Lecture 10 - Virtualization without direct execution or jitting

Darek Mihocka (Emulators) and Stanislav Shwartsman (Intel) · 2008. Two pure interpreters, Bochs and Gemulator, reach 100+ MIPS in portable code: how they do it, and what small ISA additions would help.

Contents

01 The argument

02 Gemulator: guest memory

03 Bochs: the CPU loop

04 Bochs: the trace cache

05 Bochs: branch prediction

06 Bochs: lazy flags

07 Measurements

08 ISA extensions

09 Takeaways

01

The argument

Three ways to run guest code

The paper asks which one is worth its cost on phones, game consoles and older PCs.

APPROACH	HOW GUEST CODE RUNS	NEEDS FROM THE HOST	WEAK SPOT
Direct execution	on the CPU, trapping to the hypervisor	same ISA plus VT-x or SVM	tied to one host architecture
Jitting (e.g. QEMU)	compiled into a code cache, then run	a code generator per host ISA	memory, cache pressure, self-modifying code
Interpretation	decoded and dispatched in C or C++	a C or C++ compiler	assumed too slow; the paper disputes this

- The paper argues for the shaded row: Bochs and Gemulator are both pure interpreters.

Design goals: portability first, peak speed second

The authors call maximizing best-case speed above all else a design flaw. Their six goals:

Portable

No host-specific optimizations, so it runs on legacy x86 and non-x86 hosts.

Bounded memory

A small hypervisor footprint for phones and game consoles.

Bounded worst case

Predictable cost per instruction, so tracing stays cheap.

Fast dispatch

Getting from one guest instruction to its handler quickly.

Fast memory mapping

Guest-to-host address translation in software.

Light hardware help

Small ISA additions, not heavyweight virtualization hardware.

Why not jit?

Dynamic recompilation is the usual route to fast emulation. With portability and bounded memory as goals, the authors found it may not beat a well-built interpreter.

01

Fresh code is slow

Newly generated code incurs long latencies before it runs at full speed.

02

Cache pressure

Translated code competes with the emulator for the code cache and L2.

03

Self-modifying code

Detecting and handling guest writes to compiled code is costly.

THE HOST CHANGED TOO

At the same 2.66 GHz clock, a Core 2 ran interpreters such as SoftMac two to three times faster than a Pentium 4. In one CPU generation, interpretation became far more attractive.

What a portable interpreter buys

01

Deterministic execution

A saved VM replays identically on any host. A host upgrade shouldn’t change guest behavior.

02

Cheap instrumentation

Pin and Nirvana can slow guests by orders of magnitude. An interpreter is already in the loop.

03

Security in the hypervisor

Virus checks or hot patches on running guest code, as Microsoft’s Vigilante showed.

04

Fewer host assumptions

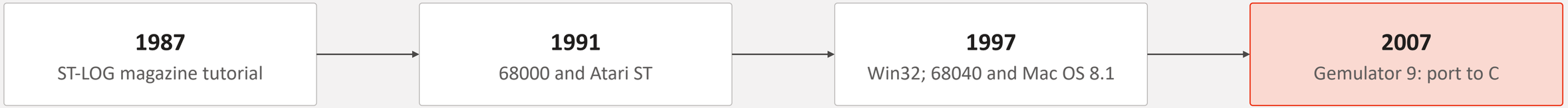
No hardware FPU or MMU required, which suits low-power devices.

02

Gemulator: guest memory

Gemulator: twenty years of one interpreter

An Atari ST and classic Macintosh emulator for MS-DOS and Windows, now being moved from assembly to C.



THE OLD WAY

An interpreter in 80386 assembly, retuned by hand for the 486, Pentium Pro and Pentium 4. Tied to MS-DOS and 32-bit Windows.

THE NEW GOAL

Portable C for 32- and 64-bit hosts that doesn’t care about byte order, with as little speed lost as possible.

Byte swapping without BSWAP: store memory backwards

The 68000 is big-endian, the 80386 little-endian, and the 386 has no BSWAP instruction. Gemulator reverses the whole guest address space instead.

THE MAPPING

```
host = K - guest    ; constant K
guest G+1          -> host H-1
guest 100..103     -> K-100..K-103
```

Guest addresses climb while host addresses fall, so all 16 MB of guest memory is one giant integer stored in reverse.

READING 32 BITS AT GUEST 100

```
last = 100 + sizeof(int) - 1   ; 103
val  = *(int *)&K[-last]       ; one load
```

Point at the last byte of the access. Its four bytes now sit in little-endian order, so an ordinary host load returns the right value.

- On MS-DOS, ~99.9% of accesses passed a bounds check, letting a 33 MHz 386 match an 8 MHz Atari ST at full speed.

XOR translation: 4 GB of guest space, in blocks

Subtracting from all-ones equals XOR with all-ones, so each power-of-two block gets its own constant.

```
0xFFFFFFFF - 0x12345678 = 0xEDCBA987
0xFFFFFFFF ^ 0x12345678 = 0xEDCBA987
```

GUEST RANGE	HOST BLOCK	XOR CONSTANT	EXAMPLE
`0x00000000–0x0FFFFFFF`	`0x30000000–0x3FFFFFFF`	`0x3FFFFFFF`	guest 0x00012345 → host 0x3FFEDCBA
`0x00000000–0x3FFFFFFF`	`0x20000000–0x5FFFFFFF`	`0x5FFFFFFF`	1 GB block, not 1 GB aligned: still works

ENDIAN AGNOSTIC

Opposite byte order: the constant’s low bits are 1. Same byte order: they are 0.

THE FLAW

I/O ranges were pushed above 0x80000000 to fail. 64-bit Windows breaks that assumption.

A software TLB: a hash lookup with a fuzzy compare

Gemulator 9 replaces the page table with 2,048 entries, each covering a 128-byte guest block and holding two words: the block’s guest base and its XOR constant.

```
void *pvGuestLogicalWrite(ULONG addr, unsigned cb)
{
  ULONG i = ((addr + cb - 1) >> 7) & 2047;
  void *p = (addr ^ tlbRW[i*2+1]) - (cb - 1);

  if (0 == (dwBaseMask & (addr ^ tlbRW[i*2])))
    return p;       /* hit: host pointer   */
  return NULL;      /* miss: slow path     */
}
```

Simplified from the paper’s Listing 2.4.

INDEX

Bits 7–17 of the last byte’s address pick one of 2,048 entries.

FUZZY COMPARE

XOR with the cached base and mask the low bits. Zero means a hit.

NEVER FAULTS

Misses, cross-block accesses and I/O all return NULL to the slow path.

- The C version compiles almost as tightly as the hand-written assembly: one extra register spill.

Three TLBs, sized by experiment

Separate tables keep a copy’s source, its destination and the program counter from evicting each other.

TABLE	BLOCK	HOST MEMORY	USED FOR
Write TLB	128 B	2,048 × 8 B = 16 KB	stores and read-modify-write
Read TLB	128 B	2,048 × 8 B = 16 KB	loads
Code TLB	32 B	16 KB + 256 KB of handler pointers	instruction dispatch, one pointer per address

ENTRIES MATTER MORE THAN SIZE

Hit rates held even at 32-byte blocks. 512 entries was acceptable, 2,048 best; more added nothing.

WHY NOT 32 BYTES FOR DATA

2,048 × 32 B wraps every 64 KB, so copies alias, and every 8th sequential access misses.

- Total for all translation and dispatch tables: just over 300 KB.

Catching self-modifying code with the TLBs

No guest block is ever cached as both writable data and executable code.

1 **Guest runs code in a 32-byte block**
code TLB entry filled; handlers start as stubs

2 **First dispatch of each address**
the stub finds the real handler and patches the entry

3 **Guest writes into that code**
the write TLB misses, because step 1 evicted its entry

4 **Guest jumps back into it**
the code TLB misses, so the new bytes are decoded fresh

Evict write entry

Patched once

Flush 6 code entries

Re-dispatch

- Six = four 32-byte code blocks under the 128-byte data block, plus a guard block each side. Step 4 is our inference.

Gemulator 9: almost everything hits the fast path

Boot Mac OS 8.1, then run Speedometer 3.23's arithmetic, floating-point and graphics tests.

98.6%

instruction dispatch hit rate

43 M misses in 3.216 B guest instructions

95.6%

data access hit rate

132 M misses in 3.014 B accesses, I/O included

15–22

host cycles per guest instruction

120–170 MIPS on a 2.66 GHz Core 2

WHAT IT NEEDS FROM THE HOST

Plain 80386 instructions, no BSWAP, no exceptions, and under 1 MB of working memory.

WHY IT'S FAST

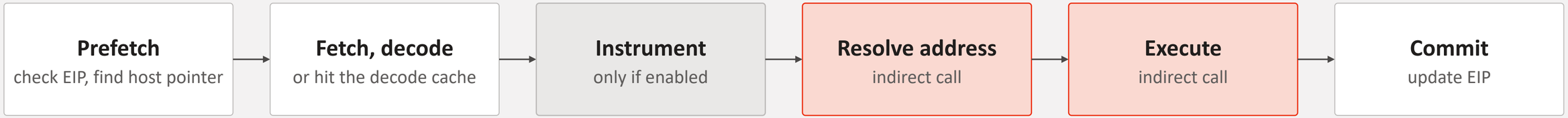
Short, predictable paths whose branches predict well. Bochs shows why that matters most.

03

Bochs: the CPU loop

Bochs 2.3.5: six stages for every guest instruction

Profiling put about half the run time in the CPU loop, mostly on fetching, decoding and dispatching rather than on executing.



THE FRONT END IS THE COST

Variable-length x86 instructions, many decode templates and three operand sizes make fetch and decode the heaviest stage.

THE RESULT OF THIS WORK

Bochs 2.3.5 to 2.3.7: over three times faster, with only portable C++ changes and no loss of accuracy.

- Shaded: the two indirect calls in the heart of the loop. Section 05 is about them.

The running example: summing an array

One small guest loop, used in every Bochs walkthrough that follows.

GUEST CODE

```
401000  8B 04 9E  mov eax,[esi+ebx*4]
401003  01 C2     add edx,eax
401005  43        inc ebx
401006  39 CB     cmp ebx,ecx
401008  75 F6     jne 401000
40100A  89 D0     mov eax,edx
40100C  C3        ret
```

STARTING STATE

esi 0x500000	ebx 2
ecx 5	edx 10
eax —	EIP 0x401000

Guest memory at esi

+8 7	+12 5	+16 9

- The body runs three times (ebx = 2, 3, 4) and returns the sum, 10 + 7 + 5 + 9 = 31, in eax.

One instruction through the CPU loop

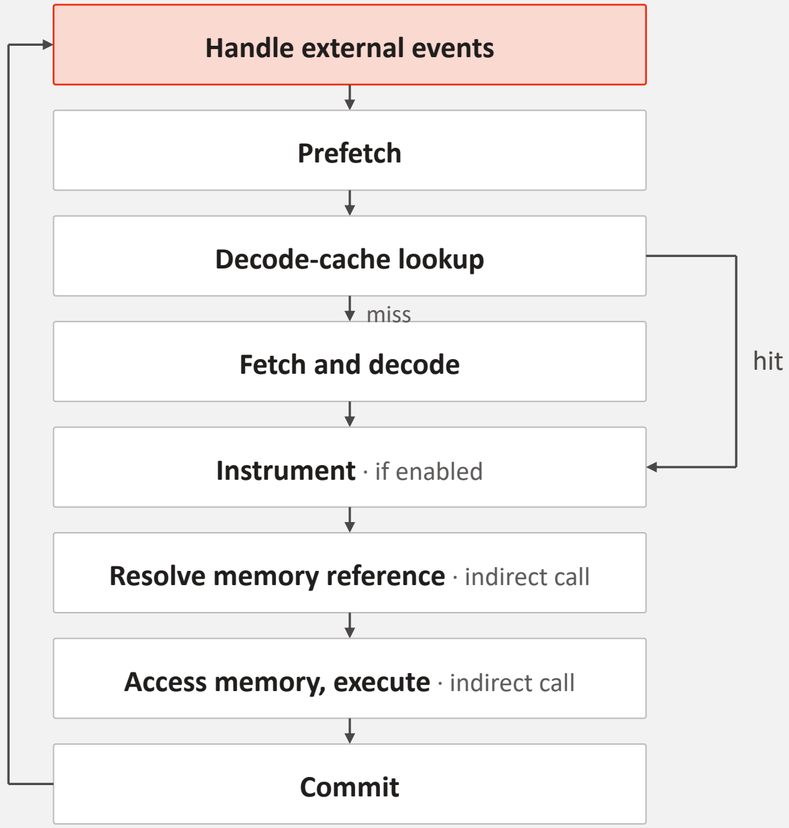
GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax —	ebx 2
ecx 5	edx 10
esi 0x500000	EIP 0x401000

this stage now / this instruction
stage done
stage skipped
next instruction

BOCHS 2.3.5 CPU LOOP



STEP 1 OF 9

Check for external events

Each pass first checks for pending asynchronous events, such as an interrupt from an emulated device. None are pending, so the loop moves on to the instruction at EIP.

