18-747 Lecture 22: Binary-to-Binary Translation

James C. Hoe
Dept of ECE, CMU
November 22, 2001

Reading Assignments: *paper on Transmeta*
"Dynamic Binary Translation and Optimization",
by E. R. Altman, K. Ebcioğlu, IBM Research

Announcements: Project 3 Short Proposal due Wednesday November 21st

Handouts: *Handout #13.A: Corrected Project 3 Description*

What is Binary Translation

- Taking a binary executable from a *source ISA* and generate a new executable in a *target ISA* such that the new executable has exactly the same *functional* behavior as the original
- Same ISA ⇒ *Optimization*
  - compiler instruction scheduling is a restricted form of translation
  - re-optimizing old binaries for new, but ISA-compatible, hardware

  Reoptimization can improve performance regardless whether implementation details are exposed by the ISA
- Across ISAs ⇒ Overcoming binary compatibility
  - two processors are "binary compatible" if they can run the same set of binaries (from BIOS to OS to applications)
  - Strong economic incentive

  How to get all of the popular software to run on my new processor?

  How to get my software to run on all of the popular processors?

What is so hard about it?

- It is always possible to "interpret" an executable from any ISA on a machine of any ISA

  Turing machine simulation

- But, naïve interpreters incur a lot of overhead and thus run slower and use more memory

  think of SimpleScalar as a PISA interpreter for PCs

- Binary translation is not interpretation
  - emits new binaries that runs *natively* on the target ISA processor
  - can be very difficult if the source ISA (e.g x86) or the source executable (e.g. hand-crafted assembly code) is not nice

Without the high-level source code, you can't always statically tell what an executable is going to do

Some Hard Problems in Translation

- Day-to-day problems
  - Floating-point representation and operations
  - Precise exceptions and interrupts
- More obscure problems

  (Executables compiled from high-level languages tend not to have these kind of problems)
  - Self-modifying code

    A program can construct an instruction (in the old format) as a data word, store it to memory, and jump to it
  - Self-referential code

    A program can checksum the code segment and compare it against a stored value based on the original executable
  - Register Indirect jumps to computed addresses

    A program might compute a jump target that is only appropriate for the original binary format and layout

    A program can jump to the middle of an x86 inst on purpose

Static vs. Dynamic Translation

- Static
  - \+ May have source information (or at least have object code)
  - \+ Can spend as much time as you need (days to months)
  - \- Isn't always safe or possible
  - \- Not transparent to users
- Dynamic
  - \- Translation time is part of program execution time
    - ⇒ Can't do very complex analysis / optimization
    - ⇒ Infrequently used code sections cost as much to translate as frequently used code sections
  - \- No source-level information
  - \+ Has runtime information (dynamic profiling and optimization)
  - \+ Can fall back to interpretation if all else fails
  - \+ Can be completely transparent to users

How can binary translation be used?

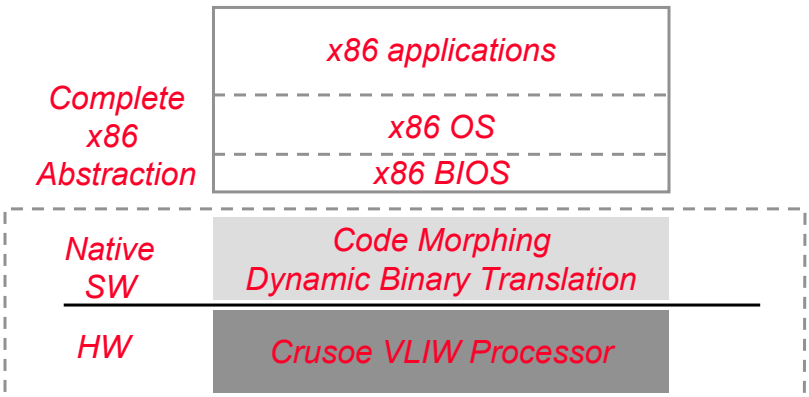
- Porting old software to new platforms (static, different-ISA)
  - e.g translator from DEC VAX to Alpha
- Binary Augmentations (static, same-ISA)
  - localized modifications to shrink-wrap binaries without sources
    - e.g. inserting profiling code, simple optimizations
- Dynamic Code Optimizations (dynamic, same-ISA)
  - profile an execution and dynamically modify the executable using techniques such as trace scheduling, e.g. HP Dynamo
- Cross-platform execution (dynamic, different-ISA)
  - using a combination of interpretation and translation to very efficiently emulate a different (often nasty) ISA
    - e.g. Transmeta Crusoe and Code Morphing
- Efficient Virtual Machines (dynamic, different-ISA)
  - using a combination of interpretation and translation to very efficiently emulate a different (nice-by-design) ISA
    - e.g. Java virtual machines and JIT (Just-in-Time) compilation

