Transmeta *Crusoe* and *efficeon*:

Embedded VLIW as a CISC Implementation

Jim Dehnert

Transmeta Corporation

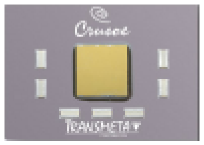
SCOPES, Vienna, 25 September 2003

Outline

- ***Crusoe*** / ***efficeon*** Background
  - System Architecture
  - Code Morphing Software Structure
  - Key hardware features
  - Benefits
- CMS Paradigm: speculation, recovery, and adaptive retranslation
  - Example: Aggressive scheduling – exceptions and aliases
  - Example: Self-modifying code
- Co-simulation for Testing
  - Simulator / emulator / self
- Summary

Transmeta Technology

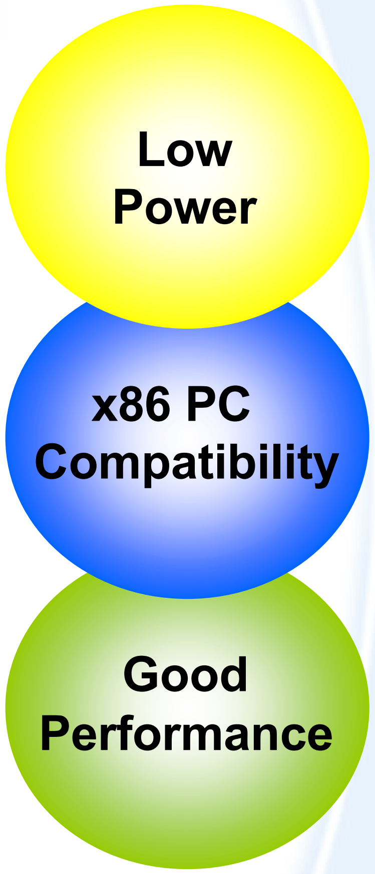
Microprocessor is the sum of



+

CMS

=



VLIW Hardware

- Very Long Instruction Word processor
- Simple and fast
- Fewer transistors

Code Morphing Software

- Provides Compatibility
- Translates binary x86 instructions to equivalent operations for a simple VLIW processor
- Learns and improves with time

Advantages of CMS Approach

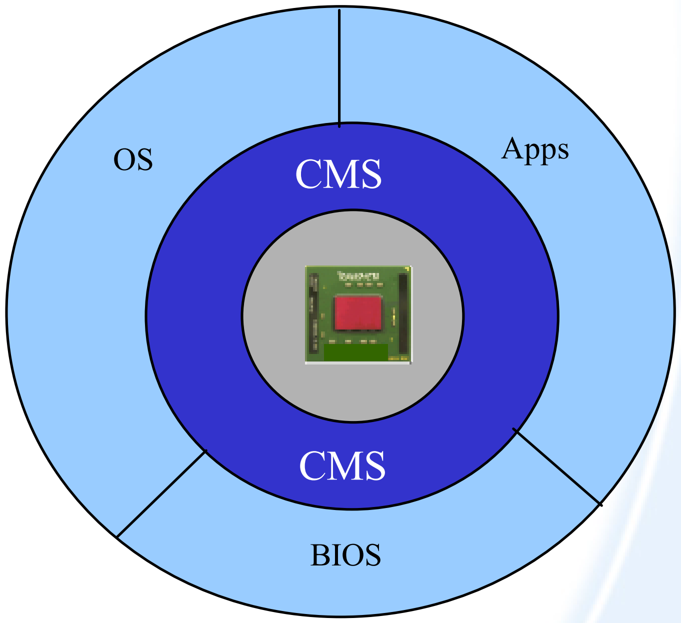
- Simple hardware allows
  - Smaller, less expensive implementation
  - Lower power consumption
- Hidden VLIW architecture allows
  - Transparent changes in architecture
  - CMS can compensate for hardware bugs
  - Performance improvement does not require hardware changes

Crusoe / *efficeon* VLIW Engines

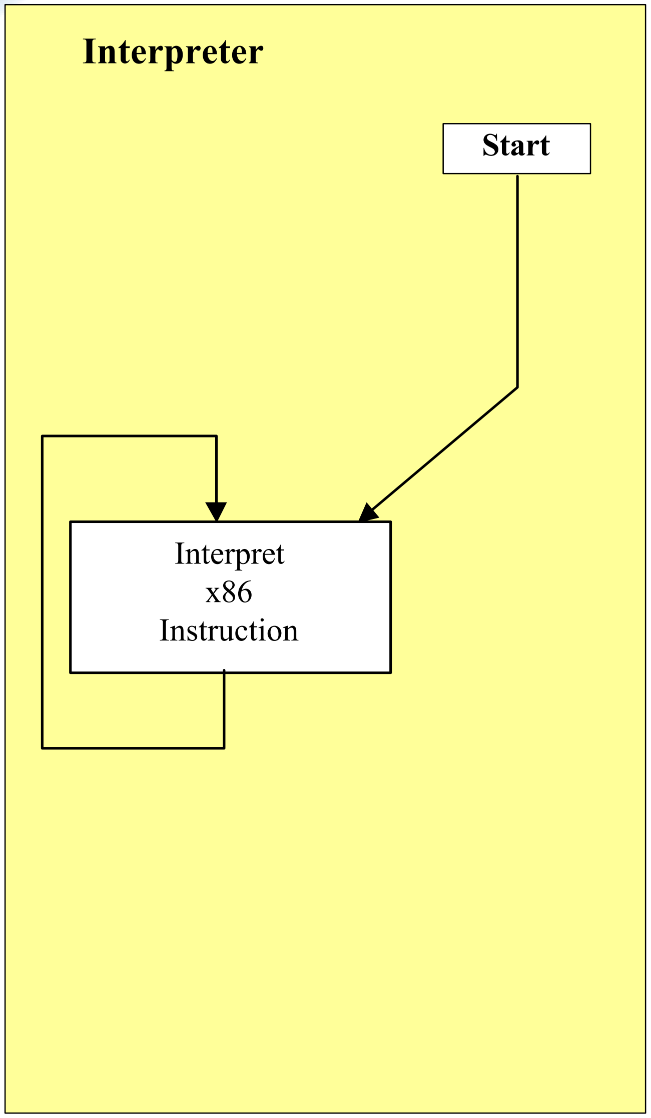
- VLIW: 2 or 4 *operations* per *instruction* in ***Crusoe***
  Up to 8 operations and modifiers in ***efficeon***
- Functional units: ALUs, memory, FP/media, branch
- Registers: 64 GPRs, 64 FPRs, 4 predicates
  dedicated x86 subset
- Few hardware interlocks (CMS avoids hazards)
- Semantic match: addressing modes, data types, partial-word operations, condition codes