STATE

EIP = 0x401000 · first pass

One instruction through the CPU loop

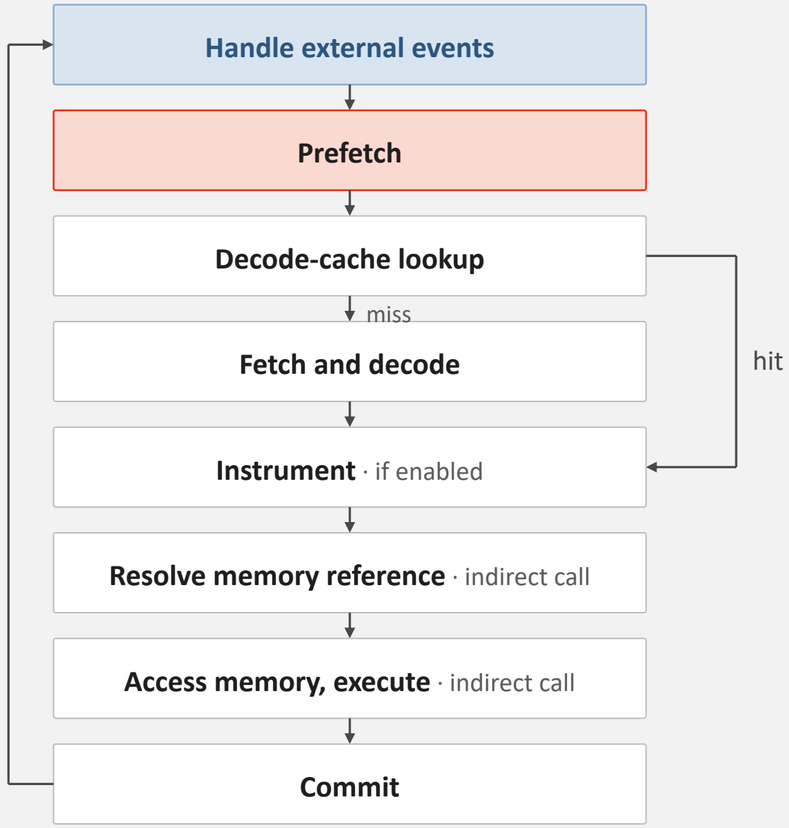
GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax —	ebx 2
ecx 5	edx 10
esi 0x500000	EIP 0x401000

this stage now / this instruction
stage done
stage skipped
next instruction

BOCHS 2.3.5 CPU LOOP



STEP 2 OF 9

Prefetch

Check that EIP is inside the code segment and that the current privilege level may fetch from it, then compute a host pointer to the guest page. Page timestamps for self-modifying code are updated here too.

STATE

host fetch pointer → guest page 0x401000

One instruction through the CPU loop

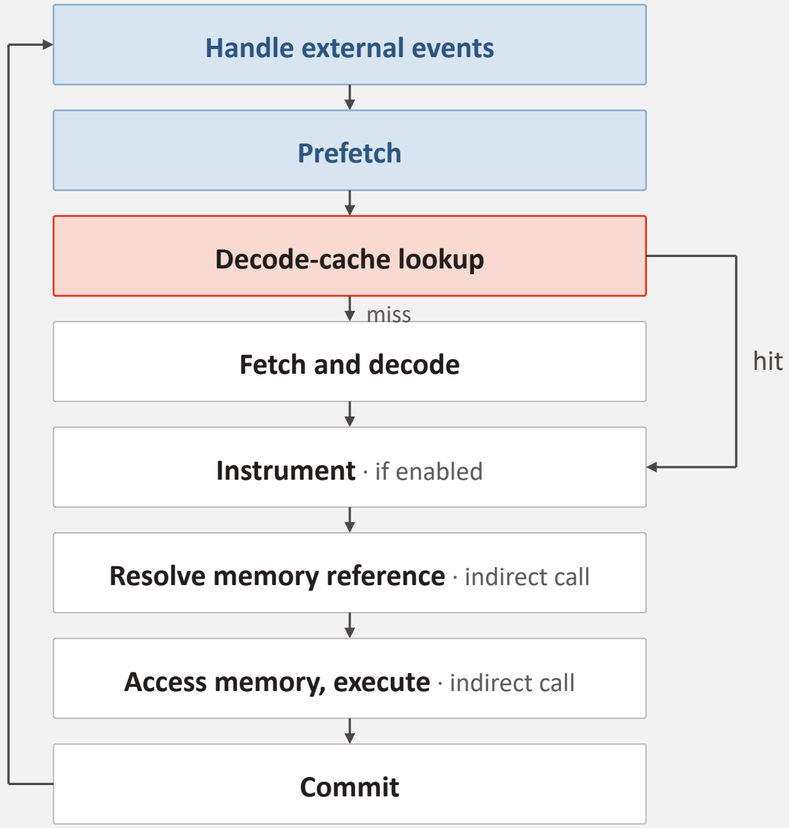
GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax —	ebx 2
ecx 5	edx 10
esi 0x500000	EIP 0x401000

- this stage now / this instruction
- stage done
- stage skipped
- next instruction

BOCHS 2.3.5 CPU LOOP



STEP 3 OF 9

Decode cache: miss

Bochs looks for an already-decoded copy of the instruction at this address. This is the first pass, so it misses and the bytes must be fetched and decoded.

STATE

lookup 0x401000 → miss

One instruction through the CPU loop

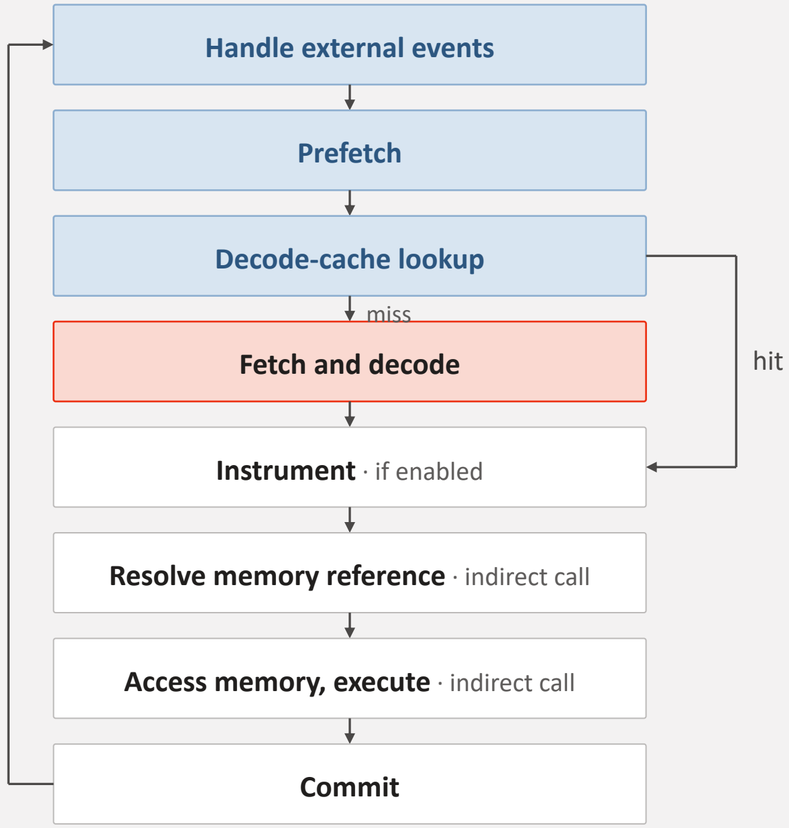
GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax —	ebx 2
ecx 5	edx 10
esi 0x500000	EIP 0x401000

this stage now / this instruction
stage done
stage skipped
next instruction

BOCHS 2.3.5 CPU LOOP



STEP 4 OF 9

Decode 8B 04 9E

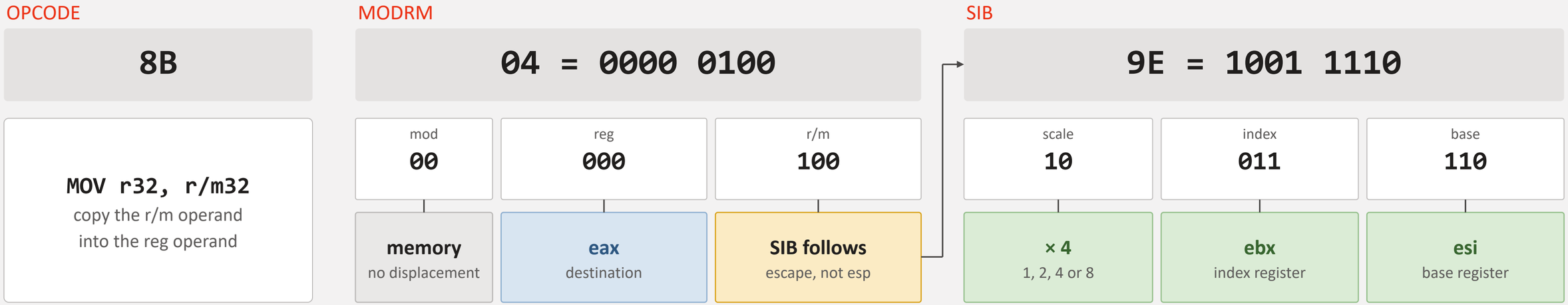
8B is MOV r32, r/m32. ModRM 04 says a SIB byte follows; SIB 9E means base esi, index ebx, scale 4. The decoder stores the handler, the address function and the operands, and caches the result.

STATE

MOV_GdEd · base + index×4 · 3 bytes

Inside 8B 04 9E: opcode, ModRM and SIB

Three bytes, three jobs. ModRM can name only one register, so r/m = 100 escapes to a SIB byte for base + index × scale.




```
eax ← [esi + ebx×4]    = [0x500000 + 2×4] = [0x500008] = 7
```

000 **eax**	001 **ecx**	010 **edx**	011 **ebx**	100 **esp**	101 **ebp**	110 **esi**	111 **edi**

- Because 100 is the escape, a plain [esp] needs a SIB byte too, and index = 100 means “no index”. Contrast add edx,eax = 01 C2: mod = 11, a register, no SIB.

One instruction through the CPU loop

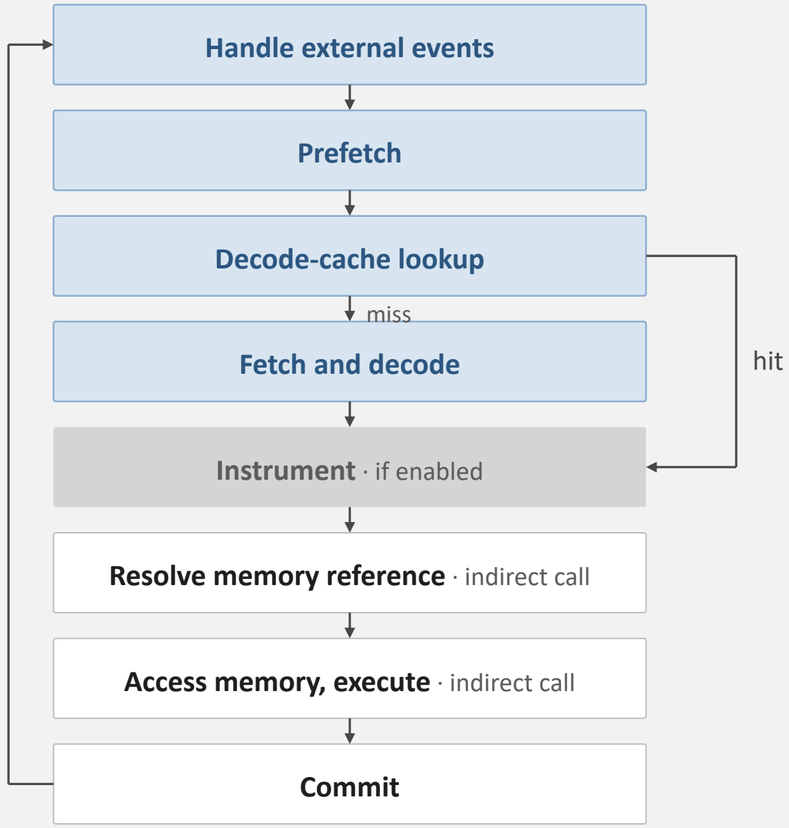
GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax —	ebx 2
ecx 5	edx 10
esi 0x500000	EIP 0x401000

this stage now / this instruction
stage done
stage skipped
next instruction

BOCHS 2.3.5 CPU LOOP



STEP 5 OF 9

Instrument: nothing to do

With no debugger or instrumentation tool attached, this stage does nothing. It lets tools observe every instruction before it runs.

