Jiunn-Yeu Chen, Wuu Yang, Jack Hung, Charlie Su, Wei Chung Hsu, A Static Binary Translator for Efficient Migration of ARM based Applications, Workshop on Optimizations for DSP and Embedded Systems, 6th Workshop on Optimizations for DSP and Embedded Systems (ODES, April 6, 2008, Boston, Massachusetts), 2008.

A Static Binary Translator for Efficient Migration of ARM based Applications

Jiunn-Yeu Chen, Wuu Yang,
CS Department, National Chiao Tung University, Taiwan
Jack Hung
CS Department, Princeton University
Charlie Su
Andes Technology, Taiwan
Wei Chung Hsu
CSE Department, University of Minnesota

Abstract

Binary translation is often used in migrating legacy binaries to new architecture based platforms. This paper describes a static binary translator which translates ARM binaries to a MIPS-like architecture designed for embedded systems. The static translator handles basic architecture translations and performs optimizations to minimize instruction overhead. The conditional execution feature in the ARM architecture requires special attention on binary translation and optimization. With several optimizations to minimize condition updates and checks, the translated code from ARM to our target architecture increases the instruction path length by only 35% on the EEMBC benchmark.

1 Introduction

Binary translation, the process of translating one binary executable into a different binary executable, has been commonly used in various applications, such as ISA (Instruction Set Architecture) migration [1][2][3][4], static binary instrumentation [5][6], fast architecture simulation [7], dynamic binary instrumentation [8][9], and runtime optimization [10][11][12].

In general purpose ISA migration, using dynamic binary translation has almost become a standard procedure. For example, Aries [2] migrates HP-PA binaries to the IA-64 architecture, Rosetta [4] migrates Power PC code to IA-32, IA-32EL [3] migrates IA-32 executables to IA-64. The primary purpose of using such process virtual machines [14] to migrate existing application executables is to support compatibility. However, some dynamic binary translation systems are used to increase the number of applications available to a new platform. For instance, DEC FX!32 [1] was developed to make numerous IA-32 applications available to the DEC Alpha platform. A successful binary translation system could certainly reduce the time-to-market requirement for having a large number of applications available for a newly defined ISA. The number of different ISAs in general purpose computing has been declining in the past several years. Few companies can afford to support and maintain their proprietary ISA. However, in the embedded system area, new architectures have often been introduced. Using binary translation to migrate embedded applications may become commonplace in the future.

In general purpose computing, dynamic binary translation has been used more often. However, a dynamic binary translation system normally will incur significant overhead on program start-up where the legacy binary must be translated on the first invocation. However, since the most important factor of binary translation in this area is to make legacy applications available to users, the performance of a migrated application is of secondary importance. For embedded systems, the consideration may be different due to some additional important requirements. For example, a migrated application should have reasonable start-up performance since embedded system users may not have the patience to wait for a slow start-up. The requirement of high power efficiency also plays an important role that limits the use of dynamic binary translation. This implies both the execution time and the space overhead of the translated binary should be acceptable. With such requirements in mind, a mix of static and dynamic translation approach to migrate embedded applications becomes more attractive.

Static translation has the advantage of avoiding the translation overhead at runtime. With that advantage, a static translation system can perform more time consuming, but performance critical optimizations to improve the code quality and the space required for the generated code. As a matter of fact, more and more embedded binaries are generated by compilers rather than hand coding. Compiler generated code is much more friendly to static binary translators. Therefore, a mixed approach would let the static translator handle most of the code in application, and let the dynamic translator handling the cases missed by static translation.

In this paper, we present a static binary translator which is part of our mixed static/dynamic binary translation system. This static binary translator converts ARM based binary code to a newly developed MIPS-like architecture. We called this new target architecture MIPS' in this paper. We show that EEMBC [17] benchmark suite can be translated correctly with minimal loss of execution efficiency (with optimization, the translated instruction path length is increased by less than 35%). In section 2, we describe the high level view of the static translator and how ~~does~~ it handles typical binary translation issues like (a) code discovery problems and (b) code location problems for indirect branches [14]. We also discuss how unique architecture features in ARM, such conditional execution, are translated to MIPS'. Section 3 provides details on the optimizations implemented in our translator to minimize the overhead of flag emulation. Section 4 gives our experiment setups, including the simulators and the benchmark used in the study. Section 5 discusses the performance of our static binary translation on the benchmark suite, Section 6 summaries and concludes this paper.

2 Code Generation Overview

For our static binary translator, the source architecture is ARM. Just like IA-32 executables dominate general purpose computing applications, ARM executables dominate embedded applications. Our target architecture is a MIPS-like architecture with some additional features such as 16-bit instructions and load/store multiple words instructions. In this section, we highlight the translation issues for major differences between the source and the target architectures.

2.1 PC-relative data access

There are data embedded in the text section of the ARM binary For example, a load instruction can access such data using PC-relative address as follows:

```
ldr   r1, [pc + 0xoffset]
```

The data embedded in the text section are mainly large immediate that cannot fit in the immediate field and the jump table for switch statements. From code generation perspective, the generated target machine code will reference the location in the text section. In section 3, we will discuss how to optimize for PC-relative data accesses. After static binary translation, we will keep the text section around so that PC-relative data access can get the needed data. PC-relative data accesses are not frequent, however, they do exist. Table 1 shows the relative frequency of such instructions in the ARM binary for the EEMBC benchmark, compiled by the GCC compiler.

Benchmark	Total ARM instructions (static)	PC-relative accesses	Percentage
EEMBC-base	10855	207	1.91%
EEMBC-speed	9919	204	2.06%
EEMBC-space	9872	209	2.12%

Table 1. Frequency of PC-relative data accesses in the EEMBC benchmark (EEMBC-base is compiled with no optimization, EEMBC-speed is compiled with optimization for speed, and EEMBC-space is compiled with optimization for space).

