Lecture: 10

VLIW & Dynamic Binary Translation

Department of Electrical Engineering
Stanford University

http://eeclass.stanford.edu/ee382a

Announcements

- HW2 is due today
  - We will post the solutions tonight

- Review session on Friday 10/23, 2-3pm, Gates 498
  - Review superscalar & VLIW techniques
  - Review HW2 solutions

Review: VLIW

- Long instruction words (or packets or bundles)
  - Each word contains multiple operations

- No data dependencies between operations in a word (parallelism)
  - No need for RAW checks

- Each operation slot corresponds to specific functional unit
  - Operation latencies are typically fixed

- Words spaced apart statically (using nops)
  - All operations are ready to execute

Review: Realistic VLIW Design using Clustering

Multi-Ported Register File

FAdd (1 cycle)
FMul 4 cyc pipe
FMul 4 cyc unpipe
FDiv 16 cycle

Instruction Memory
Condition Codes

Sequencer

How would you deal with cache misses?
Review: VLIW Today

- Servers: Intel IA-64 architecture
  - EPIC ISA (explicitly parallel instruction computing)
  - Chips Merced, Madison, Montecito, Tukwilla, ...
  - Questionable success compared to superscalars

- Embedded: TI, NXP, ST, ...
  - Very successful in low-end and high-end embedded SOCs
  - Rational: good performance/power, no need for binary compatibility
  - Large variety of ISAs, designs, optimizations points

- Interesting VLIWs of the recent past
  - Transmeta Crusoe (x86 on VLIW)

IA-64 Architecture

- 128 general-purpose registers
- 128 floating-point registers
- Arbitrary number of functional units
- Arbitrary latencies on the functional units
- Arbitrary number of memory ports
- Arbitrary implementation of the memory hierarchy

Needs retargetable compiler and recompilation to achieve maximum program performance on different IA-64 implementations

IA-64 vs. Classic VLIW

- Similarities:
  - Compiler generated wide instructions with ILP encoded in the binary
  - Static detection of dependencies
  - Large number of architected registers

- Differences:
  - Instructions in a bundle can have dependencies
  - Hardware interlocks between dependent instructions
  - Accommodates varying number of functional units and latencies
  - Allows dynamic scheduling and functional unit binding
    - Static scheduling are “suggestive” rather than absolute
  - Code compatibility across generations
    - but software won’t run at top speed until it is recompiled so “shrink-wrap binary” might need to include multiple builds

IA-64 Instruction Format

- IA-64 “Bundle”
  - Total of 128 bits
  - Contains three IA-64 instructions (aka syllables)
  - Template bits in each bundle specify dependencies both within a bundle as well as between sequential bundles
  - A collection of independent bundles forms a “group”
    - A more efficient and flexible way to encode ILP then a fixed VLIW format

\[ \text{inst}_1 \quad \text{inst}_2 \quad \text{inst}_3 \quad \text{temp} \]

- IA-64 Instruction
  - Fixed-length 40 bits long
  - Contains three 7-bit register specifiers
  - Contains a 6-bit field for specifying one of the 64 one-bit predicate registers
Interesting Features of IA64

- Predicated execution
- Speculative, non-faulting Load instruction
- Software-assisted branch prediction
- Register stack
- Rotating register frame
- Software-assisted memory hierarchy

Speculative, Non-Faulting Load

- `ld.s` fetches speculatively from memory
  - i.e. any exception due to `ld.s` is suppressed
- If `ld.s r` did not cause an exception then `chk.s r` is an NOP, else a branch is taken (to some compensation code)

Speculative, Non-Faulting Load

- Speculatively load data can be consumed prior to check
  - “speculation” status is propagated with speculated data
- Any instruction that uses a speculative result also becomes speculative itself (i.e. suppressed exceptions)
- `chk.s` checks the entire dataflow sequence for exceptions
Speculative “Advanced” Load

- \textit{ld.a} starts the monitoring of any store to the same address as the advanced load
- If no aliasing has occurred since \textit{ld.a, ld.c} is a NOP
- If aliasing has occurred, \textit{ld.c} re-loads from memory

