Appeared in the Proceedings of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization,
27-29 March 2003, San Francisco, California

The Transmeta Code Morphing™ Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges

James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson,
Thomas Kistler, Alexander Klaiber, Jim Mattson

Transmeta Corporation, 3990 Freedom Circle, Santa Clara, CA 95054

Abstract

Transmeta's **Crusoe** *microprocessor is a full, system-level implementation of the x86 architecture, comprising a native VLIW microprocessor with a software layer, the* **Code Morphing Software** *(CMS), that combines an interpreter, dynamic binary translator, optimizer, and run-time system. In its general structure, CMS resembles other binary translation systems described in the literature, but it is unique in several respects. The wide range of PC workloads that CMS must handle gracefully in real-life operation, plus the need for full system-level x86 compatibility, expose several issues that have received little or no attention in previous literature, such as exceptions and interrupts, I/O, DMA, and self-modifying code. In this paper we discuss some of the challenges raised by these issues, and present the techniques developed in Crusoe and CMS to meet those challenges. The key to these solutions is the Crusoe paradigm of aggressive speculation, recovery to a consistent x86 state using unique hardware commit-and-rollback support, and adaptive retranslation when exceptions occur too often to be handled efficiently by interpretation.*

1 Introduction

Transmeta's Crusoe® VLIW processor and CMS [20] present an approach unique among commercial architectures: a microprocessor system with an internal VLIW instruction set architecture (*ISA*) with little resemblance to the external ISA (x86) that it presents to users. This approach allows a simple, compact, low-power microprocessor implementation, with the freedom to modify the internal ISA between generations, while supporting the broad range of legacy x86 software available. Producing robust runtime performance comparable to competing x86 implementations requires that CMS deal effectively with a number of difficult problems that have usually been ignored in the literature on binary translation and dynamic optimization.

In this paper, we will sketch the structure of CMS, but our focus will be on several of the challenges it faced that set it apart from other systems described in the literature, and on the solutions we implemented. These challenges are natural consequences of CMS objectives:

- CMS must faithfully implement the complete x86 architecture: all instructions (including memory-mapped I/O), architectural registers, and complete exception behavior.
- CMS can make no assumptions about the operating system running on the processor and cannot depend on information or other assistance from the system. It is a system-level implementation, not application-level, and even executes the BIOS code. One consequence is that it does not have access to the executable files of the applications it runs; all translation is done on-line as the target software executes.
- CMS must provide robust performance for a wide variety of systems and applications, ranging from games and media processing to desktop productivity and server applications. This requires dealing with unpleasant realities like self-modifying code and precise exceptions. It is important to note in this regard that CMS is not a migration tool – unlike past commercial systems, CMS is not an interim solution to be used during transition of the code base to a new architecture, and cannot deal with unusual but

The authors warmly acknowledge the numerous Transmeta engineers who designed and implemented the Crusoe Code Morphing Software and processor. This paper is based on their excellent work.

Email contacts: dehnert@transmeta.com, grant@transmeta.com, and rjohnson@transmeta.com.

important performance problems by waiting for the code in question to be converted.

Section 2 provides background on Crusoe processor features and CMS structure for the following discussion. Section 3 describes how CMS uses speculation, recovery, and adaptive retranslation to address a number of challenges of full-system, high-performance dynamic binary translation. Section 4 surveys related work.

2 Crusoe and CMS

The Crusoe processors have microarchitectures designed for simplicity by moving complex but infrequent tasks into the software. Although a full discussion of the architecture is beyond the scope of this paper, we provide some details here relevant to the following discussion.

The Crusoe TM5800 is a VLIW processor. Each instruction (called a *molecule*) can issue two or four RISC-like operations (called *atoms*) to a subset of five functional units: two ALUs, a memory unit, a floating point/media unit, and a branch unit. It has 64 general-purpose registers and 32 floating point registers, allowing the architectural x86 registers to be assigned to dedicated native VLIW registers, with an ample set available for use by CMS.

Transmeta VLIW hardware has very few hardware interlocks. CMS guarantees correct operation by careful scheduling, inserting no-ops if necessary. Only unpredictably long-latency operations such as loads that miss in the caches have their additional latency handled automatically by the hardware. Because CMS can be tailored to the processor, future generations of the hardware can change operation latencies, or other aspects of the native ISA or microarchitecture, without affecting the visible x86 architecture.

In fact, the current TM5000 family evolved significantly from the first TM3000 family Crusoe processors, adding atoms to more efficiently implement x86 segmentation, 16-bit operations, and indirect branches, all without a change in the target ISA. The next generation of Crusoe processors, the TM8000 family, will make further native ISA changes, including a complete re-design of the instruction formats; this will all be invisible to x86 code executing on the processor.

CMS is structured like many other dynamic translation systems. Initially, an interpreter decodes and executes x86 instructions sequentially, with careful attention to memory access ordering and precise reproduction of faults, while collecting data on execution frequency, branch directions, and memory-mapped I/O operations. When the number of executions of a section of x86 code reaches a certain threshold, its address is passed to the translator.