2.2 Shifter operand and shifter carry out

The general ARM instructions support a shifter operand. For example, in the following *add* instruction:

add r0, r 1, r2, lsl #2

The third operand, register r2, shall be shifted left by 2 before it is used by the *add* instruction. Our target architecture does not support such shifter operands, so additional instructions are needed to carry out the shift operation. Fortunately, shifter operands are not that frequent in ARM binaries. Table 2 shows the frequency of some commonly used shifter operands.

As the shifter operand is generated, shifter carry out is also generated at the same time. The shifter carry out is not directly used by the operation such as the *add* in the example, it is used to update the condition flag C. Thus additional instructions are also needed to update the shifter carry out.

	LSL-immediate	LSR-immediate	ASR-immediate
EEMBC-base	2.45%	2.01%	0.83%
EEMBC-speed	2.52%	2.06%	0.85%
EEMBC-space	2.46%	2.07%	0.80%

Table 2. Frequency of the use of shifter operand in the EEMBC benchmark

2.3 Conditional execution

Conditional execution is a mechanism to perform predicated execution. In the ARM ISA, all instructions are predicated instructions. The value they used to decide if the instruction will be executed is stored in four bits, and these four condition bits are illustrated in Figure 1.

There are sixteen different conditions as shown in Figure 1. Some of the condition checks only need to examine one specific bit, and some need to check for multiple bits. The code generated for each check will be different, for example, the following *addeq* instruction only checks if one condition bit is set. In the following code example, r15 is allocated to hold condition flags, and r21 is a temporary register.

Opcode	Mnemonic	Interpretation	When to execute?
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Figure 1. All 16 condition codes in the ARM architecture

ARM :

addeq r0, r1, r2

MIPS':

btst r21, r15, 1
beqz r21, 8
add r0, r1, r2

However, the *addge* instruction in the following checks if the N flag is equal to the V flag, the translated instructions are as follows:

ARM :

addge r0, r1, r2

MIPS':

btst r21, r15, 0
btst r22, r15, 3
sub r23, r21, r22
beqz r23, 8
add r0, r1, r2

Instructions with condition code AL do not need to check any condition bits, and the instructions with NV will be translated into NOP. Our translator generates one or multiple branches to handle condition checks. If the condition specified were not met, the attempted operation is skipped. In the EEMBC benchmark, conditional execution instructions are used rather frequently. Table 3 shows the percentage of ARM instructions that specify conditions other than AL and NV .

	Check condition code
EEMBC-base	10.27%
EEMBC-speed	19.57%
EEMBC-space	19.84%

Table 3. Frequency of the condition code check in the EEMBC benchmark (dynamic measure).

2.4 Condition flags Handling

The four condition flags in the ARM ISA are N, Z, C, and V. They are located at bits [31:28] of the CPSR (Current Program Status Register) register. Two types of ARM instructions modify these four flags: the comparison instructions, and the ALU instructions or the move instructions with the s-bit set.

2.4.1 Register mapping of condition code

One target register can be preserved to store the four condition bits. However, the access and set of such bits become expensive since it would requires at least two instructions (i.e. shift and logical OR) to update one condition flag.

```
/* Instructions that calculate the new C flag */
slli  r21, r21, 2        // store  the  C flag  in bit[2]
or  r15, r15, r21        // update the flag register
```

The instruction overhead for updating condition flags is high, especially for instructions that set all four condition bits. Table 4 shows the frequency of ARM instructions in EEMBC that need to update the condition flags.

	Percentage
EEMBC-base	9.49%
EEMBC-speed	14.99%
EEMBC-space	15.18%

Table 4. Frequency of instructions that update condition flags in the EEMBC benchmark (using dynamic measure)

In order to minimize instruction overhead for condition flag updates, we decided to allocate each of the four flags in individual target registers. This would avoid the shift operation overhead for each flag update. However, the downside of this approach is the use of three additional registers which might be better used for some optimizations where more temporary registers are needed. Since our target architecture has 16 more general purpose registers than the ARM architecture, we can afford preserving four registers for the condition flags, giving that the frequency of flag update is fairly high in the ARM application binaries.

2.5 Processor mode and the Thumb instruction set

ARM supports regular execution mode and the Thumb execution mode. The Thumb execution mode allows 16-bit ARM instructions to be used. Using Thumb execution code can effectively reduce the code size, and is considered important for many embedded applications. A special instruction, *bx,* must be used to switch between the Thumb execution mode and the regular ARM mode. The *bx* instruction format is as follows:

Bx r1

The execution will enter Thumb mode if the least significant bit of register r1 is 1. Otherwise, the execution stays in ARM mode. Since the prefix of Thumb instructions is similar to that of the ARM instructions, the translator must know the execution mode to correctly parse the instructions. Therefore, this feature makes static translation difficult because the value of register r1 may not be known at translation time. Our static translator does not handle Thumb instructions. Thumb mode execution will be handled by the dynamic translator.

2.6 Register mapping

There are 31 general purpose registers in the ARM architecture, but only 16 of them are visible to programmers and compilers. The rest of the registers are used to speed up exception handling. Furthermore, registers r8-r14 are banked, which means there are multiple physical copies of each registers. Which physic registers are actually referenced depend on the current execution mode. Our static binary translator supports only user mode, so we do not need to take care of which physical register~~-~~s are used. The number of ARM registers that need~~s~~ to be maintained as part of the architecture state is 16. Our target architecture has 32 32-bit general purpose registers, so the 16 ARM registers can be mapped directly as a subset of the target architecture registers.