A New Way to Think about Architecture

- Architecture = dyn. translation + hardware implementation
  - no problem of *forward or backward binary compatibility*
    backward compatible processor: don't need new software
    forward compatible processor: don't need new processors
  - don't need increasingly fancy HW to speedup an old ISA
  - both the translator and HW can be upgraded or repaired with very little disruption to the users
- Processors (and systems) becomes commodity items (like DRAM)
  - processors can become very simple but very fast
  - slightly defective processors can still be sold with workarounds
- Old platforms and software can be cost-effectively revived and maintained forever

  "Amiga" is coming back as a soft architecture!

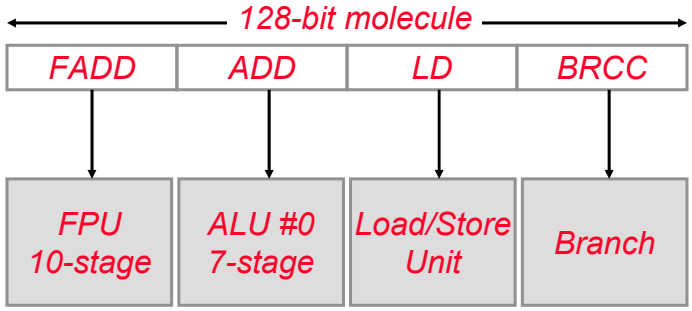
Transmeta Crusoe & Code Morphing



- Crusoe boots "Code Morpher" from ROM at power-up
- Crusoe+Code Morphing == x86 processor

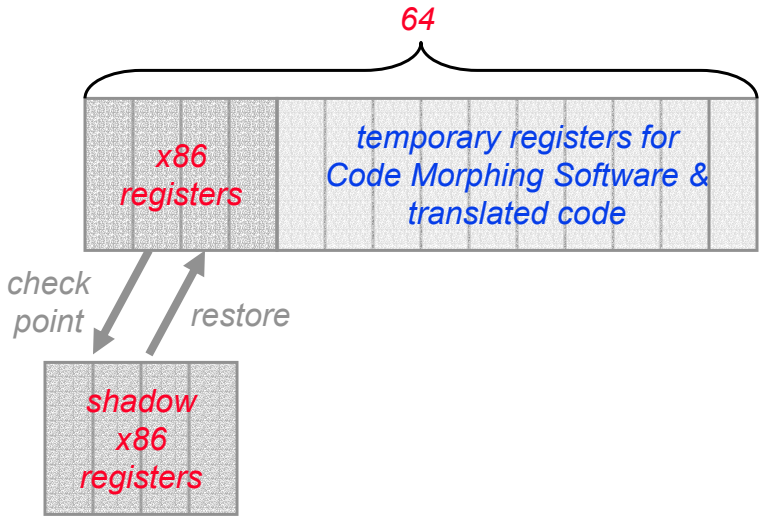
  x86 software (including BIOS) cannot tell the difference

Crusoe VLIW Processor

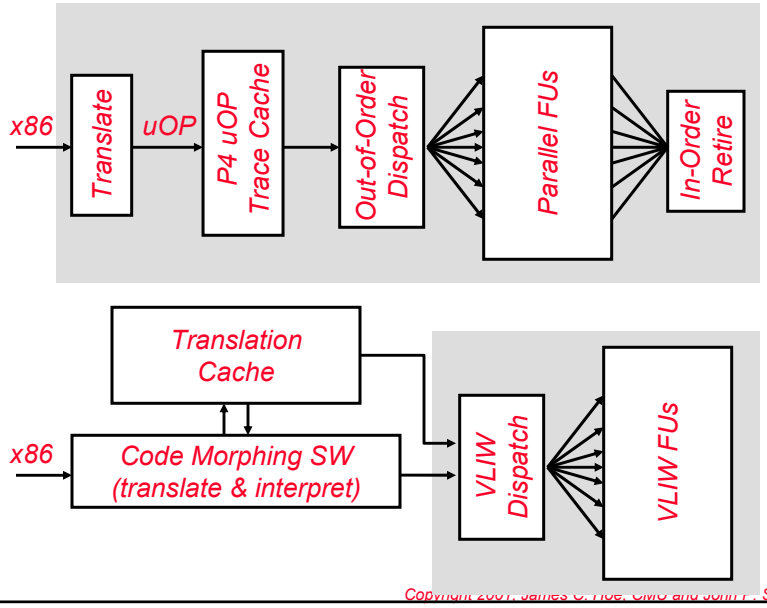


- 64 or 128-bit molecules directly control the in-order VLIW pipeline (no dependence within a molecule)
- 1 FPU, 2 ALU, 1 LSU, and 1 BU
- 64 integer GPRs, 32 FPRs + shadow x86 regs
- No hardware renaming or reordering
- Same cond. code, floating-point, and TLB format as x86

