

Transmeta™ Crusoe™ Hardware, Software, and Development

David Keppel amas-bt-2009@xsim.com

Abstract

Transmeta was founded to build simple VLIW microprocessors with high clock speed, short pipeline, and enough issue width for high performance. Crusoe is the first Transmeta product. The only programs it runs are simulators, which in turn run operating systems and applications. This paper briefly describes Crusoe's hardware, software, tools, and development process.

1. Introduction

Transmeta¹ was founded on several observations: past VLIW projects gave good performance but had compatibility problems; the Shade [CK94, CK95] simulator sometimes gave performance close to one host instruction executed per target instruction simulated, suggesting simulation as a compatibility solution; and, for the time being, x86 was the industry standard ABI. The goal was to build a low-cost, high-performance microprocessor with 100% x86 compatibility. The first product was the Crusoe microprocessor.

Crusoe is² a traditional VLIW, plus simulation support [CDE+00, DGB+03]. One kind of support is for simulating any machine. As example, Crusoe has transactional commit/abort. Code is over-optimized, with safety checks; on failure, pending work is aborted, then re-run with lower optimization. A second kind of support is specifically for x86. For example, a special hardware flags register computes flags using x86 rules. x86 simulators thus have x86 flags “for free”, while non-x86 simulators must compute flags in software.

Notably absent is hardware support for decoding simulated instructions. Instead, software does all target instruction decoding. Crusoe thus truly simulates all instructions, and can run arbitrary instruction sets, given a suitable simulator. Transmeta's first public demonstration showed a machine simultaneously running both native x86 and native PicoJava bytecodes. Quake compiled to PicoJava ran as fast as it did compiled to x86.

¹Code Morphing, Crusoe, Efficeon, Intel, Microsoft, Microsoft Windows, MMX, and Transmeta are trademarks of their respective owners.

²For simplicity, this paper uses “is” instead of “was”. Transmeta stopped selling microprocessors in 2007 [Shi97] and was sold in 2008. The term “x86” refers to generic members of the Intel 8086 family, including IA-32.

The only programs running directly on the VLIW host are simulators, called “code morphing software” or CMS. CMSes are substantially patterned after Shade, using a main dispatch loop and cached and chained translations. Where Shade was simple, CMS is complex, supporting user and kernel modes; x86 segmentation and stack floating-point; optimizations to run CMS quickly; optimizers so generated translations run quickly; and multiple implementation strategies to run a wide range of user, system, and BIOS codes with consistent performance.

As important as the final product, are tools used to build, run, and debug Crusoe. Example tools are reference x86 simulators and corner-case tests; a fast simulator and fault-isolating debugger used to run CMS both while hardware was in development, and also long after hardware was for sale to the public; and a highly automated test farm and test generators.

2. Hardware

Crusoe's conventional VLIW issue includes two integer ALUs, an SSE-style FPU, memory load/store, branch, and special register access. Up to four types may issue in one VLIW instruction, but only in some combinations. Individual operations in an instruction may be suppressed under software control, allowing an individual operation to be simulated or suppressed, without affecting or needing to simulate the others [CDS03]. Long-running operations are typically interlocked, but short-running operations are not, using software scheduling instead [DGB+03].

x86 registers may be used as two 1-byte registers, a 2-byte register, or a 4-byte register. The VLIW has 64 flat 4-byte integer registers, with narrow 2-byte and 1-byte operations that do not sign extend. 32 80-bit FP registers match the x87 FPU, with 32-bit and 64-bit operations so MMX/SSE operations do not get excess precision.

Branches have no delay slot, but indirect branches may stall to a pipe depth of 5 cycles. Indirect host branches are used commonly in interpretation and in some translations. An indirect branch pipe allows branch targets to be pre-loaded into the pipe. If the preload is at least 5 cycles ahead, a branch-to-pipe instruction executes without delay [BCH03]. A branch pipe is generally difficult to use, but a few stylized uses are practical and show substantial speedup.

A departure from conventional processors is commit and abort, which implement a transactional model. Crusoe has two main structures for commit/abort: shadowed registers and a gated store buffer [KW99, WD00]. Shadowed registers are implemented as a pair of registers. On a register write, only the “working” register is updated. On a commit, all working registers are copied to corresponding “committed” registers; on abort, committed registers are copied to working, thus erasing any changes since the last commit. Shadowed registers include some integer, FP, and special registers. Shadowing only some simplifies hardware, and gives a place to store “why” state when a transaction is aborted.

The gated store buffer is conceptually similar to a regular store buffer, but only entries which have passed a committed “gate” are allowed to drain. On a store, the working gate register is advanced. Working and committed gate values are handled as with other shadowed registers. On loads, the whole store buffer is snooped, whether or not committed. The gated store buffer is conceptually simple, but has subtle complexity. First, it is large, because translation size is often limited by the number of stores, and using smaller translations hurts performance. Second, some snooping corner cases must be resolvable without draining the store buffer, because draining implies commit.

Physical memory is divided into a CMS-only section and a target-only section. CMS memory holds the simulator code, translations, and simulator data structures. The target section contains committed target memory state. Pending uncommitted stores are never released from the store buffer until committed. A third, on-chip “local” memory is small and has guaranteed fast access and is accessed with short instructions. Local memory holds frequently-used state that is not so frequent it deserves fixed allocation in VLIW registers. Local memory holds both CMS and target state.

More support for general simulation is provided by “alias registers” [WK99, K1a00, RDP06]. Each holds the address of a memory operation and is tied to a comparator that signals an exception if it overlaps with a memory operation. CMS may then arbitrarily reorder memory operations within a translation, saving dependencies in alias registers. If a reordered memory reference overlaps another, the translation is aborted and retranslated at lower optimization. In practice, most loads and stores can be reordered.

Target-specific simulation support includes condition code flags mentioned above. A second set of condition codes is used by CMS, and avoids multiplexing a single condition code register [CK93].

The first VLIW design used no segmentation hardware, but after better workload analysis, hardware segmentation support was added for certain common cases. Although segmentation support mimics some usual x86 segmentation functionality, the bit-level layout is distinct. Performance

counters implement a superset of x86 counters, and are used both to implement x86 performance counters and internally by CMS [CKB01].

The TLB has both generic simulation support and x86-specific support. It has the same page size and basic protection controls as an x86, although the underlying organization is different. The TLB is software-invalidated. CMS must detect instruction space changes, as x86 uses implicit rather than explicit instruction/data consistency [Kep09]. Thus, the TLB has two write-protect bits, one for x86, and one set during translation from a page, so subsequent page writes fault and CMS can invalidate translations [KCW96, KCW00], and the TLB is used even when x86 paging is disabled. Later Crusoes have a small cache of sub-page write-protect bits used sometimes when code and data share a page [BAG+01].

The TLB also has four bits that compare against a special “match” register. x86-level TLB invalidations typically increment the match register, rather than performing a full invalidation. The TLB is 4-way associative and sectored, with a valid bit per sector.

An x86 has three kinds of I/O. Memory-Mapped IO, or MMIO, uses ordinary “memory” loads and stores. When they “miss” RAM, they perform device reads and writes in a 32-bit address space (36 bits with PAE). Direct Memory Access, or DMA, allows devices to asynchronously read and write RAM in the same address space as MMIO. In-Out I/O, or IOIO, uses in and out instructions that read and write devices in a separate 16-bit address space.

Various details complicate I/O. One device may be connected to both address spaces. A portion of the low megabyte of address space is mapped to a legacy video device and never to RAM. Another portion has reads and writes steered separately. One use is an x86 BIOS copy loop that does simply “read X; write X” to copy from a memory-mapped ROM to RAM, while executing from the same page that holds the data being copied. Crusoe has special steering hardware for just these addresses; it is unused for other targets.

DMA is conventionally performed by an asynchronous copy engine, but on Crusoe that introduces coherency problems for translated code. Crusoe uses a DMA interrupt and CMS performs all DMA and checks for translation invalidation [BKK+03]. Software DMA serializes DMA and removes benefits of concurrent computation and communication, but early workload studies showed DMA was rarely used. DMA is common in newer workloads, and Crusoe’s successor, Efficeon, uses a simple DMA engine which handshakes with CMS to ensure DMA writes do not overlap translations, but allows asynchronous DMA otherwise.