The translator selects a region including that address, produces native code to implement the x86 code from the region identified, and stores the translation with various related information in the *translation cache*. From then on, until something invalidates the translation cache entry, CMS executes the translation when the x86 flow of control reaches the translated x86 code region.

Initially, the exits of a translation branch to a lookup routine (the "no chain" path in Figure 1) that transfers control either to an existing translation for the next address or back to the interpreter. However, once the branch target is identified as another translation, the branch operation is modified to go directly there, a process called *chaining* (Cmelik et al. [9]). Over time, therefore, frequently executed regions of code begin to execute entirely within the translation cache, without overhead from interpretation, translation, or even branch-target lookup.

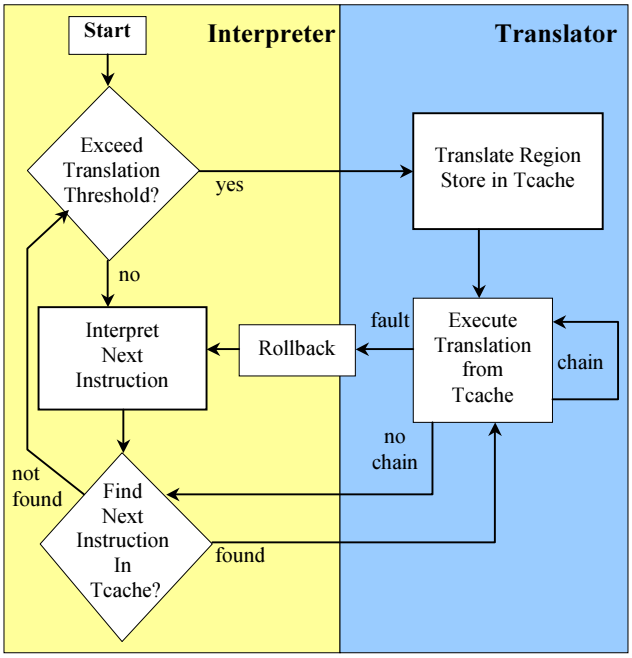


Figure 1: Typical CMS Control Flow

A variety of exceptional events may interrupt this typical control flow. This paper largely concerns the treatment of these cases, represented by the "fault" path in Figure 1.

The translator is the largest, most complex component of CMS. It comprises modules that decode x86 instructions, select a region for translation, analyze x86 data and control flow within the region, generate native VLIW code for the region, optimize it, and schedule it.

The choice of translation regions is beyond the scope of this paper, but they may be fairly large and complex, contain long traces, IF statements, and nested loops, and include up to 200 x86 instructions. This provides an extended scope for optimization. The optimizer performs a number of traditional and Crusoe-specific optimizations on the region, and schedules the final native VLIW code as a set of single-entry, multiple-exit traces. All of this is done with close attention to cost, since the translator can be a significant portion of execution time.

In addition to these components, CMS includes a runtime system to handle devices, interrupts and exceptions, power management, and garbage collection for the translation cache.

3 Speculation, Recovery, and Adaptive Retranslation

The requirement for CMS to deliver an utterly faithful yet high-performance implementation of a legacy commercial microprocessor architecture poses a significant challenge. A key paradigm that allows us to address many technical obstacles is that of speculation, recovery, and adaptive retranslation.

Speculation in this context refers to making and exploiting assumptions — unproven at translation time — about the code being translated. (For example, the translator might assume that two specific load and store instructions reference non-overlapping memory.) This type of speculation enables generation of much more efficient translations, but should one or more assumptions prove to be false, incorrect results may be produced. Hence, the assumptions must somehow be verified at runtime, with appropriate action taken when a violation is detected.

CMS uses a combination of hardware and software mechanisms to detect failing assumptions. These mechanisms trigger native exceptions that transfer control to handlers for the various modes of failure. The CMS response to failures is similar to the way it deals with normal execution. To address infrequent failures, CMS invokes the interpreter to deal with the condition. The interpreter, while much slower than executing translations, implements precise x86 semantics and guarantees correct machine state at every instruction boundary.

Because this solution has no up-front time or space cost, it works very well for the vast majority of translations, which never or seldom fail any speculative assumptions during their lifetimes. However, most varieties of speculation occasionally fail repeatedly in heavily executed translations, in which case the fault-and-interpret approach incurs unacceptable overhead. To cope gracefully with this eventuality, CMS monitors recurring failures and generates a more conservative translation when it deems the rate of failure to be excessive. To reduce the performance impact of conservative translations, CMS also attempts to confine the causes of failures to retranslations of smaller regions than the originals.

