

# On Static Binary Translation and Optimization for ARM based Applications

Jiunn-Yeu Chen, Wu Yang, Tzu-Han Hung, Charlie Su, Wei-Chung Hsu

CS Department,  
National Chiao Tung  
University, Taiwan

CS Department,  
Princeton University

Andes  
Technology, Taiwan

CSE Department,  
University of  
Minnesota



# Outline

---

- \* Motivation
- \* System Overview
- \* Implementation Issues
- \* Optimizations
- \* Experimental Results
- \* Conclusions



# Motivation

---

- \* A new ISA needs a large application base for developers to adopt.
- \* Binary translation can quickly migrate existing applications
- \* Dynamic binary translation is commonly used, but not ideal for embedded systems



# Motivation

- \* Dynamic vs Static binary translation.

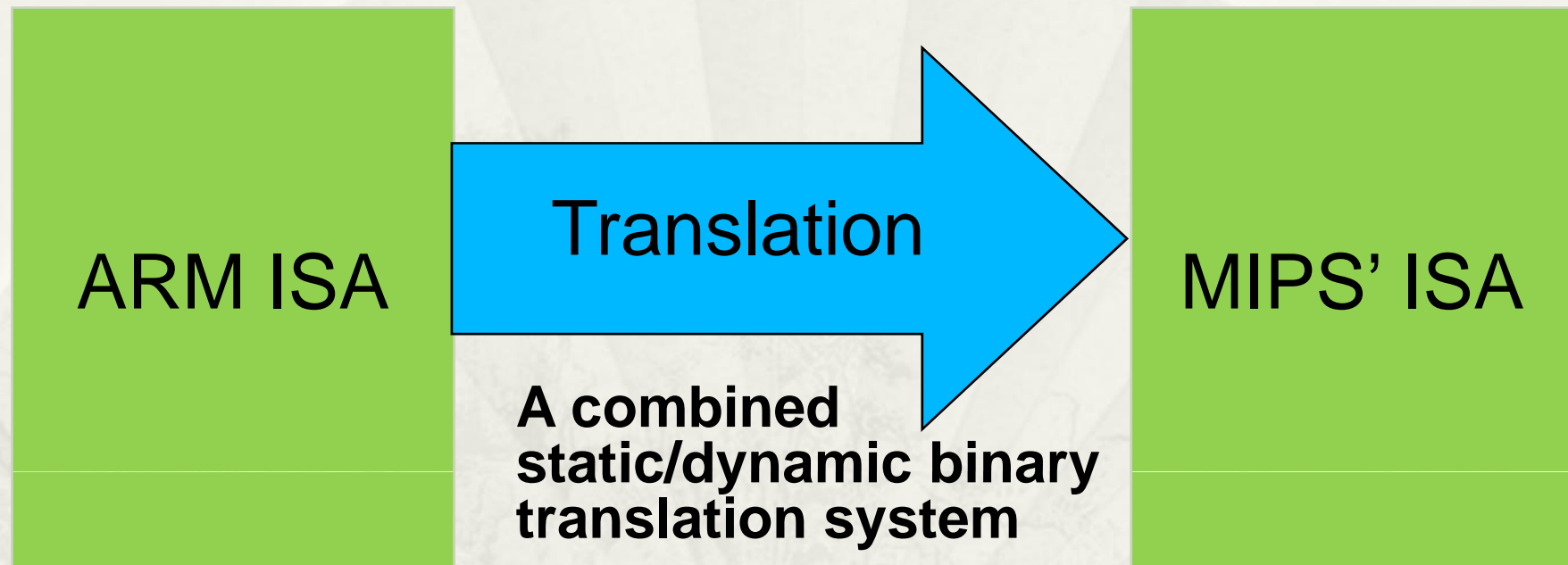
	Advantage	Disadvantage
Dynamic	Robust More adaptive	Slow start-up Less power efficient Cannot afford complex optimizations
Static	Fast start-up Enable more powerful optimizations	Still needs runtime system support Not adaptive

- \* A combined static and dynamic binary translation system may be more effective for embedded systems.
  - \* More power efficient
  - \* Faster start-up



# System Overview

- \* Our goal



# Key challenges

---

- \* Must deal with main architectural differences between ARM and MIPS'.
  - \* PC-relative data access
  - \* Shifter operand & shifter carry out
  - \* Conditional execution
- \* Must generate correct code with very limited semantic information from the executables.
- \* Must perform code optimizations to minimize instruction overhead introduced during binary translation.



