

THE TECHNOLOGY BEHIND CRUSOE™ PROCESSORS

LOW-POWER X86-COMPATIBLE PROCESSORS
IMPLEMENTED WITH CODE MORPHING™ SOFTWARE

ALEXANDER KLAIBER
TRANSMETA CORPORATION

JANUARY 2000

Property of:
Transmeta Corporation
3940 Freedom Circle
Santa Clara, CA 95054 USA
(408) 919-3000
<http://www.transmeta.com>

The information contained in this document is provided solely for use in connection with Transmeta products, and Transmeta reserves all rights in and to such information and the products discussed herein. This document should not be construed as transferring or granting a license to any intellectual property rights, whether express, implied, arising through estoppel or otherwise. Except as may be agreed in writing by Transmeta, all Transmeta products are provided "as is" and without a warranty of any kind, and Transmeta hereby disclaims all warranties, express or implied, relating to Transmeta's products, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose and non-infringement of third party intellectual property. Transmeta products may contain design defects or errors which may cause the products to deviate from published specifications, and Transmeta documents may contain inaccurate information. Transmeta makes no representations or warranties with respect to the accuracy or completeness of the information contained in this document, and Transmeta reserves the right to change product descriptions and product specifications at any time, without notice.

Transmeta products have not been designed, tested, or manufactured for use in any application where failure, malfunction, or inaccuracy carries a risk of death, bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft, watercraft or automobile navigation or communication, emergency systems, or other applications with a similar degree of potential hazard.

Transmeta reserves the right to discontinue any product or product document at any time without notice, or to change any feature or function of any Transmeta product or product document at any time without notice.

Trademarks: Transmeta, the Transmeta logo, Crusoe, the Crusoe logo, Code Morphing, LongRun and combinations thereof are trademarks of Transmeta Corporation in the USA and other countries. Other product names and brands used in this document are for identification purposes only, and are the property of their respective owners.

This document contains confidential and proprietary information of Transmeta Corporation. It is not to be disclosed or used except in accordance with applicable agreements. This copyright notice does not evidence any actual or intended publication of such document.

Copyright © 2000, Transmeta Corporation. All rights reserved.

SUMMARY

In January of 2000, Transmeta Corporation introduced the Crusoe™ processors, an x86-compatible family of solutions that combines strong performance with remarkably low power consumption. As might be expected, a new technology for designing and implementing microprocessors underlies the development of these products. As might not be expected, the new technology is fundamentally software-based: the power savings come from replacing large numbers of transistors with *software*.

The Crusoe processor solutions consist of a hardware engine logically surrounded by a software layer. The engine is a very long instruction word (VLIW) CPU capable of executing up to four operations in each clock cycle. The VLIW's native instruction set bears no resemblance to the x86 instruction set; it has been designed purely for fast low-power implementation using conventional CMOS fabrication. The surrounding software layer gives x86 programs the impression that they are running on x86 hardware. The software layer is called Code Morphing™ software because it dynamically “morphs” x86 instructions into VLIW instructions. The Code Morphing software includes a number of advanced features to achieve good system-level performance. Code Morphing support facilities are also built into the underlying CPUs. In other words, the Transmeta designers have judiciously rendered some functions in hardware and some in software, according to the product design goals and constraints. Different goals and constraints in future products may result in different hardware-software partitioning.

Transmeta's Code Morphing technology changes the entire approach to designing microprocessors. By demonstrating that practical microprocessors can be implemented as hardware-software hybrids, Transmeta has dramatically expanded the design space that microprocessor designers can explore for optimum solutions. Microprocessor development teams may now enlist software experts and expertise, working largely in parallel with hardware engineers to bring products to market faster. Upgrades to the software portion of a microprocessor can be rolled out independently from the chip. Finally, decoupling the hardware design from the system and application software that use it frees hardware designers to evolve and eventually replace their designs without perturbing legacy software.

TECHNOLOGY PERSPECTIVE

The Transmeta designers have decoupled the x86 instruction set architecture (ISA) from the underlying processor hardware, which allows this hardware to be very different from a conventional x86 implementation. For the same reason, the underlying hardware can be changed radically without affecting legacy x86 software: each new CPU design only requires a new version of the Code Morphing software to translate x86 instructions to the new CPU's native instruction set.

For the initial Transmeta products, models TM3120 and TM5400, the hardware designers opted for minimal space and power. By eliminating roughly three quarters of the logic transistors that would be

required for an all-hardware design of similar performance, the designers have likewise reduced power requirements and die size. However, future hardware designs can emphasize different factors and accordingly use different implementation techniques.

