HERMES: A Fast Cross-ISA Binary Translator with Post-Optimization

Xiaochun Zhang[*‡] zhangxiaochun@ict.ac.cn
Qi Guo[†] qguo1@andrew.cmu.edu
Yunji Chen[*§] cyj@ict.ac.cn
Tianshi Chen[*] chentianshi@ict.ac.cn
Weiwu Hu[*‡] hww@ict.ac.cn

[*] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[‡] Loongson Technology Corporation Limited, Beijing, China
[†] Carnegie Mellon University, Pittsburgh, PA, USA
[§] Center for Excellence in Brain Science, CAS

Abstract

In the era of mobile and cloud computing, cross-ISA (Instruction Set Architecture) binary translation attracts increasing attentions due to the ISA diversity of computing platforms. To easily adapt to vast guest- and host-ISAs with minimal porting efforts, existing cross-ISA binary translators (e.g., QEMU) are typically built upon ISA-independent Intermediate Representation (IR). Although IR conceals the architectural details of different ISAs, it also prevents enforcing several effective ISA-specific optimizations, which results in severe performance degradation. To improve the performance of cross-ISA binary translation without loss of portability, we present a fast cross-ISA binary translator, HERMES, by conducting post-optimization on the translated code, rather than on the IR as conventional binary translators do. The proposed post-optimization technique uses *Host-specific Data Dependence Graph (HDDG)* to significantly eliminate redundant instructions, including arithmetic, load/store and call/return-emulation instructions. To validate our approach, we implement HERMES on a commercial MIPS host system for both x86 and ARM guest. Compared with QEMU dynamic binary translator, HERMES improves the performance by a factor of 3.14x and 5.18x for x86 and ARM guest, respectively. Compared with state-of-the-art static binary translator, HERMES achieves comparable performance, while it reduces the translation overhead by 185x.

1. Introduction

As we enter the era of mobile and cloud computing, computing platforms with different ISAs (Instruction Set Architectures) are very pervasive. For example, ARM and Atom are the most common architectures of mobile devices, and AMD provides both x86 and ARM server processors for building data centers. Cross-ISA binary translation is a cost-effective way to enable the portability across various ISAs [2, 5–8]. A great challenge of cross-ISA binary translation is to adapt to different ISAs (including both guest- and host-ISAs) with minimal porting efforts. To address this challenge, existing cross-ISA binary translators are typically built upon ISA-independent Intermediate Representation (IR), which conceals the detailed architectural features of different ISAs. Although IR can minimize tedious porting efforts when migrating to different ISAs, it also prevents enforcing several effective ISA-specific optimizations, and incurs substantial redundant instructions. Therefore, existing cross-ISA binary translators always suffer severe performance degradation compared to native execution. For example, the x86-to-MIPS QEMU is 8.34x slower than native execution for SPEC CPU2000 benchmarks. Although several techniques have been proposed to improve the performance of cross-ISA binary translation [9, 17, 18, 20], reducing redundant instructions remains a challenging problem.

To eliminate redundant instructions in the translated code without loss of portability, in this paper, we present a fast cross-ISA binary translator, HERMES, by conducting post-optimization on the original translated code (rather than on the IR). The key of the post-optimization technique is that several necessary host-specific architectural characteristics (e.g., architectural registers and addressing mode) are considered for optimization. Specifically, at first the *Host-specific Data Dependence Graph (HDDG)*, which contains both host-specific instruction nodes and the related data dependence, is efficiently built from the original translated code. Then, several efficient and effective optimization passes are conducted on the HDDG to significantly eliminate the redundant instructions (including arithmetic, load/store, and call/return-emulation instructions). Because of such post-optimization, HERMES can not only improve the quality of translated code, but also still easily adapt to different guest ISAs due to the unmodified IR. To validate our approach, we implement HERMES on a commercial MIPS host system for both x86 and ARM guest. Experimental results show that, compared with the dynamic binary translator such as QEMU, HERMES improves the performance by a factor of 3.14x and 5.18x for x86 and ARM

1-4799-8161-8/15/$31.00 ©2015 IEEE

guest, respectively. Compared with state-of-the-art static binary translator, which is designed for the PowerPC-to-x86 translation [1], HERMES achieves comparable performance, while it reduces the translation overhead by 185x.

In summary, our main contributions are the following:

1. We present a fast cross-ISA binary translator by conducting post-optimization on the original translated code, which achieves both efficiency and portability.
2. Thanks to the host information offered by original translated code, we propose a HDDG-based post-optimization methodology that leverages *data dependence* between instructions for optimization passes.
3. Based on the HDDG, we propose several *lightweight* optimization passes to significantly eliminate the redundant instructions, including arithmetic, load/store and call/return-emulation instructions.
4. We conduct comprehensive experiments on both x86 and ARM guest. In addition to demonstrating good performance of HERMES for both user-level and system-level emulation, we also present several distinct observations by comparing the results of different guest ISAs.

Section 2 introduces the background and motivation. Section 3 presents details of HERMES. Section 4 discusses the implementation issues for different ISAs. Section 5 demonstrates the experimental results. Section 6 reviews some related work. Finally, section 7 concludes this paper.

2. Background and Motivation

2.1 Cross-ISA Binary Translation

Cross-ISA binary translation is a very critical technique for mobile and cloud computing. In mobile computing, cross-ISA binary translation ensures the portability of vast mobile applications across different host ISAs. In cloud computing, cross-ISA binary translation provides more potential in resource consolidation and resource utilization on pervasive heterogeneous data centers. One of the most well-known cross-ISA binary translators is QEMU [3], which currently supports several guest ISAs (e.g., x86, PowerPC, ARM and SPARC) on popular host ISAs such as x86, PowerPC, ARM and MIPS. To make QEMU easily portable (or retargetable) to different ISAs, QEMU leverages Tiny Code Generator (TCG) to produce the TCG IR[1], as shown in Figure 1. On the generic IR, QEMU conducts several simple optimization passes (e.g., *liveness analysis* and *store forwarding optimization*). Then, the optimized IR is directly translated to the host code via back-end.

Although generic IR is effective to reduce the tedious porting efforts when migrating to different ISAs, it also prevents enforcing several effective host-specific optimizations

[1] IR is a conventional technique in compiler design to separate the front-end, optimizer, and the back-end as different modules. With the help of IR, the number of different cross-ISA translators can be significantly reduced.

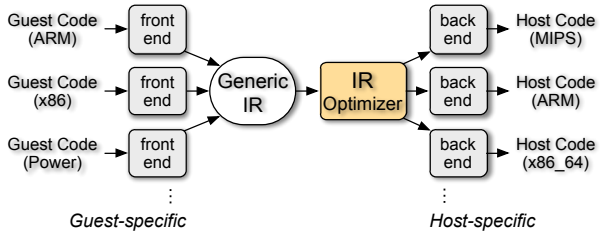


Figure 1: The framework of a generic translator with IR.