Register mapping is conducted in two steps: (a) ARM registers r0 to r11 are mapped to our target architecture registers r0 to r11, and (b) ARM registers r12to r15 are mapped to target registers r28 to r31. ARM registers r12 to r15 are usually used special registers such as PC, LR, SP, and so on. Mapping them to consecutive target registers is needed so that we may translate the load/store multi-word instructions in ARM directly into similar multiple word load/store instructions ~~-~~in the target ISA, which requires the registers be contiguous.

Other than the 16 registers preserved for mapping to ARM registers, ~~—~~the remaining registers in our target architecture are used for temporary registers, and for special usages such as the shifter operand. In our current translator, five of them are used as temporary registers, two of them are reserved for handling privileged mode, and four of them are saved for future usage.

In section 2.4.1 we discussed the need to allocate registers to hold each condition flag so that the cost of flag update and check can be significantly reduced. Initially, we allocate the four condition flags in one register (mapped for CPSR). Now we separate them out and assigned one register for each. The allocation of the target registers not mapped to the 16 ARM general purpose registers are now listed in Table 5.

Index	Usage	Index	Usage
12	Shifter operand	20	V flag
13	Shifter carry out	21	Temp register 1
14	Top of RAS	22	Temp register 2
15	Unused	23	Temp register 3
16	Special condition flags	24	Temp register 4
17	N flag	25	Temp register 5
18	Z flag	26	Reserved
19	C flag	27	Reserved

Table 5. New register allocation for the ~~-~~target architecture ~~exclude~~ excluding the ones mapped to the 16 ARM registers (i.e. r0-r11 and r28-r31).

2.7 Executable layout

A typical ARM executable layout is shown in Figure 2.The binary layout of the target program of the translator can be simply classified into three parts: ARM program sections, target program sections, control management sections. The origin ARM program sections are stored in the ARM program part of the target program. Keeping the ARM program sections allows the target program to access the data in the ARM program sections. The text section is also needed due to the load/store of PC-relative data. All the ARM sections are allocated in the same place as in the original ARM executable layout. This decision makes the memory access much easier to handle, no additional computation is needed to compute the memory address of the operands.

The target program part has the regular text, data, and bss sections. These sections are allocated in high memory since the lower addresses are used for the ARM sections. The control management part has RAS, address mapping table, and address stub sections. The purpose of these three sections will be discussed in the section 2.9.

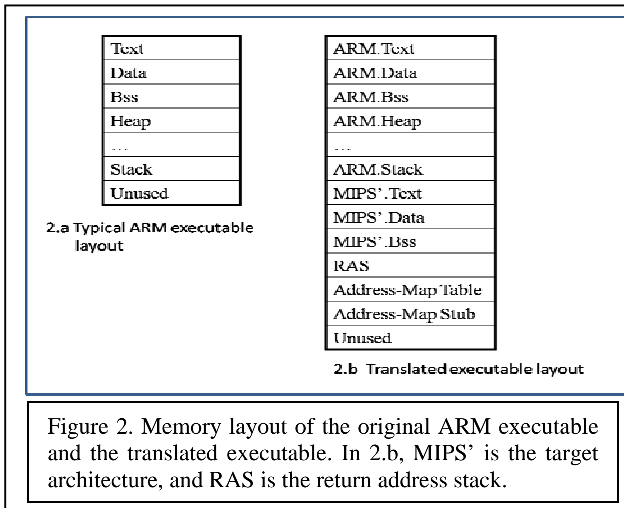

Figure 2. Memory layout of the original ARM executable and the translated executable. In 2.b, MIPS' is the target architecture, and RAS is the return address stack.

2.8 Control flow graph

Like other static binary translation tools [5][6], we also build the control flow graph of each translation unit. The control flow graph serves the purpose of transfer control management and code optimization.

Our translator constructs the control flow graph by recognizing potential basic blocks and establishing the relation between different blocks. The usual way to recognize basic blocks in binary code is to identify the leader of each basic block. Typical leaders are instructions following direct or indirect branches, instructions targeted by direct branches, and program entry point. Here we must handle one additional case which is the address for PC-relative data. The basic blocks containing PC-relative data are identified to prevent treating PC-relative data as instructions, so they are excluded from the control flow graph.

The control flow graph we built is not precise because the targets of indirect branches may not be known at translation time. In section 3, we will discuss code optimizations implemented in our translator. Since some optimizations are based on the control flow graph, if there is an indirect branch jump to the middle of a basic block, our code optimization may becomes invalid. Therefore, when the address mapping routine discovers at runtime that the target of an indirect branch is not a recognized basic block, it will throw an exception and will transfer control to our runtime translation system to handle such cases.

2.9 Program control management

Program control management includes updating the ARM-PC, and handling direct and indirect branches. The following subsections describe the mechanisms used in the translator to manage control transfers.

2.9.1 Lazy update to ARM-PC

In the generated code, updating the ARM-PC is required since other instructions may reference the PC. However, updating ARM-PC for each instruction incurs unacceptabley high overhead. Notice that only a small percentage of instructions need to reference the PC explicitly. In many cases, the PC referenced is a known value at translation time – it is an offset to the text section. Therefore, our translator employs a method that updates PC only when it is needed and cannot be resolved at translation time. This means the translator generates instructions to update the PC before it is to be referenced, for example, push the current PC onto the stack.

