Wabi Cpu Emulation

Paul Hohensee
Mat Myszewski
David Reese

Sun Microsysytems

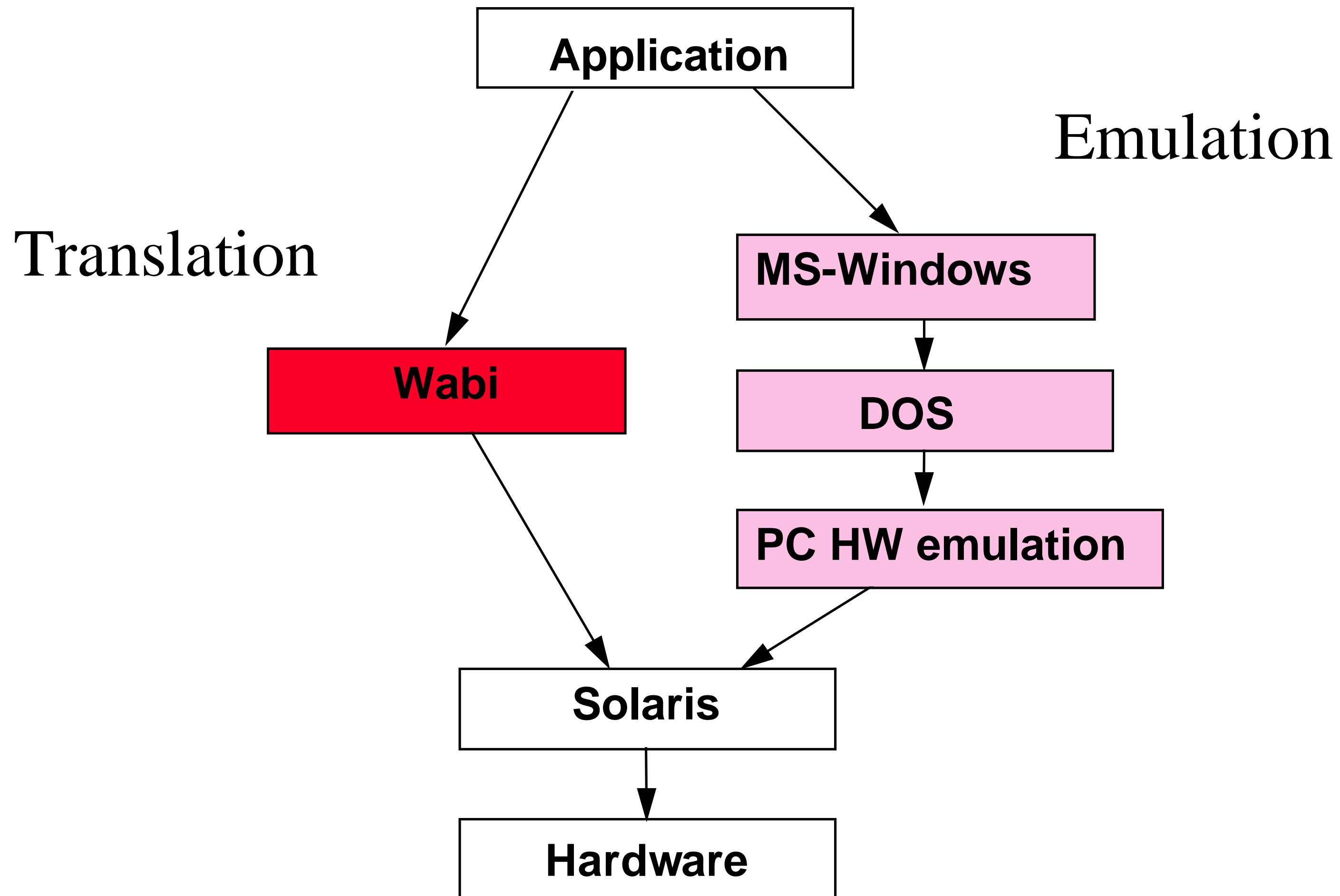
Overview

1. Wabi design
2. Cpu emulation issues
3. Hardware/software tradeoffs
4. Emulator design
5. Interpreter design
6. Translator design
7. Translator improvements
8. Memory management issues
9. x86 emulation statistics
10. Performance data