(e.g., peephole optimization, software pipelining), as it conceals the distinct features of different ISAs. As a result, there are many redundant instructions left in the generated code. By investigating the translation of QEMU, we find that the sizes of the x86-to-MIPS and the ARM-to-MIPS translated binaries are about 5.9x and 5.6x larger than that of native MIPS binary, respectively, which implies that the translated code is not properly optimized.

To enforce more aggressive optimization passes on the generic IR, several proposals transform the TCG IR to more powerful LLVM IR, and then retarget the LLVM IR to various host ISAs [11, 22]. However, since LLVM IR is still an abstraction of specific ISAs, the optimization potential may be limited. For example, due to the lack of host register information, even *register promotion* optimization of LLVM cannot effectively eliminate the memory accesses of memory-maintained variables (e.g., emulated guest CPU states). The reason is that the register promotion is confined within a code block, and many memory accesses are still indispensable across different code blocks[11].

Based on the above observation, the generic IR is not very suitable for conducting aggressive host-specific optimizations. Without impairing the portability offered by the IR, we propose to conduct post-optimization on the original translated host code, which can easily incorporate host-specific information to obtain effective and efficient optimization passes.

2.2 Optimization Passes for Binary Translation

Optimization passes are widely used in compiler design and program analysis. In compiler design, many effective compiler optimizations require multiple passes over a basic block, loop, or entire program. Taking GCC as an example, more than 200 passes are employed with "-O2" optimization level to optimize the code, including *dead code elimination*, *loop unfolding*, and *constant propagation*, etc.

In binary translation, several optimization passes are also employed to improve the quality of translated code. As introduced in previous sections, QEMU conducts simple *liveness analysis* and *store forwarding optimization* on the IR to get optimized code. To improve the quality of translated code, several "heavy" optimization passes in compiler design are utilized for binary translation. A notable example is HQEMU [11], which uses an enhanced LLVM compiler

to conduct aggressive optimization passes, such as, *register promotion optimization* and *scalar operation vectorization*, etc., on the generated LLVM IR. Compared with the QEMU, such optimization passes incur more than 60x slowdown on entire translation process. To reduce the optimization overhead, HQEMU offloads such optimization passes to other hardware threads or cores on a multicore system. However, it significantly reduces the system throughput (which is crucial for cloud computing).

Another important kind of optimization passes to eliminate redundant instructions are peephole optimization. Typically, a set of ISA-specific *peephole translation rules* should be designed and written by engineers. To automate the process, Bansal et al. propose a methodology called as *Peephole Superoptimization* [1], which exhaustively enumerates all possible translation rules to learn the optimal peephole translation rules. Then, such peephole rules are applied to any program using pattern matching. In addition to the costs to learn the peephole translation rules (including both human efforts to write the translation rules and the tedious searching process), the runtime translation overhead also cannot be neglected. For example, it takes 2-6 minutes to translate an executable with only about 100K instructions.

In a nutshell, delicate optimization passes improve the quality of translated code by greatly sacrificing the translation performance. Although the translation overhead can be amortized by long-running programs, such binary translation systems are not suitable for embedded environments where the programs have relatively short execution time [22]. Moreover, even for the long-running programs, especially for interactive applications that are common on mobile devices, the translation overhead may still have significant impacts, since the translation performance determines the start-up overhead, which refers to the consumed overhead until the program reaches the steady state [21]. The start-up overhead directly impacts another important metric: responsiveness, which in turn, affects user experience [12]. For example, it is not acceptable for the users to wait several minutes to open a program (e.g., Microsoft Word). Therefore, a practical binary translator should leverage *lightweight* optimization passes to reduce translation overhead.

3. The HERMES Cross-ISA Binary Translator

To preserve the portability offered by IR, HERMES proposes a novel post-optimization technique, which uses several *lightweight* optimization passes to eliminate the redundant instructions in the translated host code. In this section, first we introduce the overall architecture of HERMES. Then, we detail the HDDG-based post-optimization methodology.

3.1 Overall Architecture

Figure 2 presents the overall architecture of HERMES. In the traditional workflow of the binary translator, HERMES inserts a hook function in the back-end to capture the original translated instructions. All the instructions of an entire translation block are then buffered in the instruction cache. The instruction cache is treated as the input of the Host-specific Data Dependence Graph (HDDG) generator, which incorporates the *host information* (e.g., host ISA opcodes, registers and immediate) into traditional data dependence graph. Thus, the HDDG represents not only the data flow between instructions, but also the underlying host-specific view, such as register model, addressing mode, and constant operands. By leveraging the host information of the HDDG, several *lightweight* optimization passes are conducted to reconstruct the HDDG to eliminate the redundant instructions (including arithmetic, load/store, and call/return-emulation instructions), and then produce the optimized HDDG. After that, an assembler is used to generate the optimized host code according to the optimized HDDG.

In HERMES, the code generated by post-optimization is then executed without changing the traditional emulation process. In addition, the translation from guest code to generic IR is also left unmodified, which makes HERMES portable across different guest ISAs. By leveraging post-optimization covering different host ISAs, HERMES easily achieves both efficiency and portability across different hosts.

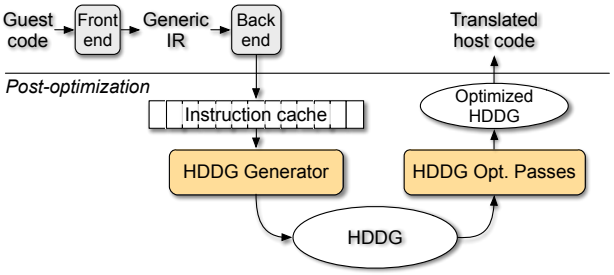


Figure 2: The overall architecture of HERMES using HDDG-based post-optimization.

3.2 HDDG-based Post-Optimization

In this section, we elaborate the proposed HDDG-based post-optimization methodology.

3.2.1 Generate HDDG.

To conduct the HDDG-based post-optimization, first we should generate the HDDG from the back-end. As shown in Figure 3, for each basic block of the guest code, the instructions in the instruction cache are retrieved to generate the *host-specific nodes*, and each node consists of the host information of the corresponding instruction. Once the host-specific nodes are built, the *data dependence* between nodes is constructed according to the source and destination register fields recorded as the host information. For example, if a previous node's destination register is used as a later one's

source register, these two nodes are linked with a directed *dependence edge*. Therefore, the entire HDDG contains both the host-specific nodes and the dependence edges.