For PC-relative data access, the ARM-PC value is directly embedded as an immediate of the target instruction as shown in the following example:

ARM :

```
        add     r1, pc, #228
```

MIPS' :

```
        movi    r31, 0x0000811c    // updated ARM -PC
        addi    r1, r31, 228
```

2.9.2 Indirect branch handling

Direct branches can be handled at translation time since the branch target address for the translated block is known. For indirect branches, typical static binary translators assume non-return indirect branches are generated for the switch statements. Hence, they search the binary backwards from the indirect branch to figure out where is the jump table. Once the starting address of the jump table is known, the remaining translation can be straight forward – each entry in the jump table will be replaced by the translated address. Our binary translator uses the shadow return address stack [14] to handle return branches, and identifies jump table to handle switch based indirect branches. However, we have a general address mapping approach as the safety net for "unstructured" indirect branches which often exist in hand crafted code. For unstructured indirect branches, their target addresses will be used to search the ARM-to-MIPS' address mapping table to obtain the translated address.

2.9.3 Address mapping table

The ARM-to-MIPS' mapping table is generated by the translator and stored as part of the data section in the translated executable. The table maps an ARM instruction address to the address of the translated instruction. It is used to provide the target address for "unstructured" (non-return, non-switch) indirect branches. To minimize the size of the table, we do not keep one entry for every ARM instruction. Instead, we allocate one entry for each basic block. If the target address of an indirect branch is not a recognized basic block, the execution will trap to the runtime system and rely on the dynamic translator to fix the corner case.

Each time an unstructured ARM indirect branch is executed, the control will transfer to a stub generated by the binary translator. This stub is used to lookup the address mapping table and check if the current entry contains the correct ARM address. Since this table lookup is performed at runtime, it must be efficient to avoid excessive overhead. A simple hash function is used to hash the ARM address into an index to the table. If the search hit, the stub will provide a target address for the execution to continue. A hash collision is solved by linear probing. However, we allow the table to grow as needed during translation time to minimize collisions. For the EEMBC benchmark suite, the generated table size is either 1K or 2K. If the search missed in the table, the stub will trap to our runtime system.

2.9.4 Return address stack

Although the addressing mapping table can basically handle the indirect branches, to search the address mapping table is not cheap. In order to accelerate indirect branch handling, a prediction mechanism is often used. If the branch target is as predicted, a direct branch can be used instead. This approach usually works well, unfortunately, return branches, which are also indirect branch, are difficult to predict since a procedure can often be called from many different places. Hence, we implement the

return address stack, also called shadow return address stack in [14], to speed up return branch handling.

3 Code Optimization Overview

In section 2, we describe code generation issues for translating ARM instructions to our target architecture, MIPS'. In this section, we discuss optimizations that we have implemented and their associated issues. To make the code examples concise, we use the new register mapping introduced in section 2.6.

3.1 PC-relative data access optimization

First of all, PC-relative data should not be translated as code since it will increase the target code size. Our translator does not translate blocks that cannot be reached. Some blocks may not be reached based on static analysis, but may be reached by indirect branches at runtime. For such a case, our address mapping table and stubs will catch this exception at runtime and generate a trap to the runtime translation system.

PC-relative data accesses are usually translated into data move instructions. However, to load a PC-relative data, we must maintain the source PC. So a simple PC-relative load may require two target instructions, one to update the source PC, one to load the data from the text section. One way to optimize for this case is to inline the PC-relative data so that only a single *movi* instruction is needed, or if the immediate is large, another *sethi* instruction can be added. However, there is a potential risk for this optimization because the PC-relative location may be modified, so that the data might change at runtime. Inlining the data at translation time would be wrong in such cases. Our translator checks all the store instruction to find the PC-relative data that might be stored, and then inline the remaining PC-relative data. We also make the text section read-only, so that if some non-PC-relative stores touch the text section, our trap handler will detect such cases.

3.2 Check condition code selectively

For conditional execution instructions, the translated code will test for the condition and skip over the actual operation if the check failed. There are optimization opportunities for multiple conditional execution instructions with the same conditions. In this case, our translator can simply generate the check condition instructions only once but carefully adjust the target of the branch to the correct place.

The code generator will check the next instruction to see whether the two instructions are under the same execution condition. Figure 3 shows one example of this optimization. In Figure 3, the baseline translation would generate four target instructions. However, since the two ARM instructions are checking the same condition, our optimization can remove the second branch by modifying the target address of the first branch to skip two instructions instead of one.

```
ARM                                  :
addeq   $Rd1, $Rs1, $Rt1
subeq   $Rd2, $Rs2, $Rt2

MIPS':

beqz            $R_FLAG_Z, 8
add             $Rd1, $Rs1, $Rt1
sub             $Rd2, $Rs2, $Rt2
```

Figure 3. Example of translating two conditional instructions with the same condition. The condition code check of the second ARM instruction is eliminated.

A similar optimization can be applied for two instructions having reversed condition. In this case, we may not eliminate the second condition check instruction, but we can create a shorter execution path. The consecutive instructions with inverse condition implies that if the condition check of the first instruction fails, the condition of the second instruction must be met, so that the offset of the condition branch in the check code of the first instruction can be modified to bypass the second check. This is illustrated in Figure 4.

```
ARM                                  :
addeq   $Rd1, $Rs1, $Rt1
subne   $Rd2, $Rs2, $Rt2

MIPS':

beqz            $R_FLAG_Z, 16
add             $Rd1, $Rs1, $Rt1
bnez            $R_FLAG_Z, 8
sub             $Rd2, $Rs2, $Rt2
```

Figure 4. Example of translating two conditional instructions with a reverse condition. The branch offset of the first instruction is modified as shown in bold face.

3.3 Update condition code flags selectively

Translating the update of condition code flags in ARM instructions will incur a high instruction overhead. For example, if we update all four condition flags for each instruction, the instruction overhead could be as high as 8 times (i.e. two additional instructions generated per update). It is one of the most critical area calls for optimization.