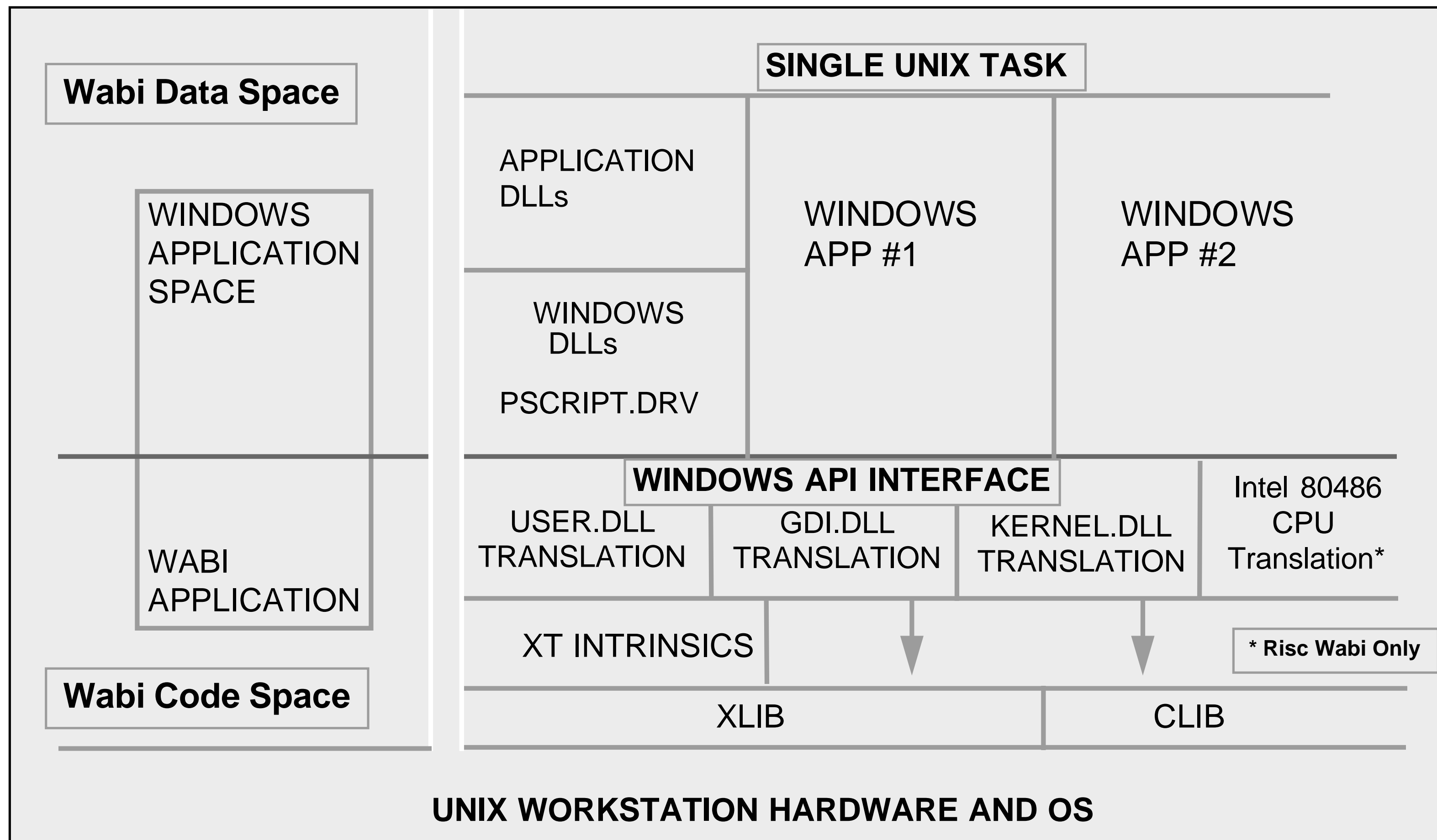
Wabi:
A personal productivity environment for the UNIX user

- Runs MS-Windows personal productivity applications
  - E.g. Microsoft Office, Lotus SmartSuite, Lotus Notes, CorelDraw, WordPerfect, Quattro Pro
- Integrates PC applications into the UNIX/X-Windows environment
- Runs on Intel and non-Intel systems
- Offers the benefits of two platforms in a single desktop

Translation vs Emulation



Wabi Architecture



Emulator Goals

- Run correct Windows applications on multiple RISC platforms
  - as fast as possible
  - in as small a footprint as possible
- Favor speed over space, within reason
- x86 memory little-endian regardless of host
  - simplifies external interfacing
- Emulate 80486 ring 3 protected mode ISA, including the fpu (using 64, not 80-bit numbers) and the single-step (trace) trap
- Maintainable and portable

Emulator Non-Goals

- No GDT or IDT management (this *is* ring 3)
- No TSS or task support
- No real or v86 mode
- No mmu or paging
- Relaxed protection checking
  - No limit checking on data accesses
  - No read-only checking
- No debug register support
- Simple user code and data selector types only (e.g., no call gates)

Emulation Design Issues

- Interface: Hardware simulator - non-reentrant, execute until an exception and exit
- Always translate on first code reference
  - no hybrid execution model to switch between interpretation and translation
- Internal memory management subsystem for temporary space, auxiliary data and host code buffers
- Testing and fault isolation must be designed in
  - single instruction and per interval step functions
  - in-vitro tracing
- Support for asynchronous interrupts

Static vs Dynamic

- Given that a static translator, e.g. DEC's FX!32, requires a dynamic component, why not go all the way?
- Advantages:
  - No extra disk space (3x expansion in FX!32)
  - Retranslate efficiently when guard predicates fail
  - Dynamically generated app code, e.g. video, optimized
  - Adaptive optimization can exceed static quality
- Disadvantages:
  - Limited time to generate code
  - Overhead varies with execution frequency