CMS Objectives

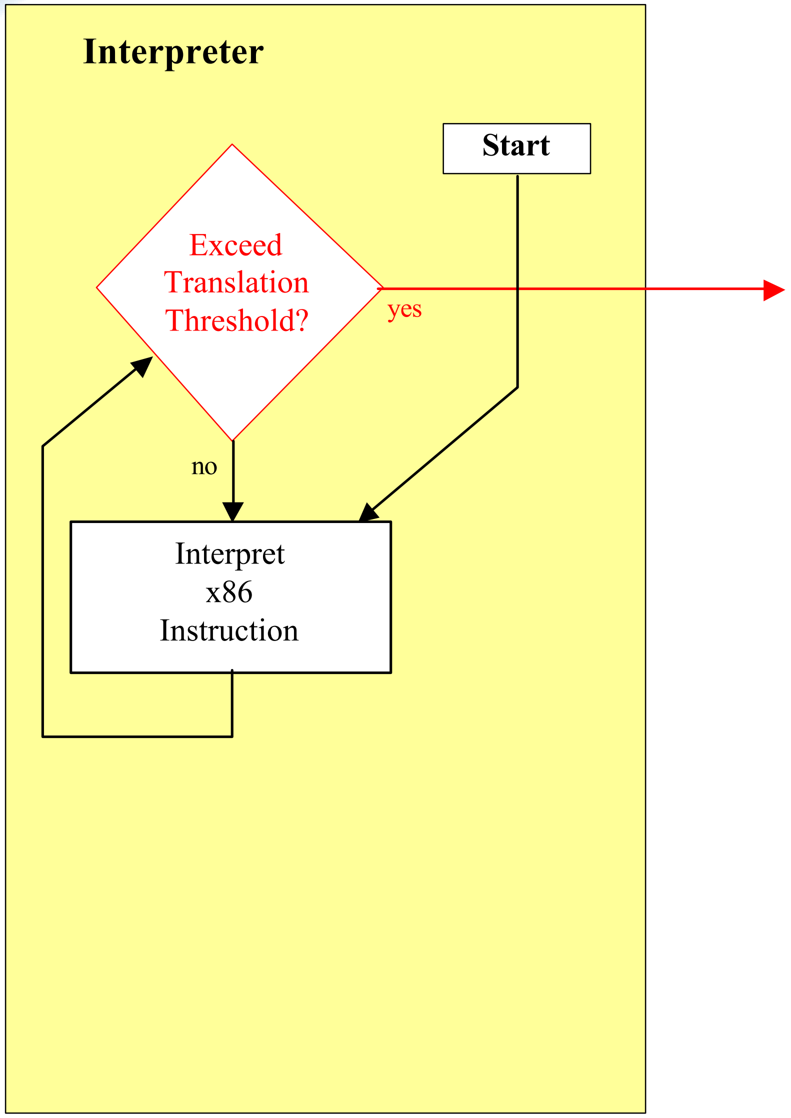
- Code Morphing Software layer provides a completely compatible implementation of the x86 architecture on the embedded VLIW processor:
  - All target instructions (including memory-mapped I/O)
  - All architectural registers
  - Compatible exception behavior
- Constraints:
  - No OS assumptions or assistance
  - Only see executed code – instructions and pages
- Robust performance required