Finally, the Code Morphing software itself offers opportunities to improve performance without altering the underlying hardware. The current system is a first-generation embodiment of a new technology that can be further optimized with experience and experimentation. Because the Code Morphing software would typically reside in standard Flash ROMs on the motherboard, improved versions can even be downloaded into processors in the field.

CRUSOE PROCESSOR FUNDAMENTALS

With the Code Morphing software handling x86 compatibility, Transmeta hardware designers created a very simple, high-performance, VLIW engine with two integer units, a floating point unit, a memory (load/store) unit, and a branch unit. A Crusoe processor long instruction word, called a *molecule*, can be 64 bits or 128 bits long and contain up to four RISC-like instructions, called *atoms*. All atoms within a molecule are executed in parallel, and the molecule format directly determines how atoms get routed to functional units; this greatly simplifies the decode and dispatch hardware. Figure 1 shows a sample 128-bit molecule and the straightforward mapping from atom slots to functional units. Molecules are executed in order, so there is no complex out-of-order hardware. To keep the processor running at full speed, molecules are packed as fully as possible with atoms. In a later section, we describe how the Code Morphing software accomplishes this.

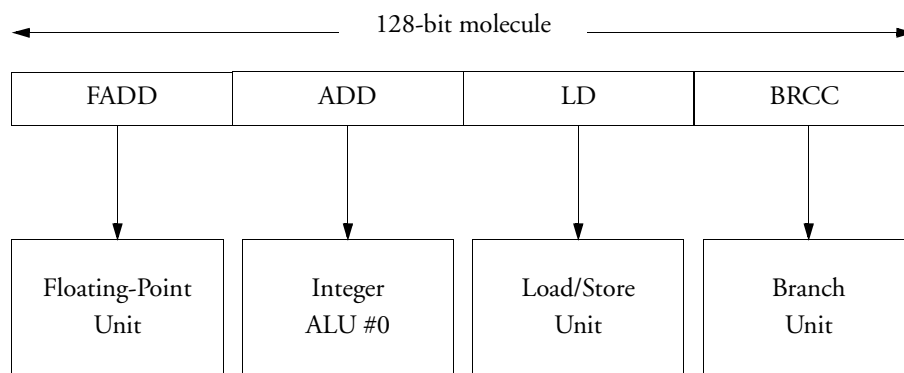


Figure 1. A molecule can contain up to four atoms, which are executed in parallel.

The integer register file has 64 registers, %r0 through %r63. By convention, the Code Morphing software allocates some of these to hold x86 state while others contain state internal to the system, or can be used as temporary registers, e.g., for register renaming in software. In the assembly code examples in this paper,

we write one molecule per line, with atoms separated by semicolons. The destination register of an atom is specified first; a “.c” opcode suffix designates an operation that sets the condition codes. Where a register holds x86 state, we use the x86 name for that register (e.g., %eax instead of the less descriptive %r0).

Superscalar out-of-order x86 processors, such as the Pentium II and Pentium III processors, also have multiple functional units that can execute RISC-like operations (micro-ops) in parallel. Figure 2 depicts the hardware these designs use to translate x86 instructions into micro-ops and schedule (dispatch) the micro-ops to make best use of the functional units. Since the dispatch unit reorders the micro-ops as required to keep the functional units busy, a separate piece of hardware, the in-order retire unit, is needed to effectively reconstruct the order of the original x86 instructions, and ensure that they take effect in proper order. Clearly, this type of processor hardware is much more complex than the Crusoe processor's simple VLIW engine.

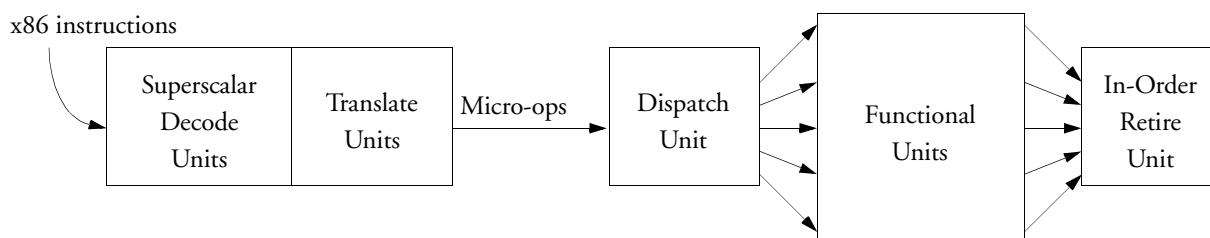


Figure 2. Conventional superscalar out-of-order CPUs use hardware to create and dispatch micro-ops that can execute in parallel.

Because the x86 instruction set is quite complex, the decoding and dispatching hardware requires large quantities of power-hungry logic transistors; the chip dissipates heat in rough proportion to their numbers. Table 1 compares the sizes of Intel mobile and Crusoe processor models.