The Transmeta native VLIW processors provide hardware assistance for various kinds of speculation and subsequent recovery; we describe this mechanism in subsection 3.1. The subsequent subsections describe the challenges CMS meets by applying the procedure of speculation, recovery, and adaptive retranslation:

- CMS must faithfully reproduce the precise exception behavior of the x86 target, without overly constraining the scheduling of its translations.
- CMS must respond to interrupts at precise x86 instruction boundaries, where the system possesses a consistent target state.
- CMS must efficiently handle memory-mapped I/O and other system-level operations, without penalizing normal (non-I/O) memory references.
- Legacy PC software, especially games, often includes performance-critical self-modifying code. Similar problems result from pages containing both code and data, common in Windows/*9X* device drivers, BIOSs, and embedded systems running a real-time operating system such as QNX.

We present a variety of data in this section to illustrate these challenges. Some of it was collected on a TM5800 system, but in most cases the desired data could not be easily extracted from the hardware, and we used data from a Crusoe simulator that provides accurate dynamic molecule counts but not cycle accuracy. The simulation benchmark set includes boots of several Windows variants, DOS, Linux, and OS/2, and benchmarks from SPECcpu92 and SPECint2000, Windows productivity applications, and media applications (see Appendix A for a list). We will generally present selected or summarized data from this set.

Note that all of the issues we discuss in this paper occur in applications, although some (e.g., memory-mapped I/O) are much more common in system code.

3.1 Hardware Support for Speculation and Recovery

Compilers typically deal with recovery from speculation by generating compensation code, which re-executes incorrectly sequenced operations, performs operations omitted from the speculative code path, and corrects mismatches in register assignments (Freudenberger et al. [13]). With this approach, hardware support is required to defer faults of potentially faulting

instructions moved above branches (e.g., *boosting,* Smith et al. [23]), to detect overlapping memory operations scheduled out of sequence, and to branch to the compensation code (e.g., *memory conflict buffers,* Gallagher et al. [14], or the Intel IA-64 *ALAT* [18]).

In contrast, Crusoe native VLIW processors provide an elegant hardware solution that supports arbitrary kinds of speculation and subsequent recovery and works hand-in-hand with the Code Morphing Software [8]. All registers holding x86 state are *shadowed*; that is, there exist two copies of each register, a *working* copy and a *shadow* copy. Normal atoms only update the working copy of the register. If execution reaches the end of a translation, a special *commit* operation copies all working registers into their corresponding shadow registers, committing the work done in the translation. On the other hand, if any exceptional condition, such as the failure of one of CMS's translation assumptions, occurs inside the translation, the runtime system undoes the effects of all molecules executed since the last commit via a *rollback* operation that copies the shadow register values (committed at the end of the previous translation) back into the working registers.[1] Following a rollback, CMS usually interprets the x86 instructions corresponding to the faulting translation, executing them in the original program order, handling any special cases that are encountered, and invoking the x86 exception-handling procedure if necessary.

Commit and rollback also apply to memory operations. Store data are held in a *gated store buffer*, from which they are only released to the memory system at the time of a commit. On a rollback, stores not yet committed can simply be dropped from the store buffer. To speed the common case of no rollback, the mechanism was designed so that commit operations are effectively "free" [27], while rollback atoms cost less than a couple of branch mispredictions.

In the following subsections, we describe several ways in which CMS takes advantage of the commit/rollback mechanism.

3.2 Precise exceptions

Without special hardware support, it is difficult, if not impossible, for a dynamic translation system on a statically scheduled host to correctly model the exception semantics of the target ISA while at the same time achieving high performance. The primary reason is that exception semantics impose severe constraints on instruction scheduling. In the x86 ISA, exceptions are *precise*: when one instruction causes an exception, all instructions preceding it must complete before the exception is reported, and none of the subsequent instructions may complete. Solving this problem without special hardware support unduly constrains the scheduling of host instructions, and/or requires compensation code, either of which can reduce performance even in the common case where no exceptions occur. But with hardware support for commit and rollback and the interpreter-based recovery procedure in place, CMS has much more flexibility in scheduling the translated instructions. It can reorder potentially faulting atoms or hoist them above conditional branches, without the bookkeeping required by traditional control speculation, and without generating space-consuming compensation code.

The consequence of this approach, however, is that for a fault that should be reflected at the x86 level, CMS must determine whether the fault is genuine or whether it is a result of speculative instruction reordering. If the interpreter re-executes the instructions of the entire translation without encountering the fault, then it was speculative and, if it is infrequent, CMS can ignore it and continue normal execution.

The preferred strategy for dealing with a recurring fault depends on its class. For genuine x86 faults, we narrow the translation size around the faulting instruction. This reduces the amount of work that must be rolled back and/or interpreted, since the neighboring regions can remain large and highly optimized. We can ultimately run translations of all but the faulting instruction, which becomes a zero-instruction translation that simply calls the interpreter to execute the faulting instruction.

For frequently recurring speculative faults, we retranslate with more conservative policies that are likely to eliminate the sort of fault encountered, after first cutting the translation into smaller regions so that much of it can still benefit from aggressive translation. The new translation keeps track of the policies used, so that if another problem arises requiring different conservative policies, CMS will add them to the existing ones to avoid bouncing between translations with incomparable policies, neither of which solves both problems.

