18-640 Foundations of Computer Architecture

Lecture 11:
"Dynamic Code Translation & Compilation"

Mauricio Breternitz, Ph.D., AMD Fellow
October 2, 2014

➢Recommended Reading Assignment:
- "Dynamic Binary Modification: Tools, Techniques, and Applications" by Kim Hazelwood, in *Synthesis Lectures on Computer Architecture*, Morgan & Claypool Publishers, 2011.

18-640 Foundations of Computer Architecture

Lecture 11:
"Dynamic Code Translation & Compilation"

A. Dynamic Binary Translation
B. Transmeta Example
C. Code Morphing System
D. Nvidia Denver Example

A. DBT: Dynamic Binary Translation

- Up to now, CPU hardware design
  - Pipelining, superscalar, speculative execution, …
  - ISA to interface hardware features (VLIW)
- For VLIW, the compiler's role has been static
  - Hence, it's effectiveness limited
- DBT provide the compiler with a dynamic role

What is Binary Translation?

- Translating programs in one binary format to another
  - 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

Why is DBT a Good Idea?

- Feature support without hardware/source-code modification
  - E.g. Binary compatibility, optimization
- Vs. static binary translation
  - Access to complete program
    - Programs are fully linked
  - Access to program state
    - Include dynamic values
  - Can handle self-modifying code
  - Can adapt to changes in program behavior
  - No need to re-link
    - Translate instruction and jump to it

Dynamic Compilation Challenges

- Managing compilation costs
  - Trade-off compilation speed for generated code quality
  - Dynamic and static memory footprint important in certain systems
- Binary sources have less information than original source
  - Decompilation of CFG/DFG from binary
  - Maintain exception behavior of original code

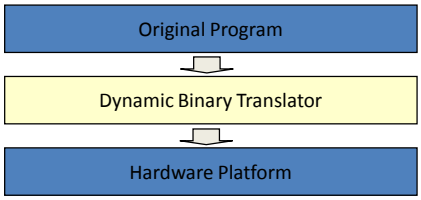
Dynamic Compilation Challenges (cont.)

- Integrating many components together
  - Compilers
  - Profilers
- On-stack replacement
  - Application
    - Dynamic optimization of procedures on the call stack
    - Mid-iteration optimization of long-running loops
  - Transition from old code to new compiled code
    - Stack frames of recompiled methods must match original if they are currently on the runtime stack
    - Optimized code may use stack in strange ways
    - Must be able to perform from inlined context

Managing Compilation Costs

- Mixed-mode / lazy compilation
  - Apply expensive optimizations to frequently executed code
  - Possible combinations
    - Combine interpreter / optimizing compiler
    - Combine fast & optimizing compiler
- Use less aggressive compiler optimizations
  - Avoid expensive/iterative global optimizations
    - Local optimizations have larger payoff
  - Use faster register allocation schemes
    - Linear scan vs. graph coloring
- Implement efficient compiler
  - Apply optimizations concurrently
  - Pipeline dataflow information between stages

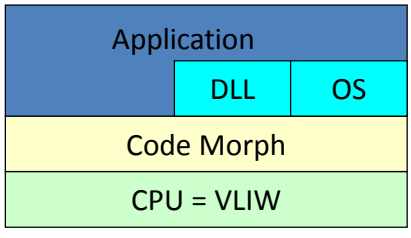
Dynamic Binary Translation Overview

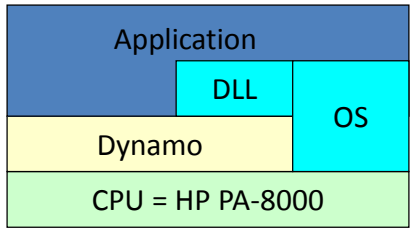


- Dynamic program modifier
  - Start with interpretation or intercept program execution
  - Observe a sequence of instructions
  - Produce new code and save in "code cache"
    - Change the jump/branch target address properly
  - Manipulate or add instructions as needed

DBT Configurations (1)

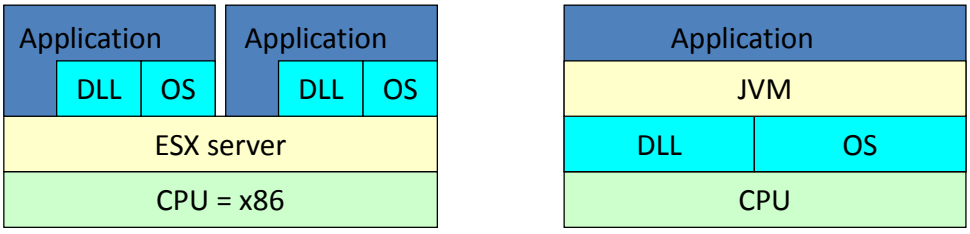
- Cross platform
  - E.g. Crusoe (Transmeta)
- Same platform
  - E.g. Dynamo (HP), PIN (x86), Dynamo-Rio(x86)