STATE

no tool attached

One instruction through the CPU loop

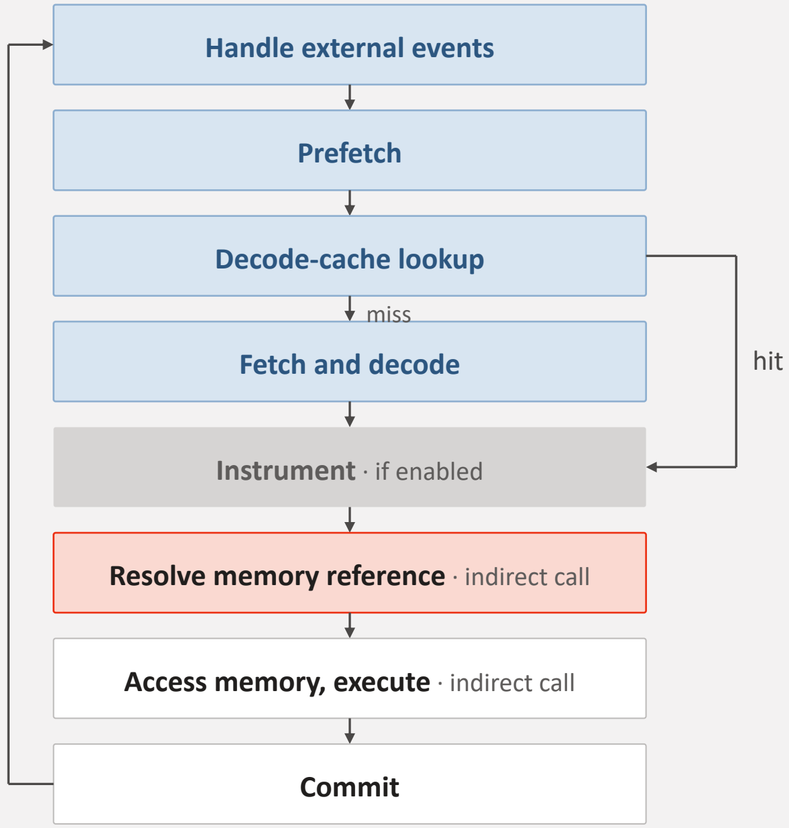
GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax –	ebx 2
ecx 5	edx 10
esi 0x500000	EIP 0x401000

- this stage now / this instruction
- stage done
- stage skipped
- next instruction

BOCHS 2.3.5 CPU LOOP



STEP 6 OF 9

Resolve the address

An indirect call reaches the function for this addressing form, which computes the effective address. The host CPU has to guess where this call goes before it knows.

STATE

EA = esi + ebx×4 = 0x500008

One instruction through the CPU loop

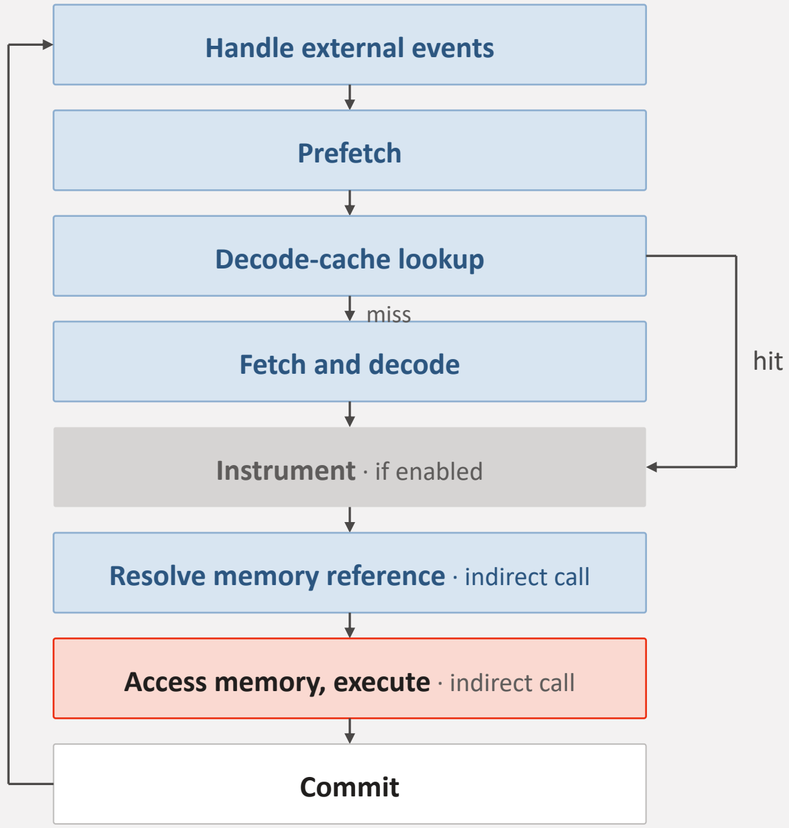
GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax **7**	ebx 2
ecx 5	edx 10
esi 0x500000	EIP 0x401000

this stage now / this instruction
stage done
stage skipped
next instruction

BOCHS 2.3.5 CPU LOOP



STEP 7 OF 9

Execute

A second indirect call reaches the MOV handler. It translates 0x500008 through Bochs's software TLB to a host address, reads the dword and writes it to eax.

STATE

eax = [0x500008] = 7

One instruction through the CPU loop

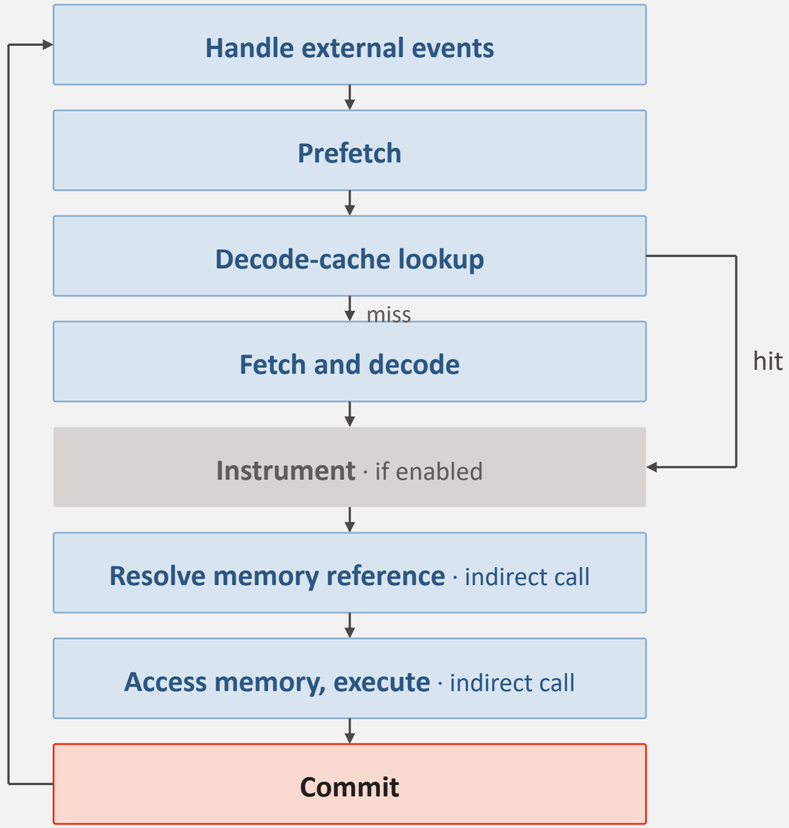
GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax 7	ebx 2
ecx 5	edx 10
esi 0x500000	EIP **0x401003**

this stage now / this instruction
stage done
stage skipped
next instruction

BOCHS 2.3.5 CPU LOOP



STEP 8 OF 9

Commit, and round again

EIP advances by the instruction's length. The old EIP is kept, so a fault in the next instruction can be restarted precisely. Then add edx,eax starts again at the top.

STATE

EIP = 0x401003 · eight stages for one mov

One instruction through the CPU loop

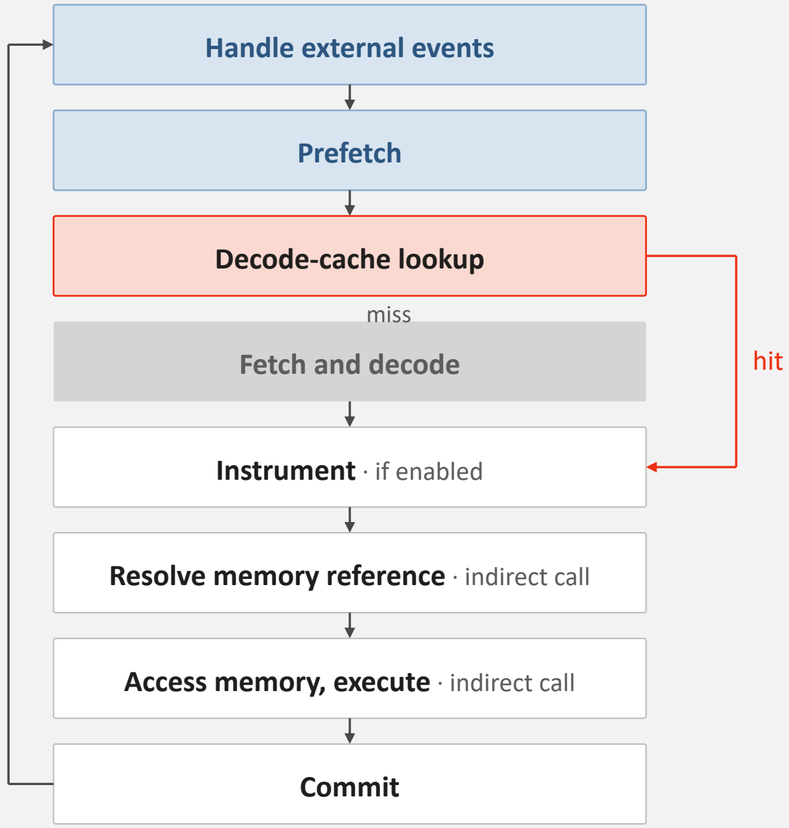
GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax 7	ebx **3**
ecx 5	edx **17**
esi 0x500000	EIP 0x401000

this stage now / this instruction
stage done
stage skipped
next instruction

BOCHS 2.3.5 CPU LOOP



STEP 9 OF 9

Second pass: a hit

Back at 0x401000 the lookup hits and the decoded copy is reused. Decoding is skipped, but events, prefetch, the lookup and both indirect calls still run for every instruction.