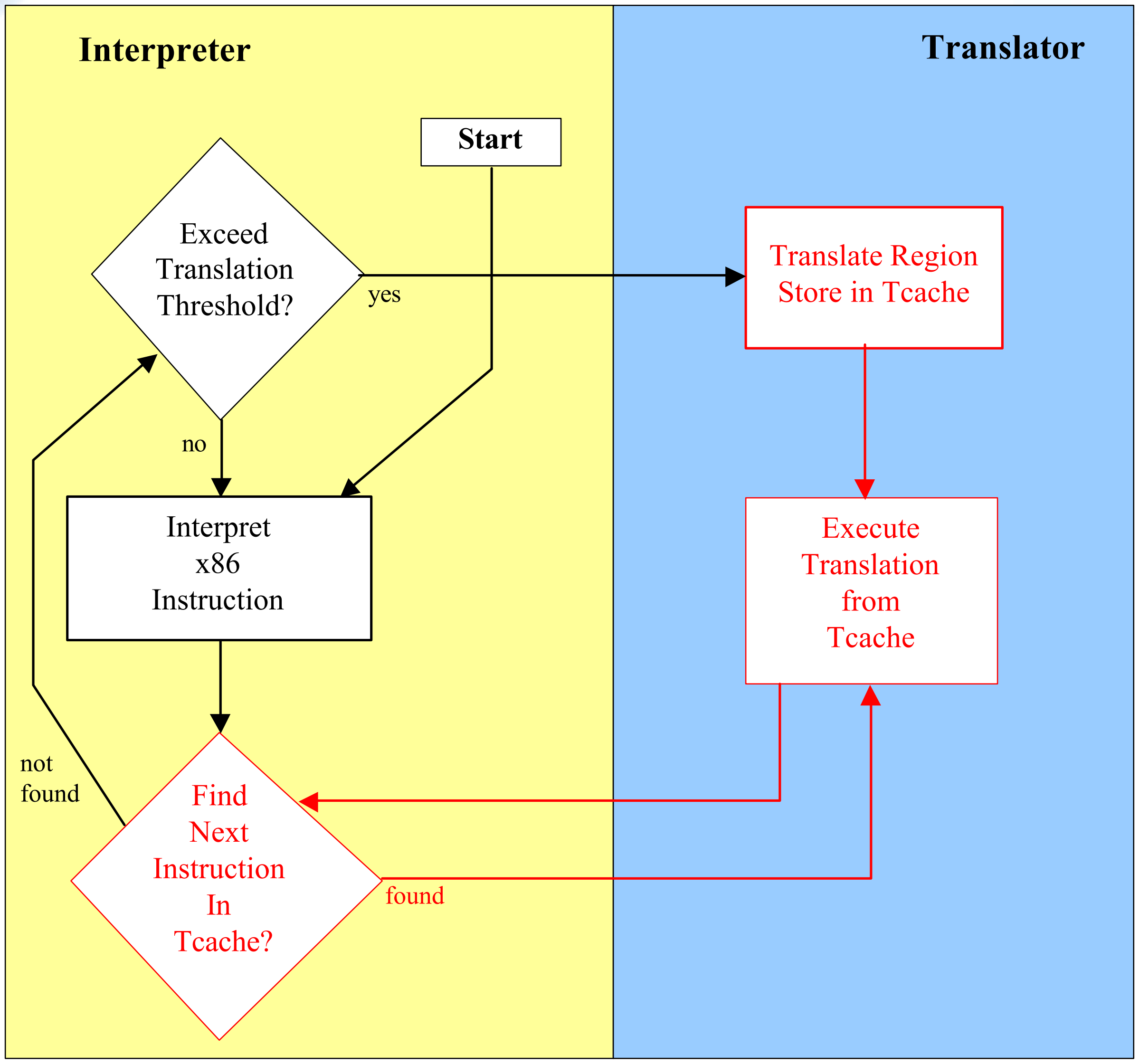
CMS Control Structure



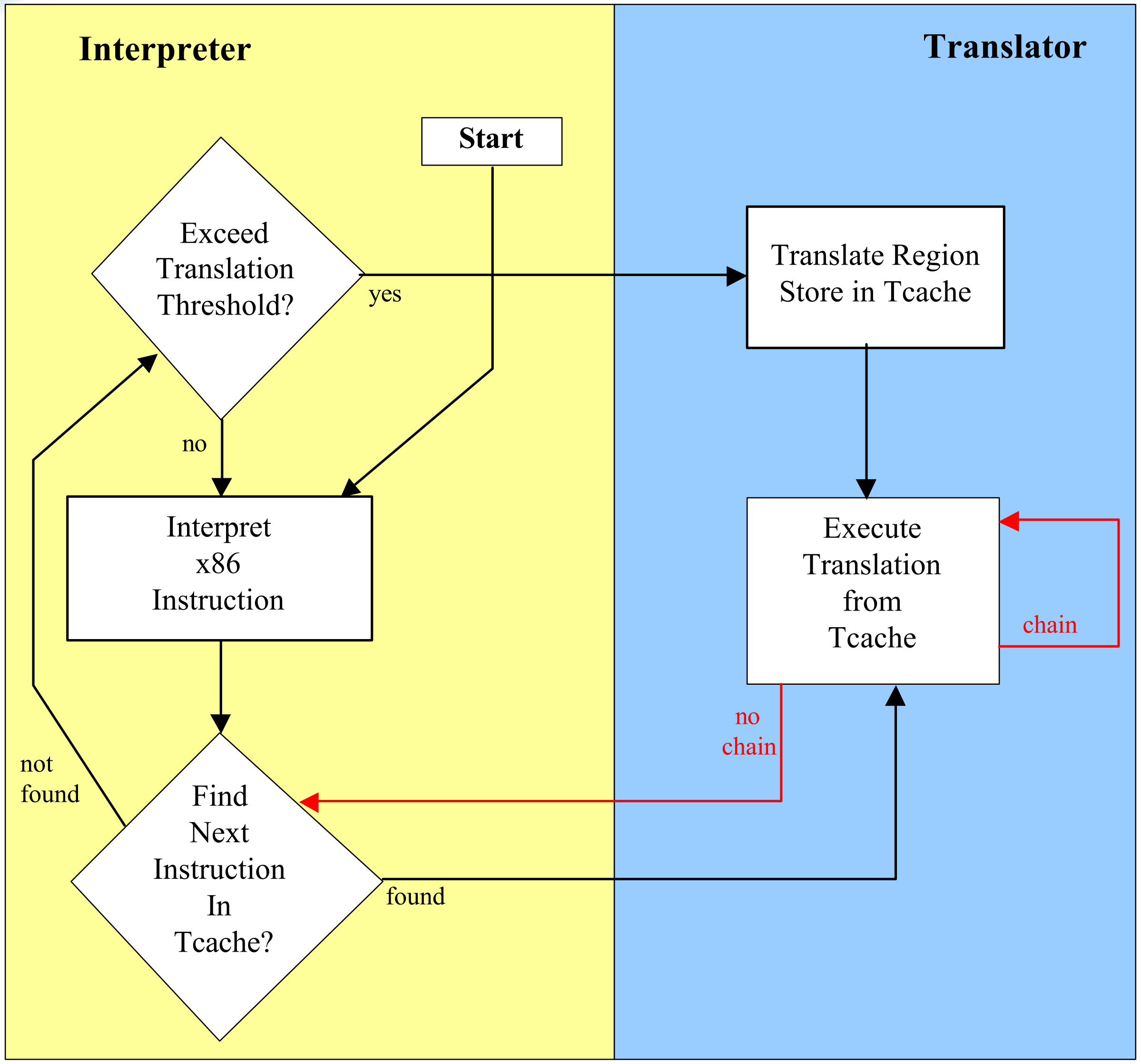
CMS Control Structure



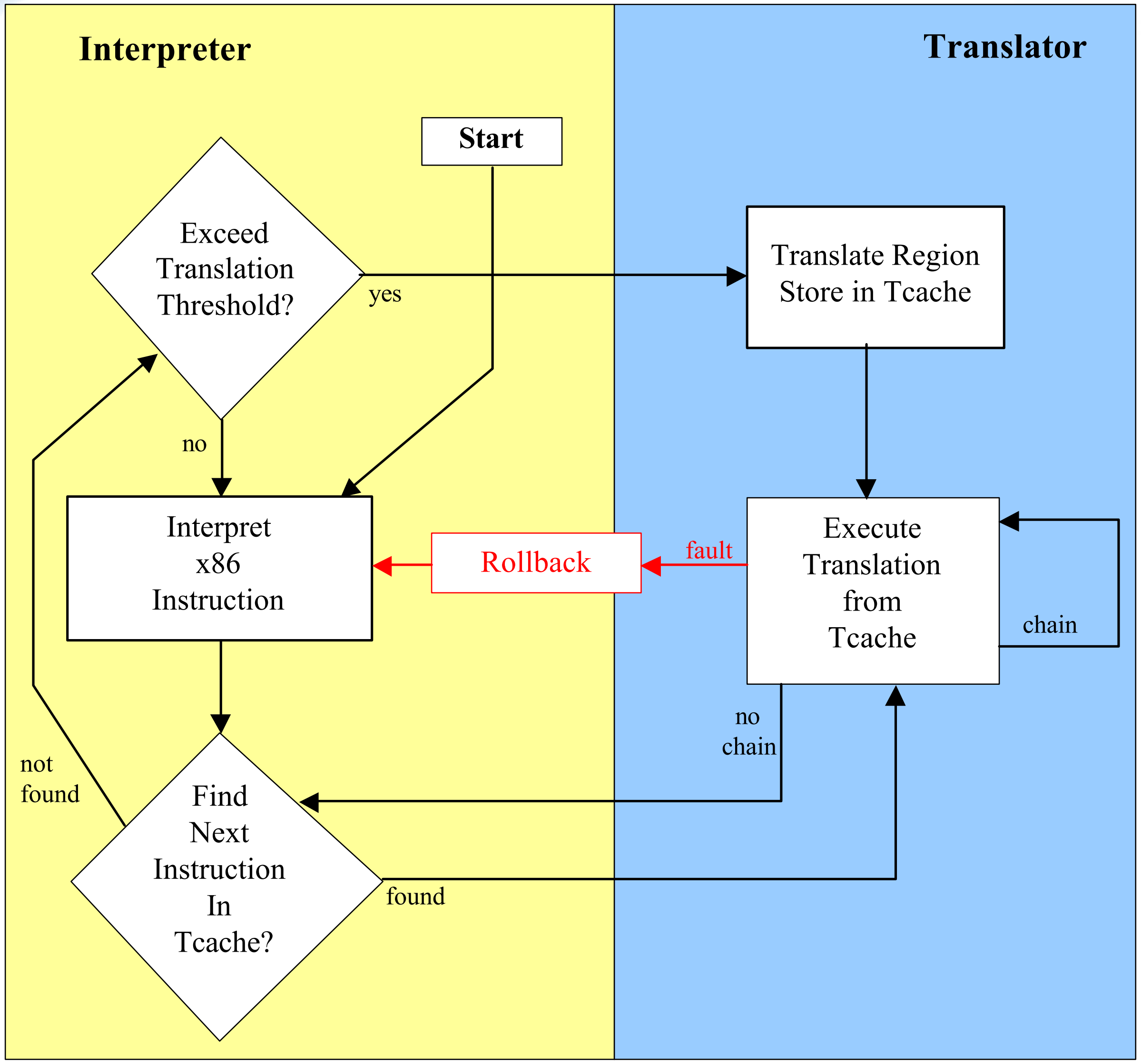
CMS Control Structure


Interpreter
Translator
Start
Exceed Translation Threshold?
yes
no
Translate Region Store in Tcache
Interpret x86 Instruction
Execute Translation from Tcache
not found
Find Next Instruction In Tcache?
found

CMS Control Structure


Interpreter
Translator
Start
Exceed Translation Threshold?
yes
no
Translate Region Store in Tcache
Interpret x86 Instruction
Execute Translation from Tcache
chain
no chain
not found
Find Next Instruction In Tcache?
found

CMS Control Structure


Interpreter
Translator
Start
Exceed Translation Threshold?
yes
no
Translate Region Store in Tcache
Interpret x86 Instruction
Rollback
fault
Execute Translation from Tcache
chain
no chain
not found
Find Next Instruction In Tcache?
found

Hardware Support for Recovery

- *Shadow* registers: *Working* and *shadow* copies of x86 registers
  - Code uses working registers
  - Consistent x86 state preserved in shadow registers
- *Memory* is analogous
  - Speculative writes to working buffer
  - Memory contains consistent x86 state
- *Commit* operation: Copies working registers to shadow registers, releases speculative memory writes -- *fast*
- *Rollback* operation: Copies shadow registers to working registers, discards speculative memory writes

CMS Is A Dynamic System

- Start with interpretation
  - – low overhead but slow execution
- Translate when repetition suggests benefit
  - – higher overhead but much faster execution
- Re-translate if the situation changes
  - – more or less optimization as appropriate

CMS Is A Dynamic System

Dynamic context gives CMS significant advantages

- Before translating, interpreter can collect useful data:
  - Branch frequencies
  - Abnormal memory accesses (memory-mapped I/O)
- Translated segments can also collect data:
  - Prologues can count entries, e.g. for tcache management
- Translator can perform optimizations not available to compilers or hardware implementations:
  - Runtime information
  - Ability to rollback to consistent x86 state