3.3 Interrupts

Commit and rollback serve a similar purpose with respect to interrupts. Because an interrupt causes a rollback to a consistent target state, translated code need not be concerned about interrupts in intermediate states that are not consistent with an x86 state between instructions. Interrupts do not trigger adaptive retranslation, since they are normally not directly related to the translation being executed when the interrupt is delivered.

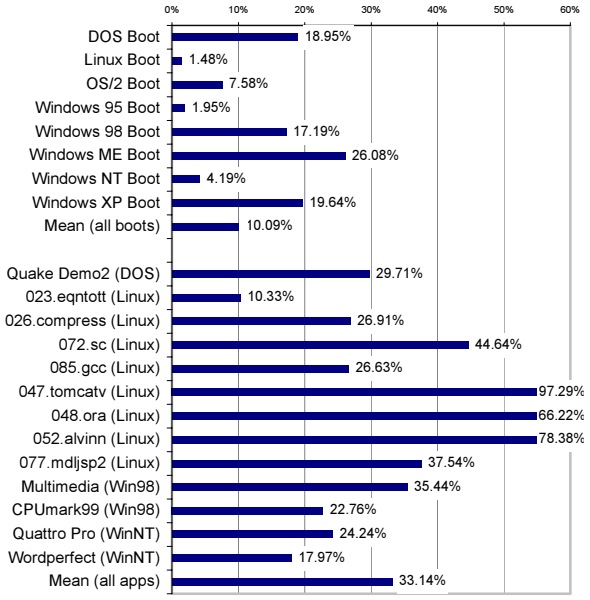
[1] Commit and rollback can equivalently be viewed as checkpoint and restart.

3.4 Memory-mapped I/O

The Crusoe system (processor and CMS) is designed to transparently run arbitrary code written for the x86 architecture, including both operating system and application software. Besides the obvious difficulty of accurately implementing the many corner cases of the x86 system-level architecture, CMS must also correctly implement low-level I/O interactions with physical devices. One of the most important rules associated with I/O transactions is that they must be performed in the original (x86) program order since they trigger irrevocable interactions with external devices.

In the x86 architecture, devices can be accessed via two different mechanisms: explicit I/O instructions ("in/out"), and memory-mapped accesses. The former are easily recognized and translated appropriately. Memory-mapped I/O, however, cannot be distinguished at translation time from regular memory accesses. In addition, a given x86 instruction can access both regular memory and I/O space over the course of program execution.

Figure 2: Degradation Caused by Suppressing Memory Reordering



Entirely suppressing memory reordering to solve this problem would be a severe handicap. To illustrate this point, we ran simulations of our benchmark suite with and without reordering of memory operations. Figure 2 above presents a representative subset of the results, along with the means for the entire set (see Appendix A). Several of the boots degraded by less than 5%, but the cost was as high as 26% in Windows/ME boot. The application degradation was much greater.

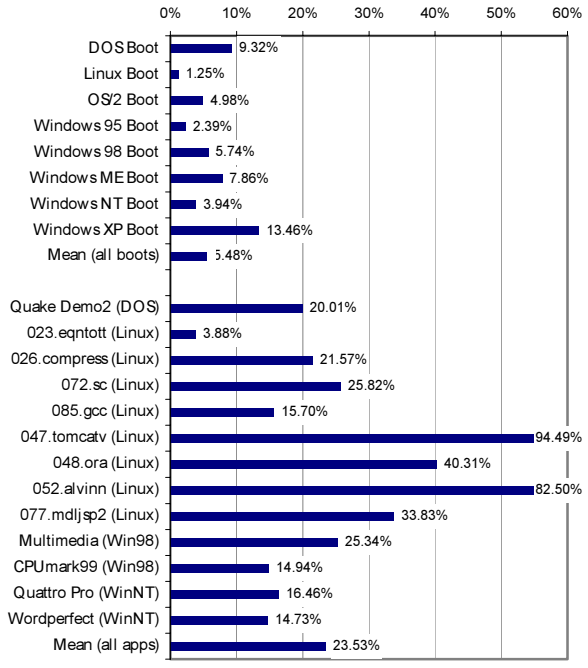
To solve the problem, load and store atoms on the Crusoe hardware specify whether they have been reordered with respect to the original x86 program. When such a speculative memory atom accesses a memory page that is mapped to I/O space, the hardware raises an exception [19]. At this point, CMS performs a rollback to the previously committed state and interprets. If the faults recur too often, CMS regenerates the translation, this time without reordering the offending memory reference.

3.5 Data speculation

Even for memory operations that access memory and not devices, it is common that the translator cannot prove that load and store addresses do not overlap; this also precludes reordering.

A key insight is that in practice, memory references rarely overlap if overlap is not obvious, so reordering is usually safe (and beneficial). Crusoe provides simple hardware support (the *alias hardware* [20]) that allows CMS to reorder selected memory references, with hardware taking on the burden of verifying at runtime that the reordered references did, in fact, not overlap. If hardware detects a violation, it raises an exception, and CMS may invoke rollback and conservative re-execution in the interpreter to handle the condition.

