSCR, in different combinations of modes, such as the flush-to-zero mode and the default NaN mode. Running a few benchmarks correctly is far from being correct for a binary translator because of low instruction coverage (an application usually makes use of less than 5% of the 1240 instructions in Neon and VFP), neglect of floating-point exceptions, and no mode switching in applications. Therefore, we developed a black-box testing framework for a binary translator, shown in Figure 7.

V. Experiments

In order to show the performance of our translation system, we compare the execution time of the guest binary with and without Neon and VFP instructions on MC2LLVM and QEMU, respectively. Since MC2LLVM can run in three different modes, we always use the pure dynamic translation mode in this experiments.

A. Environment

The experimental hardware is equipped with the Intel Core i7-4770 and 8GB memory, running 32-bit Ubuntu ver-

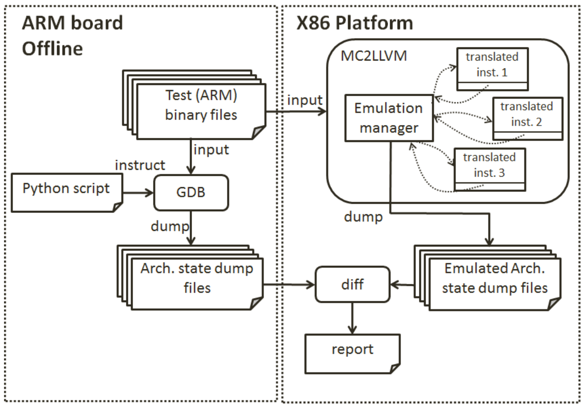
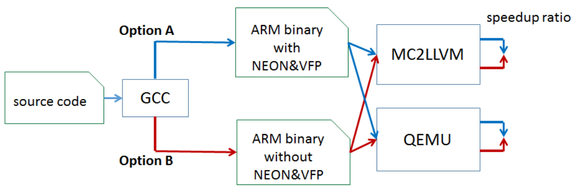
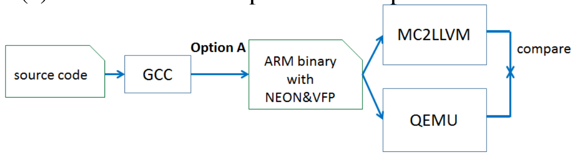


Figure 7. A verification framework.



(a) Performance comparison for options A and B.



(b) Performance comparison for MC2LLVM and QEMU.

Figure 8. Performance comparison.

sion 14.04. We use SPEC CPU2006 and LINPACK as our benchmarks. All of the benchmarks were compiled into ARM statically linked binaries with GNU GCC version 4.4.6 and linked with uClibc library 0.9.30.2. The version of QEMU we used is 2.1.50. MC2LLVM is layered on top of LLVM version 3.2.

We compiled SPEC CPU2006 with two configurations: Option A (-O3 -march=armv5 -mfloat-abi=softfp -mfpu=neon -ftree-vectorize) to tell the compiler to try to generate Neon and VFP instructions, and Option B (-O3 -march=armv5 -mfloat-abi=soft) to tell the compiler not to generate Neon and VFP instructions. The LINPACK benchmark was compiled with -O3 -march=armv5 -mfloat-abi=softfp -mfpu=vfpv3.

Among 12 integer benchmarks (CINT2006) in SPEC CPU2006, perlbench cannot be compiled and both MC2LLVM and QEMU failed to run gcc, a benchmark in CINT2006. Among 17 floating-point benchmarks (CFP2006) in SPEC CPU2006, both MC2LLVM and QEMU failed to run gromacs, cactusADM, and sphinx3, and only MC2LLVM failed to run tonto and games (some bugs happened when emulating them even with Option B, without Neon and VFP support).

B. Performance

Figure 8 (a) and (b) show our comparative approaches. Figure 9 shows the execution time ratio of SPEC CINT2006 compiled with Option A and Option B running on MC2LLVM

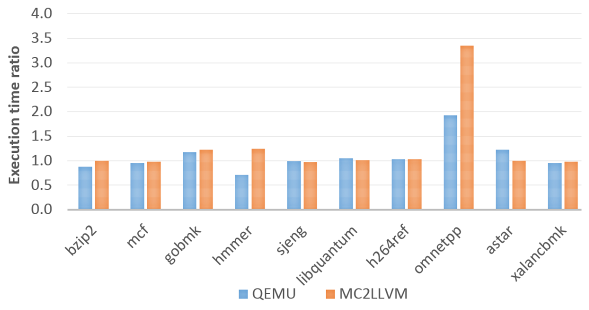


Figure 9. Execution time ratio of Option B/Option A on translating SPEC CINT2006.

and QEMU, respectively. The geometric means of the execution time ratio for MC2LLVM and QEMU are 1.174 and 1.052 respectively. The results imply that MC2LLVM could process the Neon and VFP instructions more effectively than QEMU on average.

