Improved Execution Migration

Meet Udeshi, Nirmal Boran

IIT Bombay

July 25, 2018

Introduction

- Venkat et al[1] show energy efficiency and performance gain by Heterogenous ISA CMPs.
- DeVuyst et al[2] explore cost-effective migration strategies.
  - Global variables kept common during compile-time.
  - Heap memory only touched by *malloc* function, same implementation used.
  - Keep functions at same virtual address for same pointers.
  - Stack frame size and local variable location maintained same at compile time by using padding.
  - **Dynamic modification:** Hard-coded stack offset in instructions, function arguments, stack object pointers.
- DeVuyst et al look at only function calls as potential migration points, and suggest using binary translation for immediate shift until a migration point is reached.

[1] A.Venkat, D.Tullsen - Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor

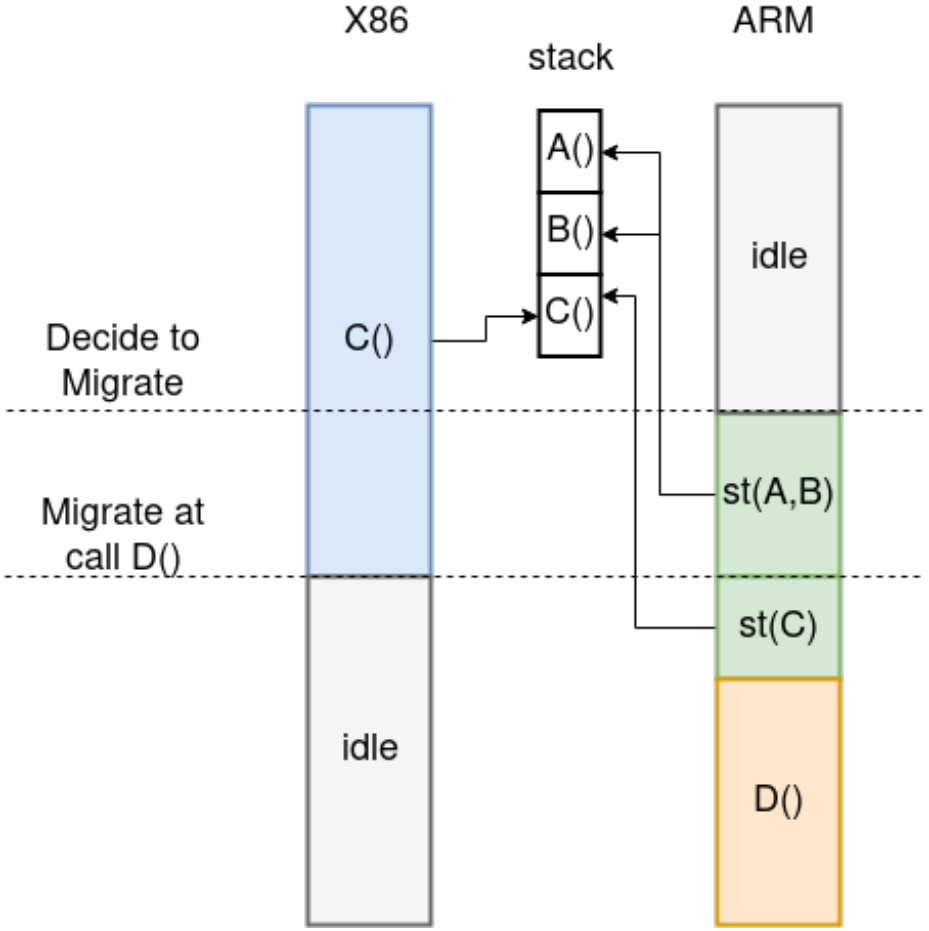
[2] M.DeVuyst et al - Execution Migration in a Heterogeneous-ISA Chip Multiprocessor

Implementation and Challenges

- LLVM compiler (`clang` and `llc`) for analysis of function stack-frames for stack transformation.
- Mapping variables in C code, or even LLVM-IR to stack slots is difficult when using `-O1` or higher. Resort to compiling with `-O0`.
- Callee-saved spilled register mapping depends heavily on live variables at call-site.
- Pointer values in callee have to be changed when caller stack is transformed. Mapping pointer to variable is difficult, because a function can be passed pointers by multiple functions.

Simultaneous transformation

Only last frame on the stack (current function) is currently in use. Start transformation of other frames simultaneously on other processor before migration point is reached.