Register Files



Executing x86 to as uOPs or atoms



Code Morphing Software (CMS)

- The only software written natively for Crusoe processors
  - begins execution at power-up
  - fetches previously unseen x86 basic block from memory
  - translates a block of x86 instructions at a time into Crusoe VLIW
  - caches the translation for future use
  - jumps to the generated Crusoe code for execution, execution can continue directly into other blocks if translation is cached
  - regains control when execution reaches a unknown basic block
  - interprets the execution of “unsafe” x86 instructions
  - retranslates a block after collecting profiling information
- CMS uses a separate region of memory that cannot be touched by code translated from x86
- Crusoe processors do not need to be binary compatible between generations

  ⇒ *can make different design trade-offs but needs a new translator with a new processor*

Cost of Translation

- Translation time is part of execution time!

  Translation cost has to be amortized over repeat use

- 1st pass translation must be fast and safe
  - almost like interpretation
  - x86 instructions are examined and translated byte-by-byte
  - CMS constructs a function that is equivalent to the basic block
  - CMS jumps to the function and regain control when the fxn returns
  - collects statistics, i.e. execution frequency, branch histories
- Re-translate an often "repeated" basic block *(after ~50 times)*
  - examines execution profile
  - applies full-blown analysis and optimization
  - builds inlined Crusoe code that can run directly out of the translation cache without intervention by CMS
  - can do cross-basic block optimizations, such as speculative code motion and trace scheduling
- Caches translation for reuse to amortize translation cost

Example of a Translation

x86 Binary Code

```
A:  addl  %eax,  (%esp)     // load data from stack, add to %eax
B:  addl  %ebx,  (%esp)     // load data from stack, add to %ebx
C:  movl  %esi,  (%ebp)     // load from mem (%ebp) into %esi
D:  subl  %ecx,  5          // subtract 5 from %ecx
```

literal translation

1st Pass Sequential Crusoe Atoms

```
ld      %r30,   [%esp]          // A:  load data from stack, save to temp
add.c   %eax, %eax, %r30        //     add to %eax, set condition code

ld      %r31,   [%esp]          // B:  load data from stack, save to temp
add.c   %ebx, %ebx, %r31        //     add to %ebx, set condition code

ld      %esi,   [%ebp]          // C: load from mem (%ebp) into %esi

sub.c   %ecx, %ecx, 5           // D: subtract 5 from %ecx
```

Example of an Optimization

1st Pass Sequential Crusoe Atoms

```
ld      %r30,   [%esp]
add.c   %eax, %eax, %r30        // cc is never tested
ld      %r31,   [%esp]          // %r31 and %r30 are common sub-expr
add.c   %ebx, %ebx, %r31        // cc is never tested
ld      %esi,   [%ebp]
sub.c   %ecx, %ecx, 5
```

basic optimizations

2nd Pass Optimized Crusoe Atoms

```
ld      %r30,   [%esp]          // [%esp] is loaded once and reused
add     %eax, %eax, %r30        // don't need to set condition code
add     %ebx, %ebx, %r30        // don't need to set condition code
ld      %esi,   [%ebp]
sub.c   %ecx, %ecx, 5
```

Optimizations include common sub-expr elimination, dead-code elimination (include unnecessary cc), loop invariant removal, etc. (see L16 for more)

Example of Scheduling

2nd Pass Optimized Crusoe Atoms

```
ld      %r30,   [%esp]
add     %eax, %eax, %r30
add     %ebx, %ebx, %r30
ld      %esi,   [%ebp]
sub.c   %ecx, %ecx, 5
```

VLIW Scheduling (see L17&18)