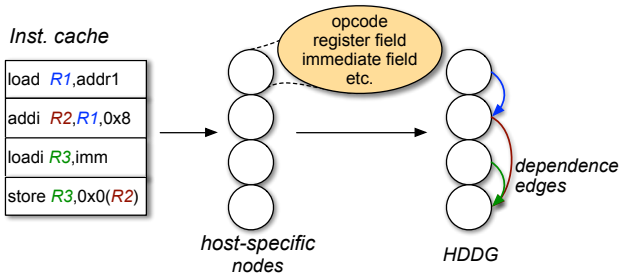


Figure 3: An illustrative example to generate the HDDG, which contains both the host-specific nodes and dependence edges.

3.2.2 Overview of HDDG-based Optimization.

As stated, all the optimization passes are conducted based on the HDDG. During the optimization pass, the host information, i.e. the opcode, immediate field, and the register field, of a node N_0 is considered to match with several predefined *instruction templates*, e.g., move-like, addi-like, and load/store instructions, etc. Once node N_0 is matched with an instruction template, for example, N_0 is a move-like instruction, a corresponding *optimization rule* is enforced on N_0 and the set of its destination nodes ($\{N_i | i = 1, ..., t\}$) according to HDDG. In this case, the optimization rule breaks the data dependence between N_0 and N_i by redirecting the input edge of N_i from N_0 to the source node of N_0. Then, if finally there is no output edge of node N_0, it could be marked as a "redundant" target for further processing.

After all the optimization passes are processed, the next step is to generate the host code from the optimized HDDG. By traversing the HDDG, the nodes that are marked as "redundant" targets would not produce the corresponding host instructions. However, to guarantee the precise emulation of guest exceptions, some instructions (i.e., load/store to emulated guest memory space, multiply, and divide, etc.) cannot be eliminated in both user-level and system-level emulation, even they are marked as "redundant" targets.

3.3 Optimization Passes On HDDG

We propose several optimization passes to reconstruct the HDDG. The optimization passes are classified into three categories based on the optimization targets, i.e., *arithmetic*, *load/store*, and *call/return-emulation* instructions.

3.3.1 Arithmetic.

Many redundant instructions are arithmetic instructions, and we consider three different instruction templates (and the related optimization rules) to eliminate such redundancy.

Move-like. A move-like node (e.g., move instruction) produces the same output as its input. Thus, if a move-like node is the source node of a dependence edge, the destination node of the dependence edge can directly use the input of the move-like node as its operand by redirecting the dependence edge. Actually, the move-like instruction template is the basis of load/store optimization, which will be detailed in the upcoming section.

Addi-like. Most addi-like nodes (e.g., add immediate instruction) produce intermediate variables for emulating the "base plus offset" addressing mode of some ISAs (e.g., x86). Typically, for an addi-like node, the difference between the output and the input is only a 16-bit signed constant. Thus, if an addi-like node is the source node of a dependence edge, the destination node of the dependence edge can modify its immediate number to redirect the dependence edge.

Constant-like. For a constant-like node (e.g., load immediate instruction), which is widely used for emulating guest absolute addressing mode, the output is only a constant number. In HERMES, the constant-like node is converted to move-like and addi-like node as follow. If two constant-like nodes produce the same number, the later one could be converted to a move-like node; If the difference of two constant-like nodes' output variables is a 16-bit singed, the later one could be converted to an addi-like node. Then, the modified HDDG could be optimized by leveraging the optimization rules of move-like and addi-like node.

Generally, the move-like nodes are caused by unnecessary data movement. The addi-like and constant-like nodes are incurred by different addressing modes of the guest and host ISAs. These three instructions correspond to the majority of the redundant arithmetic instructions. In addition to these instructions, other instruction templates, which heavily depend on the host characteristics, can also be defined for optimization. We will elaborate it in later sections.

3.3.2 Load/Store.

In the original translated host code, there are two different kinds of load/store instructions. The first one corresponds to the guest load/store instructions, and these instructions access the emulated *guest code/data space*. To offer accurate guest exception emulation, such load/store instructions cannot be optimized. The second kind of load/store instructions are used for updating the guest's Emulated CPU State (ECS) (e.g., registers, conditional codes, and EFLAG state, etc.) in the *host data space*. These instructions are called as State-Emulation Load/Store (SELS). SELS is very common in traditional binary translation systems, e.g., SELS is more than 30% of the instructions in the x86-to-MIPS binaries. Since SELS is not the source of guest exception, aggressive optimization passes can be considered to eliminate SELS.

To optimize SELS, the basic intuition is to first convert the relatively costly memory-targeted SELS to register-targeted move instructions, and then some of such move in-

structions can be further optimized. Concretely, we propose two optimization passes, i.e., SELS Elimination and CPU State Remapping.

SELS Elimination. SELS Elimination (SELSE) targets the entire memory space of ECS. For a load node whose *source* data already exist in the HDDG, i.e., the instruction loads a variable that have been loaded or stored by previous nodes, SELSE coverts the load to a move-like node, and adds a dependence edge from the node containing the required data to this node. Then, move-like optimization rules could be applied to this node. For a store node whose *target* data will not be used by later nodes, i.e., the node stores a variable that will be rewritten before loaded, SELSE marks the store node as a "redundant" target.

To efficiently find the potential redundant SELS instructions that access the same ECS variable for a given SELS, we use a table to record the SELS, and the table is indexed by ECS variables. However, once encountering exception instructions in system-level emulation or auxiliary functions (e.g., helper functions in QEMU), we need to clear the entire table to start a new SELSE process. This mechanism ensures that the program gets the correct ECS values after returning from the exceptions, since the original ECS values may be modified in the exception handlers or auxiliary functions.

CPU State Remapping. In contrast to SELSE, CPU State Remapping (CSR) only targets a part of the ECS memory space that is most frequently updated. CSR works by statically allocating host registers for several ECS variables that are originally maintained in memory, and thus the relevant SELS can be eliminated. The key of CSR is to select a small number of ECS variables from all ECS variables[2] for remapping, by balancing the number of used host registers and achieved performance gain. Apparently, the selection of such critical ECS variables varies on different ISAs, and we will elaborate it in later sections.

After CPU state remapping, the critical ECS variables are accessed by move instructions that are converted from original SELS instructions. For the original load instruction, a mapped register is allocated as the *source* register and for the original store instructions, a mapped register is allocated as the *destination* register. In HDDG, such converted move nodes are slightly different from the conventional move nodes. More specifically, the orginal-load and original-store move nodes do not have input edges and output edges, respectively. To optimize such move nodes, we further define two instruction templates, i.e., *fetch-like* and *update-like* template. Once a move node is matched with the fetch-like template, the destination nodes of its output edges can be allocated with the same source register as this node. Similarly, once a move node is matched with the update-like template, the source node of its input edge can be allocated with the same destination register as this node. After that, such move nodes can be marked as "redundant" targets.