Desirable Hardware

- Want software equivalent of the P6 hardware interpreter
  - access control on 4Kb pages
  - load/store swapped on big-endian systems
  - load/store misaligned
  - ALU and register resources of the P6 core (mmu emulation) (flat register model)
  - 16-bit literal instruction operands
  - sub-register moves, e.g. PA-RISC & PowerPC
  - integer divide overflow trap
  - 80-bit floating point

Floating Point

- Mapping x87 stack to host fp registers
  - guard code checks potential stack over/underflow and height
  - ABI must support at least 8 callee-saved registers
- Denorm operand trap and inexact roundup (C1) require hardware
- 80-bit precision quagmire, most cpu emulators use 64-bit
- x87 delayed exception model
  - requires user-settable traps or checks in generated code

Emulator History & Definitions

- Four emulators: three interpreters and one dynamic translator
  – 1992: 286 ring 3 protected mode interpreter
  – 1993: 486 ring 3 protected mode interpreter
  – 1994: 486 ring 3 protected mode dynamic translator
- Interpreter
  – Execute one (sometimes two) x86 instructions per dispatch
  – Emulate each x86 instruction via ‘microcode’ routine
  – Cache only ‘microcode’ routine addresses
- Translator
  – Execute many x86 instructions per dispatch
  – Generate and execute SPARC code on-the-fly
  – Cache SPARC code for later reuse

gcc

- High-Level Assembler
  - seven uses of ‘asm’ in the interpreter
- Standard source language across all platforms
  - IBM is a particular offender
- Compiler source availability
  - Independent of other group’s resources and schedules
  - Flow analyzer and register allocator now handle *very* large routines: 1000’s of BB’s, > 65k temporaries
- Performance-critical language extensions
  - First-class labels enable threaded interpreter implementation
- Generally excellent generated code as of late 1992

Interpreter Overview

- Relatively small working set, though image is large
- Instruction Execution
  - Pure threaded interpreter
  - Delayed x86 ICC materialization
  - Host registers used for x86 machine state, visible and hidden
  - Instruction overrides handled by duplicated prefix code and base+index jump in dispatch
- Instruction Decode
  - 16 bits at a time
  - High frequency combinations directly executed, others dispatch to slower generic versions

Interpreter Internals

- Enormous C function - 116K SPARC instructions
- Callouts for:
  - 64-bit arithmetic
  - libm fp functions
  - code and data segment management
- Use gcc as a high-level assembler for
  - global register assignment
  - internal routines via label variables
  - a few assembly language macros

Interpreter Instruction Dispatch

- 16 + 2 bit initial dispatch
  - Simple indexed jump on 16 bits of x86 opcode plus 2 bits for data/address size:

    10 SPARC instructions
  - 1Mb dispatch vector, dcache pig, thus...
- Thereafter, dispatch on low bits of mapped x86 instruction pointer
  - Save addresses of SPARC code corresponding to decoded x86 instructions in per-code-segment dispatch vector:

    6 SPARC instructions

Translator Overview

- Dynamic compiler
  - Always compile, fast and dumb, per interval
  - Interval: "augmented" basic block of x86 instructions that may include simple loops and multiple terminating conditional branches