	Mobile PII	Mobile PII	Mobile PIII	TM3120	TM5400
Process	.25m	.25m shrink	.18m	.22m	.18m
On-chip L1 Cache	32KB	32KB	32KB	96KB	128KB
On-chip L2 Cache	0	256KB	256KB	0	256KB
Die Size	130mm ²	180mm ²	106mm ²	77mm ²	73mm ²

Table 1. The Code Morphing software simplifies chip hardware.

Viewing power dissipation as heat, Figure 3 and Figure 4 contrast the operating temperatures of a Pentium III and a Crusoe processor running a software DVD (Digital Versatile Disk) player. The model TM5400 requires no active cooling, whereas the Pentium III processor can heat to the point of failure if it is not aggressively cooled.

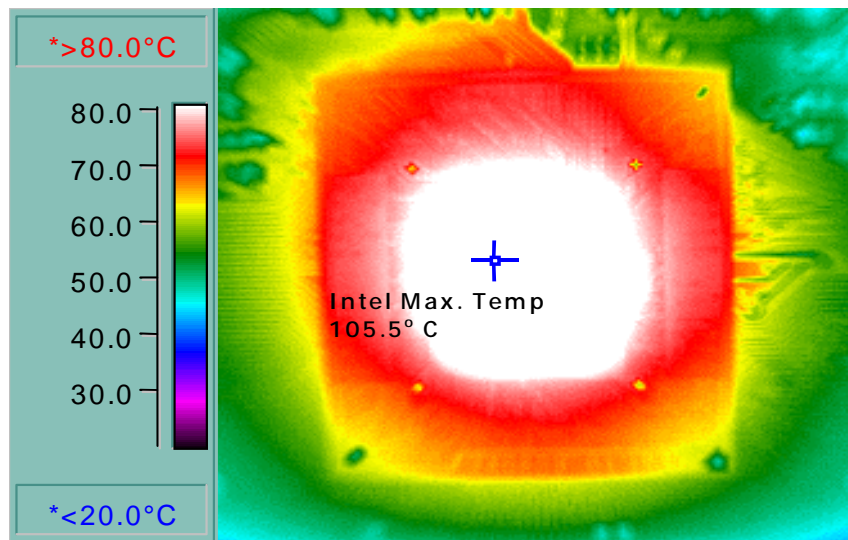


Figure 3. A Pentium III processor plays a DVD at 105° C (221° F).

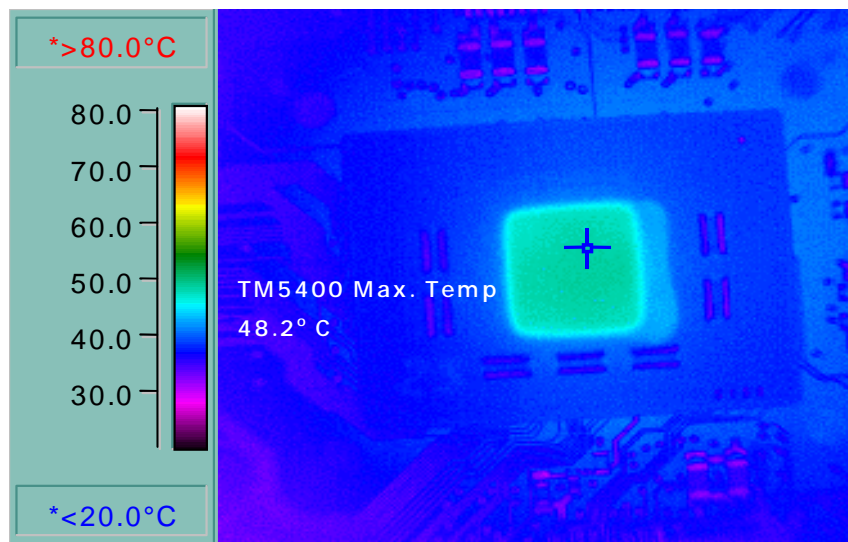


Figure 4. A Crusoe processor model TM5400 plays a DVD at 48° C (118° F).

THE CODE MORPHING SOFTWARE

The Code Morphing software is fundamentally a dynamic translation system, a program that compiles instructions for one instruction set architecture (in this case, the x86 *target* ISA) into instructions for another ISA (the VLIW *host* ISA). The Code Morphing software resides in a ROM and is the first program to start executing when the processor boots. The Code Morphing Software supports ISA, and is the only thing x86 code sees;; the only program written directly for the VLIW engine is the Code Morphing software itself. Figure 5 shows the relationship between x86 code, the Code Morphing software, and a Crusoe processor.