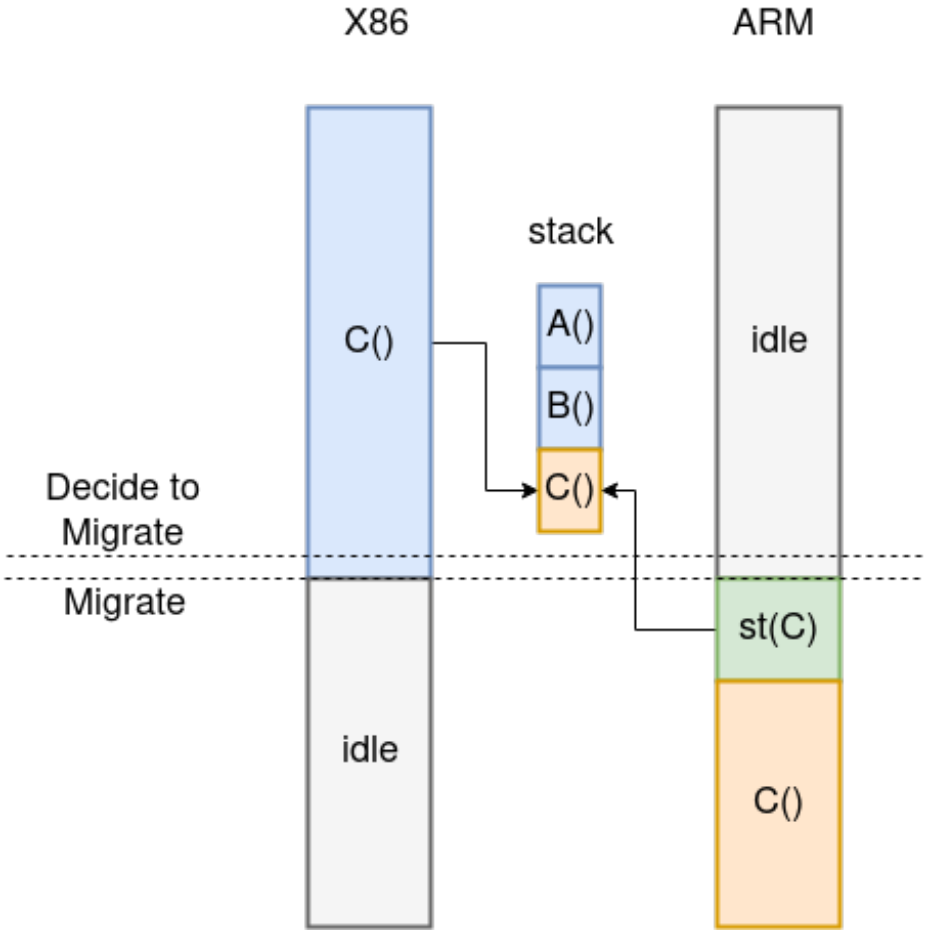
Simultaneous transformation

Problem:

- Stack transformer implementation given in paper requires two-pass transformation
- First pass is frame-wise and can be handled in parallel.
- Second pass fixes corner-case of pointers and does register mapping in between ISA based on live variables.