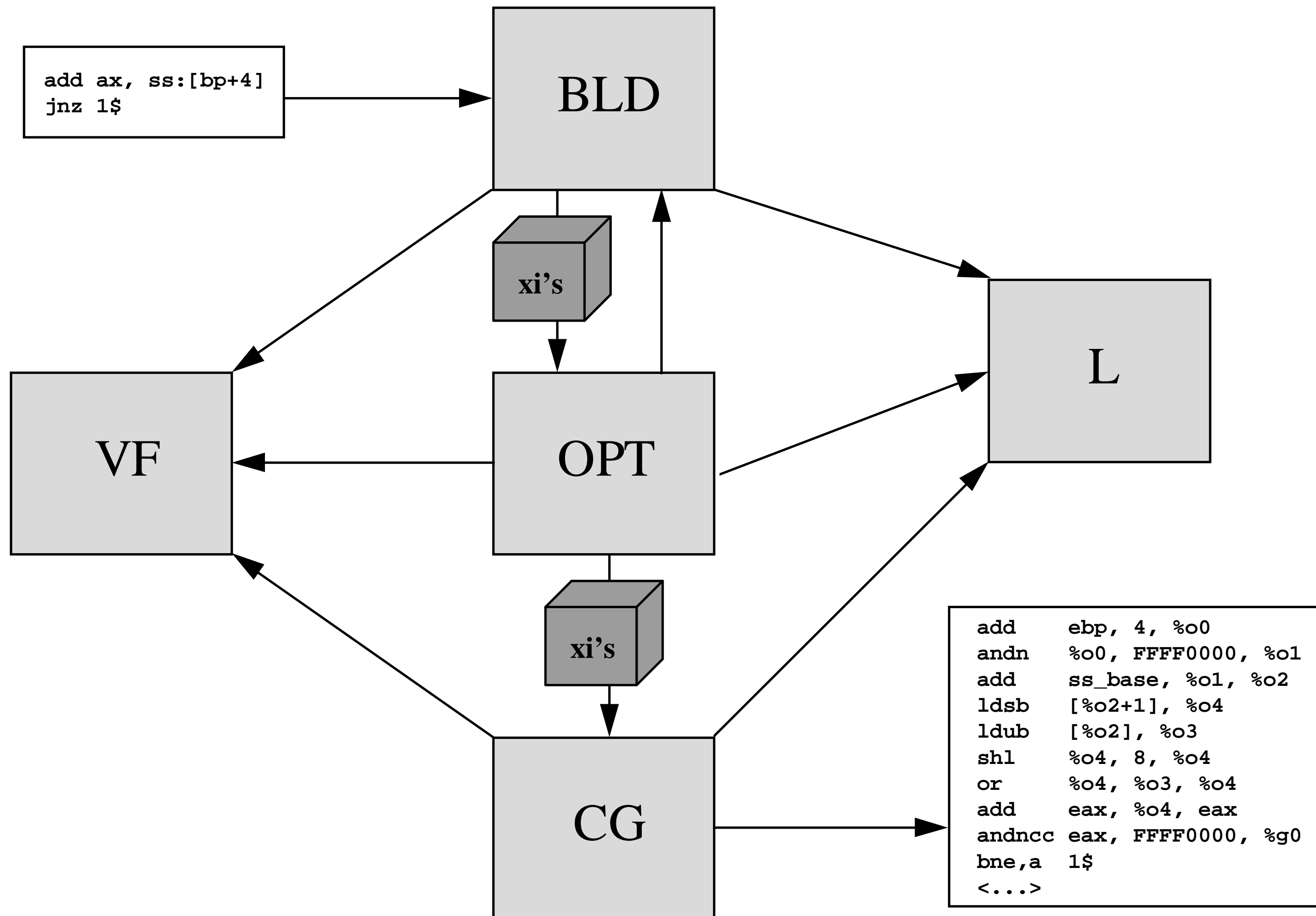
```
        mov     eax, [bp+12]
1$      push    eax
        call    near foo
        jeq     1$
        jc      2$
```

Translator Overview

- Code Cache
  - 1 - 64Mb, default = (physical memory / 4 + 8 Mb)
  - Allocation per code selector: generated code, etc.
  - Oldest generated code discarded when full
- Microcoded x86 FPU (in gcc, via internal routines)
- Overhead

200%	low reuse	=> app launch
< 1%	cpu intensive	=> spreadsheet recalc
< 10%	typical	=> slide show

Translator Design



Translator Advantages

- x86 instructions decoded once, barring invalidation
- All x86 registers mapped to SPARC registers
- No instruction dispatch overhead except for indirect control transfers (mostly subroutine returns)
  - SPARC code for contiguous intervals is jammed
  - Indirect transfers use hash table
- Unnecessary x86 ICC computation reduced
  - Most x86 instructions write ICC’s, almost all are dead
  - Most that aren’t are read-next, so kept in SPARC ICC’s

Translator Advantages

- Most dead x86 register high halves detected

```
andn    src, <register containing 0xFFFF0000>, tmp
and     dst, <register containing 0xFFFF0000>, dst    =>    mov    src, dst
or      dst, tmp, dst
```

  - 10-20% performance boost in 16-bit x86 code

- x86 instruction peepholes

```
    xor   ax, ax   =>   clr      ax   # write zero, don't xor
    or    ax, ax   =>   tst      ax   # don't write ax
    jcc   1$       =>   revjcc   2$   # reverse conditional branch
    jmp   2$
1$
```

- Code and data selector TLB's reduce far control transfer and segment register load overhead
  - ~8% of all instructions in x86 code using 16-bit addressing
  - ~20% performance improvement
  - Data segment register load: same = 4 SPARC instructions, hit = 22, miss > 100

Translator Enhancements

- Reduce ICC materialization by
  - implementing a demand-driven delayed ICC model
  - doing ICC lookahead through 2 levels
  - using larger (multiple BB) intervals
- Extensive x86 instruction sequence idiom recognition
- Inline small functions, especially far ones
- Maximize host register utilization within intervals
- Reallocate x86 memory data to SPARC registers/memory
- Use adaptive optimization to regenerate code for high frequency intervals

ICC Optimization

- Interpreter uses delayed ICC evaluation
  - only ALU instructions delay all ICCs
  - others delay "result" ICCs (ZF,SF,PF)
- Initial translator algorithm computes all live ICCs using lookahead to minimize live sets
  - lookahead is restricted to a single interval
  - indirect control transfers are problematic
- New translator algorithm uses a hybrid algorithm
  - All ICC's needed within an interval are computed
  - ICC's live out of an interval are delayed
  - Delayed state can encode 2 computations