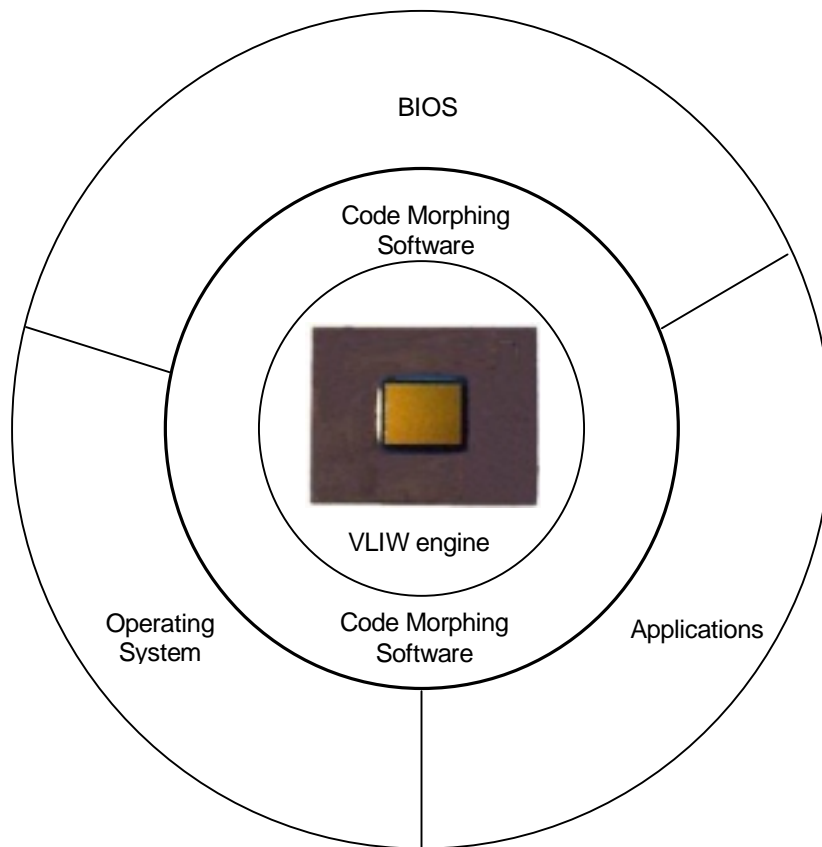


Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.

Because the Code Morphing software insulates x86 programs—including a PC's BIOS and operating system—from the hardware engine's native instruction set, that native instruction set can be changed arbitrarily without affecting any x86 software at all. The only program that needs to be ported is the Code Morphing software itself, and that work is done once for each architectural change, by Transmeta. The feasibility of this concept has already been demonstrated: the native ISA of the model TM5400 is an

enhancement (neither forward nor backward compatible) of the model TM3120's ISA and therefore runs a different version of Code Morphing software. The processors are different because they are aimed at different segments of the mobile market: the model TM3120 is aimed at Internet appliances and ultra-light mobile PCs, while the model TM5400 supports high-performance, full-featured 3-4lb. mobile PCs.

Coincidentally, hiding the chip's ISA behind a software layer also avoids a problem that has in the past hampered the acceptance of VLIW machines. A traditional VLIW exposes details of the processor pipeline to the compiler, hence any change to that pipeline would require all existing binaries to be recompiled to make them run on the new hardware. Note that even traditional x86 processors suffer from a related problem: while old applications will run correctly on a new processor, they usually need to be recompiled to take full advantage of the new processor implementation. This is not a problem on Crusoe processors, since in effect, the Code Morphing software always transparently "recompiles" and optimizes the x86 code it is running.

The flexibility of the software-translation approach comes at a price: the processor has to dedicate some of its cycles to running the Code Morphing software, cycles that a conventional x86 processor could use to execute application code. To deliver good practical system performance, Transmeta has carefully designed the Code Morphing software for maximum efficiency and low overhead.

DRAWING THE HARDWARE-SOFTWARE LINE

Virtualizing an x86 CPU is a challenging undertaking because of the complexity of the x86 architecture. Choosing which functions to implement in hardware and which in software is a major engineering challenge, involving issues such as cost and complexity, overall performance and power consumption. Clearly, there are many possible choices, influenced by market demands, or the latest hardware technologies available.

For its initial products, Transmeta has drawn the line between hardware and software so that software handles the complex task of decoding x86 instructions and generating explicitly parallel molecules, which the hardware executes using a very simple, high-speed, VLIW engine. A few unique hardware features, described later in this paper, were added to better support dynamic translation. The hardware-software line might be drawn differently for another kind of product, for example, a high-end server processor.

DECODING AND SCHEDULING

Conventional x86 superscalar processors fetch x86 binary instructions from memory and decode them into micro-operations, which are then reordered by out-of-order dispatch hardware and fed to the functional units for parallel execution.