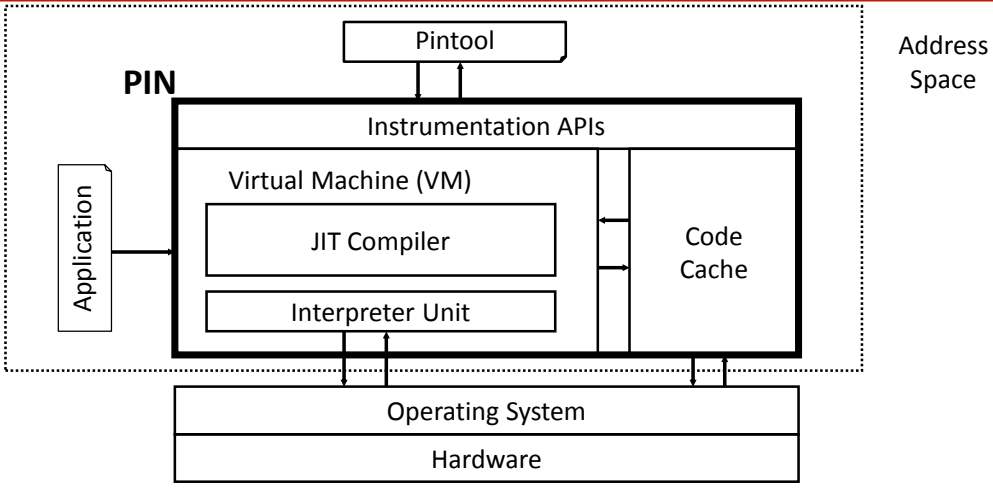


DBT Configurations (2)

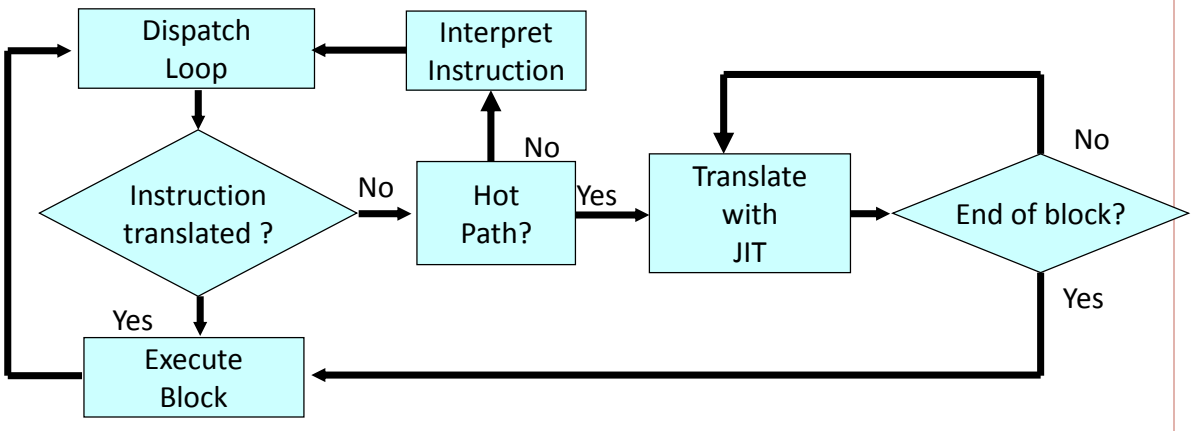
- Virtual Machine
  - E.g. ESX server (vmWare)
- JIT compilation
  - E.g. JVM (Sun), C#(MS)