Adaptive Optimization

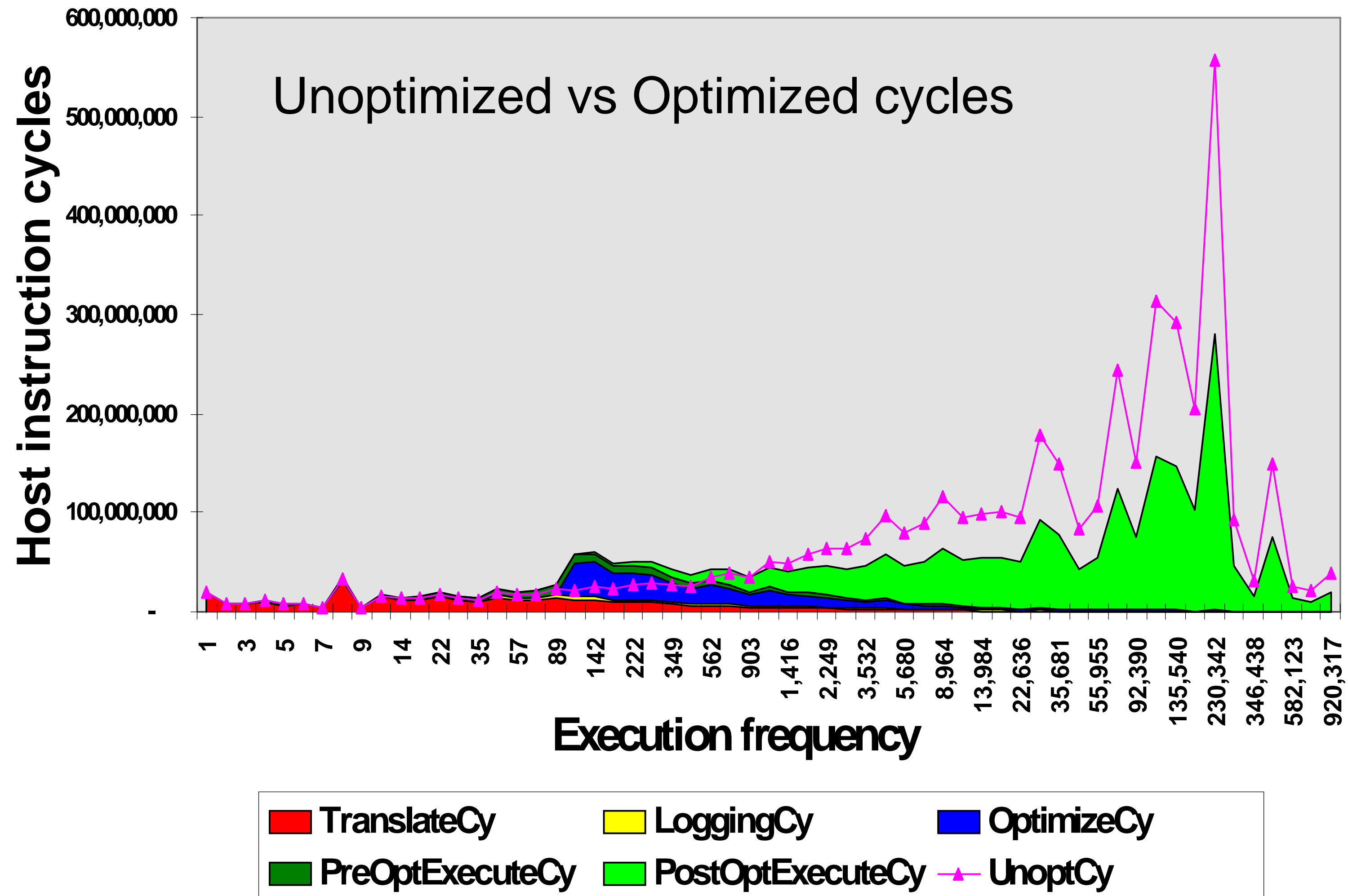
- Based on techniques from OO systems and profile-guided feedback
- Insert counters to trigger reoptimization when execution frequency exceeds threshold
- Mathematical model based on Excel 5 benchmark trace

Parameters:

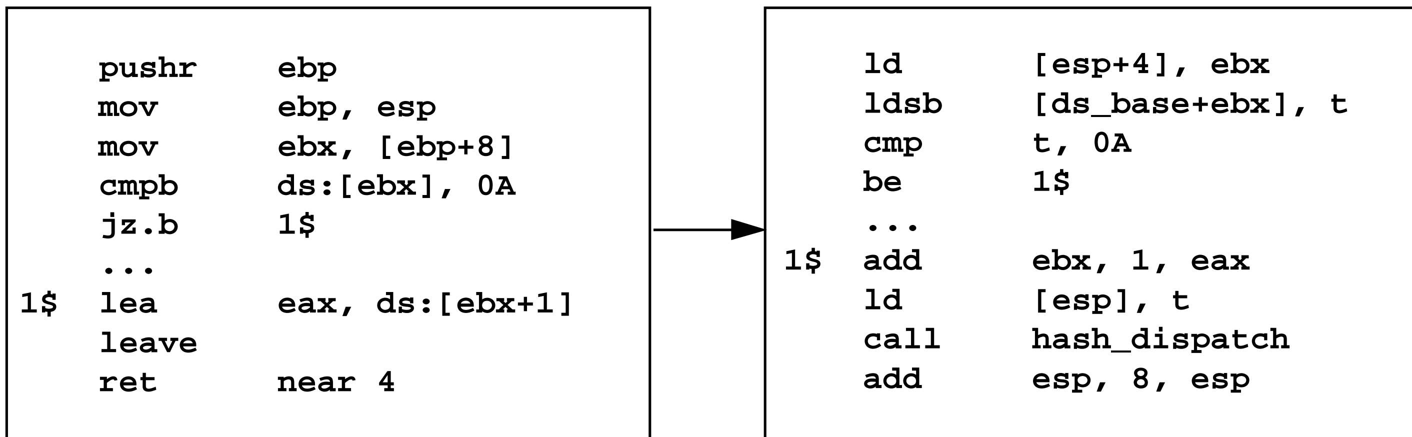
host cpi	1
translator cycles per x86	1015
optimizer cycles per x86	3045
logging cycles per interval	20
unopt to opt host code ratio	0.5
trigger frequency	90