In contrast (besides being a software rather than a hardware solution), Code Morphing can translate an entire group of x86 instructions at once, creating a *translation*, whereas a superscalar x86 translates single instructions in isolation. Moreover, while a traditional x86 translates each x86 instruction every time it is executed, Transmeta's software translates instructions *once*, saving the resulting translation in a *translation cache*. The next time the (now translated) x86 code is executed, the system skips the translation step and directly executes the existing optimized translation.

Implementing the translation step in software as opposed to hardware opens up new opportunities and challenges. Since an out-of-order processor has to translate and schedule instructions every time they execute, it must do so very quickly. This seriously limits the kinds of transformations it can perform. The Code Morphing approach, on the other hand, can *amortize* the cost of translation over many executions, allowing it to use much more sophisticated translation and scheduling algorithms. Likewise, the amount of power consumed for the translation process is amortized, as opposed to having to pay it on every execution. Finally, the translation software can *optimize* the generated code and potentially reduce the number of instructions executed in a translation. In other words, Code Morphing can speed up execution while at the same time reducing power!

CACHING

The translation cache, along with the Code Morphing code, resides in a separate memory space that is inaccessible to x86 code. (For better performance, the Code Morphing software copies itself from ROM to DRAM at initialization time.) The size of this memory space can be set at boot time, or the operating system can make the size adjustable.

As with all caching, the Code Morphing software's technique of reusing translations takes advantage of "locality of reference". Specifically, the translation system exploits the high *repeat rates* (the number of times a translated block is executed on average) seen in real-life applications. After a block has been translated once, repeated execution "hits" in the translation cache and the hardware can then execute the optimized translation at full speed.

Some benchmark programs attempt to exercise a large set of features in a small amount of time, with little repetition—a pattern that differs significantly from normal usage. (When was the last time you used every other feature of Microsoft Word exactly once, over a period of a minute?) The overhead of Code Morphing translation is obviously more evident in those benchmarks. Furthermore, as an application executes, Code Morphing "learns" more about the program and improves it so it will execute faster and faster. Today's benchmarks have not been written with a processor in mind that gets faster over time, and may "charge" Code Morphing for the learning phase without waiting for the payback. As a result, some benchmarks do not accurately predict the performance of Crusoe processors.

On typical applications, due to their high repeat rates, Code Morphing has the opportunity to optimize execution and amortize any initial translation overhead. As an example, consider a multimedia

application such as playing a DVD—before the first video frame has been drawn, the DVD decoder will have been fully translated and optimized, incurring no further overhead during the playing time of the DVD. In summary, we find that the Crusoe processor's approach of caching translations delivers excellent performance in real-life situations.

FILTERING

It is well known that in typical applications, a very small fraction of the application's code (often less than 10%, sometimes as little as 1%) accounts for more than 95% of execution time. Therefore, the translation system needs to choose carefully how much effort to spend on translating and optimizing a given piece of x86 code. Obviously, we want to lavish the optimizer's full attention on the most frequently executed code but not waste it on code that executes only once.

The Code Morphing software includes in its arsenal a wide choice of execution modes for x86 code, ranging from interpretation (which has no translation overhead at all, but executes x86 code more slowly), through translation using very simple-minded code generation, all the way to highly optimized code (which takes longest to generate, but which runs fastest once translated). A sophisticated set of heuristics helps choose among these execution modes based on dynamic feedback information gathered during actual execution of the code.

PREDICTION AND PATH SELECTION

One of the many ways in which the Code Morphing software can gather feedback about the x86 program is to *instrument* translations: the translator adds code whose sole purpose is to collect information such as block execution frequencies, or branch history. This data can be used later to decide when and what to optimize and translate. For example, if a given conditional x86 branch is highly biased (e.g., usually taken), the system can likewise bias its optimizations to favor the most frequently taken path. Alternatively, for more balanced branches (taken as often as not, for example), the translator can decide to speculatively execute code from both paths and select the correct result later. Analogously, knowing how often a piece of x86 code is executed helps decide how much to try to optimize that code. It would be extremely difficult to make similar decisions in a traditional hardware-only x86 implementation.