Example DBT : PIN for x86



Translation Loop in VM



* Some DBTs have no interpretation mode

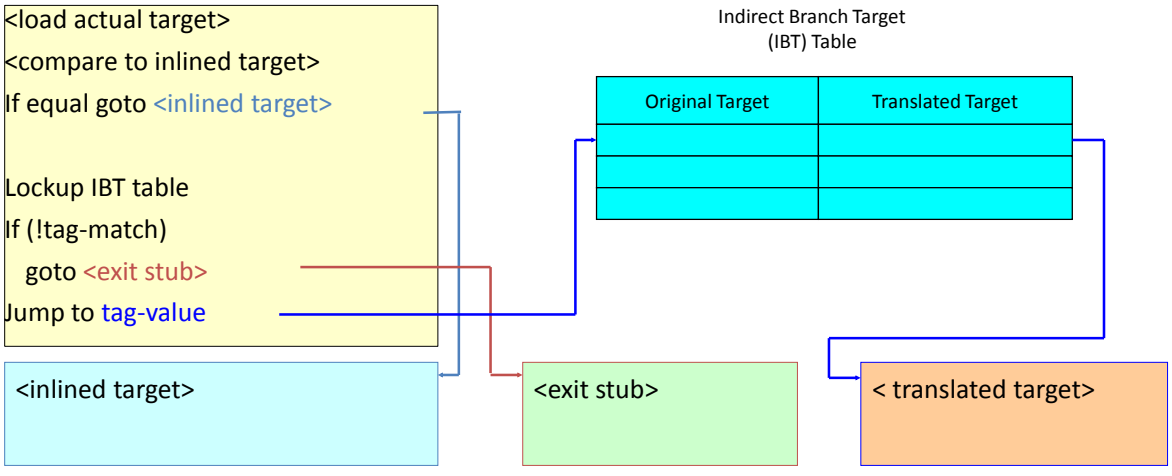
Code Cache

- Software cache in virtual address space
  - Keep translated code in memory for later reuse
  - Essential to leverage the high cost of translation and optimization
  - 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 switches to interpretation)
  - Trace (better amortization of switch overhead but need 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
- Link translated code
  - Direct branches are easy (branch to proper code cache offset)
  - 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
  - Invalidate corresponding translated code of the page in code cache
  - Next time, the code is re-translated

Multicore – Access to shared data

- Speculative execution assumes data unchanged
- Access by another core invalidates this assumption

- Solution:
  - "Victim" cache stores speculative state
  - Access to shared data aborts "commit" of translation

Example Use for DBT (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 scheduling: 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

Other DBT Uses

- Profiling
- Architectural studies
- Security/debugging tools
- …

- Most of this uses require instrumentation API
  - Define uses through the API by defining which instructions are examine and what code is inserted

DBT Used for Binary Instrumentation

- MIPS pixie/pixstats ~1987
  - Statically modified binaries to count instructions
- Sun Spix/Spixstats/Span/Shadow/Shade ~1988-95
  - Success led to more functionality
  - Ability to patch in analysis routines while running
  - Shade ~1990 enhanced dynamic techniques
    - able to run MIPS and x86 code on SPARC, about 1/3 native speed
    - Amazed at the performance of non-native code
- Lessons –Critical realizations in 1995
  - Co-design the underlying hardware to reduce inefficiencies
  - Could move from less than native to net performance gain
  - Meant Hybrid processors could compete with mainstream

B. Transmeta – A Real-Life DBT system

- Transmeta
- Hardware Support for Binary Translation
- DBT Optimizations

Binary Translation Uses

- Pentium Pro and its successors
  - Translates x86 binary code into internal UOPS via HW
- Intel IA32EL
  - Translates user level x86 programs on Itanium via SW
  - Performance about 60% of native machine
- JAVA JITs / Microsoft MSIL / VMware
- Transitive Technologies
  - Rosetta runs Apple PowerPC user programs on x86
- Transmeta Crusoe and Efficeon mobile processors
  - Translates x86 binary code into internal UOPS via sw
  - Transparent full system level translation as real products
- Performance comparable to Intel's mobile CPUs

Binary Translation History

- Good work worth mentioning:
  - XDOS — Early static DOS to Unix
  - DEC FX!32 — x86 user apps to Alpha
  - IBM DAISY, BOA — PowerPC on VLIW
  - Elbrus nArch, E2K — SPARC, x86 to VLIW
  - Sun SoftWindows — x86 to SPARC/Solaris
  - Intel PINx86 — binary instrumentation
  - Intel StarDBTIntel — research translator

  and many others ...
  - see Virtual Machines book by Jim Smith & Ravi Nair

Transmeta "Hybrid Processor"

- A Hybrid processor –in Transmeta's proposal

  Specifically means a microprocessor implemented as a

  hardware/software co-design using software binary translation and optimization with special purpose hardware

  so that

  it can function as a fully compatible microprocessor with performance comparable to pure hardware implementations.