VI. Conclusion

We have finished the Neon and VFP extensions in our binary translator MC2LLVM:

1) 1240 translations of the Neon and VFPv3 instructions are emulated.
2) Emulation of the machine features of Neon and VFPv3 architectures is included.

We enhance the translation capability and increase opportunities of using host SIMD and floating-point units to improve performance in MC2LLVM. We also propose new methods by diagnosing the input and output of a floating-point operation to detect floating-point exceptions if the IR layer of a binary translation system do not provide the relevant information about their occurrences instead of emulating a floating-point operation in software, which takes more processor time. We also developed a verification framework for testing the emulated instructions in Neon and VFP to gain confidence in the correctness of our approaches. The experiment results indicate that MC2LLVM is, in average, 1.24X and 2.27X faster than QEMU on the SPEC CPU2006 integer benchmarks and floating-point benchmarks, respectively, and have 3.36X more throughput of the floating-point operations than QEMU benchmarked by LINPACK.

Acknowledgment

The work reported in this paper is partially supported by National Science Council (NSC), Taiwan, Republic of China, under grants MOST 103-2221-E-009-105-MY3, NSC 100-2218-E-009-009-MY3, and NSC 100-2218-E-009-010-MY3.

References

[1] ARM Limited, ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition Errata Markup, ARM DDI 0406B, 2011.

[2] V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo: a Transparent Dynamic Optimization System,” ACM SIGPLAN Notices, 35, 5, pp. 1-12, 2000.

[3] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” In *Proc. 2005 USENIX Annual Technical Conf.*, pp. 41-46, 2005.

[4] J.-Y. Chen, W. Yang, T.-H. Hung, H.-M. Su, and W.-C. Hsu, “A Static Binary Translator for Efficient Migration of ARM based Applications,” In Proc. 6th Workshop on Optimizations for DSP and Embedded Systems, 2008.

[5] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S.B. Yadavalli, and J. Yates, “FX!32 - A Profile-Directed Binary Translator,” IEEE Micro 18, 2, pp. 56-64, 1998.

[6] K. Hazelwood, G. Lueck, and R. Cohn, “Scalable Support for Multithreaded Applications on Dynamic Binary Instrumentation Systems, In *Proc 2009 International Symp. Memory management* (Dublin, June 19-20, 2009), 2009.

[7] J.L. Henning, “SPEC CPU2006 Benchmark Descriptions,” ACM SIGARCH Computer Architecture News, 34, 4, pp. 1-17, 2006.

[8] C.A. Lattner, LLVM: An Infrastructure for Multi-Stage Optimization, Master’s Thesis, Comp. Sci. Dept, Univ. Illinois at Urbana-Champaign, 2002.

[9] W. Kahan, IEEE Standard 754 for Binary Floating-Point Arithmetic, Lecture Notes on the Status of IEEE 754, pp. 94720-1776, 1996.

[10] MCDisassembler, http://llvm.org/docs/doxygen/html/classllvm_1_1MCDisassembler.html.

[11] R.W. Moore, J.A. Baiocchi, B.R. Childers, J.W. Davidson, and J.D. Hiser, “Addressing the Challenges of DBT for the ARM Architecture,” In Proc. 2009 ACM SIGPLAN/SIGBED Conf Languages, Compilers, and Tools for Embedded Systems (LCTES 09), pp. 147-156, 2009.

[12] B.Y. Shen, J.Y. You, W. Yang, and W.-C. Hsu, “An LLVM-based Hybrid Binary Translation System,” In *Proc. 7th IEEE International Symp. Indurstrial Embedded System* (SIES 12, Karlsruhe, Germany, June 20-22), 2012.

[13] B.-Y. Shen, J.-Y. Chen, W.-C. Hsu, and W. Yang. 2012. “LLBT: an LLVM-based Static Binary Translator,” In Proc. 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES’ 12), pp. 5160, October 2012.

[14] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson, “Binary Translation,” Communications of the ACM, 36, 2, pp. 6981, February 1993.

[15] J.E. Smith and R. Nair. Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann, June 2005.

[16] Texas Instruments. Pandaboard. OMAP4430 SoC dev. board, revision A2, 2012.

[17] VMware Inc, VMware Workstation, 2013.

[18] C. Zheng and C. Thompson, “PA-RISC to IA-64: Transparent Execution, No Recompilation,” Computer 33, 3, pp. 47-52, March 2000.