Outline

- ***Crusoe*** / ***efficeon*** Background
  - System Architecture
  - CMS Structure
  - Key hardware features
  - Benefits
- CMS Paradigm: speculation, recovery, and adaptive retranslation
  - Example: Aggressive scheduling – exceptions and aliases
  - Example: Self-modifying code
- Co-simulation for Testing
  - Simulator / emulator / self
- Summary

The CMS Paradigm

To produce high performance while remaining perfectly faithful to the x86 architecture, the translator must optimize aggressively:

- Speculation: Translator makes aggressive assumptions about code to achieve higher performance
- Example assumptions:
  - operations won't raise exceptions
  - memory operations unaliased, *normal* (not to I/O space)
  - no self-modifying code
  - … and many more …

The CMS Paradigm

To produce high performance while remaining perfectly faithful to the x86 architecture, the translator must optimize aggressively:

- Speculation: Translator makes aggressive assumptions about code to achieve higher performance
- Recovery:
  - Commit x86 state at convenient points
  - Check assumptions and rollback if false
  - Interpret sequentially for precise conformance

The CMS Paradigm

To produce high performance while remaining perfectly faithful to the x86 architecture, the translator must optimize aggressively:

- Speculation: Translator makes aggressive assumptions about code to achieve higher performance
- Recovery:
  - Commit x86 state at convenient points
  - Check assumptions and rollback if false
  - Interpret sequentially for precise conformance
- Adaptive retranslation: If recovery is required too often:
  - Retranslate with less aggressive assumptions
  - Retranslate smaller regions to minimize impact
  - Keep both translations if more aggressive usually works

Example: Aggressive Scheduling

- CMS performance depends on aggressive reordering and scheduling of code

x86 code:

```
L: lea   %ecx = (%edi,%edi,1)
   lea   %eax = 0x1(%ebx)       # %eax is invariant
   fldl  (%esi,%eax,8)          # address is invariant
   faddl (%esi,%ecx,8)
   fmull 0x6959c8               # address is invariant
   fstpl 0x40(%ebp,1)
   inc   %edi
   cmp   %eax,%edi
   jbe   L
```

efficeon code (with liberties):

```
E:{calculate rt1=%ecx, rt2=%eax; flda ft1 = [0x6959c8]}
  {fld   ft2 = [%esi+rt2*8];     flda ft3 = [%esi+rt1*8]}
L:{fadd  f7  = ft2+ft3;     %ecx = rt1;   rt1 += 2}
  {fmul  f7  = f7*ft3;      %eax = rt2;   %edi += 1}
  {sub.c r63 = %edi-%eax;   flda ft3 = [%esi+%ecx*8]}
  {fst   f7, [0x40+%ebp];   test p3 = leu;     brc p3, L}
```

Aggressive Scheduling – Exceptions

- **Problem 1**: x86 has precise exception semantics

x86 code:

```
L: lea   %ecx = (%edi,%edi,1)
   lea   %eax = 0x1(%ebx)
   fldl  (%esi,%eax,8)
   faddl (%esi,%ecx,8)
   fmull 0x6959c8
   fstpl 0x40(%ebp,1)
   inc   %edi
   cmp   %eax,%edi
   jbe   L
```

x86 order:
ecx, eax, f7a, f7b, f7c, edi

efficeon order:
f7b, ecx; f7c, eax, edi

efficeon code:

```
E:{calculate rt1=%ecx, rt2=%eax; flda ft1 = [0x6959c8]}
  {fld    ft2 = [%esi+rt2*8];     flda ft3 = [%esi+rt1*8]}
L:{fadd   f7  = ft2+ft3;      %ecx = rt1;    rt1+=2}
  {fmul   f7  = f7*ft3;       %eax = rt2;    %edi +=1}
  {sub.c r63 = %edi-%eax;    flda ft3 = [%esi+%ecx*8]}
  {fst    f7, [0x40+%ebp];    test  p3 = leu;     brc p3, L}
```

Aggressive Scheduling – Exceptions

- **Problem 1**: x86 has precise exception semantics
- **Speculation**: CMS translations scheduled assuming no exceptions will occur
- **Recovery**: Exception causes rollback to preceding commit point, sequential interpretation
- **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

Aggressive Scheduling – Aliases

- **Problem 2**: data speculation -- memory ops may be aliased

x86 code:

```
L: lea   %ecx = (%edi,%edi,1)
   lea   %eax = 0x1(%ebx)
   fldl  (%esi,%eax,8)        # invariant?
   faddl (%esi,%ecx,8)
   fmull 0x6959c8             # invariant?
   fstpl 0x40(%ebp,1)
   inc   %edi
   cmp   %eax,%edi
   jbe   L
```

efficeon code:

```
E:{calculate rt1=%ecx, rt2=%eax;    flda ft1 = [0x6959c8]}
  {fld    ft2 = [%esi+rt2*8];       flda ft3 = [%esi+rt1*8]}
L:{fadd   f7  = ft2+ft3;       %ecx = rt1;   rt1+=2}
  {fmul   f7  = f7*ft3;        %eax = rt2;   %edi +=1}
  {sub.c r63 = %edi-%eax;      flda ft3 = [%esi+%ecx*8]}
  {fst    f7, [0x40+%ebp];     test  p3 = leu;    brc p3, L}
```