Ordinary memory can be read multiple times without side-effects, which in turn allows speculative operations. Memory-mapped I/O, however, must be read exactly once

for each target read request. This is complicated to simulate, as any memory reference, including an instruction fetch, can go either to real memory or to memory-mapped I/O. To make translation practical, VLIW loads and stores use two address space identifiers, or ASIs: “normal” (#n) and “abnormal” (#a). RAM is accessed only with #n. A load or store to an MMIO address with a #n ASI causes a “normal assumed” exception. Similarly, MMIO uses only #a. These ASIs enable translation optimizations for RAM accesses, while ensuring correct behavior on MMIO. Large objects, such as 80-bit FP and multiword x86 complex data, are implemented using multiple VLIW loads and stores. To ensure “all or nothing” behavior, a sequence of check instructions precede and protect large loads and stores. Check instructions fault if the corresponding load or store would fault [KSD03].

The PC architecture extends beyond the x86 instruction set, and includes bridges for memory and external devices. Crusoe uses a memory controller integrated with the CPU. The North Bridge, which is closest to the CPU, is implemented in software, with underlying VLIW CPU support for memory control registers, etc. x86 bootstrapping with an integrated memory controller is somewhat complicated, as RAM is checked by CMS before the first x86 instruction runs, but x86 software can reconfigure RAM. The South Bridge is a commodity part, as used with other x86 CPUs.

Older x86es use the PCI bus for graphics. AGP became popular during Crusoe development and offers much higher performance via a faster bus and RAM-like behavior. AGP is not included in Crusoe hardware, limiting both graphics performance and GPU compatibility. Crusoe’s successor, Efficeon, has AGP.

Crusoe power management includes LongRun [K1a00, HAC+06, RHK07]. LongRun allows the processor to quickly stop, adjust clock speed and voltage up or down, and restart. CMS controls LongRun, increasing speed when the processor is continuously busy, and decreasing it when the processor is mostly idle over some interval.

Savings are substantial because lower frequencies allow proportionally lower voltages, and dynamic power consumption is roughly proportional to V^2F and leakage is roughly proportional to V^2 . Thus, running at 90% frequency uses roughly 70% static and 80% leakage power compared to running at full speed. LongRun2, introduced with Efficeon, extends the V^2F observation, using an adjustable substrate bias to cut both static (leakage) and dynamic (switching) power, and to increase switching speed. Actual energy use varies, as a slower processor takes longer to complete work and thus spends less time in deeper power management states, but in practice CPU energy savings are often substantial; and in a typical laptop, the CPU is the second biggest energy user, after the display backlight.

Performance at 90% clock speed is also typically better

than 90%: many workloads spend substantial time blocked on memory, and memory typically stays at 100%. Thus, cache misses are effectively faster when the CPU is slowed.

LongRun complicates characterization and binning because transistors are not precisely linear, so transistors may “bin” differently depending on the specific voltage and frequency setting. Yields are improved by using higher voltages, but at the expense of higher power consumption. Different settings could have been allowed for each part, but at intolerable manufacturing expense to manage multiple V/F mappings for each part.

VLIW registers allow software to choose higher or lower power/performance settings, then request a level change. The request briefly stops all execution, and in hardware changes power supply voltage and frequency multiplier, resynchronizes the PLL, then restarts execution. Some care is needed to ensure the power supply does not glitch during voltage changes. Since no work is done during changes, LongRun software must balance what V/F settings to use, when to change, and how often to change.

All exceptions go to a common trap vector. Exception types include usual processor exceptions plus “special” exceptions to support simulation. Exceptions are typically serviced two ways. One is “fix up and continue.” For example, on a TLB miss fault, a simulation routine reads target page tables, loads the hardware TLB, then restarts the faulting instruction. Another example is to simulate a faulting operation, then restart the faulting instruction with the relevant pipe disabled [CK04]. A second service category is “abort and go to main loop.” For example, if the store buffer overflows from too many uncommitted stores, the current translation is aborted and the simulator creates a new translation with fewer stores between commits. (The number of stores is often not known during translation, see §3.) x86 interrupts, including DMA, are also handled via abort to ensure handling when x86 state is precise.

Exceptions set a “fault” bit in hardware, and return from exception clears the bit. If another fault occurs while the bit is set, control vectors to a special “double fault” location in the exception vector, whose handler carefully services the nested fault [AK04]. Nested faults are dynamically rare, but there are many corner cases that can give rise to them. For example, placing page tables in video memory causes TLB miss handling to take a normal-assumed fault; the translation is aborted, then during interpretation the miss handler’s #n load is promoted to #a. Nested fault handlers increase software complexity because there are multiple handlers for each fault type, but nesting speeds common-case handlers, e.g., by allowing a dedicated temporary register or by ignoring uncommon cases and letting them double fault.

3. Simulator: CMS

Crusoe hardware runs only one application: CMS. At a high level, CMS is a “traditional” translator like Shade [CK94], with data structures holding the target state (TS), and simulation code and data including a main loop, translator, translation cache (TC), and a translation map (TM) used to find a suitable translation given the current virtual state. In addition, CMS keeps potentially extensive information for each translation (TCinfo), and CMS uses an interpreter: it interprets instructions several times before translating them, and interprets to handle certain corner-case behaviors. Other components include “out of line” sub-routines called from translations or the interpreter, called “OOLs”; dynamically-generated repeat string OOLs called “DROOLs”, an exception-handling kernel called the “nucleus”, a simulated “virtual north bridge”, LongRun, power-on reset, and cross-debugger support.

Target state is held principally in four places. First, target memory is stored in a target-only part of host RAM. Second, ordinary integer and floating-point/media (FP) target registers are allocated to shadowed VLIW registers. Third, complex non-memory target state is held in local memory and cached in VLIW registers as needed. Targets with many registers allocate target registers to VLIW local memory and cache them in VLIW registers on demand. For the x86, hard allocation to VLIW registers simplifies bookkeeping, especially given the x86’s asymmetric treatment of registers. Finally, target device state is held in target devices: CMS state exists only in the processor, RAM, and ROM, and no target devices besides ROM and RAM are assumed or required.

On entry to the main loop, CMS looks for a translation that matches the current program counter (PC) and context (described below). If no translation is found, the main loop calls the interpreter, which updates instruction and branch counters as it runs [TA05]. When a given instruction is executed a few tens of times, the interpreter returns, and instructions are translated using the branch statistics to guide translation.

Interpretation costs on the order of 100 VLIW instructions per simulated target instruction, where translations are typically under 1 VLIW instruction per simulated target instruction. However, interpreting avoids translating rarely-used instructions, which may cost 10,000 VLIW instructions per translated target instruction. And, interpretation collects profile data that greatly improves performance of many translations. The interpreter also implements precise x86 semantics and computes exact x86 state at every instruction boundary. The interpreter is thus used for delivering faults and for handling corner cases that are too rare or expensive to handle in translations.

CMS finds translations by hashing the current target program counter (PC) to subscript the TM. CMS matches both

virtual and physical addresses: the same code at a different virtual address may need to be translated differently, as not all code is PC-relative; and mapping the same virtual address to a different physical address may reference different code. The V-to-P check uses the TLB and so also performs needed protection checks. CMS also matches saved information called “context” [KCB01]. For example, an instruction may execute differently in real mode than protected mode. Rather than generate one general translation that works for both modes, CMS generates a streamlined translation for each situation.

Other approaches are possible, such as TM invalidation on each mode transition. But using context allows fine-grained choices. For example, the user/kernel mode is in the context, and mode checks are performed at translation time rather than during execution. The context risks duplicating translations, but what goes in the context is chosen so in practice most code is executed in only one context. A problematic case was emulating the mapping from x87 stack to VLIW registers. Most code executes a given FP instruction with only one mapping, but non-FP code is often run with many FP mappings. This was solved by moving FP stack mapping to hardware [RDD+04].

The main loop has various entry points: translations may return to the main loop in various ways depending on their status. The interpreter also branches to the main loop. And while nucleus aborts largely sanitizes x86 state, there is simulator state which merits special treatment on exceptions. For example, “store buffer overflow” is handled by retranslating with more frequent commits; “x86 page fault” is handled by executing instructions one-at-a-time until the fault reoccurs, then by branching to code which simulates page fault delivery then resumes x86 execution [TK01].