STATE

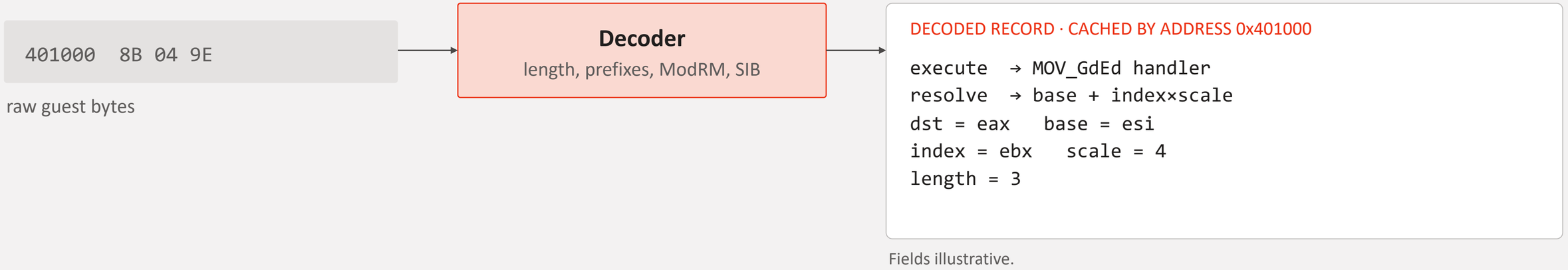
ebx = 3 · decode skipped

04

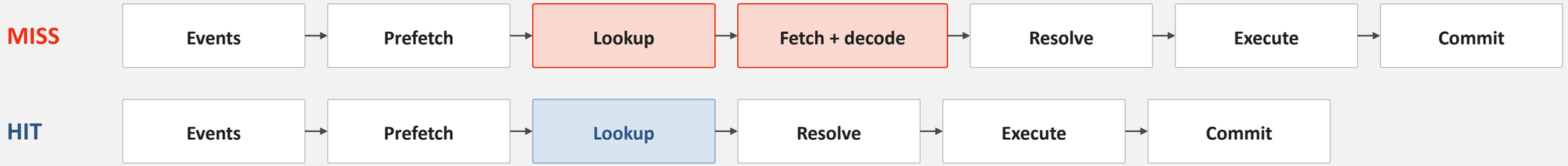
Bochs: the trace cache

Decode once: the decoded-instruction cache

Bochs 2.0 (2002) kept decoded instructions, keyed by guest address. It almost doubled speed.

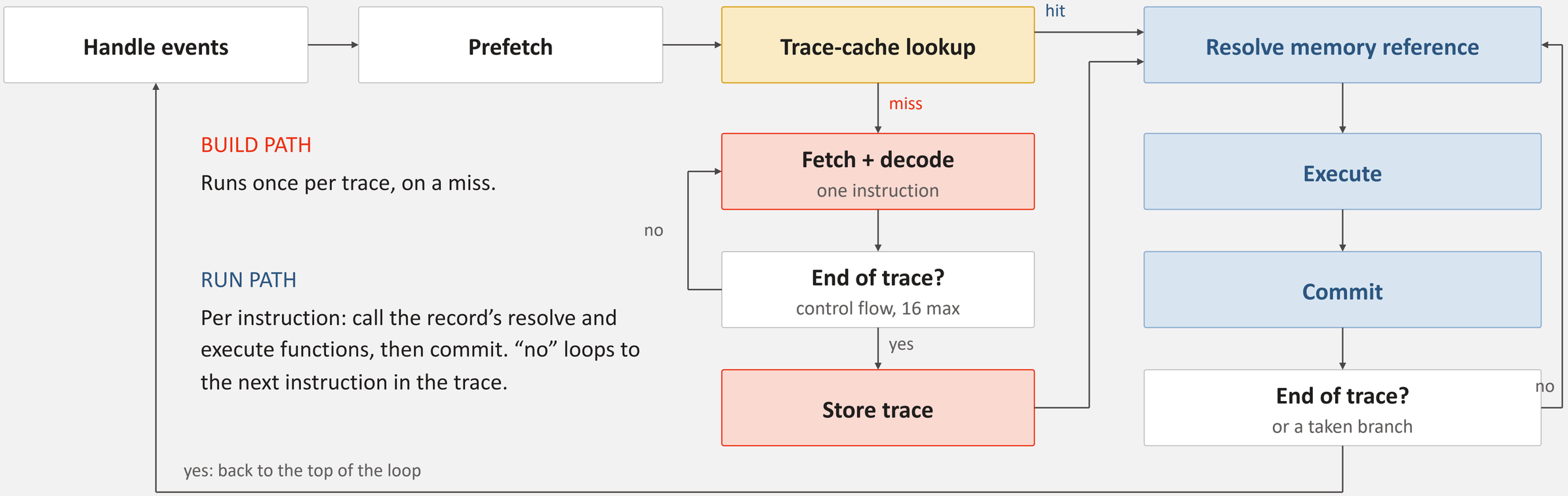


PER INSTRUCTION, MISS VS HIT



With a trace cache, the loop has a build path and a run path

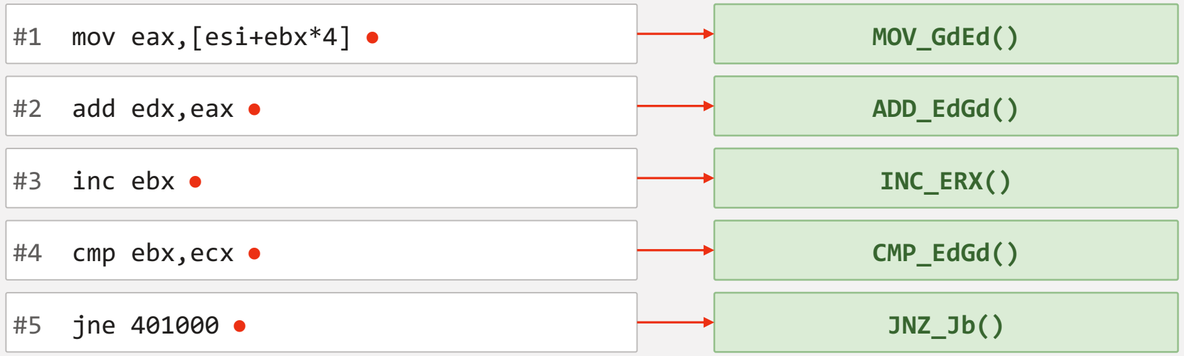
Bochs 2.3.6. Events, prefetch and lookup now happen once per trace, not once per instruction.



A trace holds function pointers, not host code

The host CPU never jumps into the trace cache. Bochs’s loop walks the decoded records and calls each one’s handler through a pointer.

TRACE-CACHE ENTRY 0x5040 · DATA



● = the record’s execute pointer. Handler names illustrative.

RUNNING A TRACE · SIMPLIFIED

```
for (each record i in the trace) {
  i->ResolveModrm(i);  // indirect call
  i->execute(i);       // indirect call
  commit EIP;
  if (branch taken) break;
}
```

CACHE	WHAT IT HOLDS	WHO EXECUTES IT
Pentium 4 trace cache	decoded micro-ops	the CPU’s execution units, in hardware
QEMU translation cache	generated host machine code	the host CPU jumps straight into it
Bochs trace cache	decoded records with function pointers	Bochs’s loop, calling each handler

Building and running a trace

GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
40100A  89 D0      mov eax,edx
40100C  C3         ret
```

eax –	ebx 2
ecx 5	edx 10

being decoded now
stored in the trace
handler called from the trace
skipped: early out

TRACE CACHE

lookup(0x401000) → **miss**

hash: (X + (X<<2) + (X>>6)) mod 65536 = 0x5040

TRACE-CACHE ENTRY 0x5040 · RECORD → HANDLER

#1	empty
#2	empty
#3	empty
#4	empty
#5	empty
#6	empty
#7	empty
#8–16	unused

STEP 1 OF 7

Look up the trace

Bochs hashes the guest address to pick a trace-cache entry. Entry 0x5040 is empty, so it switches to build mode.

STATE

entry 0x5040 empty → build

Building and running a trace

GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
40100A  89 D0      mov eax,edx
40100C  C3         ret
```

```
eax –                ebx 2
ecx 5                edx 10
```

being decoded now
stored in the trace
handler called from the trace
skipped: early out

TRACE CACHE

lookup(0x401000) → **miss**

hash: (X + (X<<2) + (X>>6)) mod 65536 = 0x5040

TRACE-CACHE ENTRY 0x5040 · RECORD → HANDLER

```
#1   mov eax,[esi+ebx*4]   → MOV_GdEd
#2   add edx,eax           → ADD_EdGd
#3   inc ebx               → INC_ERX
#4   cmp ebx,ecx           → CMP_EdGd
#5   empty
#6   empty
#7   empty
#8–16  unused
```

STEP 2 OF 7

Decode until control flow

The builder fetches and decodes one instruction after another into the entry. None of these four can change control flow, so it keeps going.

STATE

4 instructions decoded

Building and running a trace

GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
40100A  89 D0      mov eax,edx
40100C  C3         ret
```

```
eax –                ebx 2
ecx 5                edx 10
```

being decoded now
stored in the trace
handler called from the trace
skipped: early out

TRACE CACHE

lookup(0x401000) → **miss**

hash: (X + (X<<2) + (X>>6)) mod 65536 = 0x5040

TRACE-CACHE ENTRY 0x5040 · RECORD → HANDLER

```
#1   mov eax,[esi+ebx*4]   → MOV_GdEd
#2   add edx,eax           → ADD_EdGd
#3   inc ebx               → INC_ERX
#4   cmp ebx,ecx           → CMP_EdGd
#5   jne 401000            → JNZ_Jb
#6   empty
#7   empty
#8–16  unused
```

STEP 3 OF 7

jne: keep tracing past it

jne is a conditional branch. Bochs may trace through a branch that might not be taken, so it carries on into the fall-through code.

STATE

5 decoded · branch may fall through

Building and running a trace

GUEST CODE

```
401000  8B 04 9E  mov eax,[esi+ebx*4]
401003  01 C2     add edx,eax
401005  43        inc ebx
401006  39 CB     cmp ebx,ecx
401008  75 F6     jne 401000
40100A  89 D0     mov eax,edx
40100C  C3        ret
```

```
eax –                ebx 2
ecx 5                edx 10
```

being decoded now
stored in the trace
handler called from the trace
skipped: early out

TRACE CACHE

lookup(0x401000) → **miss**

hash: (X + (X<<2) + (X>>6)) mod 65536 = 0x5040

TRACE-CACHE ENTRY 0x5040 · RECORD → HANDLER

```
#1   mov eax,[esi+ebx*4]   → MOV_GdEd
#2   add edx,eax           → ADD_EdGd
#3   inc ebx               → INC_ERX
#4   cmp ebx,ecx           → CMP_EdGd
#5   jne 401000            → JNZ_Jb
#6   mov eax,edx           → MOV_EdGd
#7   ret                   → RET_near
#8–16  unused
```

STEP 4 OF 7

ret ends the trace

mov eax,edx is ordinary, but ret always changes control flow, so the trace stops and is stored. Seven instructions, decoded once.

STATE

trace stored: 7 instructions

Building and running a trace

GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
40100A  89 D0      mov eax,edx
40100C  C3         ret
```

eax **7**	ebx **3**
ecx 5	edx **17**

being decoded now
stored in the trace
handler called from the trace
skipped: early out

TRACE CACHE

lookup(0x401000) → **run**

hash: (X + (X<<2) + (X>>6)) mod 65536 = 0x5040

TRACE-CACHE ENTRY 0x5040 · RECORD → HANDLER

```
#1   mov eax,[esi+ebx*4]   → MOV_GdEd
#2   add edx,eax           → ADD_EdGd
#3   inc ebx               → INC_ERX
#4   cmp ebx,ecx           → CMP_EdGd
#5   jne 401000            → JNZ_Jb
#6   mov eax,edx           → MOV_EdGd
#7   ret                   → RET_near
#8–16  unused
```

STEP 5 OF 7

Run it; jne leaves early

Bochs calls each record's handler in turn, with no prefetch or decode in between. ebx = 3 is not ecx = 5, so jne is taken and the early-out stops the trace at slot 5.