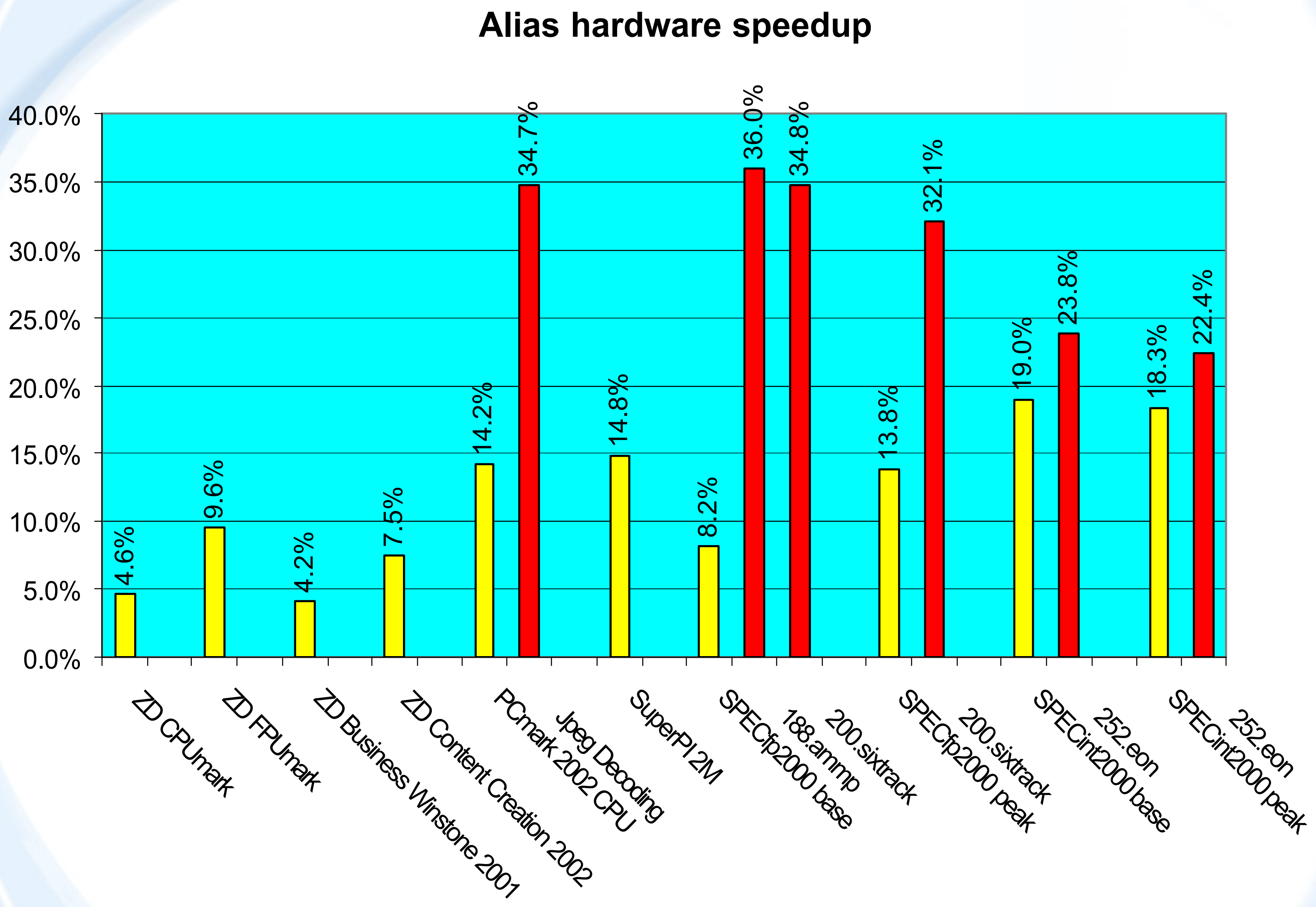
Aggressive Scheduling – Aliases

- **Problem 2**: data speculation -- memory ops may be aliased
- **Speculation**: CMS assumes memory operations don't alias unless it can determine otherwise. Loads or stores are moved past a store that might alias.
- **Recovery**: Speculated operations set an *alias register* unless proven not to alias. Potentially aliasing store checks alias regs.

efficeon code:

```
E:{calculate rt1=%ecx, rt2=%eax;   flda ft1 = [0x6959c8] [a1]}
  {fld   ft2 = [%esi+rt2*8] [a2];  flda ft3 = [%esi+rt1*8]}
L:{fadd  f7  = ft2+ft3;       %ecx = rt1;   rt1+=2}
  {fmul  f7  = f7*ft3;        %eax = rt2;   %edi +=1}
  {sub.c r63 = %edi-%eax;    flda ft3 = [%esi+%ecx*8] [a3]}
  {fst   f7, [0x40+%ebp] [check a1,a2,a3];
                              test  p3 = leu;    brc p3, L}
```

Aggressive Scheduling – Aliases



Aggressive Scheduling – Aliases

- **Problem 2**: data speculation -- memory ops may be aliased
- **Speculation**: CMS assumes memory operations don't alias unless it can determine otherwise. Loads or stores are moved past a store that might alias.
- **Recovery**: Speculated operations set an *alias register* unless proven not to alias. Potentially aliasing store checks alias regs.
- **Adaptive retranslation**: Translation that takes alias faults too often is translated with conservative reordering
  - Enabling adaptive retranslation improves 3D vector component of PCmark2002 by a factor of 47.5 on HW

Self-Modifying Code (SMC)

- **Original problem**: If the x86 code is modified, 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
  - DMA looks like SMC
- **Costs incurred by CMS**:
  - Handling fault, invalidating translations, special processing
  - Generating new translations for modified code

SMC: Fine-Grain Protection

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

  Only needed for a few pages at a time, allowing tiny hardware cache

- Greatest speedups with fine-grain protection:

	Faults	Speedup
Win95 Boot	98.1%	2.2x
Win98 Boot	98.3%	3.8x
MultimediaMark	97.9%	1.6x
WinStone Corel	98.2%	2.1x
Quake Demo2	87.0%	1.02x

Recovery based on detection by fine-grain protection.
But how do we adapt translations?

SMC: Self-Revalidating Translations

- **Subproblem:** Data stores into same region as code
- **Adaptive retranslation:**
  - Install *prologue* to check code before executing translation
  - Must retranslate to capture x86 code
  - Fine-grain fault enables prologue and disables faults
  - Prologue performs check and re-enables faults
- **Tradeoff:** Checking code is expensive since it runs in sequence with the translation, but efficient if translations are executed many times between (clusters of) writes
- **Example:** Quake Demo2 frame rate improves 28%

SMC: Self-Checking Translations

- **Subproblem:** Frequent data stores in same region as code, so self-revalidation prologue overhead is significant
- **Adaptive retranslation:**
  - Integrate checking code with translation
  - May be scheduled for maximum overlap with translation
  - Must always check after any stores that might modify code
  - Minimize expense by making smaller translation first
  - Disable fine-grain protection for self-checking translation
- **Tradeoff:** Much better than faults, but still expensive:
  - Code-size mean increase 83% (58-100%)
  - Path-length mean increase 51% (11-124%)

Stylized SMC

- **Subproblem:** True self-modifying code often just replaces immediate fields in instructions, for instance to adjust the part of an array referenced:

```
label: lea %eax = 16(%esi)
```

- **Adaptive retranslation:**
  – Translated code gets value from x86 code space:
```
ld  %temp = [label+2]
add %eax = %esi + %temp
```
  – Use self-checking or self-revalidation for bytes not used directly

SMC: Translation Groups

- **Subproblem:** True self-modifying code that cycles among a small number of distinct versions (Windows/9X device-independent BLT driver)
- **Adaptive retranslation:**
  - Keep multiple translations for a single x86 address range
  - If current translation fails self-revalidation, try to match others
  - If another matches, make it the current one

CMS Paradigm Summary

- Robust performance requires dealing with a wide variety of relatively unusual cases that are expensive when they occur
- All three paradigm components are important:
  - Speculation
  - Recovery
  - Adaptive retranslation
- Several hardware mechanisms are vital:
  - Commit / rollback
  - Alias registers
  - Fine-grain protection

Outline

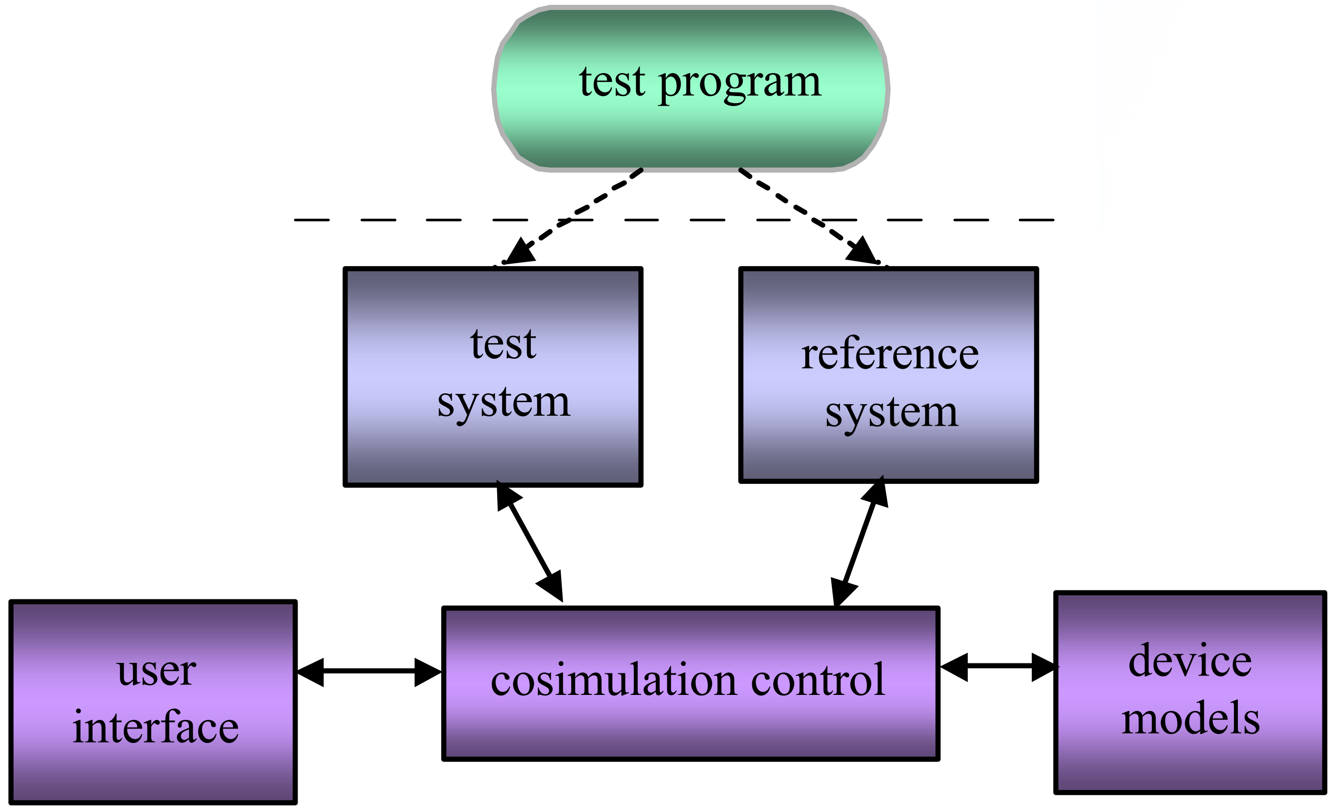
- ***Crusoe* / *efficeon*** Background
  - System Architecture
  - CMS Structure
  - Key hardware features
  - Benefits
- CMS Paradigm: speculation, recovery, and adaptive retranslation
  - Example: Aggressive scheduling – exceptions and aliases
  - Example: Self-modifying code
- Co-simulation for Testing
  - Simulator / emulator / self
- Summary