MAKING A TRANSLATION

To conclude this section, we illustrate by way of a simple example how the Code Morphing system translates a chunk of x86 code into equivalent code for the Crusoe processor's VLIW engine.¹ Assume that the filtering and path selection algorithms have chosen the following four x86 instructions, (A) through (D), for translation.

```
A. addl %eax, (%esp)      // load data from stack, add to %eax
B. addl %ebx, (%esp)      // ditto, for %ebx
C. movl %esi, (%ebp)      // load %esi from memory
D. subl %ecx, 5           // subtract 5 from %ecx register
```

In a first pass, the *frontend* of the translation system decodes the x86 instructions and translates them into a simple sequence of atoms. At this stage, it is still fairly easy to discern the correspondence between the original and generated code. (Registers %r30 and %r31 are used as temporaries for the memory-load operations.)

```
ld %r30, [%esp]          // load from stack, into temporary
add.c %eax, %eax, %r30    // add to %eax, set condition codes.
ld %r31, [%esp]
add.c %ebx, %ebx, %r31
ld %esi, [%ebp]
sub.c %ecx, %ecx, 5
```

In a second pass, the *optimizer* applies well-known compiler optimizations to the code, such as common subexpression elimination, loop invariant removal, or dead code elimination (including unnecessary settings of the condition codes). This exemplifies optimizations that a hardware-only x86 implementation cannot do: a software-based translation system can actually *eliminate* atoms from the instruction stream, rather than just reorder them. In this example, all but the last setting of the condition code is unnecessary (allowing for greater flexibility in scheduling), and one of the load atoms is redundant, leaving fewer atoms to be executed.

```
ld %r30, [%esp]          // load from stack only once
add %eax, %eax, %r30
add %ebx, %ebx, %r30      // reuse data loaded earlier
ld %esi, [%ebp]
sub.c %ecx, %ecx, 5       // only this last condition code needed
```

In a final pass, the *scheduler* reorders the remaining atoms and groups them into individual molecules. This process is similar to what out-of-order processors do in their dispatch hardware. However, by using software to schedule the code, it becomes feasible to use more effective scheduling algorithms and

1. As a reminder, we write VLIW code one molecule per line, with atoms separated by semicolons. The destination register of an atom is specified first; a ".c" opcode suffix designates an operation that sets the condition codes.

consider a larger window of instructions than would be reasonable in hardware. After scheduling, we have reduced the four original x86 instructions down to just two molecules:

```
1. ld %r30,[%esp];    sub.c %ecx,%ecx,5
2. ld %esi,[%ebp];    add %eax,%eax,%r30;    add %ebx,%ebx,%r30
```

There are two important points to observe here:

- Though the molecules are executed in-order by the hardware, they perform the work of the original x86 instructions out of order.
- The molecules explicitly encode the instruction-level parallelism, hence they can be executed by a simple (and hence fast and low-power) VLIW engine; the hardware need not perform any complex instruction reordering itself.

CRUSOE HARDWARE SUPPORT FOR CODE MORPHING

Dynamic translation on conventional processors would result in unsatisfactory performance. In contrast, the Crusoe hardware can achieve excellent performance because it has been designed specifically with dynamic translation in mind. Below, we discuss three simple hardware features that support exceptions, speculation, optimization of memory operations, and self-modifying code.

EXCEPTIONS AND SPECULATION

Without special hardware support, it is in general very difficult for a dynamic translation system to correctly model the exception semantics of the target ISA while at the same time achieving high performance. The reason is that exception semantics impose severe constraints on instruction scheduling. Consider again the example from the previous section, where the following x86 code:

```
A. addl %eax, (%esp)
B. addl %ebx, (%esp)
C. movl %esi, (%ebp)
D. subl %ecx, 5
```

was translated into the following two molecules:

```
1. ld %r30,[%esp];    sub.c %ecx,%ecx,5
2. ld %esi,[%ebp];    add %eax,%eax,%r30;    add %ebx,%ebx,%r30
```

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. Observe that in the translation above, atoms occur out of order with respect to the original x86 code

order. Now imagine that during execution, the load instruction in molecule 2, corresponding to x86 instruction (C), takes a page fault. However, by that time, the processor has already executed code in molecule 1 corresponding to instruction (D), which violates the rules of precise exceptions.

Solving this problem without special hardware support unduly constrains the scheduling of host instructions, or requires extra host instructions to be issued, either of which reduces performance even in the common case where no exceptions occur.

It is worth noting at this point that out-of-order processors, too, have this problem. They typically employ complex hardware mechanisms to delay or undo the effects of micro-ops that have been executed “too soon”.

The Crusoe host processor provides a much simpler hardware solution that works hand-in-hand with the Code Morphing software. All registers holding x86 state are *shadowed*, i.e., there exist two copies of each register, a *working* and a *shadow* copy. Normal atoms only update the working copy of the register. When execution reaches the end of a translation without encountering an exception, a special *commit* operation copies all working registers into their corresponding shadow registers, indeed committing the work done in the translation. On the other hand, if any x86-level exception occurs inside the translation, the runtime system undoes the effects of all molecules executed since the start of the translation. This is done via a *rollback* operation which copies the shadow register values (last committed at the end of the previous translation) back into the working registers. At this point, the Code Morphing software re-executes the x86 instructions conservatively, that is to say in their original program order, to determine the actual location of the exception.