A goal of CMS is to minimize trips through the main loop and instead branch directly to suitable code. Examples of this include translation chaining, retranslating to avoid speculation exceptions, and so on. A case which is common and difficult to avoid is indirect target branches, which may go to any address. Where branches are dynamically predictable, a flavor of receiver caching [DS84] is efficient. However, many indirect branches are unpredictable. For example, return from a procedure such as `strlen()`, which often has many callers. For unpredictable branches, CMS uses a fast lookup in a small cache of recent lookups, falling back on misses to a general TM lookup. Early Crusoe uses a software cache; late Crusoe uses a small hardware cache [BCT+03].

A translator for a Shade-like simulator is straightforward: for each target instruction, emit a “canned” sequence of host instructions, and repeat until a branch, until some threshold is reached, or until an especially tricky case is reached. In contrast, the CMS translator is an optimizing compiler whose source language is decoded x86 in-

structions, and whose optimization strategies include conventional things like common subexpression elimination, VLIW-specific instruction scheduling, and unconventional things like alias protection for speculative code motion.

The translator first reads x86 instructions and decodes them to an intermediate representation (IR). The translator typically reads instructions that have already been interpreted, but may read new instructions and thus must perform all usual fetch-time checks. The interpreter collects conditional branch profiles, and the translator follows branches but often avoids branch-to-self to avoid store buffer overflow – Crusoe translations are single-entry multiple exit, and while they may contain embedded loops, they tend to be more like DAGs rather than full sub-graphs [DS84, May87]. Once a translation is generated, it is sometimes retranslated in to several simpler translations, but translations are not merged [May87] except for chaining.

TM lookup performs a page protection check on the first instruction in the translation. When a translation spans pages, the translator generates checks for other pages in the translation [BKB05]. Chaining to a translation on a different page goes by way of a page protection check called an “interpage prologue” or IPP [BKB05].

CMS optimizations include “traditional” compiler optimizations as well as simulation-specific and Crusoe-specific optimizations. All optimizations would ideally provide good translation speedup with little translation-time cost. However, choosing optimizations is tricky, as benefit varies widely with the workload, and even cost varies some with the workload.

It is the nature of machine code that translations often contain redundant subexpressions, especially for memory addressing. Constant folding and propagation are also useful. Translations often speculate branches based on interpreter profiles, and commit and branch to another translation for the not-predicted path. Thus, translations typically have few internal branches, and various loop optimizations are of less value. Short target subroutines may be inlined in a translation – being sure to push and pop the target PC. Translations do, however, have branches to call OOLs.

The VLIW has eight alias registers, which allows loads and stores to be moved speculatively. If they overlap, an alias exception is delivered, but except for memory references that “obviously” overlap, most do not overlap, so the compiler often does a good job choosing non-overlapping operands. The compiler avoids using alias registers where it can: one translation often has tens of loads and stores, making alias registers a scarce resource. Some cases are amenable to translation-time analysis, for example adjacent offsets from the stack pointer within 4096 bytes never overlap, even if adjacent stack pages map to the same physical page. (In real mode, segments can be arbitrary sizes, but

segment wrap only occurs with full-size segments, which are a multiple of page size.)

VLIW instruction scheduling typically schedules the longest path then fills available slots where possible. Alias registers and the corresponding load/store motion often makes translations significantly shorter. Alias speculation occasionally fails and requires retranslation. Failure is rare, so for nearly all applications it is at least a slight win, and for some nearly doubles performance [DGB+03].

Register allocation is somewhat complicated, as some registers are dedicated to target state and so are live on entry and cannot be renamed on exit. With many registers, modest issue width and a short pipe, register pressure is not typically a problem. The x86 flags register is set by most x86 integer instructions, and some x86 instructions set only some flags leaving others with their earlier values. Thus, analysis is needed to safely elide flags operations and keep flags calculations from being a bottleneck.

Crusoe also simulates x87 FP, whose architecture includes eight FP registers, a “valid” bit per register, and a “top of stack” (TOS) register, often used to treat the registers as a stack. FP instructions contain offsets which are added to TOS, mod eight, to form absolute register numbers. `fadd` is a two-operand add, where one operand is offset zero. `faddp` also increments TOS; in typical use, the second operand has offset one, so `faddp` has the effect of popping the first register and making the second register the new top of stack, which the next instruction accesses with offset zero. Crusoe uses a hardware TOS (but see §5).

Each x87 register also has a valid bit. Reading an invalid register or overwriting a valid register causes an exception. In typical use, valid bits are set for register numbers TOS and higher, and clear for lower-numbered registers. CMS detects this and only simulates valid bits explicitly when they follow a different pattern.

The x87 has deferred errors, where state about each FP instruction is saved internally, and executing a later FP instruction delivers pending errors. The saved state is: the code segment, PC, opcode, data segment, and operand offset in the data segment. Saving that state substantially complicates and slows simulation. However, target FP instructions are often clustered, so a single translation has only one “last” FP instruction that is complicated to simulate. Most but not all FP instructions check for pending errors; errors may be ignored or delivered; and errors may be delivered as exceptions or interrupts, optionally routed through off-chip logic, and optionally cause the x86 to stall until a non-maskable or non-masked interrupt.

The VLIW store buffer is 32 entries, but translation typically stops at fewer than 32 x86 stores. Many stores take up more than one store buffer entry because misaligned stores may span store buffer entries and because x86 memory accesses cause implicit “accessed” or “dirty” bit writes to the

page tables, again using store buffer space. Some CMS state writes use buffered stores. A few x86 instructions use more than 32 entries. For example, `fnsave` directly uses 29 to 37 store buffer entries, depending on the size and alignment of the operand – plus any used implicitly for TLB accessed and dirty bits. CMS does not directly translate large cases but instead the translation calls an OOL. The OOL pre-checks all stores for possible faults; if all are safe, stores are committed incrementally.

Translation also stops for other reasons. One reason is target instructions that make pervasive state changes, making further translation difficult. For example, state loaded from target memory that changes the context. Translation also typically stops soon after MMIO loads, because many translated operations handle corner cases by aborting, and MMIO loads cannot be aborted and may not be repeated. Translation need not stop immediately, but many instructions cannot follow “tricky” instructions, and these are usually encountered soon enough.

To reduce duplication, translation sometimes stops on x86 instructions that are already translated. Translation also stops after at most 200 x86 instructions [DGB+03].

Finally, translation may stop due to adaptive retranslation: CMS records when and why a translation fails, so retranslation can create better code for the situation. As an example, when a particular x86 instruction faults frequently, CMS may generate a new optimized translation that ends just before the faulting instruction, followed by a translation that branches directly to the interpreter to handle faults [DGB+03]. Adaptive retranslation is also used to deoptimize when speculation is too aggressive, though this does not necessarily shorten translations [DGB+03].

Translations are stored in a conventional TC. Shade-like translators typically fill the TC then invalidate it completely. CMS translation is expensive, typically thousands of VLIW instructions per translated x86 instruction, so CMS reclaims TC space incrementally to save retranslation overhead. The TM matches virtual address, physical address, and context.

The interpreter and translations both call OOLs. OOLs typically perform rare or expensive operations that would otherwise bloat translations without performance gain. Examples include simulation of call gates, segmentation changes, north bridge simulation, and so on. Some OOLs return to the caller and do not commit; other OOLs change state in a way that makes chaining tricky and so commit then return to the main loop.

A special case is x86 `rep` instructions, which repeatedly scan or move bytes in memory. For example, “scan for the byte value B” or “move N words”. There are many `rep` variants, and some combinations of alignment and transfer size are much faster with special-case CMS code. `rep` code is large enough it is best to avoid inlining it in translations, but like `BitBlt` [Loc87], there are many special cases, and

static specialization would bloat CMS. Thus, CMS demand-compile DROOLs tailored to particular calls. Although there are many `rep` cases, many programs dynamically use just a few. Thus, performance is improved at low space cost.

User-level simulators use the host to handle most exceptions. Exceptions that are simulated typically poll whether an exception is needed. System-level simulators typically simulate all checks needed for delivery. CMS relies on the underlying VLIW to deliver most exceptions and interrupts, which are vectored through a “mini OS” called the nucleus. CMS also simulates some checks, and also typically vectors them through the nucleus with a “conditional trap” instruction, which is smaller than a conditional branch and also reduces contention for the VLIW condition codes.