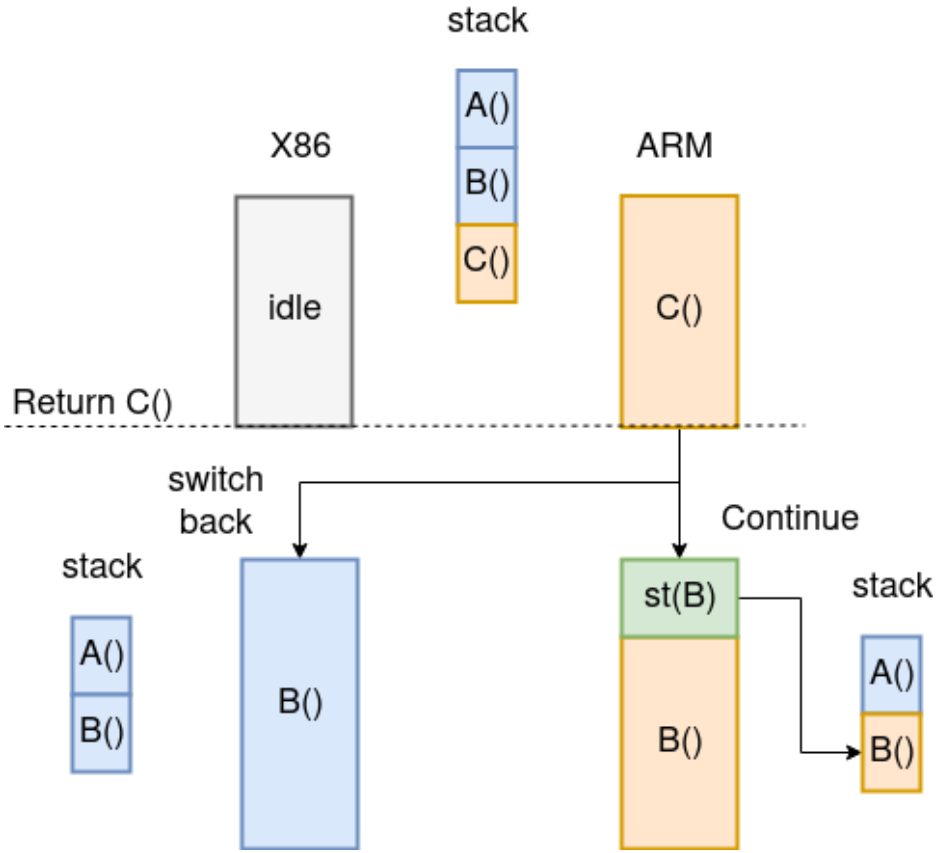
Single frame transformation

Only transform the stack frame of current function.



Single frame transformation

When returning to previous function, decide whether to migrate to current processor or switch to previous processor



Single frame transformation

Advantages:

- Pointer problem solved. Pointer value not changed because previous stack not modified.

Problems:

- Live variables in registers would have to be moved twice, or added to the stack.
- Migration on function return, would require live variable analysis.

Transformation pseudo-code

Function: `BZ2_bzDecompressInit(bz_stream*, int, int)`

Slot	size	align	location
x86:			
fi.0	4	4	[SP-28]
fi.1	8	8	[SP-24]
fi.2	4	4	[SP-36]
fi.3	4	4	[SP-32]
fi.4	8	8	[SP-16]
ARM:			
fi.0	4	4	[SP-20]
fi.1	8	8	[SP-32]
fi.2	4	4	[SP-36]
fi.3	4	4	[SP-40]
fi.4	8	8	[SP-48]

```
tmp1 = frame[loc[1]]
frame[loc[1]] = frame[loc[0]]

for i=2..len(loc):
    tmp2 = frame[loc[i]]
    frame[loc[i]] = tmp1
    tmp1 = tmp2
```

Pointer Analysis

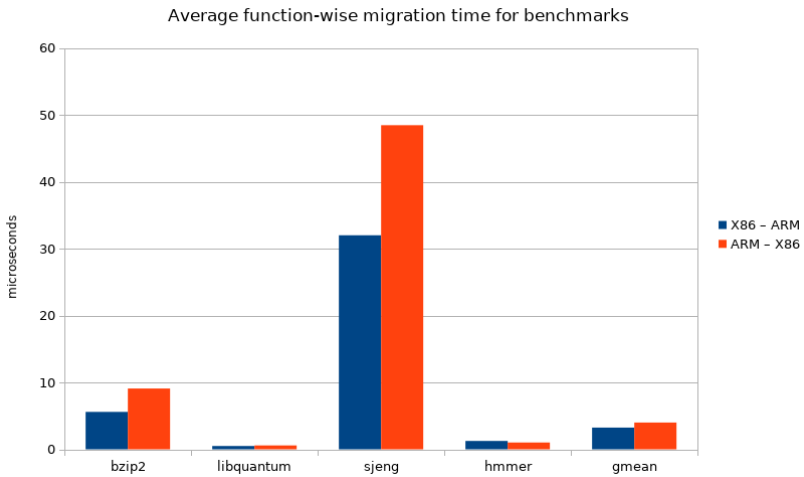
```
define @functionA
  %2 = alloca i32
  %3 = alloca i64
     .
     .
  %51 = load i64, %3 i64*
  %52 = call @functionB(%51 i64, %2 i32*)
```

In LLVM-IR, alloca instruction return the pointer to stack variable, which is kept in the first few SSA variables of the function. We can determine these SSA variables by looking for alloca instructions. Thus, use of these variables in function calls would mean a pointer to the stack is being passed to child function.
Only 5 such cases were found in the entire bzip2 package.

Simulation method

- Migration between X86_64 and Aarch64 ISA
- Implemented migration code in `C` as `migrate()` function
- `migrate()` call is inserted at end of (almost) every function in codebase
- Used `gem5` to calculate time of execution for `migrate()`

Simulation Results



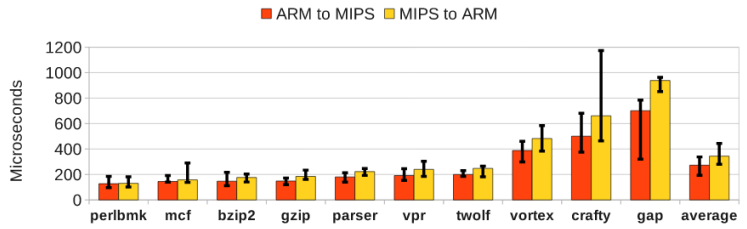


Figure 2. The average costs of state transformation for migration. Lines indicate the minimum and maximum measured transformation times.

The End