- Speedup: 1.55

Adaptive Optimization



Translator Example



Emulation Performance Limits

- Detecting and reporting x86 exceptions is expensive
  - limits optimization and instruction scheduling
  - requires additional memory to recover state information
  - fpu implementation requires checks in generated code
- Self-modifying code forces retranslation
  - detection via page protection and checksum
- Shortage of host registers for x86 and emulator state
- Shortage of host machine cycles

Dispatch Table Management

- First version used dispatch table with one entry per byte in code segment or fixed size hash table with collision chaining
  - sparse tables (number of entries independent of code size)
  - large contiguous allocation => fragmentation
- New algorithm uses dynamic size hash table with linear collision resolution
  - tables start small and grow based on loading factor
  - up to 3x faster on small (32Mb) systems

Code Buffer Management

- First version used whole code segment as unit of allocation
  - works with many small code segments
  - persistent dead code wastes cache space
  - useless in Win95 memory environment
- New version uses software paging
  - works just as well with large and small code segments
  - old code discarded: most is dead, what isn't is retranslated, minimizing internal fragmentation
  - better space utilization compensates for extra bookkeeping
  - can still invalidate entire code segments

x86 Memory Accesses

- 16-bit code statistics
  - 50% of x86 instructions reference memory
  - 90% of memory references are non-byte width
  - 5% of the memory references are misaligned
  - 16-bit operands: 34% theoretical speedup
    - 7 vs 3 instructions on SPARC V8 vs V9
  - 32-bit operands: 41% theoretical speedup
    - 13 vs 3 instructions on SPARC V8 vs V9
- Misaligned access exception handling and fixups halve theoretical speedups

x86 Memory Accesses

- 32-bit code statistics
  - % of x86 instructions reference memory
  - % of memory references are non-byte width
  - % of the memory references are misaligned
  - 16-bit operands: % theoretical speedup
    - 6 vs 2 instructions on SPARC V8 vs V9
  - 32-bit operands: % theoretical speedup
    - 12 vs 2 instructions on SPARC V8 vs V9

Interval and Instruction Sizes

Excel 5 Benchmark (Win 3.1)

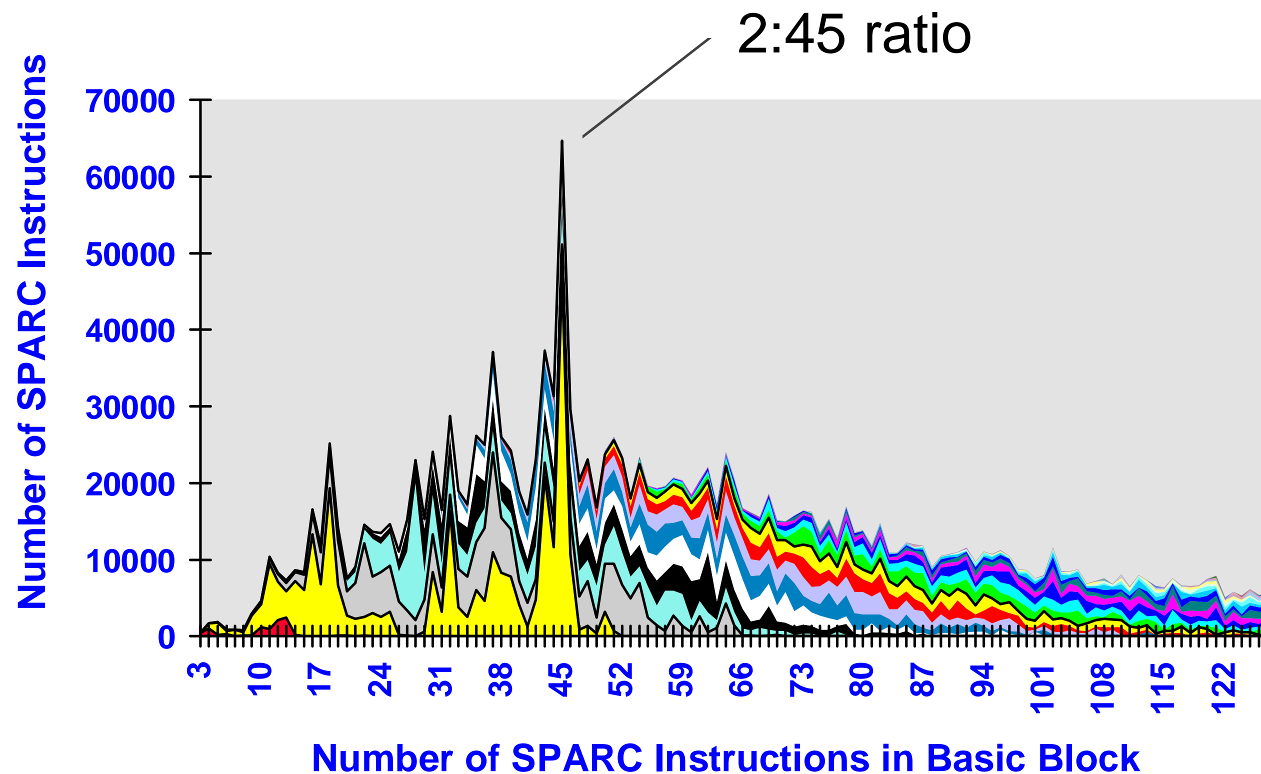
	Static Count			Executed		
	x86	SPARC	Ratio	x86	SPARC	Ratio
Instructions per Interval	5.6	48.6	8.7	5.0	40.7	8.1
Bytes per Interval	15.6	194.4	12.5	13.6	162.7	11.9
Instruction Length	2.8	4.0	1.4	2.7	4.0	1.5