Nevertheless, if the mapped register will be *updated* between a fetch-like node and its destination node, or the mapped register will be *fetched* or *updated* between a update-like node and its source node, the above optimization rules cannot apply. The reason is that removing such fetch-like or update-like nodes may break the *reaching definition* for the later nodes that depend on the results of such nodes.

[2] The total number of all ECS variables could be more than 200 for x86 guest, and more than 300 for ARM guest in QEMU.

3.3.3 Call/Return Emulation.

For a function call, the related *return* instruction is emulated as a conventional indirect branch, which requires expensive guest-to-host address translation. The basic optimization of call/return-emulation is to record the Translated Return Address (TRA) during *call* emulation, and then to directly use the TRA in the related *return* emulation.

The first challenge of *post-optimization* is to distinguish the call/return emulation from the other branch emulation in translated host code. To efficiently address this problem, our approach is built on the following observations. In spite of different guests, a "call" emulation is characterized by a constant-like node that generates the Guest Return Address (GRA), and the TRA should be recorded once the GRA is generated. A "return" emulation is characterized by a variable that is loaded from the guest stack and then stored in emulated PC register. According to data dependence on the emulated PC register, all instructions for address translation are recognized. These instructions are then replaced with much less instructions for optimized *return* emulation.

Traditionally, TRAs are maintained by a *return cache* [23], where TRAs are indexed by the starting addresses of the related guest *call* targets. Although return cache works well for the x86 host, it still brings considerable overhead for RISC hosts (e.g., MIPS and SPARC) that do not support 32-bit immediate addressing mode [10]. To efficiently obtain the TRA on RISC hosts, we propose Compressed Return Shadow Stack (CRSS) addressed by Guest Stack Pointer (GSP). Compared with traditional *shadow stack* [5] that may cause stack overflow (e.g., emulation of recursive functions), CRSS restricts the stack as a fixed size, and addresses it by a simple masked value of the GSP, to avoid costly checking of stack overflow. In this case, different GSP may be mapped to the same TRA in the CRSS. To efficiently determine whether the mapped TRA is the correct return target, the *return-site* GRA, which is determined during return, is compared with the *call-site* GRA. As shown in Figure 4, during call emulation, the call-site GRA is placed in the translated code segment, which is pointed by the corresponding TRA with a constant offset. In return emulation, the call-site GRA can be retrieved according to the TRA recorded in the CRSS. Once the retrieved GRA matches the return-site GRA, the execution jumps to the TRA. According to the direct branch at the TRA, the execution can be further redirected to the

real translated return address. The main advantage of the proposed process is that we do not need to generate additional pointers (which are 32-bit immediate) to store and to reload GRAs and TRAs, which especially benefits the RISC hosts.

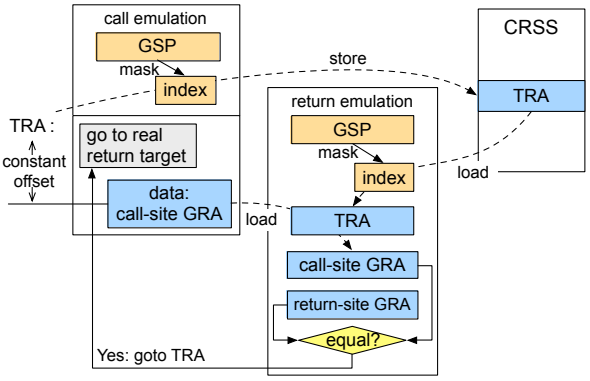


Figure 4: The workflow of CRSS-based call/return emulation.

4. ISA-Specific Features

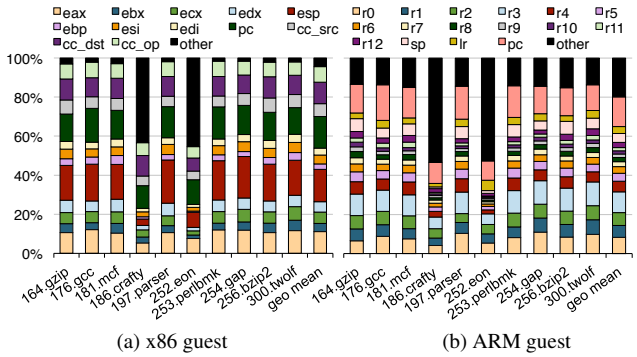
Although HERMES is a general approach that can be applied to any guest-ISA and host-ISA, several ISA-specific features need be considered to achieve optimal performance.

4.1 Guest Features

As stated, CPU State Remapping (CSR) is critical to the performance, and the selection of critical ECS (Emulated CPU States) variables varies for different guest ISAs. To validate this, we investigate the number of accesses to different ECS variables in x86-to-MIPS and ARM-to-MIPS translation. As shown in Figure 5, for x86 guest, accesses to 12 ECS variables (e.g., EAX, EBX, and ECX, etc.) cover more than 95% ECS accesses. Therefore, we map these variables to the host registers (except PC that is used for indirect branch and optimized by CRSS) to improve the performance. Since other variables (e.g., EFLAG state, FPU state, and exception state etc.) incur a small number of SELS instructions, they are still emulated by memory to save the precious host registers. For the ARM guest, accesses to 16 ECS variables cover about 80% ECS accesses. By balancing the number of used registers and the potential performance gain, we only map 11 ECS variables, i.e., R0-R9 and SP, to the host registers.

4.2 Host Features

For different host ISAs, the proposed post-optimization techniques could leverage host-specific features to extend the instruction templates, and thus further improve the performance. For example, for the ISA that provides the multiply-add instruction, an add instruction could generate an efficient multiply-add instruction, if the output of the add instruction is the source of a multiply instruction. Even for the same



(a) x86 guest (b) ARM guest

Figure 5: Breakdown of accesses to different ECS variables (on hot traces) for x86 and ARM guest.

instruction template, the corresponding optimization rules may vary on different hosts. For example, on x86 hosts, the constant-like nodes could be commonly optimized by the instructions with 32-bit immediate, while on MIPS hosts that do not support 32-bit immediate, constant-like nodes can only be optimized by converting to move-like and add-like node. Therefore, when migrating HERMES to different hosts, distinct host-specific features should be considered to revise the instruction templates (or optimization rules) to improve the potential performance gain.