Transmeta Crusoe Characteristics

- Transmeta's Crusoe microprocessor is a full, system level implementation of the x86 architecture, comprising a native VLIW microprocessor with a software layer, the Code Morphing Software (CMS)
  - 128bit VLIW(very long instruction word) engine
    - The Crusoe processor features a 128-bit wide VLIW (Very Long Instruction Word) engine that can issue up to 4 instructions per clock cycle.
  - Code Morphing Software
    - Code Morphing Software (CMS) layer provides the Transmeta Crusoe processor with x86 compatibility
  - Integrated Architecture
    - To ease system design and enhance performance, Transmeta has integrated Northbridge functionality — SDR and DDR SDRAM memory controllers, a 32-bit, 33MHz PCI bus controller and a Serial ROM interface controller — directly into the Crusoe processor die.
  - LongRun Technology
    - Transmeta LongRun technology allows the Transmeta Crusoe processor to conserve power by dynamically adjusting its voltage and clock frequency.

Transmeta

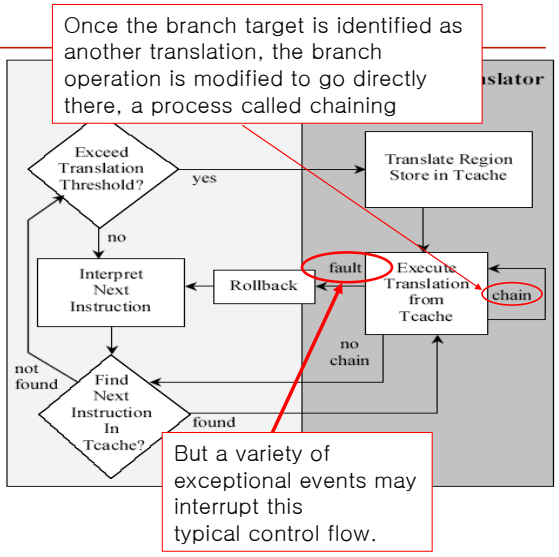
- Transmeta was founded in 1995, with the idea to co-design hardware and software together in a machine whose goal was provide an efficient base for binary translation of x86 programs
- Spent $600M in R&D over 12 years to learn the tricks
- CMS was invisible to users, no compatibility bugs
- Five generations of processors designed
  - First two never shipped, insufficient performance
  - Crusoe product–Pentium class for mobile (250nm-180nm)
  - Efficeon product–Pentium-M class for mobile (130nm-90nm)
  - Next generation design that never shipped
- Lesson: Worked well and was on a rapid learning curve
- IPC improving at roughly 2x / generation

C. Transmeta CMS – "Code Morphing System"

- To produce high performance while remaining perfectly faithful to an existing architecture, translator must optimize aggressively:
  - –Speculation: Translator makes aggressive assumptions about code to achieve higher performance
  - –Recovery: Check assumptions and rollback to commit points if they prove to be false, for precise interpretation
  - –Adaptive retranslation: If recovery is required too often, retranslate with less aggressive assumptions

Overview of CMS Control Flow

- Typical CMS control flow
  - CMS is structured like many dynamic translation systems.
  - Initially, an interpreter decodes and executes x86 instructions sequentially.
  - When the number of executions of a section of x86 code reaches a certain threshold, its address is passed to the translator.
  - The translator selects a region and stores the translation with various related information in the translation cache.
  - From then on, until an event invalidates the translation cache entry, CMS executes the translation when the x86 flow of control reaches the translated code region.



Hardware Support for Recovery

- *Shadow* registers:
  - *Working* and *shadow* copies of x86 registers
- *Commit* atom: Copies working registers to shadow registers, commits memory writes
- *Rollback* atom: Copies shadow registers to working registers, discards memory writes

Example: Precise Exceptions

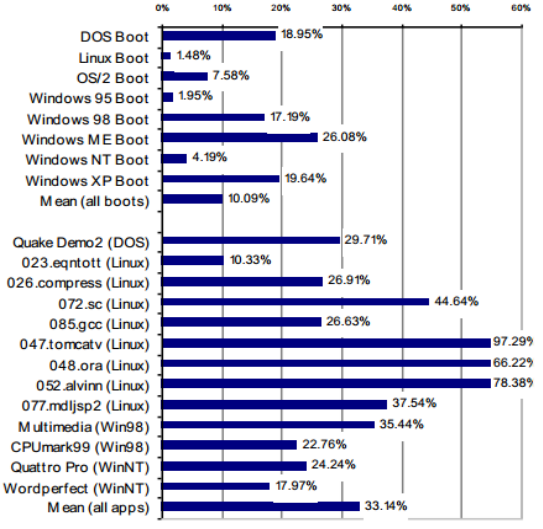
- **Problem**: CMS rearranges operations in a translation, but x86 has precise exception semantics
- **Speculation**: CMS translations scheduled assuming no exceptions will occur
- **Recovery**: If an exception occurs, rollback to preceding consistent commit point, and interpret sequentially
- **Adaptive retranslation**: An instruction causing exceptions too often is isolated, and the rest of the original translated code is retranslated so it won't need rollback