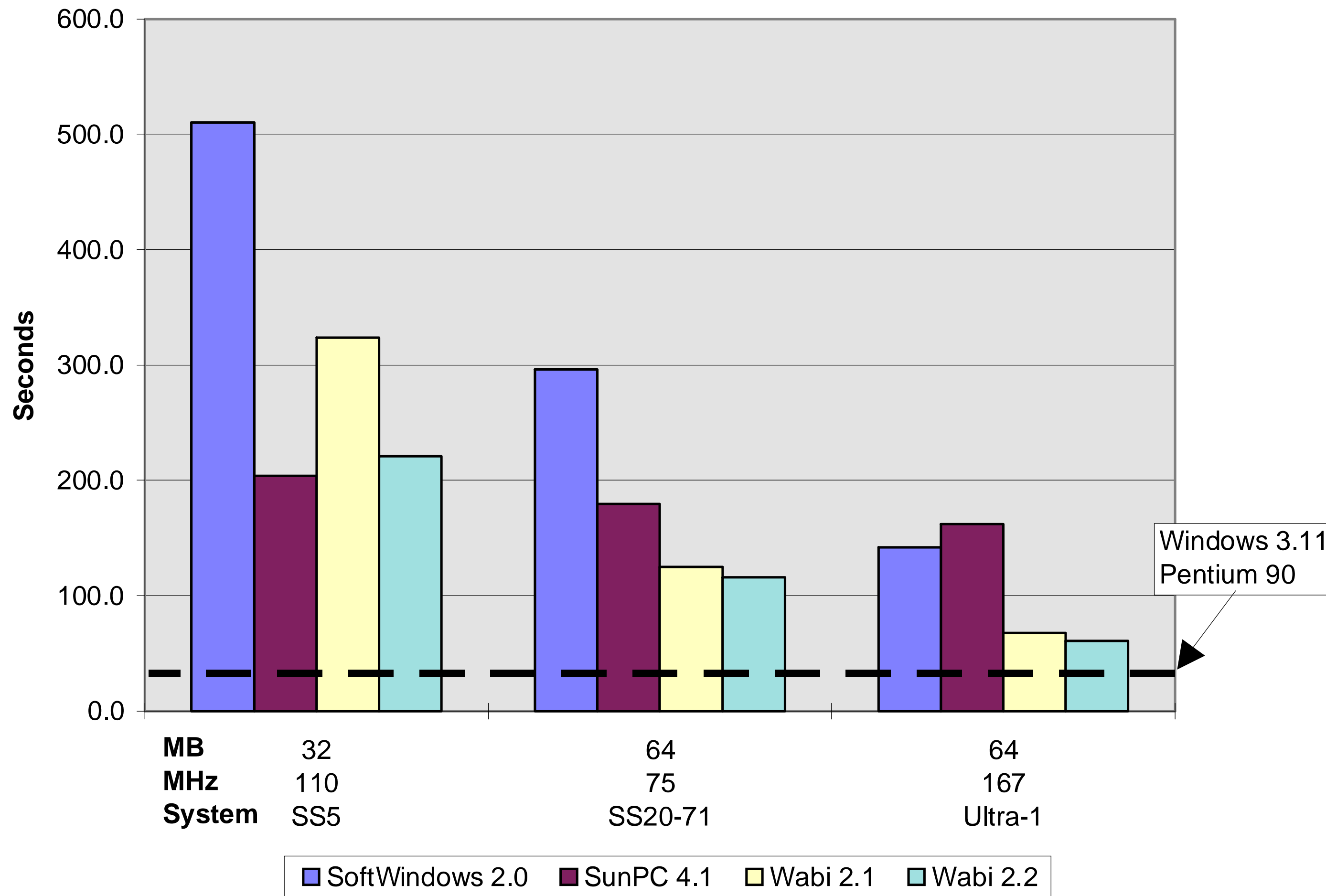
Module, Segment, Procedure, and Interval Sizes