Figure 3: Degradation Caused By No Alias Hardware



Simulation data from our benchmark suite demonstrates the performance benefit of the alias hardware. Figure 3 above presents the performance degradation that results from not using the alias hardware, which is almost as severe as not reordering at all.

Crusoe's alias hardware is much simpler than that required to enforce memory constraints in an out-of-order processor. It is also simpler than other approaches suggested for VLIW processors, such as the memory conflict buffer [14] or IA-64 ALAT [18]. Those use fully associative tables with hardware mechanisms to determine which protected-address table entries to overwrite and to check for each out-of-order memory operation, whereas Crusoe requires the translator to explicitly specify this.

Recurring faults are handled by cutting the faulting translation into smaller regions and by scheduling any regions that still fault without speculative load/store reordering.

3.6 Self-modifying code (SMC)

Programs that modify themselves during execution can cause significant problems for any microarchitecture. For CMS, this manifests itself in the problem of keeping the translation cache consistent with its associated x86 code. Once again, CMS speculates, this time that the x86 code it translates does not change. In this case, both the detection of problems and adaptation techniques are of interest.

The original Transmeta approach to detecting SMC was to simply write-protect an x86 memory page whenever an x86 code fragment on that page was translated by CMS. If data on that protected page were later modified, either explicitly by the program or implicitly by system paging activity, then a fault occurred, and CMS would discard the affected translation(s).

Page-level protection is adequate for correctness, and critical to performance in the common case, but it does not efficiently handle self-modifying code. It also does not deal well with the sharing of code and writable data on the same page, if such occurs in performance-critical loops (e.g., for graphics processing in games). Although these are becoming less common in modern compiled applications, device drivers, games like Quake, embedded code, etc., use techniques such as assembly modules that intermix code and local (static) data. This is particularly common in BIOS and embedded software, which are subject to space constraints and often use assembly code extensively.

There are two costs incurred by SMC. The first is handling the fault when the page is written and invalidating translations associated with the page. The second is re-generating the translations the next time code on the page is executed.

The following subsections discuss three techniques for minimizing the cost of detecting writes to pages containing mixed code and data, and then two techniques for adapting to code that is actually self-modifying.

3.6.1 Fine-Grain Protection

The Crusoe processor provides hardware support for write-protecting memory at granularity finer than full pages [5]. The key insight is that finer granularity is only needed for a few pages at a time (e.g., the few pages that share writable code and data). As a result, only a few pages need to have fine-grain entries in a hardware cache, and a software fault handler can update the cache from memory on misses, allowing a small, simple hardware structure. The granularity supported cannot always identify a single translation affected, but typically narrows the impact to a few, reducing both the number of faults and the number of invalidated translations for each.

In order to avoid excessive processing for the common case of paging virtual memory, DMA writes to a protected page invalidate all translations for the page.

We simulated several benchmarks to demonstrate the benefit of fine-grain protection, comparing the number of protection faults with and without the fine-grain feature, and the overall slowdown in molecules executed per x86 instruction. The results are given in Table 1 below. The "faults" column gives the ratio of the number of protection faults without fine-grain support to the number with fine-grain support, and the "slowdown" column shows the impact on molecules executed per x86 instruction as a result.

Table 1: Slowdown Without Fine-Grain Protection

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

3.6.2 Self-Revalidating Translations

If CMS determines that a translation is encountering legitimate protection faults due to data stores in the same region as code, it can make the faults less expensive by adding a *prologue*, which is a code segment that is invoked just before a translation is entered. Prologues are generally used for temporary monitoring purposes, and allow easy installation and removal without disturbing the translation. Inserting a prologue involves removing any

existing chains to the translation, and replacing its start address by the prologue address.

Once a candidate translation for self-revalidation is identified, it is flagged. The next time it is encountered, it is re-translated in order to capture the translated x86 code (which is not preserved initially). Later, if the handler for a fine-grain protection fault determines that the translation(s) might be affected, it enables the prologue and turns off protection to avoid the cost of faulting again. When the translation is next invoked, the prologue verifies that the x86 code corresponding to the translation has not changed, re-enables protection, re-verifies the x86 code, disables the prologue, and then executes the translation.

This technique does not eliminate protection faults due to writes. But it executes the fault handler and checks at most once per write to the protected area, and at most once per execution of the translation, so it can be quite efficient if the writes are much less frequent than executions of the affected translations. After retranslation to capture the x86 code to be checked, the translation executes at normal optimized speed unless there are writes to the protected area. As an example of the benefit, the Quake Demo2 benchmark achieves a 28% higher frame rate with self-revalidation than without it.

If the protection faults do happen frequently, the overhead of the fault handler and the checks is significant, since a revalidation is likely to be at least as expensive as executing the translation. Further, this technique does not work if it is the translation itself that is writing on its associated x86 region, since the write occurs after the checking prologue has completed, causing a new fault and preventing forward progress. For such cases, the following technique for optimizing fault detection may work better.