Undoing changes to memory is slightly more complicated. The Crusoe processor handles x86 store operations by holding store data 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 (no exceptions), the Crusoe hardware is designed such that commit operations are effectively “free”.

ALIAS HARDWARE

The more freedom the scheduler has to move atoms around to fill molecules, the better code it can usually generate. One of the biggest limits on this freedom comes from potential dependencies between memory operations. In particular, it is often desirable to be able to reorder load instructions ahead of store instructions. However, doing that is incorrect if the load happens to use data from the preceding store, and since it is generally hard to prove otherwise at translation time, a translator often has to make overly conservative assumptions. (This is also a problem for traditional compilers and microprocessors.)

The Crusoe host provides innovative *alias* hardware that addresses this problem. When the translator moves a load operation ahead of a store operation, it converts the load into a *load-and-protect* (which in addition to loading data also records the address and size of the data loaded) and the store into a

store-under-alias-mask (which checks for protected regions). In the (unlikely) event that the store operation overwrites the previously loaded data, the processor raises an exception and the runtime system can take corrective action. Using this mechanism, it is always safe to reorder memory loads and stores. Again, Crusoe hardware provides a very simple hardware mechanism that in concert with software solves a thorny problem.

The alias hardware can be put to even better use than moving atoms around: it can help to eliminate redundant load/store atoms. Consider the case where a datum is loaded from memory twice, but there is an intervening store operation (a code sequence that is actually fairly common in processors with few registers, like the x86):

```
ld %r30,[%x]           // first load from location X
...
st %data,[%y]          // might overwrite location X
ld %r31,[%x]           // this accesses location X again
use %r31
```

As long as the intervening store operation does not overlap with the first load, the second load is redundant, but all too often a translator or compiler cannot prove that this is the case. Using the alias hardware, it is a simple matter to protect the first load, have the store check pending aliases, and eliminate the second load:

```
ldp %r30,[%x]          // load from X and protect it
...
stam %data,[%y]         // this store traps if it writes X
use %r30                // can use data from first load
```

Notice that the use of the loaded data can now also be scheduled earlier, further speeding up the generated code. To our knowledge, no out-of-order processor can perform a similar feat!

COPING WITH SELF-MODIFYING CODE

At times, x86 instructions in memory get overwritten, either because the operating system is loading a new program, or because an application is using self-modifying code. When this happens to code that has already been translated, the Code Morphing software needs to be notified to keep it from erroneously executing a translation for the old code. To this end, whenever the system translates a block of x86 code, it write-protects the page of x86 memory containing that code. It does so by setting a dedicated “translated” bit in that page’s entry in the processor’s memory management unit. (As with other details of the VLIW hardware, that bit is invisible to x86 software.) When a protected page is written to, the simplest remedy is to invalidate the affected translation(s). As the runtime system dynamically learns more about the program’s behavior, it switches to more sophisticated strategies (beyond the scope of this paper).

EXAMPLE: A COMPLEX TRANSLATION

We close our review of translation technology with a slightly longer example taken from an actual x86 application running on Windows NT, illustrating more of the sophisticated capabilities of Code Morphing. The following twenty x86 instructions (which in a conventional processor would generate more than twenty micro-ops):

```

1.      movl    %ecx,$0x3
2.      jmp     lbl1
...
3.  lbl1:  movl    %edx,0x2fc(%ebp)
4.      movl    %eax,0x304(%ebp)
5.      movl    %esi,$0x0
6.      cmpl    %edx,%eax
7.      movl    0x40(%esp,1),$0x0
8.      jle     skip1
9.      movl    %esi,$0x1
10. skip1:  movl    0x6c(%esp,1),%esi
11.      cmpl    %edx,%eax
12.      movl    %eax,$0x1
13.      jl      skip2
14.      xorl    %eax,%eax
15. skip2:  movl    %esi,0x308(%ebp)
16.      movl    %edi,0x300(%ebp)
17.      movl    0x7c(%esp,1),%eax
18.      cmpl    %esi,%edi
19.      movl    %eax,$0x0
20.      jnl     exit1
      exit2:

```

were translated into the following ten VLIW instructions:

```

1.      addi     %r39,%ebp,0x2fc
2.      addi     %r38,%ebp,0x304
3.      ld       %edx,[%r39];      add %r27,%r38,4;      add %r26,%r38,-4
4.      ld       %r31,[%r38];      add %r35,0,1;      add %r36,%esp,0x40
5.      ldp      %esi,[%r27];      add %r33,%esp,0x6c;  sub.c %null,%edx,%r31
6.      ldp      %edi,[%r26];      sel #1e,%r32,0,%r35
7.      stam     0,[%r36];          sel #1,%r24,%r35,0;  add %r25,%esp,0x7c
8.      stam     %r32,[%r33];      add %ecx,0,3;      sub.c %null,%esi,%edi
9.      st       %r24,[%r25];      or %eax,0,0;      brcc #1t,<exit2>
10.     br       <exit1>