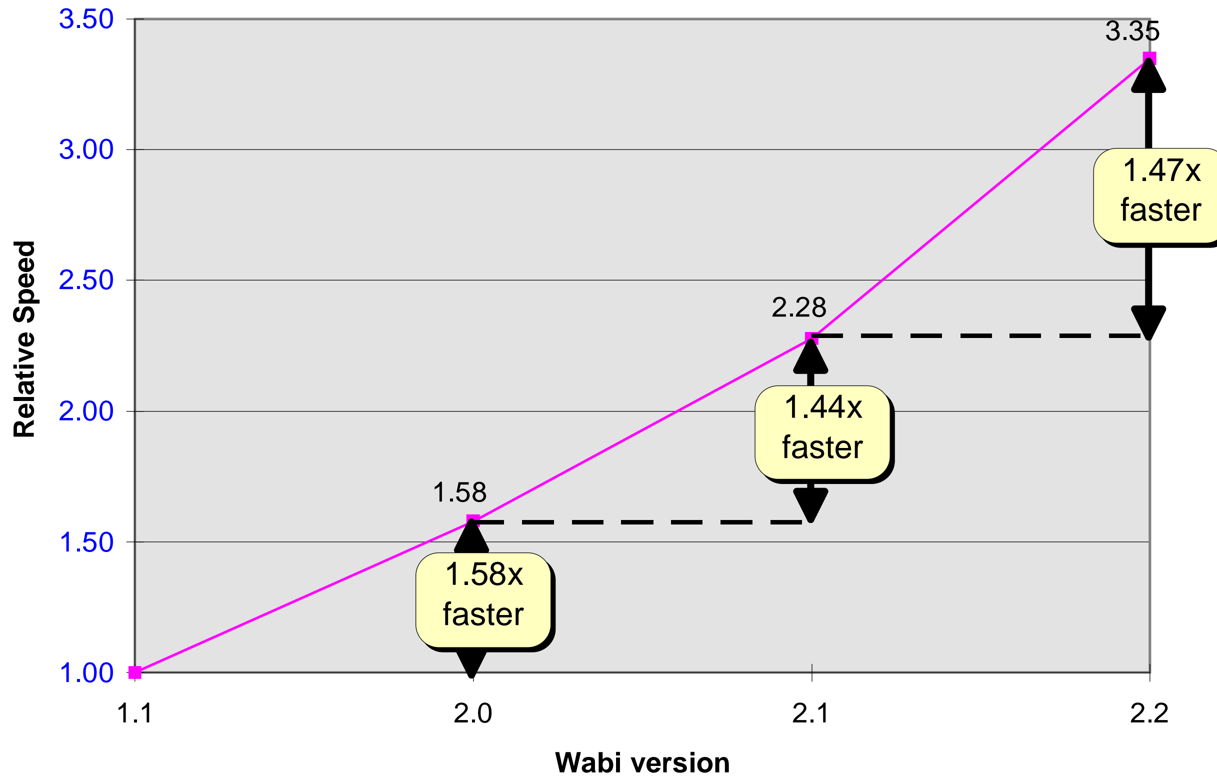
Excel 5 Benchmark (Win 3.1)

	Number	Seg- ments	Proce- dures	Inter- vals	x86 Instrs	Host Instrs	x86 Bytes	Host Bytes
Module	153	12.7	204.0	1966.2	10977.5	95572.5	30664	382290
Segment	1937		16.1	155.3	867.1	7549.1	2422	30196
Procedure	31213			9.6	53.8	468.5	150	1874
Interval	300829				5.6	48.6	15.6	194

Number of SPARC Instructions in Basic Blocks of Given Size




Excel Benchmark Performance
Seconds
600.0
500.0
400.0
300.0
200.0
100.0
0.0
Windows 3.11
Pentium 90
MB
MHz
System
32
110
SS5
64
75
SS20-71
64
167
Ultra-1
SoftWindows 2.0
SunPC 4.1
Wabi 2.1
Wabi 2.2


Wabi Performance
(Excel Benchmark)
Relative Speed
3.50
3.00
2.50
2.00
1.50
1.00
3.35
2.28
1.58
1.47x faster
1.44x faster
1.58x faster
1.1
2.0
2.1
2.2
Wabi version

Conclusions

- Performance must be comparable or better than volume shipping PC running MS-Windows
- Intel to RISC performance ratio is < 2x today
- We expect to achieve a 3 to 1 ratio of generated SPARC to x86 instructions on V9 systems
- Working set size will be larger than an x86 based solution
- Challenges
  - exception handling
  - startup costs
  - reoptimization heuristics