# PC-relative data access

---

- \* PC-relative data is embedded in the text section mainly for
  - \* Large immediate
  - \* Immediate field for jump table
- \* The ARM text section is kept in the MIPS' program and the translated code can still reference back to the same address.

**ARM :**

*ldr r1, [pc + offset]*

**MIPS' :**

*mov rt1, armpc*

*ldr r1, [rt1 + offset]*



# Shifter operand& shifter carry out

- \* Shifter operand and shifter carry out are temporary values generated by shifter operands in ARM instructions.
- \* In the translated MIPS' code, additional instructions are needed for different kinds of shift operands in ARM binaries.

**ARM :**

```
add    r0, r1, r2, lsl 2
```

**MIPS' :**

```
sll    r_shiftO, r2, 2    //update shifter operand  
btst  r_shiftC, r2, 30  //update shifter carry out  
add    r0, r1, r_shiftO
```



# Conditional execution : condition code check

---

- \* Almost all ARM instructions are conditionally executed.
- \* Condition flags are used to determine if the condition code were matched.
- \* Checking condition flags in ARM is translated into conditional branch instructions in MIPS'.

**ARM :**

*addeq r0, r1, r2*

**MIPS' :**

*btst rt1, r\_flags, 1*

*beqz rt1, 8*

*add r0, r1, r2*



# Conditional execution : flag update

---

- \* There are 4 condition flags in ARM.  
(Negative(N), Zero(Z), Carry (C), Overflow (V))
- \* One register is preserved to store the flags, an additional shift instruction is needed to update each bit in the register.

***/\*update C flag bit with value in register rt1\*/***

**Use a single register**

```
slli rt1, rt1, 2           //shift the new bit to the current position  
or r_flags, r_flags, rt1 //update the C flag bit
```



# Key challenges

---

- \* Must deal with main architectural differences between ARM and MIPS'.
- \* **Must generate correct code with very limited semantic information from the executables.**
  - \* Architecture register mapping
  - \* Control flow management
- \* Must perform code optimizations to minimize instruction overhead introduced during binary translation.



# Architecture Register mapping

---

- \* Number of visible registers.
  - \* ARM : 16
  - \* MIPS' : 32
- \* The 16 registers in ARM are mapped as follow.
  - \* Registers r0 ~ r11 are direct mapped to MIPS' r0 ~ r11
  - \* Registers r12 ~ r15 are mapped to MIPS' r28 ~ r31
- \* The registers r12 ~ r15 are used as special registers (e.g. PC, LR), so the mapping is preferred due to easily translate of the load/store multiple word instructions.



# Program control management

---

- \* Program control management includes :
  - \* Update ARM PC
  - \* Indirect branch handling
- \* Lazy update to ARM PC
  - \* We only update the ARM PC when it is needed.
- \* Indirect branch handling
  - \* Address mapping table
  - \* Return address stack



# Address mapping table

---

- \* The address mapping table is used to map ARM instruction address to MIPS' instruction address, the entry of the table contains  $\langle \text{ARM\_address}, \text{MIPS}'\_address \rangle$ .
- \* It is a hash table built at translation time.
- \* When an indirect branch is executed, a hashing function stub will be invoked to find the corresponding entry. If the branch target and the ARM PC in the entry were equal, the execution will be directed to the MIPS' PC.
- \* The hash table size used for our benchmark is usually 1K or 2K entries. It takes about 16KB to store the hash table.



# Return address stack

---

- \* Return address stack (RAS) is mainly used for translating function calls and returns to avoid the overhead brought by looking up the address mapping table.
- \* Additional instructions for function call.
  - \* Push next ARM PC and MIPS' PC on top of the RAS
- \* Additional instructions for return.
  - \* Pop both addresses from the top of the RAS
  - \* If the ARM address and the target address in the register were equal, branch to the MIPS' PC.
  - \* Check address mapping table if the check failed



# Key challenges

---

- \* Must deal with main architectural differences between ARM and MIPS'.
- \* Must generate correct code with very limited semantic information from the executables.
- \* **Must perform code optimizations to minimize instruction overhead introduced during binary translation.**
  - \* Check condition code selectively
  - \* Update condition code selectively
  - \* Special condition update/check
  - \* Combine conditional branch
  - \* Register remapping



# Check condition code selectively

- \* A group of instructions with the same condition code can be identified at translation time, and only one branch is needed to skip the entire group.
- \* A similar optimization can also be implemented by identifying a group of instructions with inverse condition codes.