Testing with Cosimulation

- ***Crusoe*** / ***efficeon*** have usual processor testing issues.
- The software layer adds complexity:
  - Translation vs. interpretation
  - Changing translations
  - Rollback and re-execution
- Established methodology: use simulation during early development and compare against expected results
- Tremendous benefit from extended simulation testing methodology: *cosimulation*

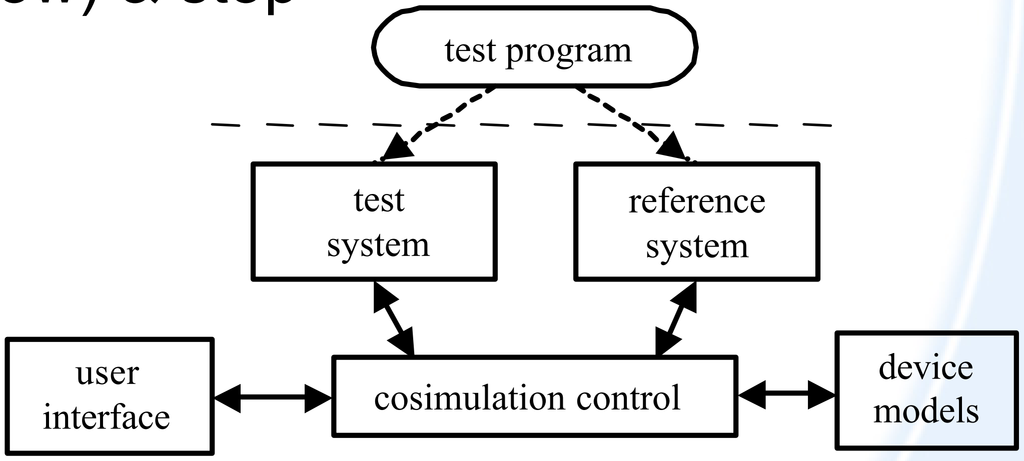
Cosimulation Overview

- Find bugs by comparing test and reference models

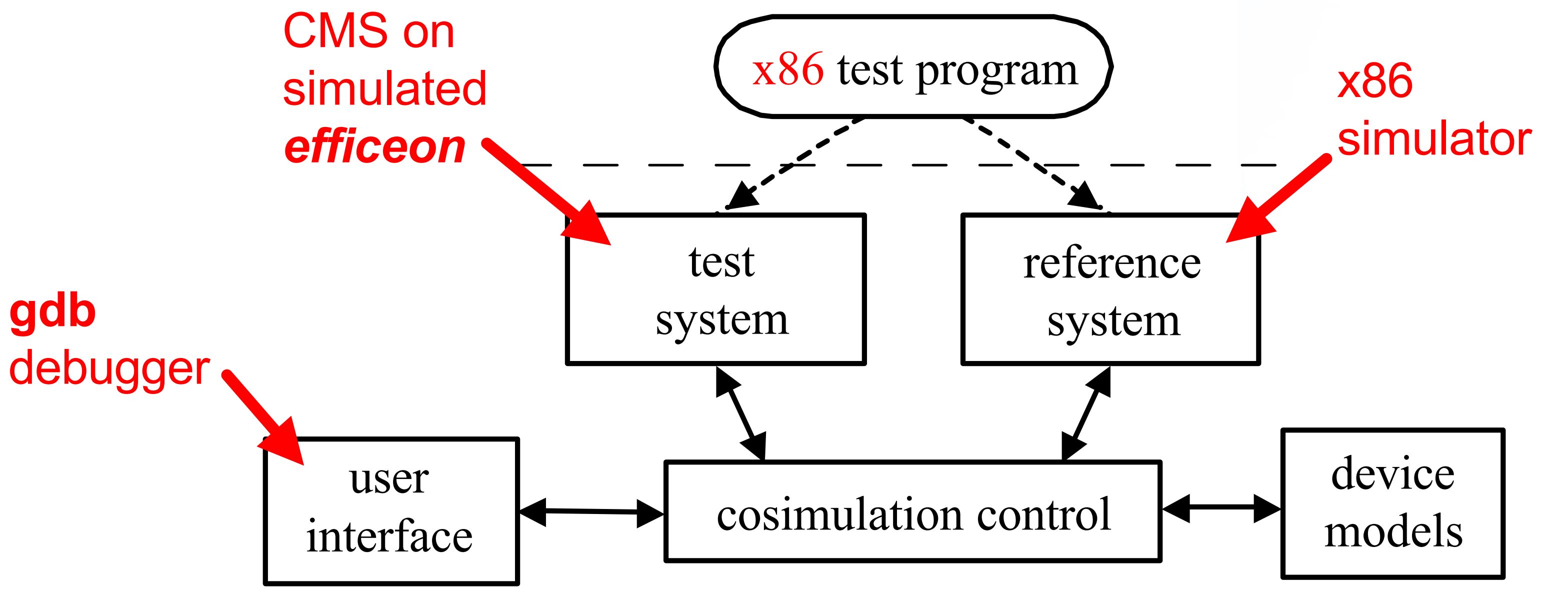


Cosimulation Control

```
forever {
    advance test system to time (now + N)
    advance reference system to same time
    if (state matches)
        checkpoint
        last = now
    else
        isolate_fault(last, now) & stop
}
```