Another notable host feature is the number of host registers that is also crucial to the CPU State Remapping (CSR). For ISAs that have relatively abundant registers, such as MIPS and SPARC that have 32 registers, a set of the registers can be dedicatedly used for CSR. However, for some ISAs that have less registers (e.g., x86), it is challenging to allocate appropriate number of registers for CSR, since the precious registers are also important to other parts of the emulation. A potential solution for x86 hosts is that some other registers such as vector registers could be used for CSR. Moreover, we also consider to provide hardware support to accelerate the guest CPU state emulation in the future.

5. Experiments

5.1 Experimental Setup

Our experiments are conducted on a Godson-3 processor, which is a Quad-core (1.0 GHz) MIPS-compatible system [13]. The host system has 2GB memory size. The operating system is a Fedora Linux with kernel version 2.6.32. The evaluated benchmarks are most widely used SPEC CPU2000 with *ref* inputs, and all benchmarks are compiled by GCC with "-O2" optimization flag and linked statically.

We implement HERMES for both x86 and ARM guest to demonstrate its portability. First we present the overall performance of HERMES for both x86 and ARM guest. In addition to user-level emulation, we also demonstrate the performance of system-level emulation. Then, we analyze the effectiveness of different optimization passes. We also

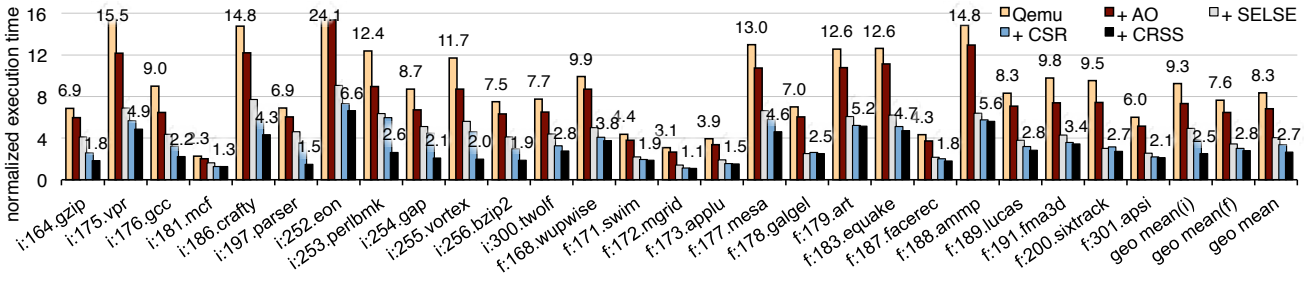


Figure 6: Overall emulation performance of HERMES for x86 guest, and all results are normalized to the native execution. The experiments are conducted by gradually enforcing different optimization techniques, i.e., AO, SELSE, CSR, and CRSS.

evaluate the translation overhead of HERMES. Finally, we compare the performance of HERMES with state-of-the-art static binary translator as *peephole superoptimization*.

5.2 Emulation Performance of HERMES

5.2.1 Emulation Performance for x86 Guest.

Figure 6 shows the emulation performance of HERMES for x86 guest, which is normalized to the performance of native execution. The achieved performance gains are presented by gradually enforcing different optimization passes, i.e., Arithmetic Optimization (AO), SELSE, CSR, and CRSS. On average, HERMES is 3.14x faster than QEMU for all evaluated benchmarks and is only 2.66x slower than native runs. Concretely, we make the following interesting observations: 1) HERMES achieves better performance for the CINT than CFP (i.e., 3.67x versus 2.71x). 2) AO offers relatively stable performance gains for all benchmarks, since it targets the common redundancy in the translated code. On average, AO yields 1.22x speedup for all benchmarks. 3) Load/store optimization, i.e., SELSE and CSR, provides the dominant performance improvement (2.02x speedup for all benchmarks), which implies that CPU state emulation is a major source of performance degradation. Besides, we also see that SELSE is more effective than CSR. 4) CRSS has more impacts on CINT than CFP, since there are more call/return instructions in CINT benchmarks. For example, for benchmark *perlbmk* that is call/return-intensive, CRSS improves the performance up to 131% even after conducting extensive optimizations such as AO, SELSE and CSR.

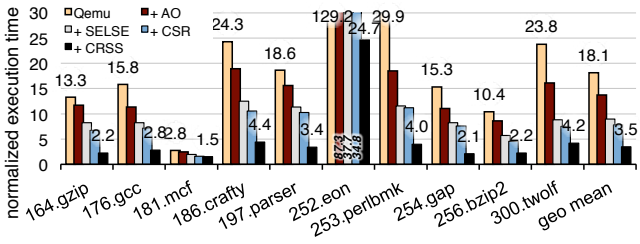


Figure 7: Overall performance of HERMES for ARM guest. All results are normalized to the native execution.

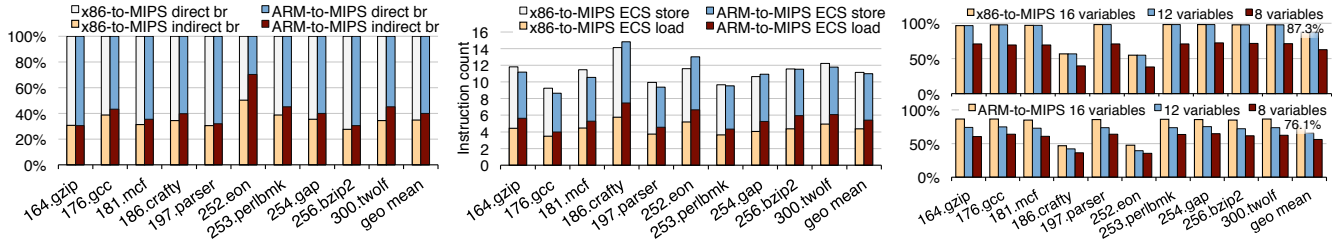
5.2.2 Emulation Performance for ARM Guest.

The main effort to make HERMES adaptable to the ARM guest is only to specify the critical ECS variables for register remapping. Actually, we remap *R0-R9* and *SP* to the host registers for ARM guest. Figure 7 shows the emulation performance of HERMES, which is normalized to the performance of native execution[3]. On average, HERMES is 5.18x faster than QEMU and is 3.50x slower than native execution. Compared with x86-to-MIPS emulation with the same benchmarks, the performance of ARM-to-MIPS emulation is 2.11x slower before optimization. While after optimization, the performance of ARM-to-MIPS emulation is only 1.46x slower. In other words, the performance improvement of HERMES is 1.45x higher in ARM-to-MIPS emulation. To reveal the reasons of such different performance improvements, we profile the number of indirect branches and SELS instructions, as shown in Figure 8.