In [1], the FX!32 translator deals with the same challenge since the x86 architecture has a similar condition code architecture as ARM. FX!32 traces the condition code dependency in the control flow graph of each translated unit. A flag is not updated unless it is actually been used. In other words, we try to locate those unnecessary flag updates, and avoid generating flag updating instruction. For example, if instruction A updates all flags, and instruction B, which follows instruction A, also updates all the flags, then there is no need to translate the flag update for A, since no other instructions will use such information. All the flag updates from instruction A will be overwritten by instruction B. Through this analysis, we can selectively update the condition code flag and eliminate most of the redundant updates to condition flags. As performed in FX!32, we also traverse successor blocks to further reduce unnecessary condition flag updates cross blocks. The performance impact of cross block redundant flag updates elimination is very significant.

3.4 Special condition code

The previous two subsections deal with redundant condition check/update elimination. However, there are cases we must update the condition flags and the instruction overhead is high. There are cases where multiple condition flags need to be checked. For example, the GT condition indicates Z, N and V flags must be checked. To avoid the cost of updating three flags, however, we could combine multiple condition check into one special condition

to check. As illustrated in Figure 5, the check of GE condition can be implemented using the set-less-than condition in MIPS'. In order to carry out this optimization, we must ensure there is only one type of check between two condition code updates.

The multi condition code update/check cases include HI, LS, GE, LT, GT, and LE. Since these six condition codes are set based on a less-than test, we can use only one set-less-than instruction to update the register allocated to represent the special condition code. This optimization reduces multiple condition bit updates and checks to only one update and check.

```
ARM                                                    :
cmp     Rd1, 0
add     Rd2, Rs1, Rs2
addge   Rd3, Rs3, Rs4
cmp     Rd1, 0

MIPS':
movi    R_TEMP_1, 0
slts    R_FLAG_SPECIAL, Rd1, R_TEMP_1
//Set the flag if Rd1 is great than or equal to R_TEMP_1
add     Rd2, Rs1, Rs2
bnez    R_FLAG_SPECIAL, 8
//If the flag were not set, the condition code is not matched
add     Rd3, Rs3, Rs4
```

Figure 5. Example of replacing ordinary condition flags with special condition flag.

3.5 Combine conditional branch

ARM use condition codes to carry out conditional branches. In some cases, while the MIPS only requires one compare-and-branch instruction, the ARM may need two instructions, one to set condition code and one to branch based on the condition code set, to get the job done. Such cases include:

1) CMP + conditional branch
2) TST + conditional branch
3) TEQ + conditional branch

Although it seems like the above cases favor MIPS-like architectures, in practice, the advantage is not very significant. This is because the condition flags set may be used in instructions more than the conditional branch, so the update of the condition flag can not be eliminated.

Conditional branch optimization gives some interesting results on the translated EEMBC code. There are some functions, the translated binary has even fewer target instructions than the source instructions. This is rather unusual when translating a more complex ISA (i.e. ARM) to a simpler ISA (i.e. MIPS').

3.6 Other optimizations

The optimization methods introduced in the previous subsections are part of the code generation components which work on the ARM IR's. Optimizations at code generation time are rather limited since they do not have the knowledge of other instructions that have not yet been emitted. For example, the redundant condition code check elimination and the inverse condition check optimization are limited to consecutive instructions. More powerful optimizations can be implemented after the target IR's are generated. For example, some classical peephole optimizations such as DCE (Dead Code Elimination), CSE (Common Sub-expression Elimination), CF (Constant Folding) and CP (Constant Propagation and Copy Propagation) can be applied with a larger scope. Adding this phase of optimization may serve two purposes: 1) There may be optimization opportunities missed by the ARM compiler (could be compiled with no optimizations) and 2) There may be new opportunities introduced during our binary code translation.

We have implemented a peephole optimization phase which identifies and reports on opportunities existing in the target IR's. This phase also estimates the potential performance gain in terms of number of instructions can be eliminated. This optimization phase is iterative. Based on profile analysis, we collect code patterns that may be optimized. Such patterns are given to the peephole optimization phase to identify and report on the complete set of benchmark. Based on the estimated performance gain, we set priorities on what transformations to implement. The performance estimation of those peephole optimizations is discussed in section 5.

4 Simulation Environment

The target platform and the tool chains for our binary translation system are still under development. The processor chip has been under sampling and some test boards are available. However, the evaluation of the performance of our binary translator is carried out using simulations where we could have hooks to collect more detailed profiles on benchmark execution.

4.1 Simulators

The simulator we used for simulating the MIPS' programs is a modified SID [15]. SID is a simulator that supports multiple ISAs, and also provides support for testing, validation and debugging. A version of SID is also used to run ARM binaries. Although there are existing ARM test boards to run ARM binaries, using an ARM simulator is more convenient for profiling and verification. We modify the GDB 6.3 [16] to support profiling of the ARM programs.

4.2 Benchmarks

The benchmark we selected is the EEMBC [17] benchmark suite version 1.1. EEMBC benchmark is commonly used for embedded system developers to tune their hardware design and software tool chains. There are 55 subprograms and are divided into six main categories: 8-16 bit, automotive, consumer, networking, office, and telecom. The EEMBC benchmark can be compiled as normal versions or as lite versions. To speed up our simulations, we use the lite versions.

The ARM compiler we used to compile EEMBC is GCC 3.4.3 with static linking. To test our static translation comprehensively, we compiled the EEMBC benchmark with 3 different options: EEMBC-base, EEMBC-speed, and EEMBC-space. EEMBC-base is compiled with option "-O0", EEMBC-speed is compiled with option "-O2", and EEMBC-space is compiled with "-Os". Since our translator does not translate the Thumb instruction set, we did not create versions with Thumb instructions.