STATE

slots 6–7 skipped · next EIP 0x401000

Building and running a trace

GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
40100A  89 D0      mov eax,edx
40100C  C3         ret
```

eax **5**	ebx **4**
ecx 5	edx **22**

being decoded now
stored in the trace
handler called from the trace
skipped: early out

TRACE CACHE

lookup(0x401000) → **hit**

hash: (X + (X<<2) + (X>>6)) mod 65536 = 0x5040

TRACE-CACHE ENTRY 0x5040 · RECORD → HANDLER

```
#1   mov eax,[esi+ebx*4]   → MOV_GdEd
#2   add edx,eax           → ADD_EdGd
#3   inc ebx               → INC_ERX
#4   cmp ebx,ecx           → CMP_EdGd
#5   jne 401000            → JNZ_Jb
#6   mov eax,edx           → MOV_EdGd
#7   ret                   → RET_near
#8–16  unused
```

STEP 6 OF 7

Second pass: a hit

The lookup hits, so Bochs calls the five cached handlers: one lookup for five instructions. jne is taken again (4 ≠ 5) and leaves early.

STATE

edx = 17 + 5 = 22

Building and running a trace

GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
40100A  89 D0      mov eax,edx
40100C  C3         ret
```

eax **31**	ebx **5**
ecx 5	edx **31**

being decoded now
stored in the trace
handler called from the trace
skipped: early out

TRACE CACHE

lookup(0x401000) → **hit**

hash: (X + (X<<2) + (X>>6)) mod 65536 = 0x5040

TRACE-CACHE ENTRY 0x5040 · RECORD → HANDLER

```
#1   mov eax,[esi+ebx*4]   → MOV_GdEd
#2   add edx,eax           → ADD_EdGd
#3   inc ebx               → INC_ERX
#4   cmp ebx,ecx           → CMP_EdGd
#5   jne 401000            → JNZ_Jb
#6   mov eax,edx           → MOV_EdGd
#7   ret                   → RET_near
#8–16  unused
```

STEP 7 OF 7

Last pass: fall through

Now ebx = 5 equals ecx, so jne is not taken and the speculative slots pay off: mov eax,edx and ret run from the same trace.

STATE

17 executed · 7 decoded · 3 lookups

Traces are short, and there’s more to gain

Windows XP boot, measured inside Bochs.

~2×

decoded-instruction cache

Bochs 2.0, 2002

> 20%

more from the trace cache

Bochs 2.3.6

< 7

instructions in most traces

almost none longer than 16

WHAT THE AUTHORS PLAN NEXT FOR TRACES

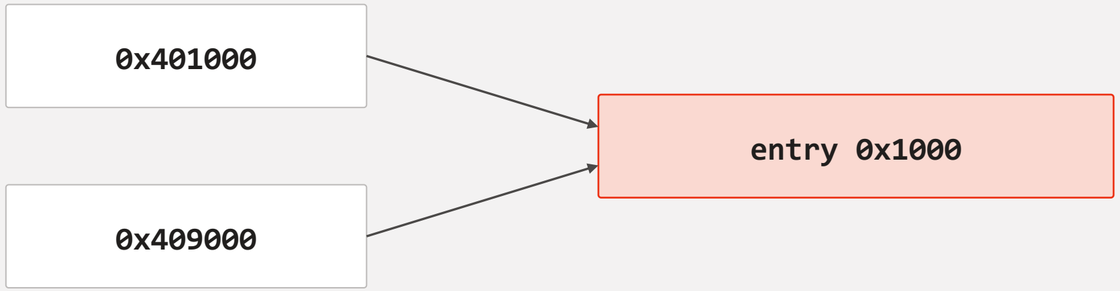
1. Split complicated x86 instructions into simpler micro-ops inside the trace.
2. Optimize hot traces like a compiler: drop no-ops, register moves and redundant flag updates.

Why the trace cache hashes its index

Two hot loops exactly 32 KB apart land on the same entry of a direct-mapped cache, and keep evicting each other.

PLAIN DIRECT MAPPING

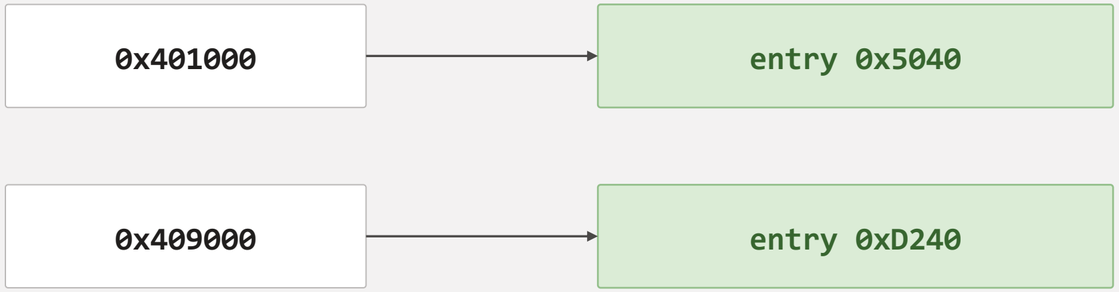
index = X mod 32768



Collision: each loop evicts the other’s trace on every switch.

BOCHS 2.3.6

index = (X + (X<<2) + (X>>6)) mod 65536



The right shift mixes higher bits into the index, so the two spread apart.

- Software set associativity was rejected: the extra lookup branches would mispredict.

05

Bochs: branch prediction

Three places the host can mispredict

For every guest instruction, the host CPU guesses where two indirect calls in the CPU loop go, and which way one test inside the handler goes.

BOCHS 2.3.5 CPU LOOP · SIMPLIFIED

```
for (;;) {
  i = fetchDecode(EIP);
  if (i has a memory operand)
    i->ResolveModrm(i);          1
  i->execute(i);                 2
  commit(i);
}
```

Call 2 runs the handler on the right; test 3 is inside it.

ONE HANDLER, CONDENSED

```
void SUB_EdGd(bxInstruction_c *i)
{
  op2 = READ_REG(i->nnn());
  if (i->modC0()) {              3
    ...register form...
  } else {
    ...memory form...
  }
  SET_LAZY_FLAGS_SUB32(...);
}
```

1 The address call

indirect call · one site, ~30 targets

Fixed in 2.3.7 with a call site per handler and 2 targets. About 40%.

2 The dispatch call

indirect call · one site, a target per opcode

Every guest instruction passes through it. Not fixed in the paper.

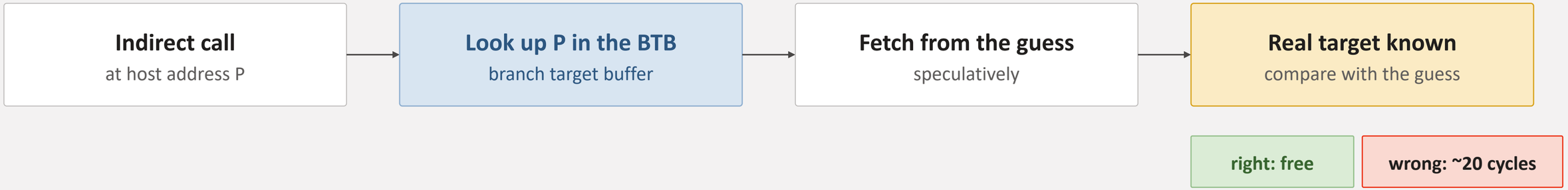
3 The register-or-memory test

conditional branch · direction set by the guest

Fixed by choosing the handler at decode time. About 15%.

How the host predicts an indirect call

Simplified model: each call site remembers where it went last time, and the CPU guesses that again.



WHY THE NUMBER OF TARGETS PER CALL SITE MATTERS

A, B and C are different functions one call site can jump to, e.g. the address functions of the loop's three memory instructions. Eight calls from the same site:

Always the same target

goes to	A	A	A	A	A	A	A	A
BTB guesses	—	A	A	A	A	A	A	A
outcome	no entry	right	right	right	right	right	right	right

The guess is the last target, which is the next one too. Right every time after the first call.

Three targets in turn

goes to	A	B	C	A	B	C	A	B
BTB guesses	—	A	B	C	A	B	C	A
outcome	no entry	wrong	wrong	wrong	wrong	wrong	wrong	wrong

The guess is the last target, which is never the next one. Wrong every time: ~20 cycles each.

Case 1 · The loop behind the address call

Bochs 2.3.5's CPU loop. The source line i->ResolveModrm(i) compiles to one indirect call instruction, and every guest memory instruction goes through it.

BOCHS 2.3.5 CPU LOOP · C++, SIMPLIFIED

```
void BX_CPU_C::cpu_loop()
{
  for (;;) {
    handleAsyncEvents();
    bxInstruction_c *i = fetchDecode(EIP);

    if (i has a memory operand)
      i->ResolveModrm(i);       // call site 1
    i->execute(i);              // call site 2
    commit(i);
  }
}
```

WHAT THE COMPILER EMITS · ILLUSTRATIVE

```
loop_top:
          call  handleAsyncEvents
          ...                   ; %ebx = i

          ...                   ; no memory? skip
0804A1F0  call  *0x10(%ebx)     ; ResolveModrm
0804A1F8  call  *0x0C(%ebx)     ; execute
          ...                   ; commit
          jmp   loop_top
```

ONE LINE, ONE MACHINE INSTRUCTION

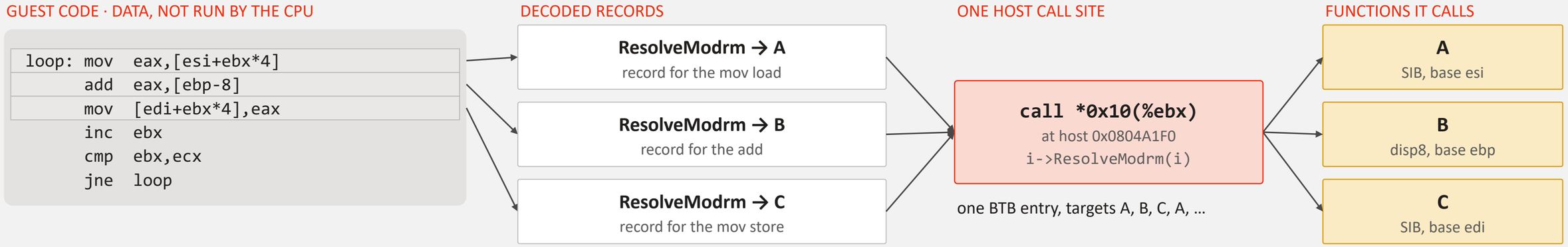
Every guest memory instruction passes through 0x0804A1F0, each time with a different pointer in its record.

IN BOCHS 2.3.7

The call moves into each handler, such as MOV_GdEd, so each handler's body has its own call instruction.

Case 1 · Calls inside Bochs, not the guest

The CPU runs Bochs’s compiled code, not the guest’s. In 2.3.5 every guest memory instruction goes through the same host call instruction, so its prediction keeps changing.



BOCHS 2.3.7 · THE CALL MOVES INTO EACH HANDLER: THREE CALL SITES, ONE TARGET EACH

MOV_GdEd
own call at 0x0805B210
always → base + index×scale + disp

ADD_GdEd
own call at 0x0805C4A0
always → base + disp

MOV_EdGd
own call at 0x0805B6F0
always → base + index×scale + disp

- Host addresses are made up. The loop’s other call site, i->execute(i), still rotates among six handlers.

Case 1 fix · The call moves from the loop into each handler

Simplified, in Bochs's style. The same C++ line in every handler compiles to a separate call instruction.

BOCHS 2.3.5 · THE CALL IS IN THE LOOP

```
// CPU loop
for (;;) {
  i = fetchDecode(EIP);
  if (i has a memory operand)
    i->ResolveModrm(i);      // one call site
  i->execute(i);
}

void MOV_GdEd(bxInstruction_c *i) {
  op = read_dword(RMAddr(i));  // already computed
  ...
}
```

BOCHS 2.3.7 · THE CALL IS IN EACH HANDLER

```
// CPU loop
for (;;) {
  i = fetchDecode(EIP);
  i->execute(i);             // no address call here
}
void MOV_GdEdM(bxInstruction_c *i) {
  eaddr = i->ResolveModrm(i);  // call site in MOV
  BX_WRITE_32BIT_REG(i->nnn(), read_dword(eaddr));
}
void ADD_GdEdM(bxInstruction_c *i) {
  eaddr = i->ResolveModrm(i);  // call site in ADD
  ...
}
```

THE TWO REMAINING ADDRESS FUNCTIONS · MISSING BASE OR INDEX = NULL REGISTER, ALWAYS 0

```
EA_BaseDisp(i)       = R[i->base()] + i->displ32();
EA_BaseIndexDisp(i)  = R[i->base()] + (R[i->index()] << i->scale()) + i->displ32();
```

Case 1 · The address call, before and after

GUEST CODE

```
loop: mov  eax,[esi+ebx*4]  A
      add  eax,[ebp-8]      B
      mov  [edi+ebx*4],eax  C
      inc  ebx
      cmp  ebx,ecx
      jne  loop
```

ADDRESS FUNCTION EACH ONE NEEDS

A SIB, base esi

B disp8, base ebp

C SIB, base edi

All three go through the same host call instruction.