```

There are several interesting points to note:

- The x86 unconditional `JMP` has no corresponding instruction in the translation: the path selector simply “follows” the branch and continues translation at the target of the `JMP`.
- Registers have been aggressively renamed in software; there is no need for a complex (and power consuming) register renamer in hardware.
- The scheduler has rearranged the instructions to execute out of order relative to the original x86 “source” code.
- The translator has replaced the two internal conditional branches with “select” instructions (which conditionally pick one of two results). In effect, the Code Morphing system is speculatively executing both legs of a branch and picking the correct result later. Reducing the number of branches is highly desirable, since they often cause inefficiencies in pipelined processors. We know of no out-of-order processor that can completely eliminate conditional branches.
- The Crusoe alias hardware has been used in the translation (in molecules 5 through 8) to hoist loads above stores and thus pack the code more effectively.

LONGRUN™ POWER MANAGEMENT

Although the Code Morphing software’s primary responsibility is ensuring x86 compatibility, it also provides interfaces to capabilities available only in Crusoe processor models. LongRun power management is one example—a facility in the TM5400 model that can further minimize that processor’s already low power consumption.

In a mobile setting, most conventional x86 CPUs regulate their power consumption by rapidly alternating between running the processor at full speed and (in effect) turning the processor off. Different performance levels can be obtained by varying the on/off ratio (the “duty cycle”). However, with this approach, the processor may be shut off just when a time-critical application needs it. The result may be glitches, such as dropped frames during movie playback, that are perceptible (and annoying) to a user.

In contrast, the TM5400 can adjust its power consumption without turning itself off—instead, it can adjust its clock frequency on the fly. It does so extremely quickly, and without requiring an operating system reboot or having to go through a slow sequence of suspending to and restarting from RAM. As a result, software can continuously monitor the demands on the processor and dynamically pick just the right clock speed (and hence power consumption) needed to run the application—no more and no less. Since the switching happens so quickly, it is not noticeable to the user.

Finally, the Code Morphing software can also adjust the Crusoe processor's voltage on the fly (since at a lower operating frequency, a lower voltage can be used). Because power varies linearly with clock speed and by the square of the voltage, adjusting both can produce *cubic* reductions in power consumption whereas a conventional CPUs can adjust power only linearly. For example, assume an application program only requires 90% of the processor's speed. On a conventional processor, throttling back the processor speed by 10% cuts power by 10%, whereas under the same conditions, LongRun power management can reduce power by almost 30%—a noticeable advantage!

CONCLUSION

In 1995, Transmeta set out to expand the reach of microprocessors into new markets by dramatically changing the way microprocessors are designed. The initial market is mobile computing, in which complex power-hungry processors have forced users to give up either battery running time or performance. The Crusoe processor solutions have been designed for lightweight (two to four pound) mobile computers and Internet access devices such as handhelds and web pads. They can give these devices PC capabilities and unplugged running times of up to a day.

To design the Crusoe processor chips, the Transmeta engineers did not resort to exotic fabrication processes. Instead they rethought the fundamentals of microprocessor design. Rather than “throwing hardware” at design problems, they chose an innovative approach that employs a unique combination of hardware and software. Using software to decompose complex instructions into simple atoms and to schedule and optimize the atoms for parallel execution saves millions of logic transistors and cuts power consumption on the order of 60–70% over conventional approaches—while at the same time enabling aggressive code optimization techniques that are simply not feasible in traditional x86 implementations. Transmeta's Code Morphing software and fast VLIW hardware, working together, achieve low power consumption without sacrificing high performance for real-world applications.

Although the model TM3120 and model TM5400 are impressive first efforts, the significance of the Transmeta approach to microprocessor design is likely to become more apparent over the next several years. The technology is young and offers more freedom to innovate (both hardware and software) than conventional hardware-only designs. Nor is the approach limited to low-power designs or to x86-compatible processors. Freed to render their ideas in a combination of hardware and software, and to evolve hardware without breaking legacy code, Transmeta microprocessor designers may produce one surprise after another in the new millennium.

To learn more about the Transmeta Crusoe processor family, consult <http://www.transmeta.com>.

ACKNOWLEDGEMENTS

This paper represents the work of a brilliant team of fellow Transmeta engineers. The author would like to thank everyone who has contributed to this paper.