As an example of nucleus operation, consider a store instruction to the `#n` ASI which takes a “normal assumed” exception because the store actually goes to MMIO. If the faulting instruction was translated – that is, is in the TC – the nucleus aborts and branches to the main loop, indicating “retranslate abnormal assumed”. Static code typically does not have “abnormal” variants, as that would roughly double the size of OOLs and the interpreter, yet still leave tricky corner cases such as a page-spanning read or write that touches both RAM and MMIO. Instead, the nucleus copies the faulting VLIW memory operation, changes the ASI in the copy to `#a`, executes the copy, then restarts the excepting VLIW instruction with that memory pipe suppressed, so the original load or store is not reexecuted. Page-spanning loads and stores are split as needed, and MMIO byte lanes masked. Any MMIO, once performed, must not be interrupted or aborted, so the interpreter and interpreter-like OOLs execute each instruction under lock, and all static code in the interpreter or callable from the interpreter follows certain rules to ensure aborts are never needed once a `#n` load or store is promoted to `#a`.

Note a copied instruction may itself take an exception, for example due to a TLB miss or page table dirty bit write. In that case, the nucleus takes a doublefault exception and carefully saves state so the nested exception handler does not step on the first-level handler. The nested fault is serviced, then state is restored and the first-level exception resumed. On a dirty write, the store buffer may overflow, causing the doublefault handler to be invoked recursively, called a triple fault. Store buffer overflow and other such cases cause an abort, so interpreter code is constructed to avoid cases that can give rise to overflow. As example, `fnsave` checks all permissions before performing any memory operation, then commits stores incrementally to avoid store buffer overflow.

x86 and DMA interrupts are also vectored through the nucleus. The nucleus aborts and branches to the main loop. All of CMS must therefore be built so all nonshadowed accesses are safe against interrupts. As example, a nonshad-

owed register may be read and then written between commits only if interrupts are locked out. Similarly, all memory references that use the ASI #a or which might be promoted by the nucleus to #a are locked, stay locked until the next commit, and must not abort.

Simulation of self-modifying x86 code is problematic in part because the x86 relies on hardware instruction/data coherency, where most other processors have a software mechanism to indicate I/D changes. “Self-modifying” code takes many forms, including paging (code pages are reused to hold different code), dynamic compilation at various grain sizes and reuse rates, one-time instruction patching such as used by some dynamic linkers, and patching of individual bits or fields in some instructions [Kep09].

To respond with good performance, CMS uses several mechanisms. In most cases, the primary goal is to avoid frequent retranslation, as that is the highest cost, often two orders of magnitude slower than interpretation.

By default, code pages are write-protected during translation. The VLIW TLB has a second write-protect bit called the “T” (translate) bit. For each page translated, CMS sets a bit in a simulator data structure and invalidates TLB entries if the bit was clear. On TLB miss, the TLB’s T bit is loaded from the data structure.

Write faults invalidate all translations for the page, so executing from a modified page forces retranslation. If fault rates for a page are high, CMS uses a variety of strategies that are asymptotically slower but also less sensitive to change.

“Self-revalidating” translations use a VLIW structure that divides a page in to subpages and keeps a bit per subpage. On a first write, a write fault handler marks the subpage as writable but not executable; subsequent writes do not fault. Special translations for these pages check the corresponding bits for all subpages used by the translation. When all subpages are executable, the translation simply executes. If a subpage is marked writable, the translation compares current x86 memory against bytes saved at translation time. If all match, the bits are marked executable, and further uses of the translations do not need to re-check. If any mismatch, the translation is invalid and CMS uses another strategy. This strategy efficiently handles most cases where code and data are on the same page and code rarely changes. Although the VLIW structure is small, applications rarely have many pages sharing code and writable data [BAG+01, DGB+03].

“Self-checking” translations use unprotected pages: every time a translation is invoked, it compares the current and saved bytes [BAB+03]. Self-checking translations are thus slower than self-revalidating translations, but can compare only those bytes used for the translation itself, rather than a whole subpage, and they eliminate write faults and state ping-ponging of self-revalidating translations [DGB+03].

The x86 assures coherency after a branch, after executing 16 bytes of instructions, or after a serializing instruction. Thus, some checking must be done late in the translation, after any possible writes.

When only immediate fields are changing, translations can fetch the values from the x86 instructions, being sure to skip immediates in checks [DGB+03]. Where instructions change but only to a few “stylized” patterns, CMS keeps a list of previously-generated translations and checks if any have become valid again [DGB+03]. Finally, CMS can generate a translation which simply calls the interpreter, thus avoiding further overhead to “rediscover” where interpretation is needed.

CMS is also heavily involved in power management. LongRun manages two types of state transitions. When an x86 is idle, the OS executes a HALT. LongRun tracks these and slows the CPU the more often it is idle. At idle, LongRun also sets a countdown timer; when there is no idle time the timer expires, signaling LongRun to increase speed. At top speed, no timer is needed, so LongRun has no overhead.

CMS bootstrapping is different than most simulators: At power on, CMS runs – slowly! – from the system ROM. It copies code to local memory, executes from local memory to configure RAM, then copies CMS to the CMS part of RAM, initializes the x86 state to power-on-reset then starts executing the BIOS.

CMS is stored in a programmable ROM along with a backup “fail safe” CMS. CMS changes are cryptographically protected against downloading arbitrary code, and the backup ensures that even a power failure during reprogramming is recoverable. Primary and backup CMS versions are in isolated physical parts of the ROM so power failure during CMS write cannot corrupt the backup CMS. CMS also has a fixed-format configuration table at a well-known address, used to hold configuration data such as CMS and LongRun tuning parameters. It is not writable by users, but supports vendor configuration for each platform.

32-bit x86 was the only Crusoe released publically. x86-64 was implemented on Crusoe hardware [KF03]. It had about 100x lower performance, due partly to multiplexing 64-bit operations on 32-bit hardware, but due mostly to interpreting only – no translation. Though incomplete, the x86-64 implementation passed all existing validation tests and could boot and run OSes and applications. It was probably the first hardware x86-64 processor, predating even FPGA implementations.

4. Supporting Software

CMS exists only because of supporting software: reference simulators, a fast VLIW simulator, error detection and isolation tools, bug and source code management tools, build

and debug tools, test generation and test suites, test and farm management, and performance monitoring, visualization and navigation tools.

One part of building “an x86” is discovering and defining “what is an x86”, then committing that to a reference simulator. A reference simulator is a standard for comparison to find bugs in the product. There are many problems discovering what is an x86: there are multiple standards, standards sometimes conflict, x86 implementations sometimes disagree with standards, and some applications, including OSes, depend on nonstandard behavior. Parts of some operations are undefined and vary from implementation to implementation. Applications may depend on combinations of behaviors that are nominally independent; such applications are arguably buggy, but some have historically been supported anyway.

For all these reasons, the reference simulator changes continually to reflect newfound understanding of the x86. Further, the reference simulator is also used to simulate various non-product x86es to diagnose changes between execution on the product x86 and a non-product x86. Thus, the reference simulator needs to be easy to change, configurable, reliable, and fast enough to not be serious bottleneck. In practice, execution speed is typically over 100 host instructions per simulated target instruction.

CMS development started years before working VLIW hardware. Development uses a fast VLIW simulator that uses dynamic translation and runs on an x86. It typically runs about 30 host instructions per simulated target instruction. As CMS matured and approached 1:1 x86:VLIW, overall performance under the VLIW simulator was so good it could boot and run x86 operating systems in something approaching real time – visibly slow, but still fast enough to enable many kinds of work that would have been infeasible on a slower simulator.

Even with VLIW hardware available, fast simulation is still vital. At first, only a few systems are available, they are in high demand, and are potentially flaky. As systems are more widely deployed, the superior environment in simulation still makes it preferable for many tasks. Finally, as hardware is revised, simulation is again available before revised hardware, allowing development, testing, and measurement to proceed with workarounds removed and new features added.

The VLIW simulator is itself a study in simulator construction: the host x86 has few registers, and the target VLIW often shadows resources to speed up simulation, so the fast VLIW simulator needs to simulate both sets of resources. For example, the fast VLIW simulator keeps an “undo” log. When writing a shadowed resource, it copies the old value and address into the undo log. On commit, the undo log is discarded. On abort, undo log values are written back to memory. However, nonshadowed values can also

be clobbered, so there is also an “undo-redo” log that is replayed after “undo”.

The VLIW simulator’s commit/abort mechanism is extended to save total machine state, called a “checkpoint.” Reloading an earlier checkpoint restarts execution from earlier in the execution. Checkpoints include an I/O log, so even interactive sessions with live input can restart from an earlier checkpoint and replay I/O, thus reproducing execution exactly.