Memory-Mapped I/O Problem

- **Problem**: Arbitrary x86 memory operations may be memory-mapped I/O, which must not be reordered and may not be rolled back
- **Speculation**: CMS translates memory operations as *normal*, reorders them at will (unless marked by interpreter as *abnormal*)
- **Recovery**: A normal reference to abnormal memory causes a hardware fault, rollback, and appropriate interpretation
- **Adaptive retranslation**: A memory op that is abnormal too often is retranslated with commits before and after it, and with other constraints necessary for memory-mapped I/O

Memory-Mapped I/O

Figure 2: Degradation Caused by Suppressing Memory Reordering



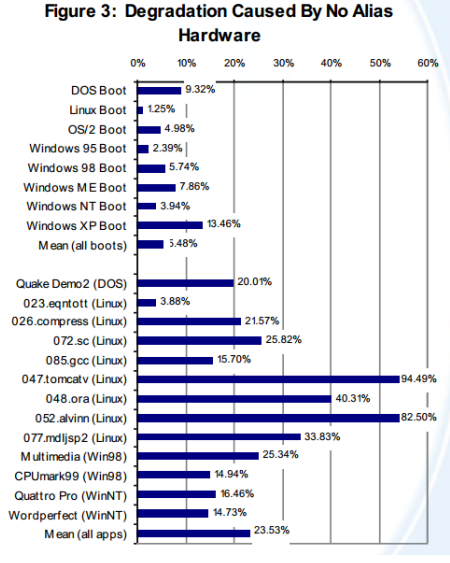
Data Speculation (Aliases)

- **Problem**: To achieve effect of out-of-order processors on memory operations, CMS needs to reorder memory operations
- **Speculation**: CMS assumes memory operations don't alias unless it can prove otherwise, reorders accordingly
- **Recovery**: Memops scheduled earlier set an *alias register;*later memops that might alias check the register and trap on overlap
- **Adaptive retranslation**: Translation that takes alias faults too often is translated without reordering

Data Speculation

Simulations obtained without using alias HW

No adaptive retranslation cuts PCmark2002 3d vector performance by a factor of 47.5 on hardware



SMC – Self-Modifying Code

- **Original problem**: If the x86 code modifies itself, the CMS translations must be invalidated or otherwise adapt.
- **Speculation**: Normal translations assume no SMC
- **Simple recovery**: Write-protect x86 code pages, find and invalidate corresponding translations if a fault occurs
- **Secondary problems**:
  - –Inefficient for self-modifying code: granularity too large
  - –Can't distinguish data in same page as code
- **Costs incurred by CMS**:
  - –Handling fault, invalidating translations, special processing
  - –Generating new translations for new code

SMC- Fine-Grain Protection

- **First refinement**: Hardware support for sub-page granularity

  Only needed for a few pages at a time, allowing tiny hardware cache
- Without fine-grain protection, the worst cases:

	Faults	Slowdown
Win95 Boot	52.8x	2.2x
Win98 Boot	59.4x	3.8x
MultimediaMark	46.8x	1.6x
WinStone Corel	54.2x	2.1x
Quake Demo2	7.7x	1.02x

Transmeta – 10 Hardware Support Features

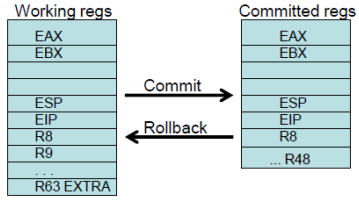
1. Software controlled state. Commit/Rollback/Abort
2. More registers than architected state
3. Alias detection under software control
4. Self modifying code detection with fine grain support
5. Auto-typing of pure memory vs I/O
6. Fast traps supported by underlying runtime system
7. Instruction primitives for fast interpretation
8. Private memory. ~5% of DRAM for translated code
9. Private non-volatile storage (FLASH ROM) with every chip
10. Competitive CPU hardware, ie uOp ISA, IPC and clock rate

Speculative Execution and x86 State Control

- Hardware for software controlled Commit/Rollback enables wide optimization regions