3.6.3 Self-Checking Translations

Instead of protecting the x86 page when creating a translation, it is possible to leave the memory page unprotected, and have the translation itself check that the source x86 bytes have not changed, by fetching them and comparing them to their values when the translation was created.

We can merge the checking code into the normal translation code, since if it detects a mismatch we can rollback any translation effects that have already occurred. There are scheduling constraints that must be observed for the inserted checking code. The fetches for checking an x86 operation must appear logically after any stores up to and including the operation being checked, and on the same control flow path as the operation being checked. However, fetching for self-checking can be reordered relative to stores using the alias hardware, as outlined in section 3.5. Hence, the overhead of self-checking a translation once is many times smaller than that of self-revalidating it once, although its average cost may be much higher if the translation is executed many times between protection faults.

To evaluate the typical cost of self-checking translations, we ran simulations of our benchmark suite normally, and with all translations forced to be self-checking. Self-checking adds a mean of 83% to the code size (ranging from 58% to 100%), and a mean of 51% to the molecules executed (ranging from 11% to 124%); because of cache effects, the actual runtime impact would be higher.

Although self-checking translations are less expensive than interpretation or re-translation, we can see from the data above that their overhead is still significant, especially for long translations (in absolute cost). Therefore, even if we can use this technique to eliminate unnecessary self-modification failures, we first attempt to adapt by producing smaller translations so that a minimum of code must be checked.

3.6.4 Stylized SMC

The above techniques are helpful only if the code is not actually changing, i.e. if the protection faults result from data being written in the same page as code. The last two techniques described here are methods of adapting to genuinely self-modifying code

Many PC applications that rely on self-modifying code do so in very stylized ways. A common approach, for example, is to modify the immediate or offset fields in instructions inside an inner loop, just before entering that loop.[2] It is possible to avoid continual retranslation in this special case by translating the original x86 code in such a way that the translation loads (at runtime) the immediate fields in question from the code stream. Consider the x86 instruction:

```
label: add  %eax, 0x123456
```

This can be translated into Crusoe code

```
ld  %temp,[label+1]
add  %eax, temp
```

This translation is valid regardless of how the "`0x123456`" immediate field is modified by the application. Note, however, that this technique must be used in conjunction with self-checking or self-revalidation, to verify that instruction fields other than the immediate operands have not been modified.

[2] This approach seems to be particularly popular on register-poor machines such as the x86, when there are loop-invariant constants but no registers to hold them throughout the loop. The game Doom uses it in critical loops, for instance, and it also occurs in current applications such as Adobe Premiere. However, it rarely appears in portable, compiled benchmarks like SPEC.

3.6.5 Translation Groups

Sometimes self-modifying code repeatedly writes and executes one of a small number of versions of the rewritten x86 code. For example, the device-independent BLT driver in Windows/*9X* uses up to 33 versions in benchmarks we have checked, with the version depending on the operation to be performed and the graphics chip's hardware capabilities. In such cases, it is desirable to have the old translation available when an old version reappears. CMS keeps such translations in *translation groups*. These are lists of translations of the same x86 code region, with the currently active translation first on the list. If the first translation fails its self-check after a protection fault, the others are checked for a current match with the x86 code before a new translation is produced, and any matching translation found becomes the current one.

As a result of these techniques, in cases encountered in practice, CMS robustly obtains good performance for both self-modifying code and mixed code and data.

4 Related Work

CMS is most closely related to the emulation, binary translation, and dynamic optimization literature, which has a long history. In the comments that follow, we focus on software emulation systems, although some may have hardware features to facilitate emulation.

We classify software emulation systems as *interpreters* (instruction-at-a-time with no memory), *static translators* (offline), and *dynamic translators* (online). (This is the classification of Altman et al. [2], which uses "emulator" instead of "interpreter.") CMS includes both an interpreter and a dynamic translator (which we call simply the *translator*). We call the emulated architecture the *target*, and the emulating architecture the *host*.

Many emulation systems are *self-hosting*, that is the host and target architectures are the same. Such systems are generally created for purposes of optimization or instrumentation. A well-known recent dynamic optimization system is Dynamo from HP Labs (Bala et al. [3,4]) and its successor DELI [10]. Dynamo's high-level architecture is similar to that of CMS, but it can fall back on efficient native execution, so there is no need to attempt translation for code that is problematic, or just cannot be improved. For this reason, the tradeoffs of self-hosting systems are quite different from systems like CMS.

Another rich area of research has been *virtual target* emulators, where the target architecture is a specially designed virtual machine rather than a physical architecture. One interesting example comes from the IBM migration of the AS/400 system to the PowerPC architecture, which was based on a static translation of an abstract machine code included by the AS/400 compilers in application object code (Soltis [24]). Java virtual machines are a much better known example. They emulate an abstract byte code designed specifically to be efficiently interpreted on a wide variety of machines [29]. From the early interpreter-only systems, these emulators have developed into sophisticated dynamic optimizers, such as Sun's HotSpot [26], IBM's Jalapeño (Burke et al. [6]), and LaTTe (Yang et al. [28]). These systems have a great deal in common with CMS, including tradeoffs between translation cost and code quality. But the virtual machine semantics are tightly controlled, avoiding most of the problems we have discussed in this paper.