Using Speculative Load Results

Branch Prediction

- Static branch hints can be encoded with every branch
  - \textit{taken} vs. \textit{not-taken}
  - whether to allocate an entry in the dynamic BP hardware
- SW and HW has joint control of BP hardware
  - “brp” (branch prediction) instruction can be issued ahead of the actual branch to preset the contents of BPT and BTAC
    - Itanium uses a 512-entry 2-level BPT and 64-entry BTAC
- TAR (Target Address Register)
  - a small, fully-associative BTAC-like structure
  - contents are controlled entirely by a “prepare-to-branch” inst.
    - a hit in TAR overrides all other predictions
- RSB (Return Address Stack)
  - Procedure return addr is pushed (or popped) when a procedure is called (or when it returns)
  - Predicts nPC when executing register-indirect branches

Register Renaming

- 128 general purpose physical integer registers
- Register names R0 to R31 are static and refer to the first 32 physical GPRs
- Register names R32 to R127 are known as “rotating registers” and are renamed onto the remaining 96 physical registers by an offset
- Remapping wraps around the rotating registers such that when offset is non-zero, physical location of R127 is just below R32
Register Stack for Procedure Calls

On a procedure call, the rename offset is bumped to the beginning of output argument registers
Callee can then allocate its own working frame (up to 96 regs)
If there isn’t enough free regs to be allocated, HW automatically frees up space by spilling life contents not in the current frame to memory

• 6 wide instruction fetch and issue
  - 6 wide integer, 2 wide FP, 4 wide ld/st, 3 wide branch
  - 1 cycle L1 data cache
  - 8-stage pipeline
  - 2-threads

Memory hierarchy
- Separate L1I, L1D, L2I, and L2D
- 6MB L3
- In 4-core Tuckwilla: 30MB L3!
DBT: Dynamic Binary Translation

- Up to now, CPU hardware design
  - Pipelining, superscalar, speculative execution, ...
  - ISA to interface hardware features (VLIW)

- What is dynamic binary translation?
  - Software tool to translate a binary code to another code at runtime

- How does dynamic binary translation work?
  - Internals, DBT APIs, ...

- Use cases
  - Binary compatibility, profiling, optimizations, debugging, ...

What is binary translation?

- Translating programs in one binary format to another

- What: different types of translation
  - Different ISAs
    - E.g. PowerPC => x86: to port programs across platform
  - Same ISAs
    - E.g. x86 => x86: code optimization or feature instrumentation

- Intermediate Representation
  - E.g. Java bytecode => x86: to avoid interpretation

- When
  - Static: translation before running programs
  - Dynamic: translation while running programs

Dynamic Binary Translation Overview

- Dynamic program modifier
  - Start with interpretation or intercept program execution
  - Observe a sequence of instructions
  - Produce new code and save in code cache
    - Change the jump/branch target address properly
  - Manipulate or add instructions as needed

DBT Configurations (1)

- Cross platform
  - E.g. Crusoe (Transmeta)
- Same platform
  - E.g. Dynamo (HP), PIN (x86), Dynamo-Rio(x86)
DBT Configurations (2)

- Virtual Machine
  - E.g. ESX server (vmWare)

- JIT compilation
  - E.g. JVM (Sun), C#(MS)

---

Why is DBT a good idea?

- Feature support without hardware/source-code modification
  - E.g. Binary compatibility, virtual machine, ...

- Vs. static binary translation
  - Access to complete program
    - Programs are fully linked
  - No need to re-link
    - Translate instruction and jump to it
  - Access to program state
    - Include dynamic values
    - Handle self-modifying code
    - Can adapt to changes in program behavior

---

How does it work?