5 Experimental results

In this section we evaluate the optimizations discussed in section 3 in translating the EEMBC benchmark suite, and we present and discuss our simulation results. Since the target architecture is under development, the test board is not commonly available at this time, so the measurements we provided in this section are based on both ARM and target machine simulations.

5.1 Baseline code generation

The performance improvements from each optimization discussed in section 3 are presented in Figure 6. The baseline we

used in comparison is the translated code using basic translations described in section 2. The performance is measure by the ARM to MIPS' execution ratio. For example, the performance of the baseline translation, which is labeled as "BASELINE" is 2.58 for EEMBC-base, 3.62 for EEMBC-speed, and 3.6 for EEMBC-space. As explained before, EEMBC-base is the EEMBC benchmark suite compiled by GCC with no optimizations, where EEMBC-speed is compiled with optimization for speed and EEMBC-space is compiled with optimization for space. The number 2.58 means the dynamic number of executed MIPS' instructions to the number of ARM instructions is 2.58. In other words, for each ARM instruction, on average, the basic translation would take 2.58 translated MIPS' instruction to execute. The ratios are higher for optimized ARM binaries, 3.62 for EEMBC-speed and 3.6 for EEMBC-space. Optimized ARM binary tend to have a higher translation ratio because compiler optimizations would have eliminated many simple instructions, such as the copy operation, which can often be translated into a single target instruction. The baseline ratio of 2.58 is somewhat expected based on past experience of binary translation of various general purpose architectures

The performance bar of each optimization is tagged with a name. For example, the bar for the optimization to eliminate redundant condition check is labeled "CHECK", and the performance bars for this optimization which eliminates unnecessary flag updates is labeled "UPDATE".

.

5.2 Selectively check condition code

As shown in Table 3, a large fraction of instructions are conditional – they must check the condition code to determine if execution is needed. There are nearly 20% of such instructions in both EEMBC-speed and EEMBC-space. This indicates eliminating redundant condition check may have a good potential for performance improvement when translating ARM binaries to other RISC architectures with no predicated execution instructions. Although it may be interesting to translate ARM instructions to Itanium architecture which does have predicated instructions, there are no practical needs to do so because Itanium is not designed for embedded systems.

The bars under the name "CHECK" in Figure 6 are results from applying the redundant condition check elimination. Compared with the baseline translation, this optimization yields no gains for EEMBC-base, and small gains for both EEMBC-speed (from 3.62 to 3.57) and EEMBC-space (from 3.6 to 3.55). This seemingly low performance gain indicates although conditional execution is frequent in ARM code, there are not many consecutive instructions using the same condition in the compiled EEMBC code. The frequency of using the same condition increased slightly when ARM binaries are optimized (i.e. in EEMBC-speed and EEMBC-space).

However, there is a different way to conduct redundant condition check elimination, which would require most complex data flow analysis and incur a much higher transformation cost. Notice that there may be instructions having the same condition as execution predicates, but not next to each other. We can use a separate phase to search and group them together. For such a group, a single branch could skip the entire group. This is different from translating predicated instructions. For architectures with predicates, the analysis to determine if two instructions are under the same condition is easier – just check if the common predicate is updated between the two instructions. To determine whether two non-consecutive instructions are under the same execution conditions is a little more complex since multiple condition bits are involved. We are currently evaluating the potential for such an optimization.

5.3 Selectively update condition flag

Table 4 shows the percentage of instructions that may update condition flags in the origin ARM program. The percentage of instructions that update condition flag is almost as frequent as the instructions that check condition codes in the EEMBC-base. For EEMBC-speed and EEMBC-space, flag update instructions are somewhat less frequent than conditional execution instructions.
The performance result of applying redundant condition update elimination is shown by the bars labeled as "UPDATE" in Figure 6. The ratio of EEMBC-base is decreased from 2.58 to 1.87., a 38% of performance improvement. The other two benchmarks have even higher improvements; the ratio is dropped from 3.57 to 2.48 for EEMBC-speed, 44% of performance gain, and dropped from 3.55 to 2.44 for EEMBC-space, 46% of performance gain. The redundant flag update elimination is the most significant optimization. When translating ARM binaries to other embedded architectures, this shall be the first optimization to consider.

Although it seems that flag updating is as frequent as condition checking, the cost of flag updating is higher, thus the optimization yields a higher return.

5.4 Combined conditional branch

Combining condition code checking with a branch into a single compare-and-branch is a more interesting optimization discussed here. Most of the other optimizations eliminate redundancies introduced by binary code translation, but this combined conditional branch transformation not only eliminates redundancies but also compresses multiple instructions into one

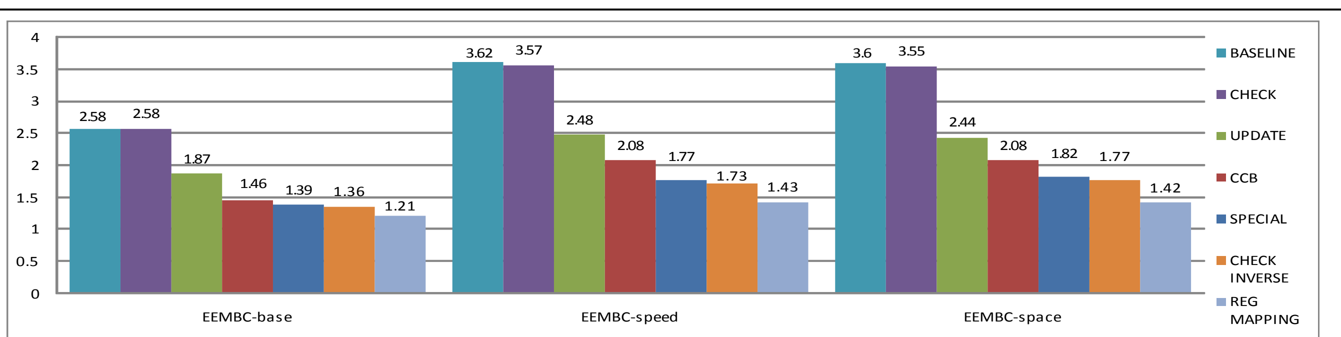

Figure 6. Performance improvement from various optimizations

instruction. It gives our translator a chance (although small) to reduce the number of translated instructions executed to be even less than the number of source instructions.