Cross-hosted emulators, emulating a target architecture on a different host, must deal more completely with the full variety of target code. A common purpose is to move code from the target architecture to the host architecture, usually to facilitate customer migration from an older architecture to a newer one intended to replace it. Examples are DEC's migration tools from DEC VAX/VMS to Alpha/OpenVMS (VEST) and MIPS/Ultrix to Alpha/OSF1 (mx, see [SCKMR92]). Performance degradation is undesirable for these systems, but they usually benefit from hosts that are significantly faster than the target, and problematic cases can be ported to native code on the new architecture. The VEST project's objectives, for instance, explicitly allowed for rejecting some target code, with diagnostic information to guide manual intervention. A later Alpha migration project is FX!32, for running Windows NT x86 code on Windows NT Alpha (Chernoff et al. [7]). FX!32 uses an interpreter with a static translator that is triggered by interpreting target code but runs offline and preserves its translations in a database. It is not a perfect emulator of the x86, for instance doing 64-bit instead of 80-bit floating point, and not supporting the Windows NT Debug API because it cannot rematerialize the x86 state at arbitrary points.

A more recent commercial project is HP's Aries, for migration from HP-PA to IA-64 (Zhang et al. [30]). It features an interpreter and dynamic translator architecture more akin to CMS. However, it does only single-block translations, and weaknesses like keeping target floating-point register images in memory likely cause severe performance problems on significant classes of code. The reference provides no performance information, and to our knowledge the project has not been completed.

Another class of migration tools is those intended by one vendor to capture applications created for another vendor's architecture. An early example is Hunter System's XDOS x86 DOS emulator [17], a static translator. The emulated software was not intended to be the primary use of the target systems, so performance requirements were modest. Difficult applications could

be handled by special-case modifications, and translation often required significant manual intervention.

All of the above systems have escape valves not available to CMS, which must seamlessly execute any x86 software, and must provide performance competitive with hardware-only x86 microprocessors that continue to improve. The most important distinction is that they are all application-level emulators that do not address system code, instead redirecting system calls to similar system calls on the host. However, CMS does not need to emulate device behavior, since its host I/O subsystem is the same as the target.

The system with objectives and constraints closest to CMS is DAISY from IBM Research (Ebcioglu et al. [11,12,21]). DAISY is a full-system implementation of a PowerPC or System/390 target on a tree VLIW host, with an interpreter and dynamic translator architecture similar to CMS. Its translation region selection is different (tree regions vs. more general code segments in CMS), it uses a state-repair approach to handle precise exceptions (Gschwind et al. [15]) rather than the commit and rollback approach of CMS, and it uses only a fine-grain protection approach to self-modifying code. The references do not discuss our other challenges.

There have been many other binary translation systems. More extensive prior work discussions may be found in Altman et al. [2,12].

5 Conclusions

CMS is a commercially available system that provides a high-performance, fully compatible implementation of the x86 ISA on a different host ISA (the Crusoe native VLIW). CMS is similar in overall architecture to a number of binary translation systems described in the literature, but a key to its success is attention to challenges such as those described in this paper. The paradigm of speculating aggressively, rolling back to a consistent state for recovery when exceptions are detected, and adaptively retranslating to deal with recurring exceptions is a powerful part of the CMS solution. These challenges do not become apparent until one attempts to run a wide variety of everyday workloads, yet it is dealing with them that makes CMS a robustly performing product instead of an experimental system.

References

[1] Erik R. Altman, Kemal Ebcioglu, Michael Gschwind, and Sumedh Sathaye, "*Advances and Future Challenges in Binary Translation and Optimization*," Proc. of the IEEE, Special Issue on Microprocessor Architecture and Compiler Technology, Nov. 2001, pp. 1710-1722.

[2] Erik R. Altman, David Kaeli, and Yaron Sheffer, "*Welcome to the Opportunities of Binary Translation*," IEEE Computer 33 (3), March 2000, pp. 40-45.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia, "*Transparent Dynamic Optimization: The Design and Implementation of Dynamo*," Tech. Report HPL-1999-78, HP Laboratories Cambridge, June 1999.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia, "DYNAMO: A *Transparent Dynamic Optimization System*," PLDI, June 2000, pp. 1-12.

[5] John Banning, Peter H. Anvin, Benjamin Gribstad, David Keppel, Alex Klaiber, and Paul Serris, "*Fine grain translation discrimination*," US Patent 6,363,336, 26 March 2002.

[6] Michael G. Burke et al., "*The Jalapeno Dynamic Optimizing Compiler for Java*," ACM 1999 Java Grande Conference, June 1999, pp. 129-141.