HOST BRANCH PREDICTION

THE HOST CALL INSTRUCTION RUNNING NOW

`call *0x10(%ebx)`

at 0x0804A1F0 · in the CPU loop · i->ResolveModrm(i)

BTB GUESSES

C · SIB, base edi

where this call went last time

RECORD POINTS TO

A · SIB, base esi

what this instruction needs

guessed wrong: ~20 cycles lost

EVERY CALL THROUGH THIS INSTRUCTION SO FAR

	end of pass 1	pass 2
goes to	**C**	**A**
BTB guessed	**B**	**C**
outcome	**wrong**	**wrong**

BOCHS 2.3.5 · STEP 1 OF 6

One call site, three targets

The loop already ran once, so the BTB has history. The last call through this instruction was the store’s, at the end of pass 1: C. So it guesses C, but this mov needs A.

STATE

pass 2 · wrong guesses: 1

Case 1 · The address call, before and after

GUEST CODE

```
loop: mov  eax,[esi+ebx*4]  A
      add  eax,[ebp-8]      B
      mov  [edi+ebx*4],eax  C
      inc  ebx
      cmp  ebx,ecx
      jne  loop
```

ADDRESS FUNCTION EACH ONE NEEDS

A SIB, base esi

B disp8, base ebp

C SIB, base edi

All three go through the same host call instruction.

HOST BRANCH PREDICTION

THE HOST CALL INSTRUCTION RUNNING NOW

`call *0x10(%ebx)`

at 0x0804A1F0 · in the CPU loop · i->ResolveModrm(i)

BTB GUESSES

A · SIB, base esi

where this call went last time

RECORD POINTS TO

B · disp8, base ebp

what this instruction needs

guessed wrong: ~20 cycles lost

EVERY CALL THROUGH THIS INSTRUCTION SO FAR

	end of pass 1	pass 2	pass 2
goes to	**C**	**A**	**B**
BTB guessed	**B**	**C**	**A**
outcome	**wrong**	**wrong**	**wrong**

BOCHS 2.3.5 · STEP 2 OF 6

Same instruction, new target

Next memory instruction, same host call instruction. The BTB now remembers A, but add needs B.

STATE

pass 2 · wrong guesses: 2

Case 1 · The address call, before and after

GUEST CODE

```
loop: mov  eax,[esi+ebx*4]  A
      add  eax,[ebp-8]      B
      mov  [edi+ebx*4],eax  C
      inc  ebx
      cmp  ebx,ecx
      jne  loop
```

ADDRESS FUNCTION EACH ONE NEEDS

A SIB, base esi

B disp8, base ebp

C SIB, base edi

All three go through the same host call instruction.

HOST BRANCH PREDICTION

THE HOST CALL INSTRUCTION RUNNING NOW

`call *0x10(%ebx)`

at 0x0804A1F0 · in the CPU loop · i->ResolveModrm(i)

BTB GUESSES

B · disp8, base ebp

where this call went last time

RECORD POINTS TO

C · SIB, base edi

what this instruction needs

guessed wrong: ~20 cycles lost

EVERY CALL THROUGH THIS INSTRUCTION SO FAR

	end of pass 1	pass 2	pass 2	pass 2
goes to	**C**	**A**	**B**	**C**
BTB guessed	**B**	**C**	**A**	**B**
outcome	**wrong**	**wrong**	**wrong**	**wrong**

BOCHS 2.3.5 · STEP 3 OF 6

Wrong every time

The store needs C, but the BTB remembers B. One call instruction cycling A, B, C is wrong on every memory instruction of every pass.

STATE

3 wrong per pass ≈ 60 cycles

Case 1 · The address call, before and after

GUEST CODE

```
loop: mov  eax,[esi+ebx*4]  MOV_GdEd
      add  eax,[ebp-8]      ADD_GdEd
      mov  [edi+ebx*4],eax  MOV_EdGd
      inc  ebx
      cmp  ebx,ecx
      jne  loop
```

WHERE EACH ONE'S CALL LIVES

MOV_GdEd `0x0805B210`

ADD_GdEd `0x0805C4A0`

MOV_EdGd `0x0805B6F0`

Three handlers, three separate call instructions.

HOST BRANCH PREDICTION

THE HOST CALL INSTRUCTION RUNNING NOW

`call *0x10(%ebx)`

at 0x0805B210 · inside MOV_GdEd · its own copy

BTB GUESSES
base + index×scale + disp
where this call went last time

RECORD POINTS TO
base + index×scale + disp
what this instruction needs

guessed right: free

EVERY CALL THROUGH THIS INSTRUCTION SO FAR

	pass 1	pass 2
goes to	B+I×S+D	B+I×S+D
BTB guessed	—	B+I×S+D
outcome	no entry	right

BOCHS 2.3.7 · STEP 4 OF 6

2.3.7: the mov's own call

The call is now inside MOV_GdEd: a different machine instruction with its own BTB entry. Last pass it went to the same function, so the guess is right.

STATE
pass 2 · wrong guesses: 0

Case 1 · The address call, before and after

GUEST CODE

```
loop: mov  eax,[esi+ebx*4]  MOV_GdEd
      add  eax,[ebp-8]      ADD_GdEd
      mov  [edi+ebx*4],eax  MOV_EdGd
      inc  ebx
      cmp  ebx,ecx
      jne  loop
```

WHERE EACH ONE'S CALL LIVES

MOV_GdEd `0x0805B210`

ADD_GdEd `0x0805C4A0`

MOV_EdGd `0x0805B6F0`

Three handlers, three separate call instructions.

HOST BRANCH PREDICTION

THE HOST CALL INSTRUCTION RUNNING NOW

`call *0x10(%ebx)`

at 0x0805C4A0 · inside ADD_GdEd · its own copy

BTB GUESSES
base + disp
where this call went last time

RECORD POINTS TO
base + disp
what this instruction needs

guessed right: free

EVERY CALL THROUGH THIS INSTRUCTION SO FAR

	pass 1	pass 2
goes to	B+D	B+D
BTB guessed	—	B+D
outcome	no entry	right

BOCHS 2.3.7 · STEP 5 OF 6

The add's own call

ADD_GdEd has its own call instruction too. Its BTB entry only ever sees base + disp.

STATE
pass 2 · wrong guesses: 0

Case 1 · The address call, before and after

GUEST CODE

```
loop: mov  eax,[esi+ebx*4]  MOV_GdEd
      add  eax,[ebp-8]      ADD_GdEd
      mov  [edi+ebx*4],eax  MOV_EdGd
      inc  ebx
      cmp  ebx,ecx
      jne  loop
```

WHERE EACH ONE'S CALL LIVES

MOV_GdEd `0x0805B210`

ADD_GdEd `0x0805C4A0`

MOV_EdGd `0x0805B6F0`

Three handlers, three separate call instructions.

HOST BRANCH PREDICTION

THE HOST CALL INSTRUCTION RUNNING NOW

`call *0x10(%ebx)`

at 0x0805B6F0 · inside MOV_EdGd · its own copy

BTB GUESSES
base + index×scale + disp
where this call went last time

RECORD POINTS TO
base + index×scale + disp
what this instruction needs

guessed right: free

EVERY CALL THROUGH THIS INSTRUCTION SO FAR

	pass 1	pass 2
goes to	B+I×S+D	B+I×S+D
BTB guessed	—	B+I×S+D
outcome	no entry	right

BOCHS 2.3.7 · STEP 6 OF 6

Every site predicts itself

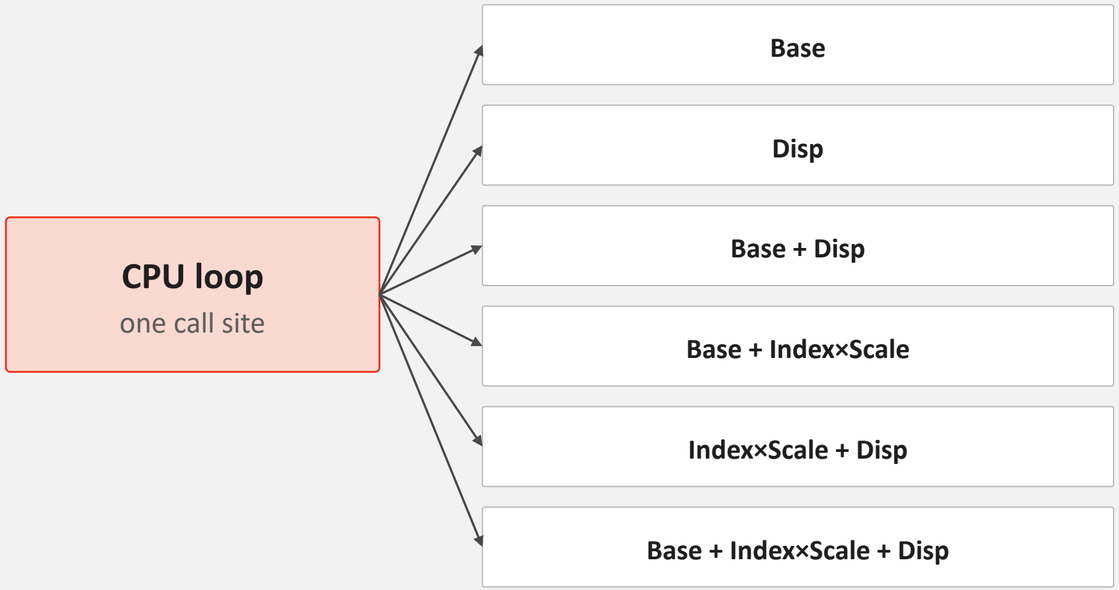
The store's handler, MOV_EdGd, is right as well. Three call sites, one target each: no wrong guesses in this loop.

STATE
0 wrong per pass · ~40% faster

Case 1 fix · Fewer targets, more call sites

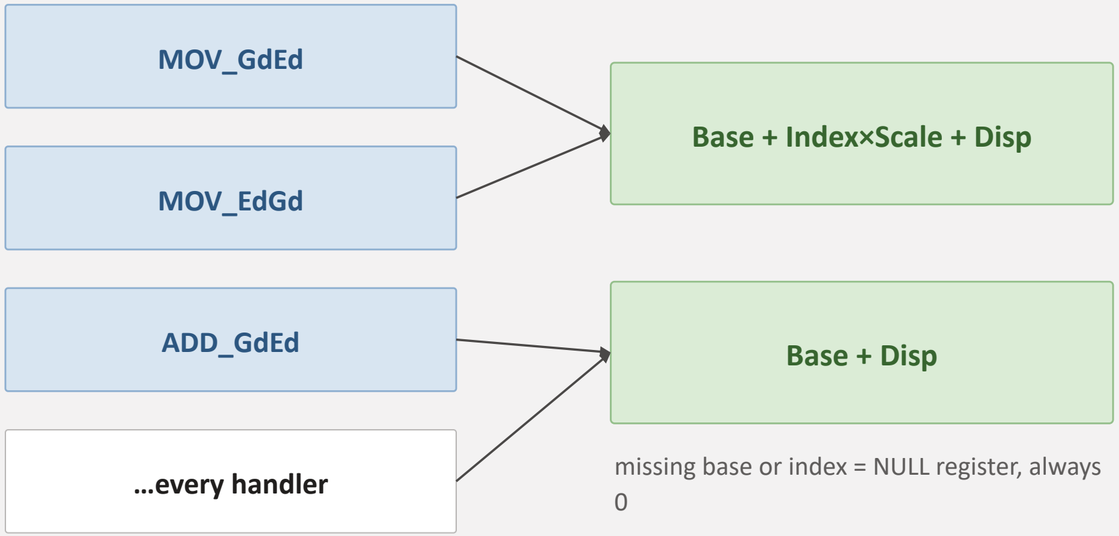
Both standard ways to help an indirect branch predictor, applied together: about 40% faster.

BOCHS 2.3.5 · ONE SITE, ~30 TARGETS



each split by base register

BOCHS 2.3.7 · A SITE PER HANDLER, 2 TARGETS



missing base or index = NULL register, always 0

- Six forms split by base register ≈ 30 functions in 2.3.5. Both changes together: about 40% faster.

Case 2 · The dispatch call: one site for every opcode

i->execute(i) is one host call instruction that every guest instruction goes through. In our loop it reaches six different handlers in turn.