  All registered state has two copies
  - Working state
  - Committed State

  Registered State includes:
  - Integer and Floating Point register files
  - Condition code flags
  - All other state registers

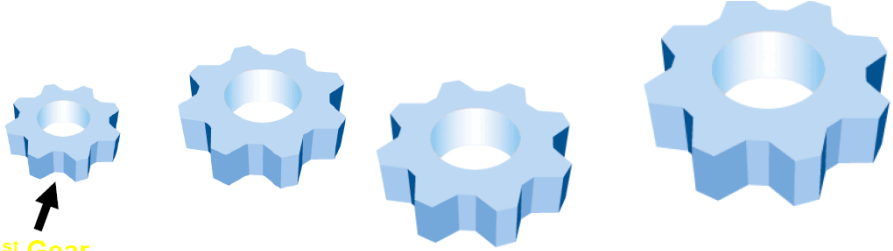


  Support for speculative stores between commit points in L1 data cache
  - New cache state
  - 1 new tag bit "speculative"
  - Flash (1-cycle) flush (on rollback) or convert to dirty (commit)
- New UOP level Instructions
  - Commit: Copies working registers to committed state (1-cycle)
  - Rollback: Copies committed registers to working registers (1-cycle)
  - Abort: To runtime trap hander, which can do fix up, rollback, and branch

Code Morphing Software

- 4 "Gear" System
  - Start with interpreter
  - Increasing levels of optimization and speculation
  - Utilize HW assistance
    - Speculation
      - Shadow Registers
      - Speculative data in cache

First Gear



1st Gear

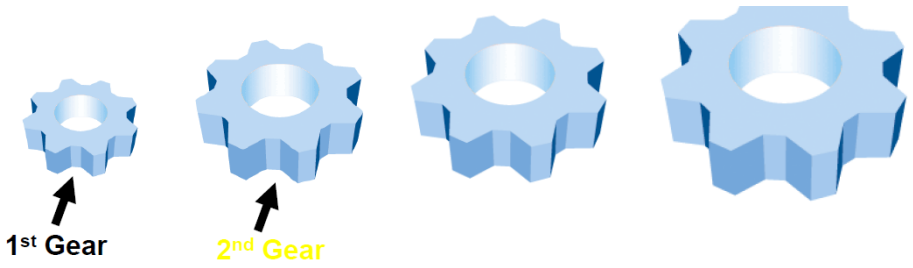
Executes 1 instruction at a time

- Profiles code at runtime
- Gathers data for flow analysis
- Gathers branch frequencies and directions
- Detects load/store typing (IO vs memory)

Filters out infrequently executed code

No startup cost
Lowest speed

Second Gear

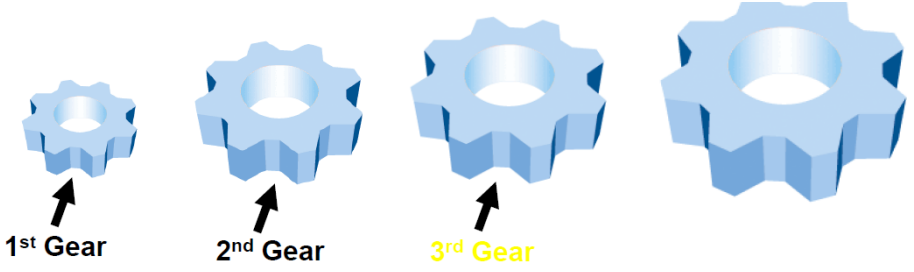


Uses profile data to create initial translations after code reaches 1st threshold.

- Translates a "Region" of up to100 x86 instructions.
- Adds flow graph "Shape" information
- Light Optimization
- "Greedy" scheduling

Low translation overhead
Fast execution

Third Gear

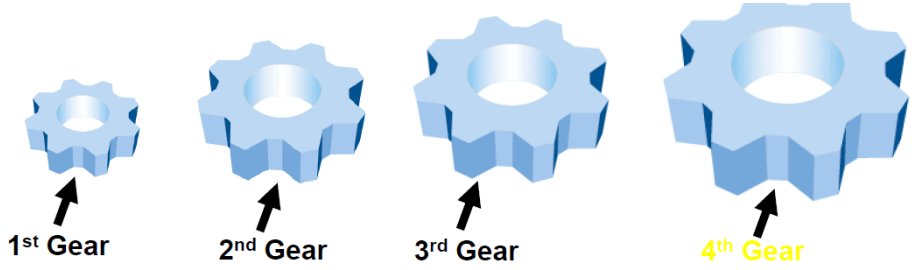


Further optimizes the 2nd gear regions

- Common sub-expression elimination
- Memory re-ordering
- Significant code optimization
- Critical path scheduling

Medium translation overhead
Faster execution

Fourth Gear



Most advanced optimizations for "hottest" code regions.