The reference simulator also supports checkpoint and restart. In principle, adding checkpoint and restart to any simulator consists of saving and restoring target state and logging target I/O for replay. In practice, simulators tend to cache target state in various tricky ways. For example, a translating simulator may cache translations based on a bit in a target register, and moving to a different checkpoint may change the bit. Thus, a simple bit copy of target state may be insufficient. However, an “official” interface to change the bit may perform consistency checks, so using it to restore state may signal an error, even though state will be consistent again when checkpoint restore finishes. In addition, the simulator may keep non-target state such as pipeline state and profiles, and that state should be updated at each checkpoint change.

Checkpoints are large – the size of the target state – so frequent checkpoints and restarts are expensive due to both the direct cost of copying state, and because restart often needs to load a checkpoint from secondary storage. Optimizations such as asynchronous writes speed up checkpointing considerably, but it is often inconvenient to store checkpoints in a format they can be resumed via demand paging. For a simple restart, performance is rarely annoying, but binary searches during fault isolation (see below) occasionally take a long time.

Checkpoints may be loaded by different debugger sessions. Automated farm testing executes long-running tests to find bugs, then leaves checkpoints so humans can quickly get to the point of failure found by automation. Checkpoints are also an easy way to hand off debugging sessions between developers.

Each simulated VLIW instruction is given a virtual timestamp called a “vtick”. Each simulated target x86 instruction executed is given a virtual timestamp called an “xtick”, whose name has stuck even for non-x86 targets. CMS calculates the xticks performed by a translation and tells the simulator using a specially-coded VLIW NOP. The simulator can stop at a given vtick; since xticks advance in chunks, it can sometimes only stop near a given xtick. Restart from checkpoint is wrapped in a command `gotov V` or `gotox X` that loads from the most recent checkpoint before `V` or `X`, then executes forward until the tick is reached. Finally, the simulator saves checkpoints automatically, so developers never “see” checkpoints, instead think-

ing in terms of executing to a specified tick. This facility is called “reverse execution.”

The reference simulator and fast simulator both have checkpoint and restart facilities, and are run together under an execution control framework called the “nexus”. In simple form, the nexus tells CMS to execute one translation then stop, then advances the reference simulator to the same xtick, then compares all of registers and memory, halting if there are differences [KBK99]. Paired execution is called “cosimulation.” Stopping every few instructions to compare millions of bytes of memory is expensive, so in practice the simulators are advanced by thousands to billions of x86 instructions. In this way, reference comparison costs are kept to a small part of the total simulation cost.

When an error is found, it could be anywhere since the last comparison. An early and vital feature of the nexus is “narrowing”: the nexus backs up to the last agreeing comparison, then executes forward half the number of ticks to the failure. If comparison fails, the error is in the first half; if not, it is in the second half. This repeats until the nexus arrives at the first xtick past failure. Thus, errors are isolated to a specific failing translation, often in a few seconds.

The debugger runs the nexus. It also looks in the “guts” of CMS. For example, the debugger prints VLIW registers along with the x86 registers they represent. Thus, with a few keystrokes it is possible to land at an error, examine the host and target code, back up to the last agreeing xtick, set breakpoints or single step, watch values changing, and even examine low-level state, such as uncommitted entries in the store buffer.

Although some errors are subtle, many are “obvious” and easy to repair. Given the reference simulator and nexus, the time from failure to repair is often just a few minutes.

Once VLIW hardware was available, the simulator was extended, where possible, to provide the same views. For example, much VLIW internal state is available via special machine registers. The VLIW simulator was extended to track even state that is not essential for VLIW simulation. On hardware, the debugger also gives access to state that is not simulated, such as cache tags and contents. Such state can be queried interactively and can also be downloaded for automated analysis – which periodically proved vital in diagnosing errors.

Many early bugs were simply found and fixed. Later bugs are often more complicated and have several people working together to discover, diagnose, and repair. Bug tracking is largely done using Gnats, though customer-facing groups use a tool with work flow. Gnats works well for developers in part because it runs “cleanly enough” within GNU Emacs, which is preferred by most software developers and used frequently by others.

CVS with wrappers is used to manage CMS source code, as well as sources for most tools. CVS has various well-

known limitations, some of which are solved by the wrappers. Fortunately, most blocks of code are relatively independent – building a specific version of CMS may require a specific version of build and simulation tools, but such dependencies are usually simple. Branches are used frequently but with frequent merges against the trunk, or with an explicit plan to never merge.

Custom build tools are used to generate VLIW object code. Much of CMS is written in C with selected extensions. Some extensions give access to special VLIW hardware, for example special registers in the VLIW. Many extensions control compiler behavior. For example, some code is checked using PC ranges, so it is vital an inline function really is inlined; a compiler directive causes compilation failure if the routine is not inlined. Similarly, tail recursion and other properties can be assured.

A fast build system is useful for all development, but vital for a code base with frequent checkins and loose checkin requirements. The build system uses both incremental and “from scratch” build servers for CMS. Incremental builds sometimes give false failures due to bad dependencies, but can validate or refute a checkin in just a few minutes. The full build server is more reliable, but can take many tens of minutes. Often, errors are fixed and checked in well before the first full build reports failure. A fast build system is a mixed blessing, in that it encourages sloppy checkins. On the other hand, fixes and features by one developer can be available to all in minutes. Especially in early development, the ability to request a fix or feature and everybody have it five minutes later is a great win.

Like the nexus, the build system does failure narrowing. If there are five checkins while build #123 is running, the build server next builds #129. If that fails, the build server both reports failure and starts a binary search for which checkin triggered the failure.

Once code passes rudimentary testing and checkin, it goes through escalating tests. The first tests are simple ones that have often failed before [Cla91]. Better and better candidates are run on longer and longer suites, some running for weeks. The best candidates, are subjected to human testing, and some difficult or risky changes are also human-tested. The fast VLIW simulator means human testing can begin long before hardware is ready; and starting human testing early means increased maturity when hardware is available.

Testing uses both VLIW and x86 suites. Tests include instruction-level tests and system level tests including performance tests. VLIW tests are all written in-house, while x86 tests include both in-house and external tests. External tests include those bought from other companies, industry-standard tests, applications and OSes used as tests, and acceptance suites used by individual makers. Note that “test” implies an acceptance or rejection criteria. Some

benchmarks have a checkable results file, but many do not. Nonetheless, “crashes” and “fails to complete” are understood to be a checkable failure. So while a benchmark may silently (uncheckably) compute wrong answers, some outcomes are easily recognized as failures. And although benchmarks are not designed for fault isolation, some failures are easy to diagnose and can expose bugs not found by any other test. Thus, it is valuable to at least triage these failures.

Several automatic instruction-level and system-level test generators are used for both VLIW and x86 testing, in addition to traditional hand-written tests and real workloads. Automated tests are used for both hardware and software development, and for both finding and diagnosing failures. Automated test generators produce tests which are complicated enough to find failures not found by simple tests, and are often simpler to diagnose than failures in real workloads. Once bugs are discovered, the test generator can be constrained to exclude or emphasize certain features in order to help find small and easy-to-debug test cases. Although “narrowing” is often time-consuming, it can be automated to a degree, and can proceed in parallel with diagnosis of test cases so far. In practice automated tests find both hardware and software bugs including tricky corner cases missed by hand-written tests.

Debugging short instruction sequences from test generators “should” be straightforward, but often “easy” bugs have already been found with hand-written directed tests, leaving difficult-to-debug race conditions and the like. In contrast, longer-running system tests less often find races and more often find state bugs that span multiple translations. State bugs can sometimes be reproduced in simulation making them easier to reproduce. Further, system tests can optionally pause occasionally and “settle” to help isolate failures, and can record their progress to simplify reconstruction of the state leading up to failure. That said, there were an unfortunate number of VLIW races and system tests sometimes found them.

A traditional problem with hand-written tests is the author “knows” certain cases are unimportant and so skips them and misses bugs. A traditional problem with pseudo-random test generators is they generate many “boring” cases for each interesting case. A technique used with good success is biased random number generation. For example, rather than testing a case with two arbitrary inputs, test with inputs whose bit pattern is *usually* mostly ones or mostly zeros. This allows the test generator author to provide guidance towards cases which are likely to be problems, but since the inputs are only guided, not firmly constrained, the random machinery still tests some cases the author thinks are “boring”.