Final Pass Scheduled Crusoe Molecules

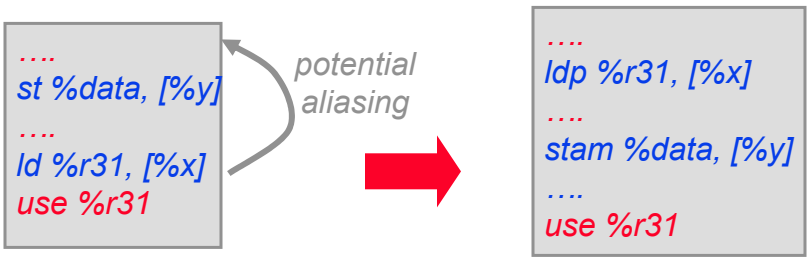
```
{ ld  %r30,   [%esp] ;   sub.c  %ecx, %ecx, 5 }
{ ld  %esi,   [%ebp] ;   add    %eax, %eax, %r30 ;   add %ebx, %ebx, %r30 }
```

In-order execution of scheduled molecules on a Crusoe processor mimics the dynamic superscalar execution of uOPs in Pentium's

Branch Prediction

- Static prediction based on dynamic profiling
- Translation can favor the more frequent traversed arm of an *if-then-else* statement by making that arm the fall through (not-taken) path
- Trace scheduling
  - construct traces such that the most frequently traversed control flow paths encounters no branches at all
  - enlarged scoped of ILP scheduling
  - needs compensation code when falling off trace
- "select" instruction
  - "SEL CC, Rd, Rs, Rt" means if (CC) Rd=Rs else Rd=Rt
  - a limited variant of predicated execution
  - supports if-conversion, *i.e change control-flow to data-flow*

Detecting Load/Store Aliasing



- *ld-and-protect* records the location and size of the load
- *store-under-alias-mask* checks aliasing against the region protected by *ldp*
- if *stam* discovers a conflict, it triggers an exception so CMS can "discard" the effects of this basic block and re-run a different translation that does not have the load and store reordered

Eliminating Repeated Loads

```
ld %r30, [%x]
….
st %data, [%y]
….
ld %r31, [%x]
use %r31
```

```
….
ldp %r30, [%x]
….
stam %data, [%y]
….
use %r30
```

- Due to limited number of ISA regs, x86 programs keep most variables on the stack

  ⇒ *the same* value is reloaded from stack for each use

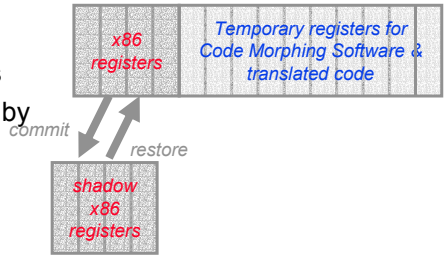
  (there isn't a spare x86 ISA register to hold it between use)
- CMS detects repeated load from the same address as common sub-expression and holds a value in a temporary register for reuse
- A store in between the loads can make the optimization unsafe
- *stam* allows CMS to optimize for the common case

Precise Exception Handling

- CMS and Crusoe must emulate x86 behavior exactly, including precise exception
- But, an x86 instruction maps to several atoms and can be reordered with atoms of other x86 instructions and can be dispersed over a large code block after optimization and scheduling
- Solution *(assumes exceptions are rare)*
  - check point x86 machine state at the start of every translated block
  - if execution reaches the end of the block without exception then continue to the next block
  - if exceptions is triggered in the middle of a block, CMS restores x86 machine state from check point and reruns the same block by "interpreting" the original x86 code, one instruction at a time

Check Pointing x86 Machine State

- Register File
  - a special "commit" instruction makes a copy of x86 register contents in the shadow registers
  - shadow registers is not touched by program execution
  - "restore" restores the shadowed values



- Gated Store Buffer
  - all stores are intercepted and held in a special buffer
  - after a commit point, all earlier gated stores are released to update cache or memory as appropriate *(Note: not all at once!!)*
  - If a restore event is triggered, the content of the gated store buffer is discarded
    - *After a commit, any earlier effects cannot be undone*
    - *An restore returns x86 machine state to the last commit point*

Performance of Transmeta's "x86"

- Execution Time
  - Comparable to direct hardware implementation by Intel or AMD
  - TM5400 at 667 MHz is about the same as a Pentium III running at 500MHz

  Unamortized translation cost leads to lower benchmark results
- Low Cost
  - Much simpler hardware

    TM5400 is a about 7 million transistors (P4 is at 41 Million)
  - Easier to design, more scalable, easier to reach high clock rate, more room for caches, better yield, etc
  - Doesn't have to worry about binary compatibility!!
- Low Power
  - less hardware $\Rightarrow$ lower power
  - Additional power management features (such as variable supply voltage and clock frequency)