- Splices together multiple regions
- Optimizes across region boundaries
- Used advanced behavioral data
- Critical path scheduling

Highest translation overhead
Fastest execution

CMS Optimizations

- Aggressive scheduling of instruction level parallelism
- Out-of-Order instruction execution obtained on In-Order hardware
- Critical path height reduction
- Common sub-expression elimination
- Uses "Address Alias Checking Hardware"
  - Re-ordering of loads and stores even with potential aliases
  - Elimination of loads and stores even with potential aliases
- Software register renaming
- Fusing operations
- Dead code elimination
- Removal of conditional branches
- Adaptive re-translation during program execution
- Loop unrolling and optimization
  - Remove Exit Branches
  - Code motion across back-edge

Loop invariant code motion
Strength reduction

HOT CODES GET MOST OPTIMIZATION

Translation Example

- translation example
  - X86 instruction

```
A. addl %eax,(%esp) // load data from stack, add to %eax
B. addl %ebx,(%esp) // ditto, for %ebx
C. movl %esi,(%ebp) // load %esi from memory
D. subl %ecx,5 // subtract 5 from %ecx registe
```

  - In a first pass, the front end of the translation – simple translation

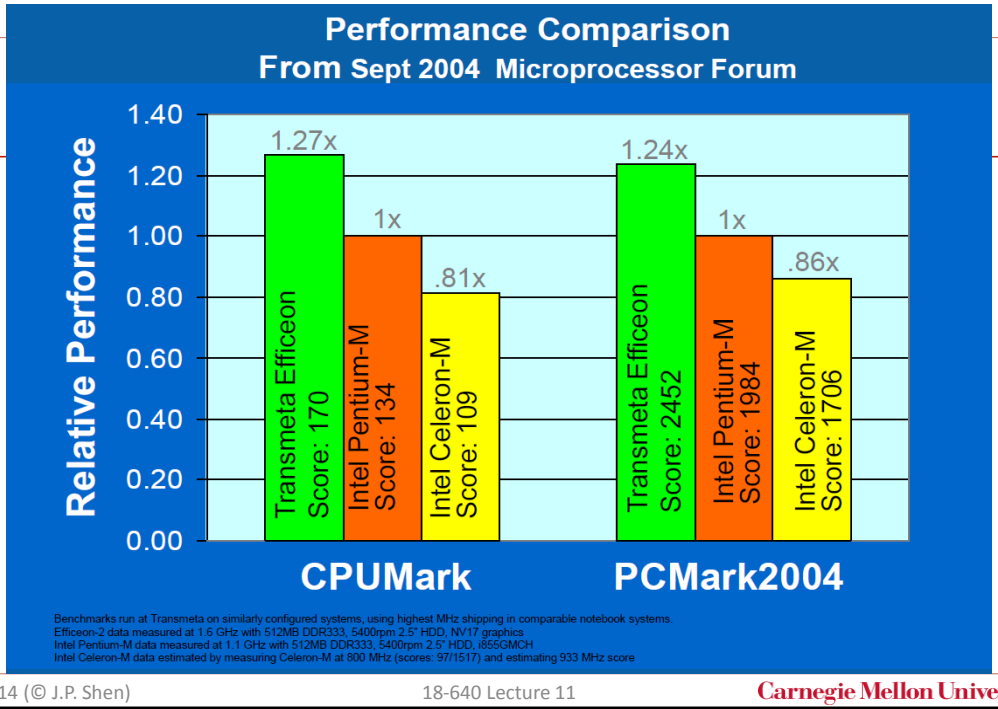
```
ld %r30,[%esp] // load from stack, into temporary
add.c %eax,%eax,%r30 // add to %eax, set condition codes.
ld %r31,[%esp]
add.c %ebx,%ebx,%r31
ld %esi,[%ebp]
sub.c %ecx,%ecx,5
```

  - In a second pass, the optimizer. applying well known compiler optimization skill such as common subexpression elimination, loop invariant removal or dead code elimination.

```
ld %r30,[%esp] // load from stack only once
add %eax,%eax,%r30
add %ebx,%ebx,%r30 // reuse data loaded earlier
ld %esi,[%ebp]
sub.c %ecx,%ecx,5 // only this last condition code needed
```

  - In a final pass, the scheduler. reordering atoms into molecules.