According to Figure 8a, ARM binaries contain more indirect branches than x86 binaries, which implies that ARM-to-MIPS emulation spends more time on handling inefficient indirect branches. In QEMU, indirect branch addresses are stored in a global hash table, and *return* instruction is also treated as an indirect branch. In contrast, *return* instruction in HERMES does not occupy any entry of the global hash table, which reduces the hash collision for processing other indirect branches. Therefore, the processing of general indirect branches could also benefit from the CRSS. Since ARM-to-MIPS emulation involves more indirect branches, CRSS potentially offers more performance gains.

According to Figure 8b, we can see that ECS-load instructions are critical to the performance. For example, *crafty* and *eon* contain more ECS-load instructions than other benchmarks, and the corresponding emulations are much slower (i.e., 14.75x and 24.12x, respectively, for x86-to-MIPS) than native execution. Compared with x86-to-MIPS emulation, the ARM-to-MIPS emulation involves more ECS-load instructions. This could also be a potential

[3] We use CINT benchmarks for ARM-to-MIPS translation without *vpr* and *vortex*, because these two programs cannot be correctly emulated by the original QEMU (v1.6.1).



(a) Breakdown of Direct and Indirect Branches (b) Number of SELS in Each Block (c) Percentage of Accesses to Hot ECS Variables

Figure 8: Detailed analysis of branch and SELS instructions in x86-to-MIPS and ARM-to-MIPS emulation.

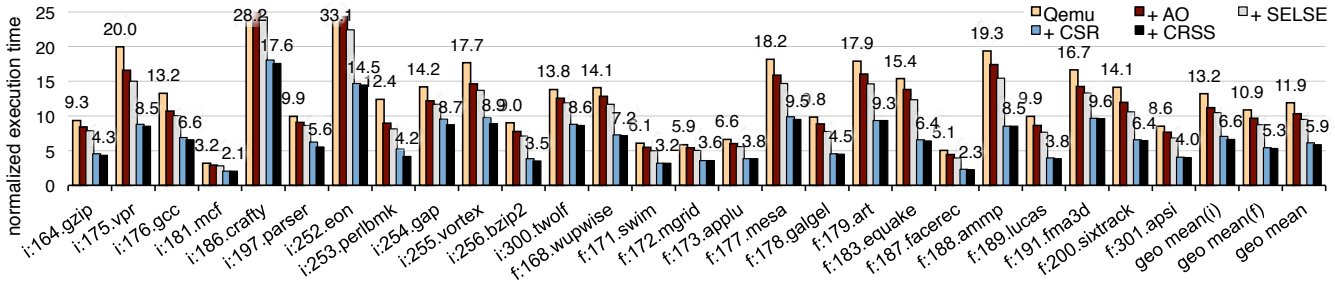


Figure 9: The system-level emulation performance for x86 guest.

reason that ARM-to-MIPS emulation is slower than x86-to-MIPS emulation.

The number of accesses to ECS variables is also very important to the emulation performance. As shown in Figure 8c, for benchmark *crafty* and *eon*, accesses to 16 hottest ECS variables are only around 50% of all accesses to ECS variables, which is also validated by Figure 5. Therefore, after remapping 11 variables to host registers in current implementation of HERMES, there is still a large number of SELS instructions, which could be an important reason of the low emulation performance for these two benchmarks. By further comparing the accesses to ECS variables for x86-to-MIPS and ARM-to-MIPS, ARM-to-MIPS emulation has less number of accesses to the "hot" ECS variables, which implies that the CSR achieves less performance improvement compared with x86-to-MIPS emulation.

In short, these experimental results demonstrate that, in addition to indirect branches, the emulation of guest CPU states is also a major bottleneck of cross-ISA binary translation. Specifically, the ECS-load instructions have distinct impacts on the emulation performance, and CSR could provide more performance improvement if most SELS instructions accessing a small number of ECS variables.

5.2.3 Performance of System-Level Emulation

HERMES also enables the system-level emulation, and Figure 9 shows the performance of system-level emulation for the x86 guest. On average, HERMES is 2.02x faster than QEMU and 5.88x slower than native execution. Apparently, the achieve performance gain is smaller than that of user-level emulation. The reason is that system-level emulation needs the costly translation from guest virtual address to host virtual address. It is expected that HERMES could achieve much higher performance gain by leveraging existing techniques, e.g., hardware-assisted translation [4], to reduce the address translation overheads.

A distinct feature of system-level emulation is that it requires accurate instruction-level exception, i.e., immediately emulating exception handling when guest exception occurs. To offer accurate exception, SELSE is restricted in the code segment that does not contain exception instructions, which limits the performance gain achieved by SELSE. Therefore, compared with user-level emulation, CSR obtains more performance gain than SELSE in system-level emulation.

5.3 Detailed Analysis of Post-Optimization

In this subsection, we detailedly analyze the effectiveness of proposed post-optimization for the x86 guest.

5.3.1 Reduction of Instructions.

To present more insights of the effectiveness of different post-optimization techniques, we further conduct experiments to show the contributions of AO, SELSE, and CSR[4] in terms of the reduction of instructions. As shown in Figure 10, we can see that the reductions of instructions contributed by these techniques well comply with the related performance gains as shown in Figure 6. For example, after

[4] We do not measure the reduced instructions by CRSS, since CRSS is used to reduce the emulation instructions of indirect branch handling rather than directly reduce the translated code.

optimization, benchmark *crafty*, *twolf*, and *eon* still contain much more instructions than native MIPS code, which result in relatively low emulation performance. While for benchmark *parser* and *bzip2*, the achieved performance gains are notable since the translated code is significantly reduced.

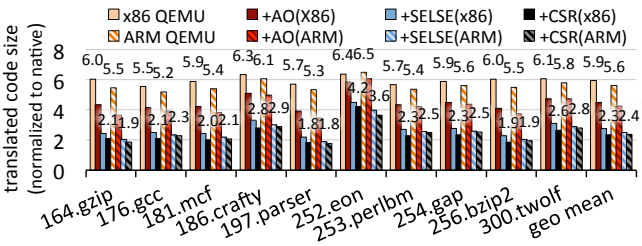


Figure 10: The size of translated code, which is normalized to the size of native MIPS code.

5.3.2 Evaluation of Register Remapping for CSR.

The CPU State Remapping (CSR) process can be conducted for different guest CPU states (e.g., general registers such as EAX and EBX) separately, and Figure 11a shows the corresponding experimental results. As shown in Figure 11a, the achieved performance gains by mapping different guest registers vary significantly for different benchmarks. For example, the performance gains by mapping *ESI* and *EDI* are obvious for *gcc* and *parser*, while the corresponding performance gain is almost 0% for *mcf*. Moreover, for *galgel*, CSR even results in about 10% performance degradation. The potential reason is that *galgel* suffers extremely high register pressure, and mapping some host registers to ECS variables significantly hurts the performance.