At the time Crusoe was being developed, some industry-standard tests relied on a human reading a report screen.

Thus, automation was hard. One part of automation was developing tools to pattern match the video output. This sometimes yielded false failures, so video frames were kept for human review.

Hardware is always in high demand and limited supply. Even with processors in volume production, the in-house debug systems are in limited supply and often need rework. In-house farm hardware and software allow automated initialization of CMS, the BIOS, and the hard disk drive. This in turn allows any machine to be used for any purpose. A machine can be allocated, configured, and used for remote interactive debugging. Then, no matter what it’s state, it can, without human intervention, be reallocated and reconfigured for another use, such as batch testing, with any desired versions of CMS, BIOS, and disk image.

A small remote debugger stub is built in to CMS. On CMS boot, the stub queries a special debug connector and if “live”, downloads a larger debug stub before proceeding. The stub provides general access, and the debugger’s user interface is the same for simulation and hardware, as much as possible, so a developer’s experience is nearly the same for debugging on both simulator and hardware. With effort, even cosimulation of hardware against a reference simulator is possible, though the lack of reverse execution on hardware is a serious shortcoming!

Debugger code is integrated with CMS bootstrap code. On reset, vital pre-reset machine state is saved. Thus, even after hard crashes, much state at the time of the crash is available for postmortem analysis, and the debugger allows downloading of the crash-time state. Then, as debugging proceeds, one can go back to the postmortem dump and answer more detailed questions about the crash.

A static theorem prover, the “Static Program Analysis Machine” (SPAM), walks a user-supplied predicate over CMS machine code, ensuring the predicate is true. For example, it is usually an error to write a nonshadowed register, abort, then read the register, since it is nonshadowed and thus contains a value from the future. A predicate to find violations is `write X ; abort; read X` for all nonshadowed X. A complication is no translations exist statically, so there are paths with unknown code. The solution is to annotate entry and exit to dynamically-generated code, including exception paths, and claim behavior of the as-yet nonexistent code.

Performance work also uses many in-house tools. The fast simulator collects low-level metrics, and hardware has performance counters [CKB01]. Hardware is accurate, but has only a few counters and has run-to-run variations. Simulation is less accurate, but collects data as fine-grained as per-instruction, and is repeatable. Fine-grained data is yet another reason the fast simulator was used heavily even long after hardware was available.

Performance visualization includes traditional plots of

bulk statistics, but it was quickly apparent detailed behavior of individual dynamically-generated translations was significant. A set of tools instrument translator and translations to collect translation-level behavior and also save both target code and the generated translation. Instrumented code is usually almost as fast as uninstrumented code, yet saves enough detail a developer can start with a screen summarizing the whole execution, then quickly zoom in to much narrower “windows” of execution. Although statistical analysis of bulk data is often important for an overall understanding of “why is this slow”, actual diagnosis and repair often depends on data from individual translations.

5. Some Development Experiences

Preceding sections describe hardware and software. This section describes some development experiences, especially those different than might be expected in conventional projects.

Much of early software development is test-driven. That is, the goal is to implement enough to execute the first instruction of the first test. Once that works, development moves to the next instruction, even if the first instruction’s implementation is incomplete and may fail on other tests. This approach keeps development very focused. Productivity relies heavily on the reference simulator, reverse execution, and the nexus. In a conventional environment, a developer knows only a test has failed, and is responsible for analyzing why. The nexus, however, reports “register X diverged at tick Y” and stops, showing total state and allowing the developer to back up one instruction and “watch” the incorrect value being generated.

Hardware development is more traditional, but still has unusual features. For example, there were three proposals for how to implement a particular operation. All would hurt, though in different ways. We dropped the operation and did it all in software, as cheaper overall than any proposed hardware solution. A hardware developer smiled broadly – “This is the first project I have worked on where instructions have been *removed* during development!”

Keeping the fast VLIW simulator consistent with real hardware is a potential nightmare, especially since CMS is built around the fast simulator’s behavior. There could have been terrible problems, but in practice problems are few and minor. A big reason is the fast simulator and development environment are so useful, hardware verification also uses it for development, thus errors are corrected even before a suitable verification test exists or hardware implementation of that feature is started.

That said, some hardware features appeared in the fast simulator late or not at all. For example, the fast simulator implements enough bootstrap-specific behaviors that a CMS boot sequence runs on both simulator and hardware,

but it is also possible to write a correct VLIW boot sequence that does not run on the simulator. This “less than 100%” implementation requires some care, but avoids simulating tricky corner cases just for CMS boot, yet allows CMS boot code development in the simulation environment.

In many situations, CMS works around hardware bugs. For example, a pair of symmetrical hardware instructions were implemented asymmetrically. Until hardware was fixed, software simulated the faulty VLIW instruction. Although simulation was much slower, it allowed CMS to run on hardware, and reduced the “push” to finish the next tapeout as soon as possible – thus increasing the number of bugs that could be found, the number of fixes in the next tapeout, and thus the rate of progress of each tapeout.

Software is also used to work around timing bugs. For example, hardware uses a “cancel” signal shared by most functional units. A new functional unit was added without suitable timing analysis. If a unit at one end of the wire applied a signal, and the unit at the other end had an in-flight operation, the in-flight operation missed the signal and the operation was not cancelled. The static compiler and CMS translator were changed to avoid scheduling operations in the same VLIW instruction if they were at opposite ends of the wire.

Software changes are also used as a diagnostic tool: given a hypothesis “the problem is due to X combined with Y”, software can be changed to avoid that combination. A variety of diagnostic features were developed and controlled with flags. In addition, many control flags introduced during development were left in for diagnostic use. Thus, many diagnostic changes are as simple as changing a few flags. Diagnostic changes may spuriously cause a symptom to go away, but if a symptom continues, that shows a hypothesis false. Flags changes are also used by people with no CMS internals knowledge to work around bugs without needing to wait for a fix. Thus, while some bugs are still show-stoppers, other serious problems are discovered and worked around in just a few minutes, with diagnosis and repair separated from other work.

Another diagnostic tool is instrumenting CMS. One way to instrument is debugger scripts run at debugger breakpoints. Another is editing CMS. Some diagnostics are sufficiently useful they are checked in to the code base. Or, a “framework” may be checked in – for example, certain bytes of local memory are allocated for logging data about the last N exceptions. The trap code to do the logging is not checked in, as what to log varies, but reserving local memory ensures certain offsets are always the same.

There is always competition for what stays in the code base. On one hand, adding instrumentation is error-prone. For example, one failure we reduced to one of two signatures that occurred with roughly equal frequency. It turned out one signature was due to my errors in adding excep-

tion logging code. On the other hand, checked-in diagnostic code makes real code harder to understand, thus introducing bugs; and, diagnostics sometimes makes code slower. One compromise is a library of code that is not checked in the main code base, but which can be patched in at less effort and with fewer errors than writing it from scratch.

Hardware and software designs changed substantially during development. For example, an early design of alias hardware was difficult to use, even in the original situations for which it was devised. Thus, alias hardware was unused when hardware first became available. A different design was implemented in a later hardware revision, where it “met up” with the CMS implementation which was developed in simulation but until then was disabled when running on hardware.

Most FP instructions are invoked with a single TOS value, so early versions of Crusoe used a flat FP register file and simulated x87 indirection by putting TOS in the context. However, non-FP code is often invoked with many TOS values, which caused context mismatches and duplicate translations. For example, most kernel code is invoked with many TOS values. All shipping versions of Crusoe have a hardware TOS value that is added, mod 8, to the VLIW instruction’s register numbers to form the VLIW hardware’s register number. CR0’s TS (task switched) and EM (emulate math) bits are also checked by x87 FP instructions; early versions of Crusoe included these in the context, but shipping versions explicitly load and test the flags as needed.

Software changes were pervasive. Most of the world’s fast simulator experts worked at Transmeta during Crusoe development, but it was a learning experience for all. Much of CMS was implemented twice – which Brooks would say is a good thing [Bro75]. Some code that shipped after being implemented only once, had a comment saying it needed to be rewritten.