- Combination of DBT framework + tool
- DBT framework
  - Analyze program
    - Instruction type and arguments
    - Basic block and control flow
    - Register and memory values
  - Provide translation primitives
    - Analyze instruction
    - Add/delete instruction
    - Change control flow
    - Read and write register values
    - Read and write memory values
  - Optimize the translated code

---

How does it work? (2)

- DBT tool
  - Perform actual translation
    - Provide two types of routine
      - Translation(instrumentation) routine : when and what to translate
      - Analysis routine : actual instructions to be translated
  - Add metadata for translation/analysis routine
**DBT Framework: PIN for x86**

Pintool

Address Space

PIN

Instrumentation APIs

Virtual Machine (VM)

JIT Compiler

Emulation Unit

Operating System

Hardware

**Application and DBT tool: Instruction Counter**

- Instrumented Applications
  - Instrumentation point
    - Before instruction
    - After instruction
    - Anywhere
      - For branch, fault-through or taken
      - Will be optimized for performance (e.g. register spill)
  - Instrumentation granularity
    - Instruction
    - Basic block
      - Single entry, single exit
    - Trace
      - A group of basic blocks executed back-to-back
  - Analysis routine signature
    - To pass useful information to instrumented function

- DBT tool (PIN style)
  - When and how to translate
    - \[ \text{counter}++; \]
    - \[ \text{sub} \ 0xff, \ %edx \]
    - \[ \text{counter}++; \]
    - \[ \text{cmp} \ %esi, \ %edx \]
    - \[ \text{counter}++; \]
    - \[ \text{jle} <L1> \]
    - \[ \text{counter}++; \]
    - \[ \text{mov} \ 0x1, \ %edi \]
    - \[ \text{counter}++; \]
    - \[ \text{add} \ 0x10, \ %eax \]

- Translation Loop in VM

  - No
  - Hot Path?
    - Yes
      - Translate with DBT tool
    - No
      - End of block?
        - Yes
        - No
        - End of block?
Code Cache

• Software cache in virtual address space
  – Keep translated code in memory for later reuse
  – Essential to leverage the high cost of translation and of good optimizations
  – Trade-off: cost of memory vs. higher reuse

• Allocation policy
  – Remember every translated blocks: doubling code size at least
  – Pick up hot path: temporal locality

• Granularity
  – Basic block
    • Easy, but frequent context switch between interpretation and direct execution
  – Trace
    • Amortize context switch overhead at the cost of path profiling

Linking Translated Basic Blocks (1)

• Context-switch overhead between interpretation and translation
  – Additional instructions for context switch
  – Register Spilling
  – Limited DBT code optimization boundary
    • E.g. Constant propagation, ILP scheduling, loop unrolling, ...

• Link translated code
  – Block reordering for direct branch
  – Indirect branch target (IBT) table for indirect branch
    • (key : value) = (original target, translated target)

• Inline preferred target
  – Inline the block of the preferred target
  – Add check code

Linking Translated Basic Blocks (2)

Exceptions

• Asynchronous exceptions (interrupts)
  – Can be delayed, easy
  – Wait until the current translated code finishes
  – Translate exception handler
  – Invoke translated exception handler

• Synchronous exception
  – During interpretation, no problem
  – During executing translated code, either revert the instruction execution and interpret
    • E.g. checkpoint support in Crusoe’s VLIW hardware
  – Or make sure to stop at exact point
    • E.g NullpointerException in Java
  – Invoke translated exception handler
Self-modifying code

• Solution 1: stick to interpretation
  – Modifying code located in heap
  – Always interpret when jumping to heap address

• Solution 2: invalidate translated code
  – When jumping to heap address, translate the code and cache it
  – Write-protect the pages with modifying code
  – Execute translated code
  – If the code changes later, page-fault exception is triggered
  – Next time, the code is re-translated

Example Use for Binary Compatibility: QuickTransit (Transitive)

• Goal: Run applications for Solaris/SPARC on Linux/x86
• Problem: Migrations are hard
  – Often 50% of legacy software system should be decommissioned
• Use DBT

Example Use for Architecture Study: Cache Simulation (1)

• Cache simulation
  – Have cache model in virtual memory
  – Instrument code to call cache model for each read/write memory instruction
  – Profile cache accesses

Architecture Study: Cache Simulation (2)