**ARM :**

```
addeq r1, r2, r3  
subeq r4, r5, r6
```

**MIPS':**

```
beqz r_flagZ, 8  
add r1, r2, r3  
beqz r_flagZ, 8  
sub r4, r5, r6
```

**ARM :**

```
addeq r1, r2, r3  
subne r4, r5, r6
```

**MIPS' :**

```
beqz r_flagZ, 8 12  
add r1, r2, r3  
bnez r_flagZ, 8  
sub r4, r5, r6
```



# Update condition flags selectively

- \* Only update the needed flags.
- \* Use control and data flow analysis to determine which flags to update.

## ARM :

```
cmp    r1, r2
addeq  r0, r1, r2
cmp    r1, r2
```

## MIPS' :

```
.....update flag N
.....update flag Z
.....update flag C
.....update flag V
beqz   r_flagZ, 8
add    r0, r1, r2
```



# Special condition code update/check

- \* It is expensive if more than one condition flag is updated.
- \* If there were only one kind of condition code check between two condition code updates, we only need to generate a single update and check for this particular condition.

## ARM :

```
cmp    r1, r2
addge  r4, r5, r6
cmp    r2, r3
```

Only one kind of check  
between two updates

## MIPS' :

```
sllt   r_special, r1, r2
beqz   r_special, r4, r5
beq    r_flagN, r_flagV
add    r4, r5, r6
/*2nd cmp*/
```

Replace the ordinary flag  
update and check with a  
special flag update and  
check



# Combined conditional branch

---

- \* A conditional branch is implemented as a compare instruction with a condition code check branch.
  - \* CMP + condition code branch
  - \* TEQ + condition code branch
  - \* TST + condition code branch
- \* We can translate the pair of instructions into a single conditional branch if there were no condition code check in between two updates.



# Register remapping

---

- \* Storing condition flags in a single register incurs large overhead.
- \* If each flag bit is allocated to a separate register.
  - + Avoid the shift instructions to extract/deposit the specific bit; reduce at least three instructions for each update.
  - Need to use more registers in the MIPS' architecture.
- \* Fortunately, MIPS' has 16 more registers than ARM.