Basic Cosimulation



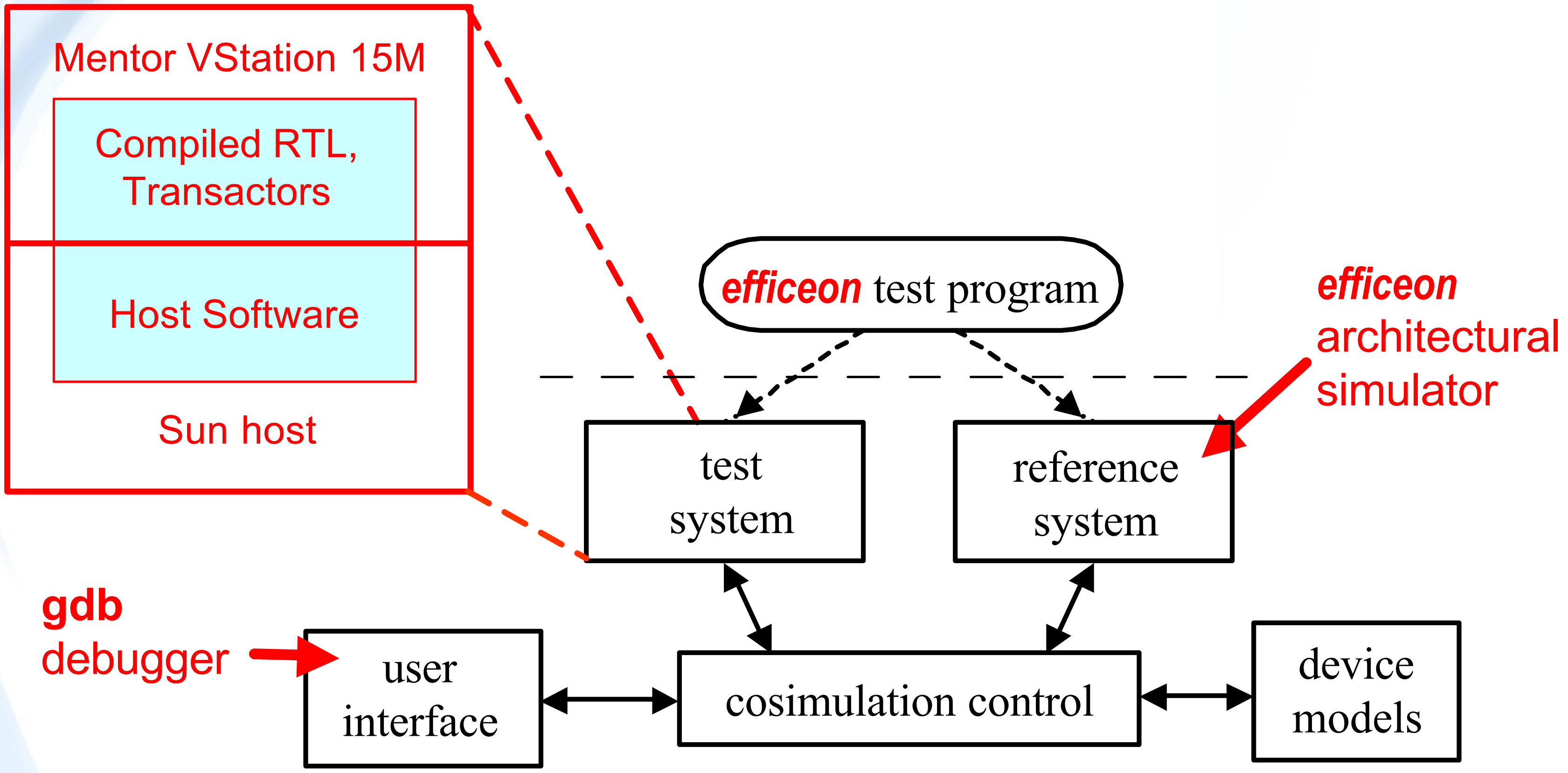
Basic Cosimulation

- Control how often simulators are compared
- Investigate state of system on mismatch
  – Automated test scripts dump information on translation, x86 state, etc., at point of failure
- Use checkpoints to automatically narrow point of failure
- System can collect statistics, traces, etc.

- *Key debugging tool for early-stage CMS*

Emulator Cosimulation

- *Can we apply cosimulation paradigm to hardware debug?*



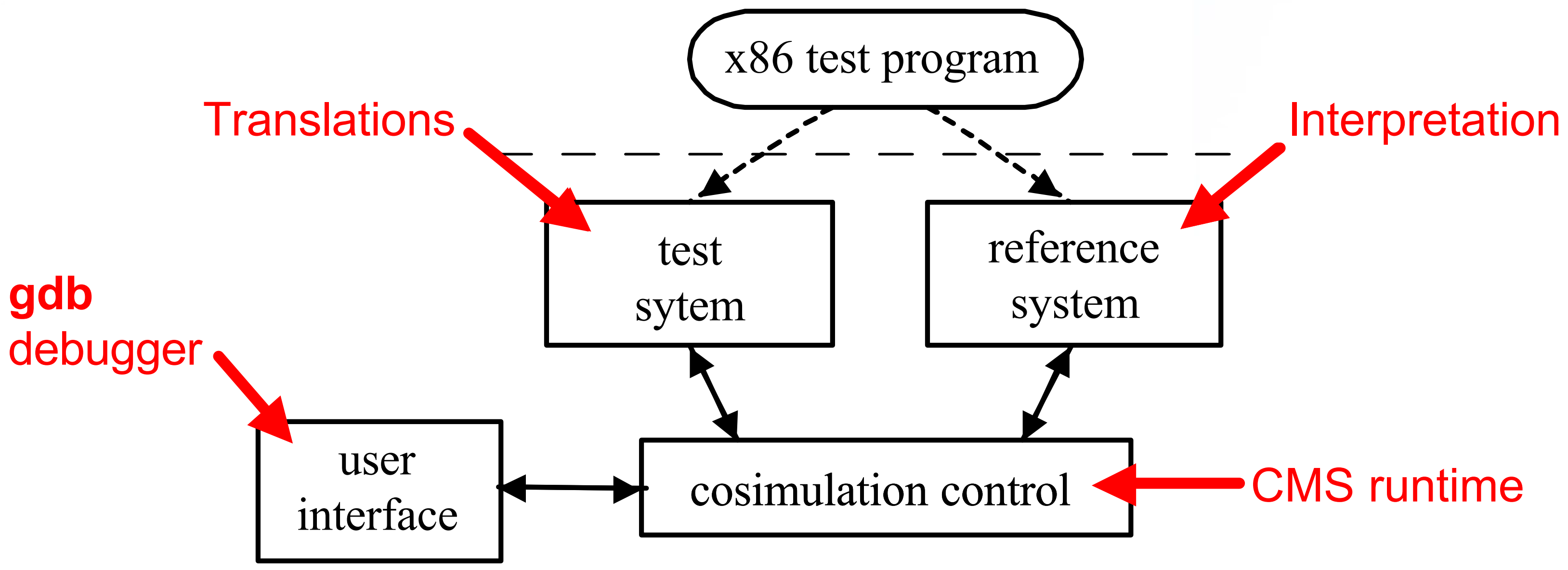
Emulator Cosimulation

- ***Transactors*** added to HW model
  - Verilog compiled with processor RTL
  - Interface to host software
  - Access to internal test system signals
  - Control of system clock
- Successful:
  - Booted many operating systems, ran applications
  - Found many bugs, holes in test suite
  - Found bugs that would have produced dead silicon
  - Helped isolate post-silicon bugs much faster