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

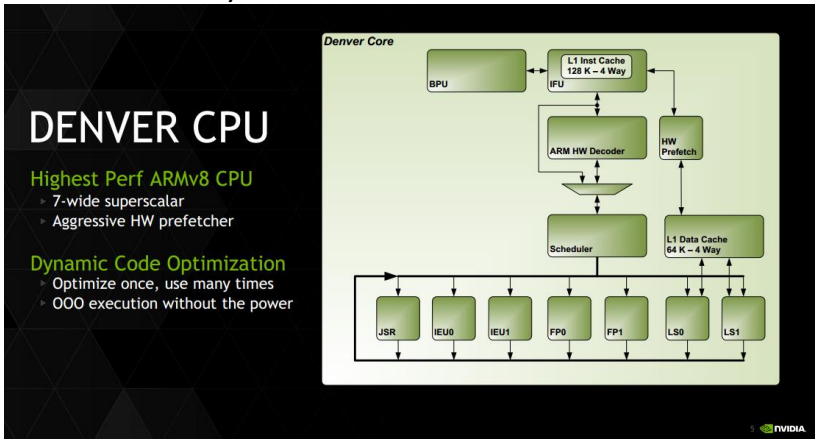



Code Morphing Software

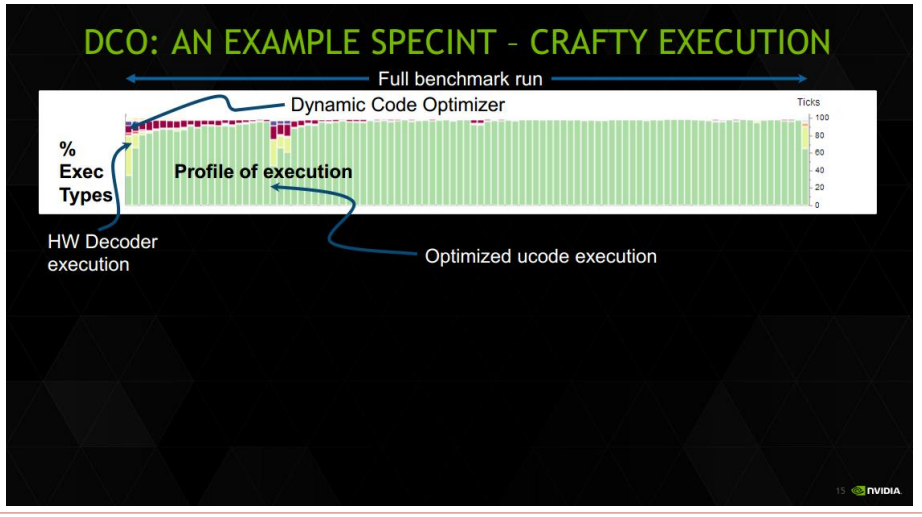
- Multicores
- Omission from Initial Transmeta proposal: multicores
  - Coherence
    - Rollback on "speculative" data hit
      - Research at Intel

D. NVIDIA "Denver" Architecture

- 128 Mb translation cache in memory
- VLIW engine



NVIDIA Denver



Binary Translation Advantages

- Innovation
  - To allow processor innovation not tied to particular instruction sets
  - Using BT to provide backwards compatibility
- Performance
  - To enable new means to improve processor performance
  - Using BT to provide backwards compatibility
- Power
  - To enable simpler and lower power processors
  - Using BT to provide backwards compatibility

Conclusion

- Binary Translation based processors can work well, no longer a theory.
- Special purpose hardware support is needed, co-designed with software.
- Software translation can be done poorly or well, special care is needed to keep translation overhead low.
- Many opportunities for clever hardware/software co-design tradeoffs
- This is a technological approach still in its infancy

References

- Wang, Cheng, et al. "Stardbt: An efficient multi-platform dynamic binary translation system." *Advances in Computer Systems Architecture*. Springer Berlin Heidelberg, 2007. 4-15.
- Chernoff, Anton, et al. "FX! 32: A profile-directed binary translator." *IEEE Micro*18.2 (1998): 56-64.
- Cmelik, Bob, and David Keppel. *Shade: A fast instruction-set simulator for execution profiling*. Springer US, 1995.
- Dehnert, James C., et al. "The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges." *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2003.
- Sites, Richard L., et al. "Binary translation." *Communications of the ACM* 36.2 (1993): 69-81.
- Ebcioglu, Kemal, et al. "Dynamic binary translation and optimization."*Computers, IEEE Transactions on* 50.6 (2001): 529-548.