GUEST CODE → EACH RECORD'S EXECUTE POINTER

```
mov  eax,[esi+ebx*4]  → MOV_GdEd
add  eax,[ebp-8]      → ADD_GdEd
mov  [edi+ebx*4],eax  → MOV_EdGd
inc  ebx              → INC_ERX
cmp  ebx,ecx          → CMP_EdGd
jne  loop             → JNZ_Jb
```

CALL SITE 2

call *0x0C(%ebx)
at host 0x0804A1F8
i->execute(i)

HANDLERS IT CALLS

- MOV_GdEd()
- ADD_GdEd()
- MOV_EdGd()
- INC_ERX()
- CMP_EdGd()
- JNZ_Jb()

ONE PASS OF THE LOOP, STEADY STATE · SIMPLE LAST-TARGET MODEL

Call site 2

goes to	MOV_GdEd	ADD_GdEd	MOV_EdGd	INC_ERX	CMP_EdGd	JNZ_Jb
BTB guesses	JNZ_Jb	MOV_GdEd	ADD_GdEd	MOV_EdGd	INC_ERX	CMP_EdGd
outcome	wrong	wrong	wrong	wrong	wrong	wrong

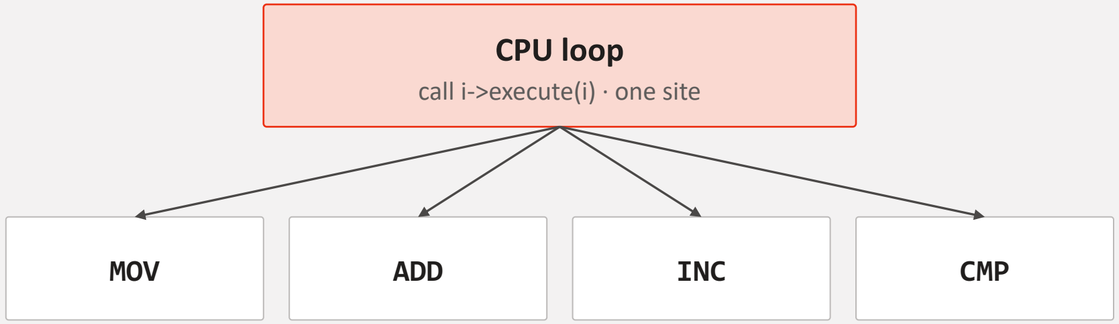
6 wrong per pass, ≈120 cycles, on top of case 1's 3.

- Handler names illustrative. Not fixed in the paper: macro-op fusion and more handler splitting are listed as future work.

Case 2 · The known remedy: replicate the dispatch

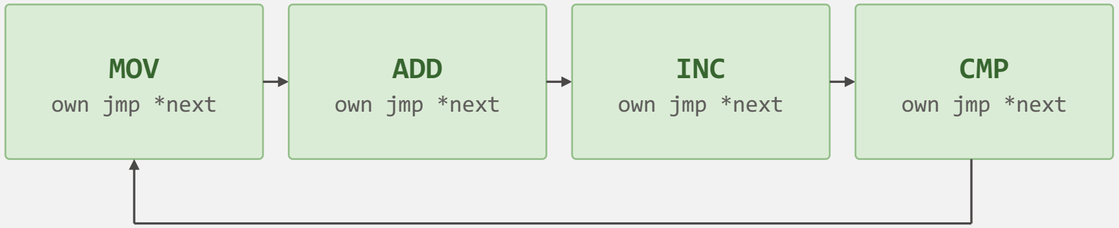
Background, not from the paper. Interpreters fix case 2 the way Bochs fixed case 1: each handler gets its own copy of the indirect branch.

SHARED DISPATCH · BOCHS



Every handler returns to the loop, so one prediction entry has to guess the next opcode for the whole program.

REPLICATED DISPATCH · “THREADED CODE”



Each handler jumps straight to the next through its own indirect jump, so each copy learns what usually follows it.

WHY NOT IN BOCHS (OUR READING)

Standard C++ has no computed goto or guaranteed tail call; this needs compiler extensions.

HARDWARE CAUGHT UP

From about 2013, CPUs predict indirect branches using history, which helps shared dispatch a lot.

Case 3 · The register-or-memory test

A conditional branch, not an indirect call: the CPU guesses which way it goes. One SUB_EdGd serves both forms, so the direction depends on the guest code.

THE HANDLER · CONDENSED FROM LISTING 3.1

```
void SUB_EdGd(bxInstruction_c *i)
{
  op2 = BX_READ_32BIT_REG(i->nnn());
  if (i->modC0()) {            // mod = 11?                    3
    op1 = BX_READ_32BIT_REG(i->rm());  ...
  } else {
    read_RMW_virtual_dword(...);       ...
  }
  SET_LAZY_FLAGS_SUB32(op1, op2, diff);
}
```

HOW THE CPU GUESSES · SIMPLIFIED

A conditional branch keeps a little history at its address. Simplest model: guess the same direction as last time.

If the guest used only one form of sub, that guess would always be right.

GUEST CODE THAT MIXES THE TWO FORMS

Six subs

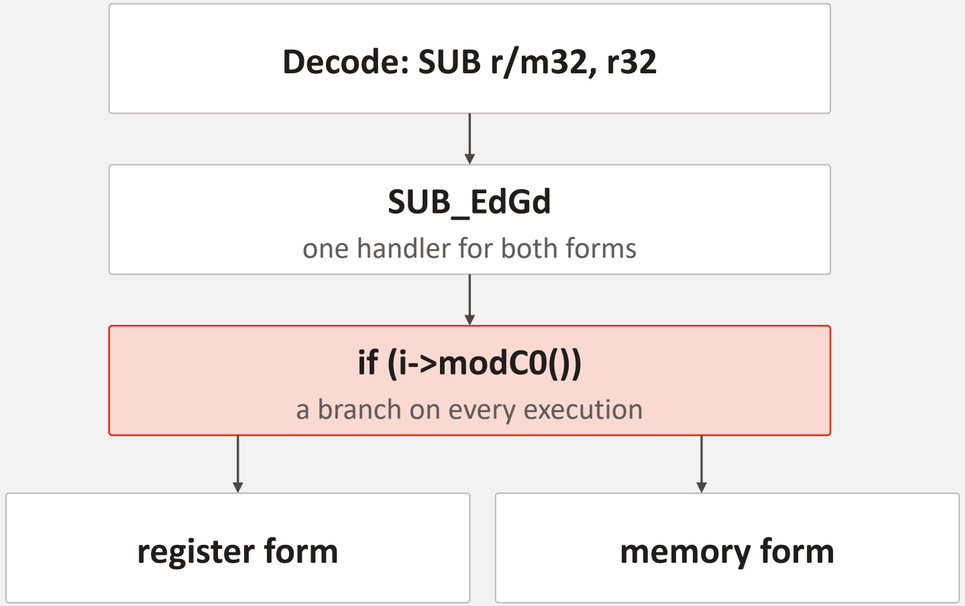
guest code	`sub eax,ebx`	`sub [esi],eax`	`sub ecx,edx`	`sub [edi],ecx`	`sub eax,ecx`	`sub [esi],edx`
guest form	reg	mem	reg	mem	reg	mem
CPU guesses	mem	reg	mem	reg	mem	reg
outcome	wrong	wrong	wrong	wrong	wrong	wrong

Every guess wrong, ~20 cycles each, in a handler a few lines long.

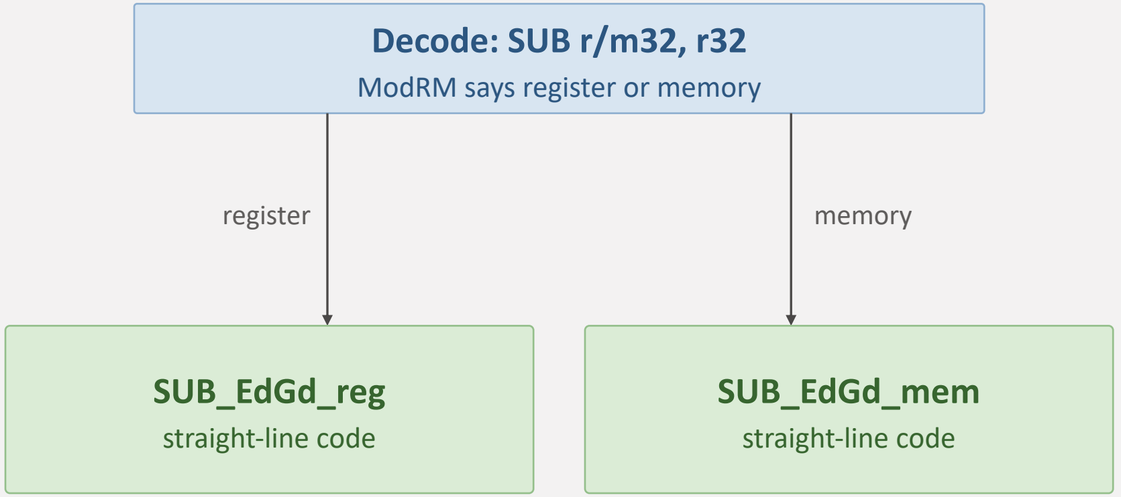
Case 3 fix · Pick register or memory at decode time

The decoder already knows the form, so each common opcode gets two straight-line handlers.

BEFORE · ONE HANDLER, A BRANCH INSIDE



AFTER · THE DECODER PICKS THE HANDLER



Handler names illustrative.

- Decoding happens once per cached instruction, so the test now costs nothing per execution: about 15% faster.

06

Bochs: lazy flags

Eager vs lazy: when are the flags computed?

The same four instructions from the loop. Only jne reads a flag, and only ZF.

	`add edx,eax`	`inc ebx`	`cmp ebx,ecx`	`jne 401000`
PUSHF approach inline assembly	**host op + PUSHF** all 6 flags	**host op + PUSHF** all 6 flags	**host op + PUSHF** all 6 flags	**read saved ZF**
Lazy flags Bochs 2.3.7	**save result, ops** no flags yet	**save result, ops** no flags yet	**save result, ops** no flags yet	**ZF = result == 0** one flag, on demand

THE COUNT

PUSHF approach: 3 host flag captures, 18 flags. Lazy: 1 comparison, 1 flag.

Lazy flags, instruction by instruction

GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax 7	ebx 2
ecx 5	edx **17**

this instruction
already run
record field updated
flag computed

WHAT BOCHS KEEPS

LAZY-FLAGS RECORD

operation	**ADD32**
op1	**10**
op2	**7**
result	**17**

GUEST EFLAGS · COMPUTED ONLY WHEN READ

CF	PF	AF	ZF	SF	OF
?	?	?	?	?	?

WORK DONE ON FLAGS SO FAR

PUSHF approach	**1 PUSHF**
Lazy flags	**nothing**

STEP 1 OF 5

add: save, don’t compute

edx = 10 + 7 = 17. The handler stores the result and both operands. No flag is computed; any of the six could still be needed later.

STATE

6 flags pending

Lazy flags, instruction by instruction

GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax 7	ebx 3
ecx 5	edx 17

this instruction
already run
record field updated
flag computed

WHAT BOCHS KEEPS

LAZY-FLAGS RECORD

operation	**INC32**
op1	**2**
op2	**—**
result	**3**

GUEST EFLAGS · COMPUTED ONLY WHEN READ

CF	PF	AF	ZF	SF	OF
?	?	?	?	?	?

WORK DONE ON FLAGS SO FAR

PUSHF approach	**2 PUSHF**
Lazy flags	**nothing**

STEP 2 OF 5

inc: overwrite the record

ebx = 3. The new record replaces the old one: add's flags were never read, so they never needed computing. (inc leaves CF unchanged, a detail Bochs handles separately.)

STATE

add's flags: never computed

Lazy flags, instruction by instruction

GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax 7	ebx 3
ecx 5	edx 17

this instruction
already run
record field updated
flag computed

WHAT BOCHS KEEPS

LAZY-FLAGS RECORD

operation	**SUB32 (cmp)**
op1	**3**
op2	**5**
result	**0xFFFFFFFE**

GUEST EFLAGS · COMPUTED ONLY WHEN READ

CF	PF	AF	ZF	SF	OF
?	?	?	?	?	?

WORK DONE ON FLAGS SO FAR

PUSHF approach	**3 PUSHF**
Lazy flags	**nothing**

STEP 3 OF 5

cmp: just a subtraction

cmp is a subtraction that keeps only the flags. Bochs saves 3 − 5 = 0xFFFFFFFE and the operands, and still computes nothing.