Figure 11b further demonstrates the accumulated performance gains by increasing the number of mapped guest registers. An interesting observation is that, for benchmark *gcc*, although solely mapping *EDI* to host register only achieves 3.77% performance gain, the accumulated performance gain is improved from 1.36x to 1.65x by adding the mapping of *EDI*, which implies that the performance gains achieved by mapping different guest registers are non-independent. We also observe that the performance gain shrinks as the number of mapped registers increases (on benchmark *mcf* the performance even decreases when 11 registers are used for mapping). This observation well validates that 11 mapped registers is an optimal design option since it balances the register pressure and performance gain.

5.4 Translation Overhead of HERMES

We measure the translation overhead of HERMES and QEMU in terms of the translation throughput, which is defined as the number of translated guest instructions in 1M CPU cycles, for x86 guest. As shown in Figure 12, on average, the translation throughput of HERMES is only 43% of that of QEMU. Such additional overhead is caused by the proposed post-optimization techniques. Fortunately, the translation process

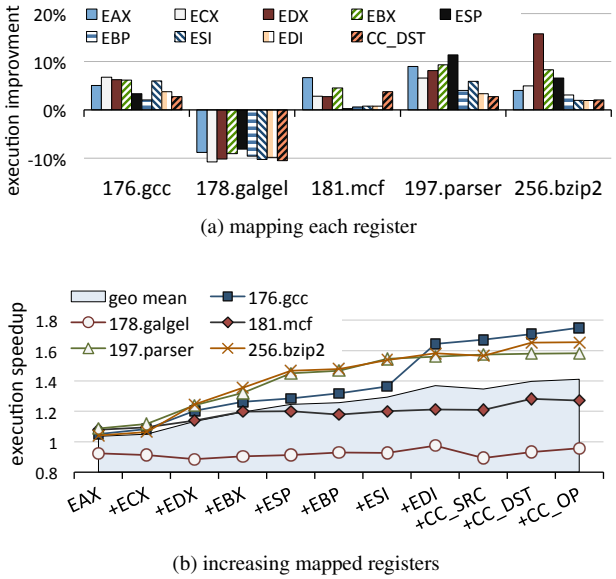


Figure 11: The achieved performance gain of CSR by mapping guest CPU states separately.

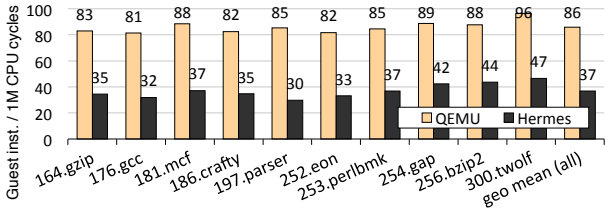


Figure 12: Translation throughput of QEMU and HERMES.

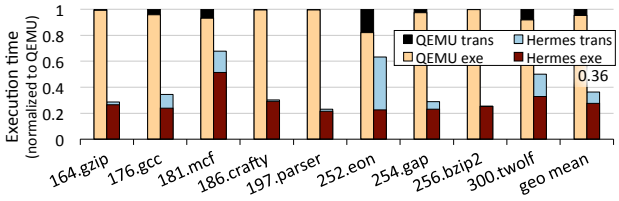


Figure 13: Overheads of translation and execution in QEMU and HERMES for x86-to-MIPS emulation, which are normalized to the entire overhead of QEMU.

is a one-time cost, and it can be significantly amortized by long-period execution.

Even for short-period execution, HERMES can still hide the translation overhead. Figure 13 compares translation and execution overheads of QEMU and HERMES for CINT benchmarks with *test* input. Although HERMES is slower during translation, it is still 2.78x faster than QEMU for entire execution. In comparison, another state-of-the-art binary translator, HQEMU is only 1.08x faster than QEMU for x86

guest with single-threaded translation for CINT benchmarks with *test* input.

5.5 Comparison with Static Binary Translator

We also empirically compare the emulation performance of HERMES with state-of-the-art static binary translator as *Peephole Superoptimizer (Peephole)*, which also focuses on improving the quality of translated code. According to the published results for PowerPC-to-x86 translation [1], Table 1 compares the performance of HERMES and *Peephole* in terms of performance speedup over QEMU and native execution on respective hosts. When comparing with the native execution, the performance of HERMES is comparable with *Peephole* for the program *gzip* and *parser*. While for program *twolf*, *Peephole* outperforms the HERMES significantly. Nevertheless, the related speedups over QEMU are comparable for *Peephole* and HERMES, as 2.62x and 2.78x, respectively. On average, the speedup over QEMU is 3.36x for HERMES, while the average speedup over QEMU is only 2.60x for *Peephole*. Generally, on the evaluated programs, we can conclude that the emulation performance of HERMES is comparable to that of *Peephole*.

In addition to performance, we also compare the the translation overhead in terms of translation throughout. According to the published results [1], the translation throughput of *Peephole* is at most 200 instructions per second. In contrast, the average translation throughput of HERMES is about 37K instructions per second, which significantly outperforms *Peephole*. In other words, HERMES reduces the translation overhead by about 185x. The above comparison results demonstrate that HERMES outperforms the state-of-the-art binary translator.

Table 1: Comparison between HERMES and Peephole Superoptimizer. Performance is evaluated by comparison with QEMU and native execution.

	HERMES		Peephole	
	to QEMU	to native	to QEMU	to native
gzip	3.72x	54.1%	2.59x	56.1%
mcf	1.81x	79.3%	1.47x	94.7%
parser	4.61x	67.1%	3.20x	67.3%
gap	4.19x	48.1%	3.22x	42.5%
bzip2	3.98x	52.9%	3.00x	73.7%
twolf	2.78x	35.9%	2.62x	153.0%
geo mean	3.36x	54.5%	2.60x	74.5%

6. Related Work

According to the difference of the guest- and host-ISA, binary translators can be roughly divided into two categories, that is, same-ISA translators and cross-ISA translators. The same-ISA translators are mainly used for program instrumentation, debugging, and security [14, 15, 19, 25]. The cross-ISA translators are used for enabling portability of compiled programs. There are many cross-ISA binary translators, e.g., Digital FX!32 [5] for x86-to-Alpah translation, IA-32 EL [2] for IA32-to-Itanium translation, and Transmeta CMS [6] for x86-to-VLIW translation, etc.