# Experiment Setup

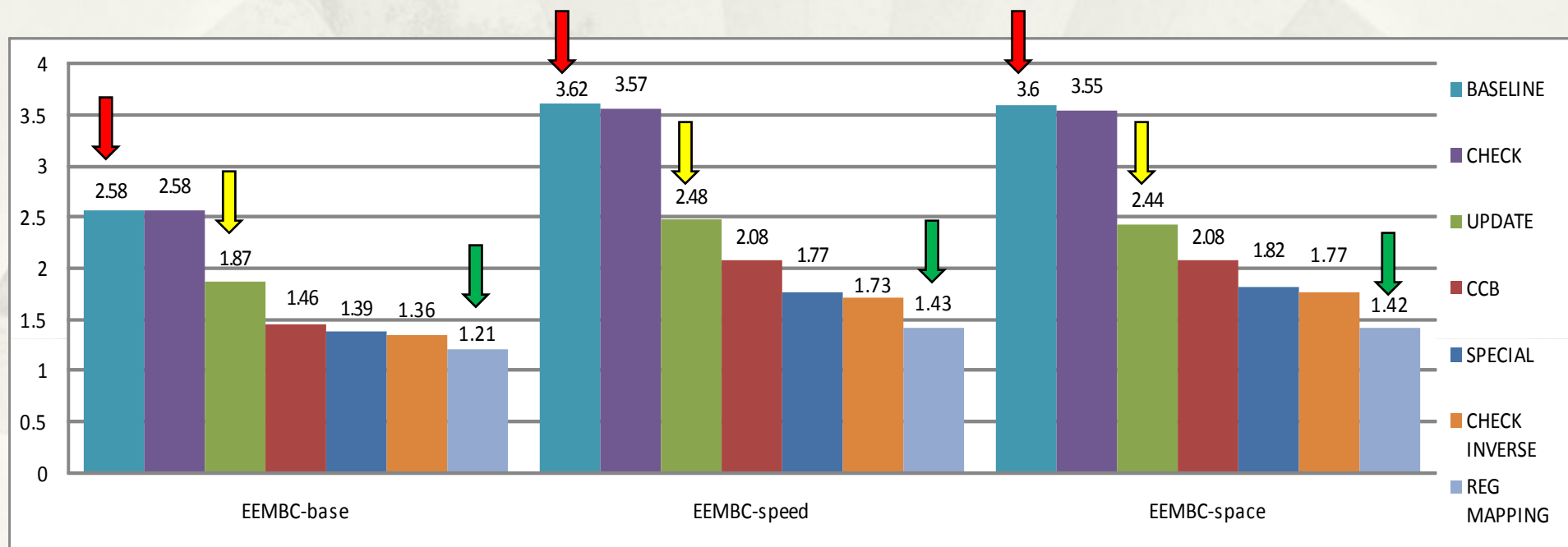
---

- \* Benchmark
  - \* EEMBC benchmark version 1.1
  - \* 55 programs divided into six categories.
- \* Compile the benchmark with three different options
  - \* EEMBC-base : -O0, no optimization.
  - \* EEMBC-speed : -O2, optimization for speed.
  - \* EEMBC-space : -Os, optimization for space.
- \* Simulations
  - \* Instruction path length measurement
    - \* Use ARM-GDB for ARM and SID/MIPS' for MIPS'.
  - \* Execution cycle measurement
    - \* Use SimpleScalar/ARM for ARM and SID/MIPS' for MIPS'



# Experimental results

- \* Measurement of the instruction path length.
- \* Later measures include previous optimizations.

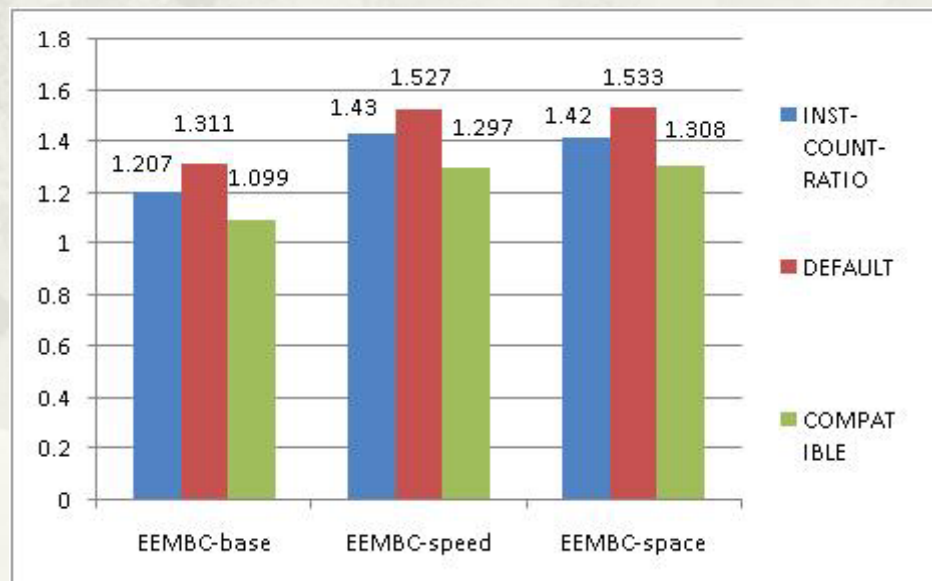


(Y-axis shows the instruction path length ratio)



# Experimental results

- \* Measurement of execution cycles.
- \* Since SimpleScalar/ARM turns some instructions into multiple micro-operations, we use two configurations:
  - \* COMPATIBLE for a similar configuration as MIPS'
  - \* DEFAULT for a configuration that SimpleScalar/ARM suggests.



# Some Interesting Cases

---

- \* The *rotate01\_lite* has a less-than-1 instruction ratio.
  - \* Selectively condition code check make a branch skip several instruction available
  - \* Combined conditional branch can replace two ARM instructions with only one MIPS' instruction.



# Conclusions

---

- \* The baseline binary translation increases the instruction path length by more than three times for the EEMBC benchmark.
- \* With several optimizations introduced, the instruction path length after the static binary translation is increased by less than 35%.
- \* With the set of code optimization techniques, static translating ARM based binaries can be attractive in ISA migration for embedded systems.



---

# Q & A