The performance result of combined conditional branch is showed by the bars labeled as "CCB" in figure 6. All three benchmarks have very good improvement. Different from previous two optimizations, this optimization improves EEMBC-base more than the other two. The ratio of EEMBC-base decreased from 1.87 to 1.47, 27% of performance gain. For EEMBC-speed and EEMBC-space, the improvement is about 19%. Figure 7 shows the frequency of combined conditional branch instructions. As shown in Figure 7, EEMBC-base has more opportunities for this optimization than the other two benchmarks.

5.5 Special condition flag

Special condition flag optimization combines multiple condition updates and checks into one condition update and check, thus saving instruction overhead for flag updates and condition checks. In Figure 6, EEMBC-base has only slight improvement from this optimization (from 1.46 to 1.39, about 5% of gain), while EEMBC-speed and EEMBC-space have much greater speed up. The improvement for EEMBC-speed is about 18% (from 2.08 to 1.77) and 14% for EEMBC-space (from 2.08 to 1.82).

5.6 Check inverse condition code selectively

The check inverse condition code optimization has a minor impact to performance. All three benchmarks benefit 2-3% from this optimization. This should be no surprise to us because section 5.2 indicates even the same condition optimization does not render notable performance gains. The approach that group instructions with the same execution conditions (i.e. predicates) and use one branch for each group can also be applied here to enhance the reverse condition check optimization.

5.7 Register remapping

In the first design, we allocate the four condition flags in one register, we called flag register. The flag checking and updating are carried out just like the ARM architecture. After learning the importance of flag emulation, we decided to keep each flag in a separate register to avoid instruction overhead to fetch/store the flag from/to the flag register.

In Figure 6, the bars with name "REGMAPPING" show the performance of this optimization. For EEMBC-base, the gain is about 12% (from 1.36 to 1.21), and the gain is more significant for the optimized ARM binaries. EEMBC-speed gains 21% (from 1.73 to 1.43) and EEMBC-space gains 25% (from 1.77 to 1.42).

With all above optimizations, the translated code can run at ratio 1.21 for EEMBC-base, 1.43 for EEMBC-speed, and 1.42 for EEMBC-space. The average ratio of the three benchmarks is 1.35. It is generally considered very cost-effective to get many applications ready for a new platform with only 35% of execution time overhead.

5.8 Peephole optimization estimation

As discussed in section 3.6, peephole optimizations such as DCE, CSE, CF and CP, may be applied to the target architecture IR's after the code generation from the ARM IR's. This is to exploit possible redundancy elimination opportunities introduced by the code translator. We implemented a peephole optimizer to identify such opportunities and estimate the potential contribution from such optimizations. Figure 8 shows the estimation – adding peephole optimizations may provide additional 5% of performance.

5.9 Discussion on Predicated Execution and Conditional Branches

In the EEMBC benchmark, there is one program which the translated code executes fewer instructions than the original ARM program. It is the *rotate01_lite* in the EEMBC-base benchmark, and the instruction ratio of ARM/MIPS' is 0.94. The reason for a lower-than-1 ratio is because of the combined conditional branch transformation. The frequent use of conditional branches in this program provides our binary translator such an opportunity.

However, we notice that there could be other opportunities for our translator to yield lower-than-1 execution ratio for more programs. This is the case we mentioned in section 5.2. A conditional executed instruction is translated into a branch and a regular instruction, where the branch may skip the regular instruction. Although we may skip the regular instruction, we will have to execute the branch instruction so that the chance of reducing translated instruction is not obvious. Nevertheless, if we add a separate phase to group instructions with the same or reversed conditions together. We will have a greater opportunity to skip more instructions. In other words, programs with more "predicated false" instructions can have potential for our translated code to achieve less-than-1 execution ratio.

This brings up the issue about the measure we used in this study. We currently use the dynamic executed instructions as a metric to evaluate the performance of optimization. For many embedded architectures, this may be reasonable. However, for migrating ARM binary to RISC architectures without predication, this could be misleading. Predicated execution will execute more instructions in general, but may minimize the cost of branch mispredictions. Our current target architecture has a short pipeline with relatively low branch misprediction penalty. However, for a fair comparison, branch misprediction should be part of the measurement. We are collecting this data and attempt to include it in the final version submitting to the workshop.

Another consideration is the performance of the memory subsystem. Since our translation must keep the original ARM code, the executable will be at least twice as large as the original ARM binary. This, however, may not have a significant impact to the I-cache performance since most instruction accesses are from the translated code, The ARM code section will be rarely referenced except for PC-relative data references, and when exceptions occur and the code must trap to the runtime system to get help from the dynamic translator.

The primary purpose of a binary translation system is not for performance – it is to make more applications available at the time a new architecture is introduced. However, the performance and power efficiency gap should not be too severe that makes the new embedded system unattractive to users. With this in mind, our static translator, with code optimizations, can provide help to application binary migration from ARM to other MIPS-like platforms.