Cache_t cacheHierarchy[NUM_CORE][NUM_LEVEL];

// Instrumentation routine
Void Instruction (INS ins, void* param) {
  if (INS_IsMemoryRead(ins)) {
    INS_InsertCall (cacheAccess, BEFORE, INT, param, BOOL, false); }
  if (INS_IsMemoryWrite(ins)) {
    INS_InsertCall (cacheAccess, BEFORE, INT, param, BOOL, true); }}

// Analysis routine
Void cacheAccess (Int address, bool isWrite) {
  // access cacheHierarchy}

Example Use for Architecture Study:
Cache Simulation (1)

Invoke Analysis/Instrumentation Routine

Application
DBT Framework

DBT API

Cache DBT Tool

Memory Address

Cache Model

// Analysis routine
Void cacheAccess (Int address, bool isWrite) {
  // access cacheHierarchy}
Example Use for Debugging: memCheck (Valgrind)

- Valgrind: Heavy DBT
  - Prefer functionality than performance
    - Debugging, profiling, ...
  - Support shadow memory
    - Maintains duplicate copy of every memory bytes and registers used by applications
- Memcheck
  - Help debugging memory errors
    - Memory leak, access to freed or un-initialized data
  - Per memory byte
    - A (addressability) bit: if set, application can reference the byte
    - V (validity) bit: if set, the data is defined
  - Heap blocks for all live memory blocks

Example Use for Dynamic Code Optimization: Dynamo

- Dynamic optimization by software
  - Dynamic: leverage runtime information (compared with compiler)
    - Software: flexible, sophisticated (compared with out-of-order execution)
      - But, with time constraints
  - Runtime information
    - Trace, DLL, function call counter, dynamic values
  - Optimizations
    - Superblock: allows additional chance for classic optimization
      - ILP scheduling, copy propagation, loop unrolling
    - Inlining: balance code size and function call overhead
    - Fast shared library invocation: remove lookup overhead
    - Register allocation: reduce register spill

DBT Overheads

- DBT framework
  - Translation (instrumentation) overhead
    - Code cache, linking translated code, and hot trace selection
- DBT tool
  - Analysis overhead
    - Runtime overhead from instrumented instructions
  - Work done in analysis routine
  - Frequency to analysis routine
  - Transition to analysis routine

Reducing Analysis Routine Overhead

// Instrumentation routine
void Instruction (INS ins, void *void) {
  BBL_InsertCall (cnt, BEFORE, ARG INT BBL_NumIns );
}

// Analysis routine
void cnt (int c ) {
  counter +=c; }

- Shift computation from analysis routine to instrumentation routine
  - E.g. counting the number of instructions in analysis routine
- Code Optimization by DBT
  - Redundant code elimination, register renaming, …
Reducing Frequency of Calling Analysis Routine

Counter += 5;
sub $0xff, %edx
cmp %esi, %edx
jle <L1>
mov $0x1, %edi
add $0x10, %eax
L1: Counter -= 2;

- Instrument at larger granularity
  - Instruction < basic block < trace

Reducing Analysis Transition Overhead

Int icount = 0;

Void samplePrint (void* ip) {
icount--;
if (icount % 1000) {
  fprintf (trace, “%p\n”, ip);
}
}

Yes Inlined

Void countDown () {
icount--;
return icount % 1000;
}

No Inlined

Void Print (void* ip) {
  fprintf (trace, “%p\n”, ip);
}

- Conditional Inlining
  - Separate if-then and if-else functions
- Instrumentation Scheduling
  - Use ANYWHERE for better register spilling

Summary

- Dynamic binary translation is a powerful software tool
  - Translate binary code to another
  - Same ISAs, different ISAs, IRs
- Use cases
  - Binary compatibility, profiling, architecture study, debugging, virtual machine, security, reliability, …
- Performance optimization
  - Code cache, linking translated blocks, hot trace selection
  - Software techniques for efficient DBT tool
  - Architectural support
- A attractive solution to provide new features without hardware/source-code modification