Some learning was about fast simulation; some about the x86 target; and some about how applications ran on that target. For example, the x86 has several segmentation modes. “Real” mode supports segments with byte-grain size, but they must be small. “Protected” mode supports large segments but the size must be a multiple of the page size. During Microsoft Windows boot, a CMS assertion tripped on a real-mode reference to an oversized segment. A check with the reference simulator showed the same reference happened there, yet the reference simulator ran Microsoft Windows. It turns out switching to real mode does not alter or invalidate sizes too large to have been established in real mode. Operating this way is sometimes called “unreal” mode [Aga91]. While not part of the ISA, all x86es support it.

Another education was how RAM and MMIO pages are interleaved. The simple model is RAM is low addresses,

and everything else is high addresses. However, a few pages of low memory are dedicated for legacy video, and a few more pages of low memory have “steering” bits, called A20M, that control reads and writes separately. A typical use is to copy a word from address X to address X – where the read comes from ROM and the write goes to RAM.

This sort of use brings up all sorts of issues. One is what happens when a single instruction spans a page boundary, part in RAM – and thus translatable – and part in ROM, so it must be fetched exactly once each time it is executed. Our initial conclusion was nobody would ever write code with one instruction in both RAM and MMIO, but doing so must not crash or corrupt CMS. CMS uses #n fetches in the translator and promotes normal-assumed instruction fetch faults to the interpreter. Interpreting is slow, but ROM fetches are even slower and are mandatory. Imagine our surprise to discover OS/2 boot has an instruction that executes from both RAM and video memory! Our guess is RAM was exhausted and somebody said “Why not execute a few instructions from video memory?” It could also have been an accident that was never noticed because it “just works”.

The gated store buffer implements memory store commit/abort. It has 32 entries, so it might seem CMS can use up to 32 stores between commits. However, a suitably misaligned store uses two store buffer entries. Further, a store may implicitly write page table “accessed” and “dirty” bits, while a page-crossing store can write accessed and dirty bits for two pages. (x86 80-bit floating-point stores can touch 3 pages; they are implemented with a pair of VLIW 16- and 80-bit stores.)

Stores that are misaligned or which change page table A and D bits are common enough translations are typically limited to around 28 VLIW stores, including stores in OOLs called by the translation. Thus, a few “unfortunate” stores do not cause overflow. Even so, store buffer overflow is not rare. Store buffer overflow is handled by retranslating with smaller translations and thus fewer stores between commits.

Transmeta’s goal is 100% x86 compatibility. In many cases there is a wide range of “legal” behaviors, and even “legal” differences risk incompatibility for a system that depends on a behavior that is not guaranteed but is nonetheless common.

One example of “legal but different” is page table “page accessed” and “page dirty” bits. Most x86 processors write them whether or not the instruction completes. Crusoe writes them using commit/abort logic, so they are written only if the instruction completes. Crusoe meets the specification, but is different. Would x86 code rely on this subtle difference? Crusoe could write them unconditionally, but at higher complexity (consider page tables in MMIO) and higher execution cost in the common case. We know of no case where this difference led to problems, but it does reflect details that, at every step, take time and effort to resolve.

Even where features agree individually with conventional x86es, it is hard to exactly emulate any one x86. Crusoe has a unique combination of feature flags. In principle, a system looks at each flag independently, so the combination is unimportant. In practice, Microsoft Windows does not examine flags individually, but instead looks for combinations seen in real processors, and defaults to a minimal x86 otherwise. Limiting feature combinations helps limit user's exposure to bugs. But it also means that since Crusoe was missing one feature, Microsoft Windows was not using any features until Crusoe systems were made available for Microsoft to test that specific combination.

Even where Crusoe is bit-for-bit compatible, the timing differs. Since there are many x86 implementations, most time-sensitive code uses wallclock time rather than instruction counts. One Linux Ethernet driver failed, however, because it self-timed a loop; the initial timing run was interpreted, and thus slow; and normal execution was translated, and thus fast. Similarly, translator pauses caused a soft modem to fail during real-time negotiation. The second try always succeeded, as the code was now compiled.

A problem reported with early dynamic translators is user-visible jerkiness during compilation. Although Crusoe also has compilation pauses, and although Crusoe's optimizing compiler is more expensive, the higher performance of modern processors – 1GHz vs. 10MHz – and Crusoe's interleaved interpretation makes the UI smooth, even when overall execution speed is slowed.

Overall execution speed is slowest when translation and interpretation rates are high and translation execution rates are low, usually OS and application startup. Tuning can improve worst-case performance. For example, interpreting more and translating less can avoid translation of low-reuse code. However, tuning for one case tends to hurt others. For example, one Microsoft Windows certification grade requires boot in a minute or less. Crusoe is tuned to boot in under a minute, even though the tuning hurts performance on some applications, including some important benchmarks.

“x86 compatible” includes both the instruction set and the PC platform. An early decision was to use the same RAM chips for both CMS and target memory. CMS bootstrap code configures memory for CMS, then CMS runs the BIOS, which configures memory. Thus, some memory configuration done by the BIOS needs to be skipped or ignored. The BIOS still has some memory “knobs” and is allowed to use those where doing so does not misconfigure CMS.

As described above, one development strategy is comparing CMS against a reference simulator. Another is comparing two CMSes, running with different configuration parameters. CMS on the fast VLIW simulator is faster than the reference simulator, and the nexus stops on any divergence between the two simulators, no matter which one made the

error. Thus, CMS vs. CMS runs a suite of configurations more quickly than running against the reference. It is, however, no substitute for execution against a reference simulator, as CMS vs. CMS misses bugs that appear in both.

Development was helped greatly by adding debugger features aggressively. Adding them as needed is too late! You want them already working and debugged when you are staring at an error, especially errors which are time-consuming to reproduce, or which appear erratically. For example, downloading cache contents and tags was added before it was needed. One hard-to-reproduce failure was positively identified as a hardware bug by dumping and examining cache tags, noting no sequence of valid instructions could give rise to the observed state, and noting the cache download code was simple and regular, thus unlikely to have bugs. This was a key to isolating the failure. We never saw the “must be hardware” corruption again, so had we added cache dumping only as needed, it might have taken weeks more to encounter another definitive case.

A few debugging stories may be of interest. One bug caused crashes roughly once a day per machine. A week of experimenting showed a corner case that failed within 5 minutes about 30% of the time. Crash analysis suggested the failure was related to an exception, but the failure case was during program start when exceptions were frequent. Unfortunately, we did not have application or OS sources, so could not readily study or instrument failing target code. We concluded it was a hardware bug, most likely in one of two units, but studying the circuits showed no errors. Further study showed a particular register had a value equal to another register – with no copy between them, but always a restarted chain pull when duplication occurred. Further hardware study showed the two units treated a shared bus signal differently: in some cases there was a one-cycle window where one unit was reading the signal, but the other was not driving it! There was only slow progress between initial discovery of the 5-minute test case and the final discovery of the mis-copied value. That discovery led within 60 minutes to positive identification of the bug's root cause.

Another error caused failures about 15 minutes before the end of a 24-hour stress test. Unfortunately, no “fast” case was discovered. Application and OS sources were not available. After about two weeks of study – about a dozen runs per machine – evidence pointed to CMS code that seemed common to all failures. Inspection revealed a bug where interrupts were supposed to be disabled across a sequence, but the disable was handed off between two interrupt disable mechanisms, and a bug reenabled interrupts between two instructions. A pending interrupt aborted execution to deliver the interrupt, leaving inconsistent state. Why did it fail only after 24 hours? The race was in a special-purpose code generator; the first time it ran was during OS boot when interrupts were also disabled at the x86 level;

after almost 24 hours, the translation was flushed and the code generator ran again, this time with interrupts enabled. A “zero-cycle” interrupt checker was added to SPAM.

A third bug appeared during testing of an isolated change. Ad-hoc testing repeatedly led to a hard crash during launch of an ordinary program. Study showed the failure was after a triple-nested nucleus fault that was not related to the change or application, but happened when starting the test in a certain time window after OS boot. Most triple faults abort the current translation. This case resumed the faulting instruction, and an error in triple-nested fault handling led to corrupted target state and a crash. Two people studying the failure found three distinct cut-and-paste errors in 20 lines of assembly that had been stable for months. One person found one error by stepping with the debugger and noting a value saved was different than the value restored – yes, the debugger can single-step through nested fault handlers. The other person inspected the code and found two more errors.