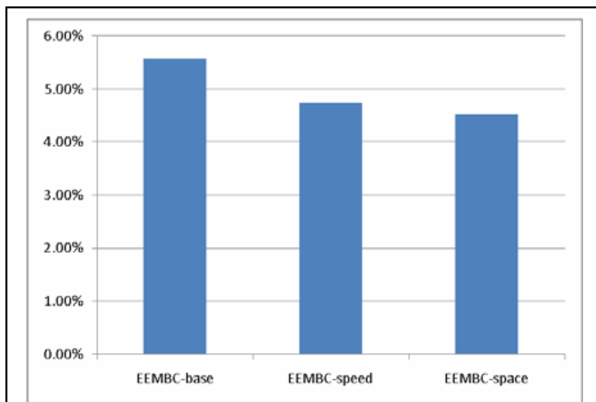

Figure 7 : Percentage of combined conditional

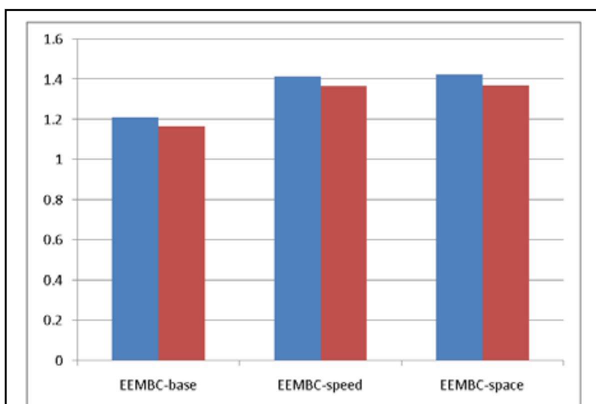

Figure 8 : Estimation of the potential of peephole optimization (Blue bars: without peephole, Red bars: with peephole)

6 Conclusion

Dynamic binary translation is a common way to migrate existing application executables to new platforms. However, dynamic binary translation is known to have some weakness like slow start-up, less efficient code space utilization, and high translation overhead for short running applications. For embedded systems, where quick start-up is important, and power efficiency is a must, and many applications may be short running, a pure dynamic translation system is less attractive. We therefore consider a mixed approach which combines the advantages of static translation and dynamic translation for migrating ARM based application binaries to other newly introduced platforms.

The ARM architecture has some features that must be handled carefully in binary translation. For example, the conditional execution instructions, the frequent condition flag updates, the PC-relative data accesses require special optimizations. Our baseline translation without optimizations achieves execution ratio of 2.58 (EEMBC-base), 3.62 (EEMBC-speed) and 3.6 (EEMBC-space), with an average of 3.27. However, with our optimization for selective condition updates, combined conditional branch generation, special condition generation, register reallocation for condition flags, and PC-relative data inlining, the execution ratio improved to 1.21 (base), 1.43 (speed) and 1.42 (speed), with an average of 1.35 for all three benchmarks. The overall performance improvement is 2.4 times. Furthermore, we have estimated a set of peephole optimizations applied to the target IR's could yield 5% of additional performance.

With the static binary translator and its optimizations, we gain confidence on that our combined binary translation system may offer an attractive solution to application migration for newly developed embedded systems.

In this study, we measured the performance using the number of instruction executed, rather than the actual execution cycles. This could be misleading since branch mispredictions and cache misses have not been taken into account. We will enhance the study with micro-architecture simulations and take measures on the real system when it is available to re-confirm the effectiveness of the static binary translator, and the role it plays in the combined binary translation system.

7 References

1) Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates, "FX!32: a Profile-Directed Binary Translator", IEEE Micro, vol. 18, no. 2, 1998
2) Cindy Zheng and Carol Thompson, "Aries -- PA-RISC to IA-64: Transparent Execution, No Recompilation", IEEE Computer, vol. 33, no. 3, March, 2000.
3) Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang and Yigal Zemach, "IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems", in Proceedings of 36th Annual International Symposium on Microarchitecture, Micro-36, December 2003.
4) Rosetta, Apple Inc., http://www.apple.com/rosetta/
5) Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey and Brian Lewis, "Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework", SUN Microsystem Technical report, TR-2002-105, 2002, "http://research.sun.com/techrep/2002/abstract-105.html
6) J. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing", In proceedings of PLDI'95, 1995.
7) Bob Cmelik and David Keppel, "Shade: a fast instruction-set simulator for execution profiling", Proceedings of the 1994 conference on Measurement and modeling of computer systems, 1994, Pages 128 – 137.
8) Nicholas Nethercote and Julian Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation", in Proceedings of PLDI 2007.
9) PIN - A Dynamic Binary Instrumentation Tool. http://rogue.colorado.edu/.
10) Vasanth Bala, Evelyn Duesterwald and Sanjeev Banerjia Dynamo, "A Transparent Dynamic Optimization System" Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation, 2000, Pages 1 – 12
11) Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, "The Performance of Runtime

Data Cache Prefetching in a Dynamic Optimization System", in Proceedings of 36th Annual International Symposium on Microarchitecture, Micro-36, December 2003.

12) Bruening, D., Garnett, T., and Amarasinghe, S. "An Infrastructure for Adaptive Dynamic Optimization". In Proceedings of 1st International Symposium on Code Generation and Optimization (CGO) (2003), pp. 265-275.
13) Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter Sweeney, "A Survey of Adaptive Optimization in Virtual Machines", in Proceedings of IEEE, Vol. 93, No.2, February, 2005.
14) James Smith and Ravi Nair, "Virtual Machines: Versatile Platforms For Systems And Processes", Morgan Kaufmann, ISBN-13: 9781558609105, 2005
15) SID, http://sourceware.org/sid/
16) GDB, "GNU Project Debugger", http://sourceware.org/gdb/
17) EEMBC, "The Embedded MicroProcessor Benchmark Consortium",http://www.eembc.com/