[7] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, B. Yadavalli, and J. Yates, "*FX!32: A Profile-Directed Binary Translator*," IEEE Micro 18 (2), March/April 1998, pp. 56-64.

[8] Robert F. Cmelik, David R. Ditzel, Edmund J. Kelly, Colin B. Hunter, et al, "Combining Hardware and Software to Provide an Improved Microprocessor," US Patent 6,031,992, Feb. 2000.

[9] Robert F. Cmelik and David Keppel, "*Shade: A Fast Instruction-set Simulator for Execution Profiling*," Proc. Sigmetrics Conf. on Measurement and Modeling of Computer Systems, 1994, pp. 128-137.

[10] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher, "*DELI: A New Run-time Control Point*," Proc. of MICRO-35, Nov. 2002.

[11] Kemal Ebcioglu and Erik R. Altman, "*DAISY: Dynamic Compilation for 100% Architectural Compatibility*," Proc. of the 24th Annual Int'l Symp. on Computer Architecture, June 1997, pp. 26-37.

[12] Kemal Ebcioglu, Erik R. Altman, Michael Gschwind, and Sumedh Sathaye, "*Dynamic Binary Translation and Optimization*," IEEE Trans. on Computers 50 (6), June 2001, pp. 529-548.

[13] Stefan M. Freudenberger, Thomas R. Gross, and P. Geoffrey Lowney, "*Avoidance and Suppression of Compensation Code in a Trace Scheduling Compiler*," ACM Trans. On Programming Languages and Systems 16 (4), July 1994, pp. 1156-1214.

[14] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen-mei W. Hwu, "*Dynamic Memory Disambiguation Using the Memory Conflict Buffer*," Proc. Sixth Int'l Conf. on ASPLOS, October 1994, pp. 183-193.

[15] Michael Gschwind and Erik R. Altman, "*Precise Exception Semantics in Dynamic Compilation*," Proc. 2002 Symp. On Compiler Construction, April 2002, pp. 95-110.

[16] Michael Gschwind, Erik R. Altman, Sumedh Sathaye, Paul Ledak, and David Appenzeller, "*Dynamic and Binary Translation*," IEEE Computer 33 (3), March 2000, pp. 54-59.

[17] Colin Hunter and John Banning, "*DOS at RISC*," Byte Magazine, Nov. 1989, pp. 361-368.

[18] Intel Corp., **Intel IA-64 Architecture Software Developer's Manual**, vol 1, Order #245317-001, January 2000.

[19] Edmund J. Kelly, Robert F. Cmelik, and Malcolm J. Wing, "Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed," US Patent 5,832,205, Nov. 1998.

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

[21] Gabriel M. Silberman and Kemal Ebcioglu, "*An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures*," IEEE Computer 26 (6), June 1993, pp. 39-56.

[22] R. Sites, A. Chernoff, Kirk, M. Marks, and S. Robinson, "*Binary Translation*," Comm. ACM 36 (2), Feb. 1993, pp. 69-81.

[23] Michael D. Smith, Mark Horowitz, and Monica S. Lam, "*Efficient Superscalar Performance Through Boosting*," Proc. 5th Int'l Conf. on ASPLOS, October 1992.

[24] Frank G. Soltis, **Inside the AS/400**, Duke Press, 1997.

[25] Standard Performance Evaluation Corp., "*SPEC OSG Frequently Asked Questions*," http://www.specbench.org/osg/faq/archive.

[26] Sun Microsystems, "*The Java Hotspot Performance Engine Architecture*," http://java.sun.com/products/hotspot/whitepaper.html, April 1999.

[27] Malcolm J. Wing and Godfrey P. D'Souza, "*Gated store buffer for an advanced microprocessor*," US Patent 6,011,908, Jan. 2000.

[28] Byung-Sun Yang et al, "*LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation*," Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, Oct. 1999, pp. 128-138.

[29] Frank Yellin and Tim Lindholm, The Java Virtual Machine Specification, Addison-Wesley, 1996.

[30] Cindy Zheng and Carol Thompson, "*PA-RISC to IA-64: Transparent Execution, No Recompilation*," IEEE Computer 33 (3), March 2000, pp. 47-52.

A. Benchmarks

The benchmarks used are:

- OS boots of DOS, Linux, OS/2, Windows95, Windows98, WindowsME, WindowsNT, and WindowsXP.
- Linux and/or Windows98 SPECcpu92: alvinn, compress, eqntott, espresso, gcc, li, mdljdp2, mdljsp2, ora, sc, spice2g6, su2cor, tomcatv, wave5.
- Windows98 SPECint2000: crafty.
- Windows98 and/or WindowsNT Winstone98: Access, Corel, Navigator, PowerPoint, QuattroPro, WordPerfect.
- Miscellaneous: MultimediaMark99, CpuMark99, Quake Demo2, WindowsME help.

Data from all of these are included in the mean values given in Tables 1 and 2.

Keywords

Binary translation, dynamic translation, dynamic optimization, emulation, speculation, self-modifying code