These last two bugs might have been found by inspection. Software inspection has been studied widely and consistently beats other strategies for finding bugs at the lowest cost. However, inspection is not used heavily at Transmeta, in part because it is not directly automatable. But despite the great success of automated testing, various “inspectable” bugs still survived long enough to demand arduous debugging. Programmers often resist classic inspection due to meetings and because it gates and thus delays checkin. However, inspection via e-mail is now common and has shown good results; and much as debug testing after checkin is common practice, inspection after checkin is also reasonable.

It is always fun to demonstrate virtualization. Shade [CK95] was once used to run nested Shades. For example, a SPARC running a Shade that was running Shade that was running Shade that was running Shade that was running a “Hello world” program (slowly). At Transmeta, various similar configurations were run using CMS and the fast VLIW simulator. For example, an x86 running the fast VLIW simulator running CMS running the fast VLIW simulator running CMS booting an OS (slowly). Or, real VLIW hardware running CMS running the fast simulator, running CMS, booting an OS (also slowly).

6. Conclusion

Transmeta’s Crusoe seamlessly runs nearly all workloads, and demonstrates a CPU based on dynamic translation is practical. Crusoe’s hardware is small and simple, using software for corner cases. This combination allowed a small team to produce a reliable x86-compatible CPU in just a few years. This paper briefly summarizes some of the designs, experiences, and lessons of development.

Acknowledgments

Thanks to Bob Cmelik and Stephen Russell for reviewing and improving earlier drafts of this paper and to Bob Cmelik for research assistance.

References

- [Aga91] R. K. Agarwal. 80x86 architecture & programming, Volume II: architecture reference. Prentice Hall, 1991.
- [AK04] H. P. Anvin, D. Keppel. Method and apparatus for handling nested faults. U.S. Pat. 6,829,719, issued 2004.
- [BAB+03] J. Banning, H. P. Anvin, R. Bedichek, G. Rozas, A. Shaw, L. Torvalds, J. Wilson. Translation consistency checking for modified target instructions by comparing to original copy. U.S. Pat. 6,594,821, issued 2003.
- [BAG+01] J. Banning, H. P. Anvin, B. Gribstad, D. Keppel, A. Klaiber, P. Serris. Fine grain translation discrimination. WIPO #WO 01/027743, issued 2001. Also as U.S. Pat. 6,363,336, issued 2002.
- [BCH03] J. Banning, B. Coon, E. Hao. Link pipe system for storage and retrieval of sequences of branch addresses. U.S. Pat. 6,640,297, issued 2003.
- [BCT+03] J. Banning, B. Coon, L. Torvalds, B. Choy, M. Wing, P. Gainer. Fast look-up of indirect branch destination in a dynamic translation system. 6,615,300, issued 2003.
- [BKK+03] P. Boyle, D. Keppel, A. Klaiber, E. Kelly. Software direct memory access. U.S. Pat. 6,668,287, issued 2003.
- [BKB05] R. Bedichek, D. Keppel, J. Banning. Interpage prologue to protect virtual address mappings. U.S. Pat. 6,845,353, issued 2005.
- [Bro75] F. Brooks. The mythical man-month. Addison-Wesley, 1975.
- [CDE+00] R. Cmelik, D. Ditzel, E. Kelly, C. Hunter, D. Laird, M. Wing, G. Zyner. Combining hardware and software to provide an improved microprocessor. U.S. Pat. 6,031,992, issued 2000.
- [Cla91] D. W. Clark. Large-scale hardware simulation: modeling and verification strategies. Chapter 9, “CMU computer science: a 25th anniversary commemorative”, R. Rashid, ed. ACM Press/Addison-Wesley, 1991, pp. 219-234
- [CK93] R. Cmelik, D. Keppel, Shade: a fast instruction-set simulator for execution profiling. Sun Microsystems Laboratories, Inc. TR SMLI 93-12; also University of Washington Department of Computer Science & Engineering TR 93-06-06.
- [CK94] R. Cmelik, D. Keppel, Shade: a fast instruction-set simulator for execution profiling. SIGMETRICS 1994.

- [CK95] R. Cmelik, D. Keppel, Shade: a fast instruction-set simulator for execution profiling. Chapter 1, "Fast simulation of computer architectures", T. Conte & C. Gimarcs eds. Kluwer, 1995.
- [CDS03] B. Coon, G. D'Souza, P. Serris. Pipeline replay support for multi-cycle operations wherein all VLIW instructions are flushed upon detection of a multi-cycle atom operation in a VLIW instruction. U.S. Pat. 6,604,188, issued 2003.
- [CKB01] B. Coon, D. Keppel, C. Price. Programmable event counter system. WIPO #WO 01/27873, issued 2001. Also as U.S. Pat. 6,356,615, issued 2002.
- [CK04] B. Coon, D. Keppel. Use of enable bits to control execution of selected instructions. U.S. Pat. 6,738,892, issued 2004.
- [DGB+03] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, J. Mattson. The Transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. CGO 2003, pg. 15-24.
- [DS84] L. P. Deutsch, A. M. Schiffman, Efficient implementation of the Smalltalk-80 system. POPL 1984, pg. 297-302.
- [KF03] M. Kanellos, M. J. Foley. Transmeta to help AMD push into servers. CNET News.com 2001/01/03.
- [KCW96] E. Kelly, R. Cmelik, M. Wing. Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed. U.S. Pat. 5,832,205, issued 1998.
- [KCW00] E. Kelly, R. Cmelik, M. Wing. Translated memory protection apparatus for an advanced microprocessor. U.S. Pat. 6,199,152, Issued March 2001.
- [KW99] E. Kelly, M. Wing. Host microprocessor with apparatus for temporarily holding target processor state. U.S. Pat. 5,958,061, issued 1999.
- [KCB01] D. Keppel, R. Cmelik, R. Bedichek. Method and apparatus for maintaining context while executing translated instructions. WIPO #WO 01/27741, issued 2001. Also as U.S. Pat. 6,415,379, issued 2002.
- [Kep09] D. Keppel. How to detect self-modifying code during instruction-set simulation. 2009 workshop on architectural and microarchitectural support for binary translation. Austin, TX.
- [KSD03] D. Keppel, P. Serris, G. D'Souza. Check instruction and method. U.S. Pat. 6,513,110, issued 2003.
- [KBK99] A. Klaiber, R. Bedichek, D. Keppel. Method and apparatus for correcting errors in computer systems. U.S. Pat. 5,905,855, issued 1999.
- [Kla00] A. Klaiber. The technology behind Crusoe processors, 2001/01. From archive.org as of 2009/03, for www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf as of 2001/06.
- [HAC+06] S. Halepete, H. P. Anvin, Z. Chen, G. D'Souza, M. Fleischmann, K. Klayman, T. Lawrence, A. Read. Adaptive Power Control. U.S. Pat. 7,100,061, issued 2006.
- [Loc87] B. Locanthi, Fast BitBlit with asm() and CPP. European Unix Users Group Conference, 1987/09.
- [May87] C. May. Mimic: a fast System/370 simulator. SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques. 1987/06.
- [RHK07] A. Read, S. Halepete, K. Klayman. Saving power when in or transitioning to a static mode of a processor. U.S. Pat. 7,260,731, issued 2007.
- [RDD+04] G. Rozas, D. Dunn, D. Dobrikin, A. Klaiber, D. Nelsen. Method and apparatus for emulating a floating point stack in a translation process. U.S. Pat. 6,725,361, issued 2004.
- [RDP06] G. Rozas, G. D'Souza, C. Price, P. Serris. Method and apparatus for enhancing scheduling in an advanced microprocessor U.S. Pat. 7,089,404, issued 2006.
- [Shi97] A. Shilov, "Transmeta quits microprocessor business", Xibt Laboratories, 2007/07/02, from <http://www.xbitlabs.com/news/cpu/display/20070207230938.html> as of 2009/05.
- [TA05] L. Torvalds, H. P. Anvin. Method of determining a mode of code generation. U.S. Pat. 6,880,152, issued 2005.
- [TK01] L. Torvalds, D. Keppel. Controlling instruction translation using dynamic feedback. WIPO #WO 01/27767 A1, issued 2001. Also as "System for using rate of exception event generation during execution of translated instructions to control optimization of the translated instruction". U.S. Pat. 6,714,904, issued 2004.
- [WD00] M. Wing, G. D'Souza. Gated store buffer for an advanced microprocessor. U.S. Pat. 6,011,908, issued 2000.
- [WK99] M. Wing, E. Kelly. Method and apparatus for aliasing memory data in an advanced microprocessor. U.S. Pat. 5,926,832, issued 1996.