STATE

result saved; flags pending

Lazy flags, instruction by instruction

GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax 7	ebx 3
ecx 5	edx 17

this instruction
already run
record field updated
flag computed

WHAT BOCHS KEEPS

LAZY-FLAGS RECORD

operation	SUB32 (cmp)
op1	3
op2	5
result	**0xFFFFFFFE**

GUEST EFLAGS · COMPUTED ONLY WHEN READ

CF	PF	AF	ZF	SF	OF
?	?	?	0	?	?

WORK DONE ON FLAGS SO FAR

PUSHF approach	**3 PUSHF**
Lazy flags	**1 comparison**

STEP 4 OF 5

jne reads ZF

Now a flag is needed. ZF is 1 only if the saved result is zero. It isn’t, so ZF = 0 and jne is taken. The other five flags are never computed.

STATE

ZF = 0 → branch taken

Lazy flags, instruction by instruction

GUEST CODE

```
401000  8B 04 9E   mov eax,[esi+ebx*4]
401003  01 C2      add edx,eax
401005  43         inc ebx
401006  39 CB      cmp ebx,ecx
401008  75 F6      jne 401000
```

eax 7	ebx 3
ecx 5	edx 17

- this instruction
- already run
- record field updated
- flag computed

WHAT BOCHS KEEPS

LAZY-FLAGS RECORD

operation	SUB32 (cmp)
op1	**3**
op2	**5**
result	**0xFFFFFFFE**

GUEST EFLAGS · COMPUTED ONLY WHEN READ

CF	PF	AF	ZF	SF	OF
1	?	?	0	1	0

WORK DONE ON FLAGS SO FAR

PUSHF approach	**3 PUSHF**
Lazy flags	**4 flags, on demand**

STEP 5 OF 5

Any flag can be recovered

Had a jb or jl followed, Bochs would derive CF from 3 < 5 (unsigned), SF from the result's top bit, and OF from (op1^op2) & (op1^result). Everything needed was saved.

STATE

jb taken (CF = 1); jl taken (SF ≠ OF)

Every flag from the saved values

Arithmetic handlers save the result (and operands when needed). Each flag is derived on demand.

FLAG	COMPUTED FROM SAVED VALUES	NOTE
ZF zero	`result == 0`	the most-read flag, for je and jne
SF sign	`top bit of result`	a single bit test
PF parity	`parity_table[result & 0xFF]`	a 256-byte lookup
CF carry	`result < op1` (unsigned add)	the sum wrapped around
AF adjust	`((op1 ^ op2) ^ result) & 0x10`	carry into bit 4
OF overflow	`((op1 ^ result) & (op2 ^ result)) < 0`	addition form; see notes

- Replaced inline assembly calling the host's PUSHF, which was non-portable and no faster.

Why the XOR trick finds carries

A + B equals A XOR B wherever no carry arrives. So (A + B) XOR (A XOR B) has a 1 in every bit that received a carry.

AF: CARRY INTO BIT 4

```
op1           0000 1111   0x0F
op2           0000 0001   0x01
result        0001 0000   0x10
op1^op2^res   0001 1110
  & 0x10      0001 0000   AF = 1
```

Bit 4 received a carry from bit 3, so the adjust flag is set.

OF: THE SIGN FLIPPED WHEN IT SHOULDN’T

```
op1       0111 0000   +112
op2       0001 0000    +16
result    1000 0000   -128
(op1^res) & (op2^res)
          1001 0000   OF = 1
```

Two positives gave a negative: both operands differ in sign from the result.

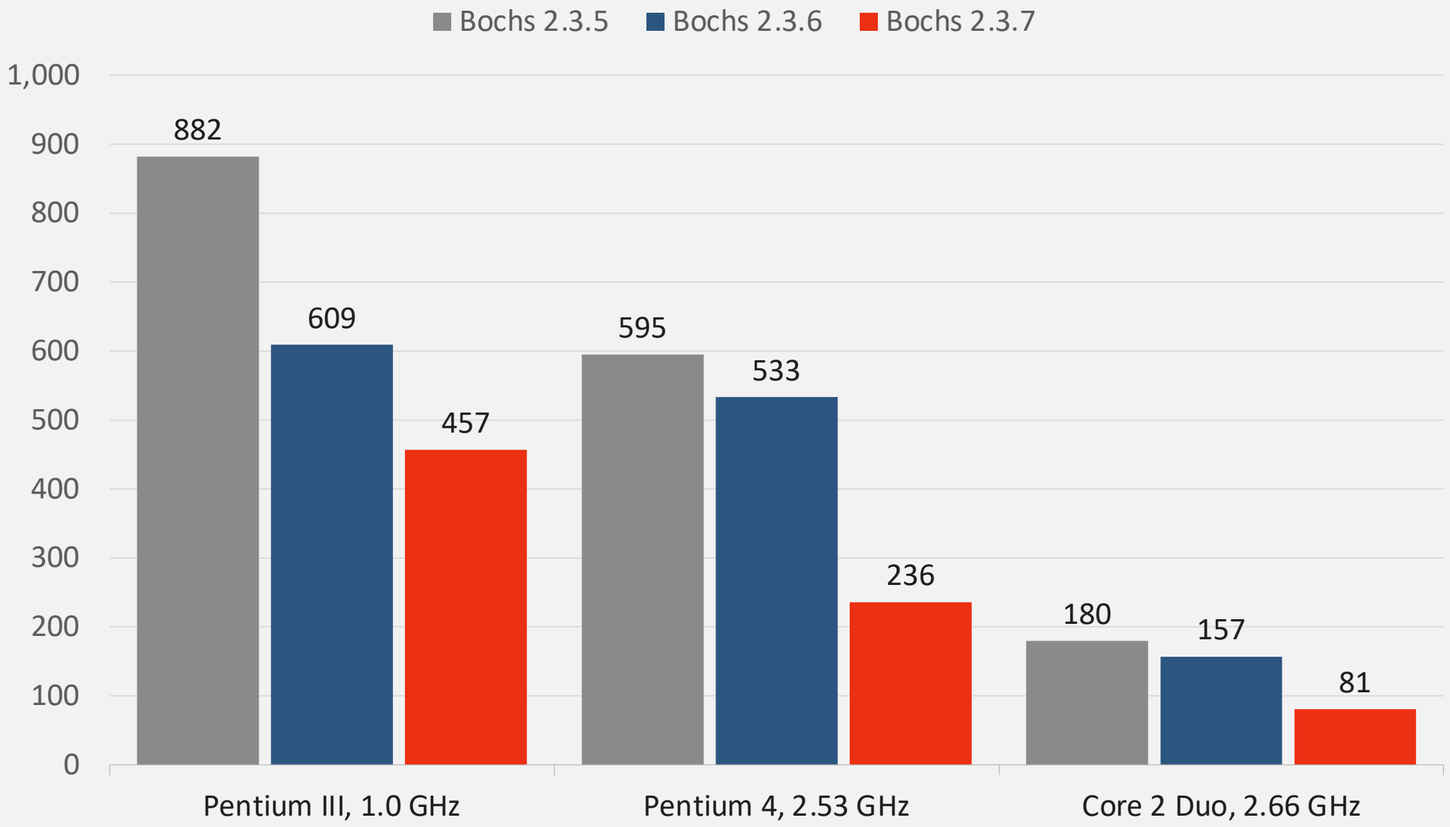
- Exercise: derive the subtraction form, (op1 ^ op2) & (op1 ^ result), which is what the paper prints.

07

Measurements

Windows XP boot time, three Bochs versions

Seconds from start of boot to a usable desktop (lower is better).



2.3.5 TO 2.3.7

1.9× faster on the Pentium III, 2.5× on the Pentium 4, 2.2× on the Core 2.

THE HOST MATTERS

Bochs 2.3.7 boots 2.9× faster on a Core 2 than on a Pentium 4 at nearly the same clock.

NOT PURE CPU

Booting includes tens of megabytes of disk I/O and device probing.

On a CPU-bound loop, Bochs 2.3.7 beats QEMU

Seconds to run 100 million calls, optimized build (lower is better). Native: 0.26 s.

```
static int foo(int i)
{
    return i + 1;
}

for (i = 0; i < 100000000; i++)
    t += foo(i);
```

10.5 s	QEMU 0.9.0 · dynamic recompilation
25 s	Bochs 2.3.5 · interpreter
8 s	Bochs 2.3.7 · interpreter

- Unoptimized build: QEMU 12 s, Bochs 2.3.5 31 s, Bochs 2.3.7 10 s. Bochs 2.3.7 sustains 87–102 MIPS.

Micro-benchmarks: where interpretation still loses

Approximate host cycles per guest operation on a Core 2 Duo (lower is better). Faster of 2.3.7 and QEMU shaded.

OPERATION	BOCHS 2.3.5	BOCHS 2.3.7	QEMU 0.9.0	WHY
Register move	43	15	**6**	dispatch overhead alone
Register arithmetic	64	25	**6**	plus the fixed lazy-flags cost
FP multiply	1,054	351	**27**	x87 emulated in software
Store a constant	99	59	**5**	a software TLB lookup
Load + store pair	193	98	**14**	two lookups
Indirect call (CALL EAX)	190	**109**	197	QEMU looks up translated code
Page fault + handler	888,666	380,857	**156,823**	guest exception path

- Peak guest rate: 62 MIPS for Bochs 2.3.5, 177 for 2.3.7, 444 for QEMU.

08

ISA extensions

HASHLU: expose the TLB to software

The software TLB is a hash lookup with a range compare, which is what TLB hardware already does. The authors propose user-mode instructions to use one.

TODAY · 9 INSTRUCTIONS, ABOUT 8 CYCLES

```
mov  edx,ebp
shr  edx,bitsSpan
and  edx,dwIndexMask
mov  ecx,ebp
add  ecx,cb-1
xor  ecx,[memtlbRW+edx*8]
mov  eax,[memtlbRW+edx*8+4]
test ecx,dwBaseMask
jne  emulate
```

PROPOSED · ABOUT 1 CYCLE WHEN PREDICTED

```
hashlu  eax,dword ptr [ebp],flags
jne     emulate
```

Look up a range, keyed by address and access size.

```
hashadd dword ptr [ebp],eax,flags
jne     error
```

Add a range. Clears ZF on a conflict or when full.

- Separate from the MMU’s TLB. It may drop entries at any time: false misses, never false hits.

SXF and LXF: the flags in a register, cheaply

Both interpreters must save and restore guest flags. Today’s instructions are partial or slow.

INSTRUCTION	WHAT YOU GET	THE COST
LAHF	5 of the 6 arithmetic flags, in AH	partial-register stall; no OF or DF
PUSHF / POPF	all of EFLAGS, via the stack	a stack write and read; POPF serializes
SXF / LXF (proposed)	6 arithmetic flags + DF in the low 7 bits	a full register: no stall, no serialization

WHO WOULD USE IT

Binary translators saving guest state before calling glue code, and any function that must preserve EFLAGS.

The long-term bet: a CPU built to host virtual machines

With these extensions and a VM in the BIOS, the authors suggest future x86 chips could shed hardware.

01

What could go

16-bit legacy support, segmentation, paging, heavyweight virtualization, rare instructions.

02

What it buys

Smaller dies and simpler hardware verification.

03

The precedent

Transmeta’s late-1990s processors ran x86 code through software.

THE OPPOSITE OF VT-X

Rather than making direct execution safe, make interpretation cheap: small primitives that fit low-power chips with tight gate budgets.

09

Takeaways

Four things to remember

01

Interpreters can pass 100 MIPS

Bochs peaks near 180 MIPS and Gemulator at 120–170, with no jit and no hardware MMU tricks.

02

Count mispredictions

Splitting handlers (~15%) and replicating indirect calls (~40%): portable C++ changes, no assembly.

03

Language and byte order barely matter

Different guests, languages and endianness, yet both interpreters land at similar speeds.

04

Portability costs under 3×

Measured against QEMU on x86, and Bochs already wins on some sequences, such as indirect calls.

For discussion

01 Every benchmark ran on an x86 host, yet portability is the paper’s main claim. What would convince you?

---

02 Bochs 2.3.7 beats QEMU on one loop but is still ~30× slower than native. When is that good enough?

---

03 QEMU and JavaScript engines still rely on jitting. What do they gain that this paper discounts?

---

04 HASHLU and SXF, or VT-x and nested paging: which kind of hardware help ages better, and for whom?