To provide a general multi-to-multi-ISA translation framework, Bellard proposes a fast full-system binary translator as QEMU, which is widely used in academia and industry [3]. QEMU emulates several CPUs (x86, PowerPC, ARM and Sparc) on several hosts (x86, PowerPC, ARM, Sparc, Alpha and MIPS). However, the performance degradation of QEMU is still very severe. To improve the performance of QEMU, many techniques have been advocated. HQEMU improves the performance of QEMU by executing the translation and optimization on different threads or cores of a multicore system, which enables several delicate optimizations on the LLVM IR [11]. DBILL is a cross-ISA dynamic binary instrumentation tool that instruments executables of one ISA running on another ISA [16]. DBILL also leverages QEMU and the heavy static optimization techniques provided by LLVM. Bansal and Aiken propose an efficient binary translation approach using superoptimization techniques [1]. The key idea is to automatically and systematically learn the translation rules between two ISAs by exhaustively enumerating all possible rules. Although this approach achieves high efficiency, the search for translation rules is a time-consuming process. Therefore, it is still an open question that whether such heavy optimization techniques can be applied to the dynamic binary translators. In contrast to these work, HERMES directly works on the translated host code by enforcing host-specific post-optimization. Thus, HERMES can use several lightweight yet effective optimization techniques to significantly eliminate the redundant instructions.

In addition to software-based binary translation, there also exists hardware-based approaches to improve the performance of binary translation. Hu et al. propose several hardware supports (e.g., new instructions, new addressing mode, and content-associated memory, etc.) to improve the translation performance of x86-to-MIPS translation on Godson-3 processors [13]. Harmonia is a ARM-to-IA (Intel Architecture) dynamic binary translation tool. In addition to software-based analysis and optimization techniques to solve the *register mapping* and *condition-code* problem, Harmonia also extends the hardware to support new instructions for CC-flag translation [20]. Yao et al. also propose a hardware-software approach to improve the performance of dynamic binary translation [24]. Apparently, HERMES is completely orthogonal to these approaches, and HERMES can cooperate with them to further improve the performance.

7. Conclusions

In this paper, we present a fast cross-ISA binary translator, HERMES, by conducting post-optimization on the translated host code. HERMES is built upon HDDG to conduct

lightweight but *effective* optimization passes to significantly eliminate the redundant instructions. HERMES is implemented on a commercial MIPS host system for both x86 and ARM guest. Compared with QEMU, HERMES improves the performance by 3.14x and 5.18x on average for x86 and ARM guest, respectively. Compared with the state-of-the-art static binary translator, HERMES achieves comparable performance, while it reduces the translation overhead by 185x.

In the future, we will investigate the performance of HERMES for supporting more host ISAs, e.g., x86 and ARM. Moreover, hardware support to accelerate the binary translation is also left as our future work.

Acknowledgments

This work is partially supported by the National Sci&Tech Major Project (No.2009ZX01028-002-003, 2009ZX01029-001-003, 2010ZX01036-001-002, 2012ZX01029-001-002-002, 2014ZX01020201), National Natural Science Foundation of China (No.61221062, 61100163, 61133004, 61173001, 61232009, 61222204, 61432016), the National High Technology Development 863 Program of China (2012AA010901, 2012AA011002, 2012AA012202, 2013AA014301). Qi Guo is partially supported by DARPA PERFECT program. Special thanks to Loongson Tech for Loongson Research Funding.

References

[1] S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In *OSDI'08*, 177–192, 2008.

[2] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 Execution Layer: A two-phase dynamic translator designed to support ia-32 applications on itanium-based systems. In *MICRO '03*, 191–201, 2003.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05*, 41–41, 2005.

[4] X. Chang, H. Franke, Y. Ge, T. Liu, K. Wang, J. Xenidis, F. Chen, and Y. Zhang. Improving virtualization in the presence of software managed translation lookaside buffers. In *ISCA '13*, 120–129, 2013.

[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):56–64, Mar. 1998.

[6] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing(TM) software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03*, 15–24, 2003.

[7] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: A new run-time control point. In *MICRO '02*, 257–268, 2002.

[8] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Trans. Comput.*, 50(6):529–548, June 2001.

[9] A. Guha, K. Hazelwood, and M. L. Soffa. Memory optimization of dynamic binary translators for embedded systems. *ACM Trans. Archit. Code Optim.*, 9(3):22:1–22:29, Oct. 2012.

[10] J. D. Hiser, D. W. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. *ACM Trans. Archit. Code Optim.*, 8(2):9, 2011.

[11] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *CGO '12*, 104–113, 2012.

[12] S. Hu and J. E. Smith. Reducing startup time in co-designed virtual machines. In *ISCA '06*, 277–288, 2006.

[13] W. Hu, J. Wang, X. Gao, Y. Chen, Q. Liu, and G. Li. Godson-3: A scalable multicore RISC processor with x86 emulation. *IEEE Micro*, 29(2):17–29, 2009.

[14] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Security '02*, 191–206, 2002.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, 190–200, 2005.

[16] Y.-H. Lyu, D.-Y. Hong, T.-Y. Wu, J.-J. Wu, W.-C. Hsu, P. Liu, and P.-C. Yew. DBILL: An efficient and retargetable dynamic binary instrumentation framework using LLVM backend. In *VEE '14*, 141–152, 2014.

[17] A. Mittal, D. Bansal, S. Bansal, and V. Sethi. Efficient virtualization on embedded power architecture platforms. In *ASPLOS '13*, 445–458, 2013.

[18] 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 *LCTES '09*, 147–156, 2009.

[19] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI '07*, 89–100, 2007.

[20] G. Ottoni, T. Hartin, C. Weaver, J. Brandt, B. Kuttanna, and H. Wang. Harmonia: A transparent, efficient, and harmonious dynamic binary translator targeting the Intel architecture. In *CF '11*, 26:1–26:10, 2011.

[21] D. Pavlou, E. Gibert, F. Latorre, and A. Gonzalez. DDGACC: Boosting dynamic DDG-based binary optimizations through specialized hardware support. In *VEE '12*, 159–168, 2012.

[22] B.-Y. Shen, W.-C. Hsu, and W. Yang. A retargetable static binary translator for the ARM architecture. *ACM Trans. Archit. Code Optim.*, 11(2):18:1–18:25, 2014.

[23] S. Sridhar, J. S. Shapiro, E. Northup, and P. P. Bungale. HDTrans: An open source, low-level dynamic instrumentation system. In *VEE '06*, 175–185, 2006.

[24] Y. Yao, Z. Lu, Q. Shi, and W. Chen. FPGA based hardware-software co-designed dynamic binary translation system. In *FPL'13*, 1–4, 2013.

[25] Q. Zhao, D. Bruening, and S. Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *CGO '10*, 22–31, 2010.