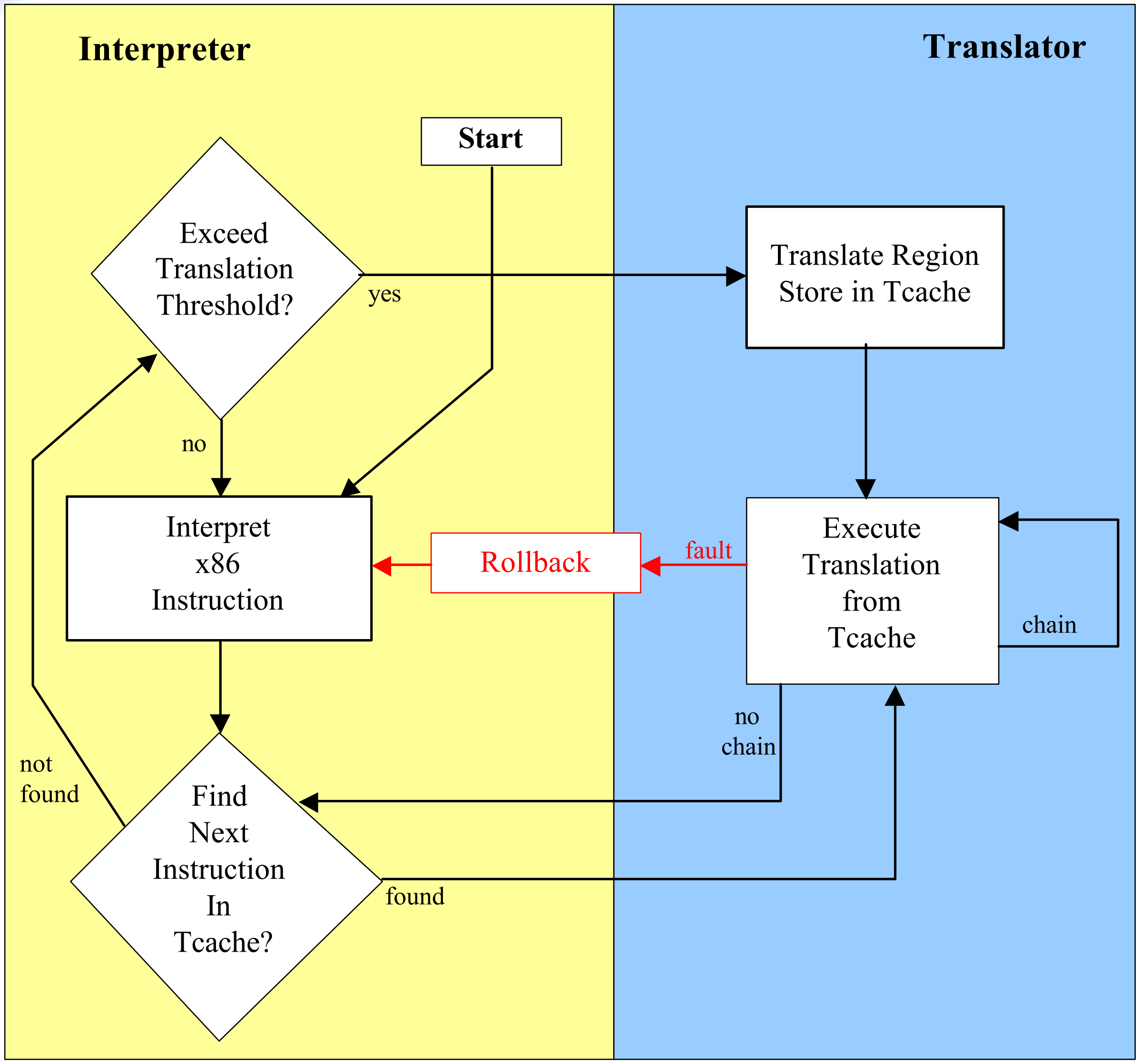
Self-Cosimulation

- Simulator- or emulator-based cosimulation is valuable but slow – simulates everything twice
- CMS has two mostly independent execution engines
- Compare them against one another!

Self-Cosimulation



Self-Cosimulation


Interpreter
Translator
Start
Exceed Translation Threshold?
yes
no
Translate Region Store in Tcache
Interpret x86 Instruction
Rollback
fault
Execute Translation from Tcache
chain
no chain
not found
Find Next Instruction In Tcache?
found

Self-Cosimulation

Instruments newly created or modified translations, s.t.:

- Entry is redirected to prologue for setup and removal of instrumentation after some number of cosimulations
- A commit or exit stops translation execution:
  - Capture x86 state
  - Rollback to previously committed state
  - Interpret to point where translation stopped
  - Compare to saved x86 state

Self-Cosimulation Details

- Cosimulating forever would take a long time
  - – Counter in prologue – remove after ***n*** executions
- Internal commits – only cosimulate between commits
  - – Replace internal commits by traps
  - – Continuing in translation after cosimulating segment terminated by commit requires saving all state
- Can’t cosimulate locked segments
- Some instructions can’t be repeated (e.g., RDTSC)

Self-Cosimulation

- Self-cosimulation behaves like normal CMS…
  - … so it can be cosimulated for debugging purposes
- Self-cosimulation on hardware:
  - Is not significantly slower than normal CMS …
  - … so it can be applied to the full test suite.
- Another highly valuable tool:
  - Found numerous hardware and CMS bugs
  - Usually easy to isolate: pinpoints a translation

Outline

- ***Crusoe*** / ***efficeon*** Background
  - System Architecture
  - CMS Structure
  - Key hardware features
  - Benefits
- CMS Paradigm: speculation, recovery, and adaptive retranslation
  - Example: Aggressive scheduling – exceptions and aliases
  - Example: Self-modifying code
- Co-simulation for Testing
  - Simulator / emulator / self
- Summary

Summary

- Commit / rollback is fundamental to architecture
  - Allows simple translations that pass off unusual cases to interpreter
  - Allows translations to make aggressive assumptions and recover if wrong
  - Central to self-cosimulation facility

Summary

- Key paradigm for resolving compatibility/performance tension is speculation, recovery, and adaptive retranslation
- The devil is in the details: a successful solution must deal with unpleasant architectural details
  - Precise exceptions
  - Interrupts and DMA
  - Memory-mapped I/O
  - Self-modifying code

  All techniques developed for real performance problems

Summary

- Valuable test paradigm: cosimulation
  - Basic cosimulation for CMS development under simulation
  - Emulator cosimulation for pre-silicon hardware
  - Self-cosimulation for CMS+HW test on silicon
  - All have been invaluable for detecting and isolating bugs

Summary

- CMS is unique:
  – Commercially available, fully compatible x86 implementation
  – VLIW architecture is simple and unconstrained
  – CMS software layer provides flexibility
  – Performance comparable to a pure hardware implementation

Questions?

For more information

[Klaiber00] Alexander Klaiber, "*The Technology Behind the Crusoe Processors*," White paper, January 2000, http://www.transmeta.com/pdf/white_papers/paper_aklaiber_19jan00.pdf .

[DGBJ+03] James C. Dehnert et al., "*The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation To Address Real-Life Challenges*," Proc. of the 2003 Int'l Symp. on Code Generation and Optimization, 23-25 March 2003.

[KlCh03] Alexander Klaiber and Sinclair Chau, "*Automatic Detection of Logic Bugs in Hardware Designs,*" Proc. of the 4th Int'l Workshop on Microprocessor Test and Verification, 29-30 May 2003.

US Patent Office

Related Work I

- Software emulation system classifications (Altman):
  - Interpreters
  - Static (offline) translators
  - Dynamic (online) translators and optimizers

  CMS contains an interpreter and a dynamic translator
- Self-hosted vs. cross-hosted

Related Work II

- Self-hosted systems: usually optimization or instrumentation
  - HP Labs’ Dynamo and DELI
  - Can fall back on native execution:
    - No need to deal with problematic or optimal code
- Virtual target emulators:
  - IBM migration of AS/400 to PowerPC
  - Java virtual machines: Sun HotSpot, IBM Jalapeño, LaTTe
  - Similar tradeoffs between translation cost and code quality, but much more tightly controlled “machine” semantics

Related Work III

- Cross-hosted emulators for system migration
  - DEC tools: static translators
    - VEST: VAX/VMS to Alpha/OpenVMS
    - mx: MIPS/Ultrix to Alpha/OSF1
  - New host usually much faster than target
  - Escape valve: port to native host code

  - DEC FX!32: x86/WinNT to Alpha/WinNT
  - Interpreter with offline static translator and database
  - Imperfect emulation: 64-bit FP, no WindowsNT debug API

  - HP Aries: HP-PA to IA-64
  - Interpreter with dynamic translator

Related Work IV

- Migration tools to capture another vendor's applications
  - Hunter System's XDOS: x86 DOS on RISC
  - Modest performance requirements
  - Special-case and manual intervention
- Closest match to CMS: IBM Research DAISY
  - PowerPC or System/390 to tree VLIW
  - Interpreter and dynamic translator
  - Different region selection (tree regions)
  - State repair for precise exceptions
  - Only fine-grain protection for SMC