Software Developer Bulletin

Crusoe™ Processor Software Optimization Guide

This document contains recommendations for optimizing x86 software for use with Transmeta Crusoe processors. These software optimization guidelines are intended primarily for model TM5x00 Crusoe processors, though many of these tips are useful for any x86 software application. The purpose of this document is to encourage software developers to produce high performance code that will run more efficiently on Crusoe processors.

Code Morphing Software

Crusoe processors consist of a hardware engine logically surrounded by a software layer. The hardware component of the Crusoe processor is a very long instruction word (VLIW) CPU capable of executing up to four operations in each clock cycle. The VLIW 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 they are running on x86 hardware. The software layer is called Code Morphing software because it dynamically “morphs” (changes) x86 instructions into native VLIW instructions. Code Morphing software includes a number of advanced features to achieve good system-level performance. Code Morphing software support facilities are also built into the underlying processor hardware.

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). Code Morphing software is the first program to start executing when the processor boots. All x86 code sees only the x86 ISA that the Code Morphing software supports. The only program written directly for the VLIW engine is the Code Morphing software itself.

The typical behavior of Code Morphing software is to execute a loop that decodes and executes x86 instructions. The first few times a specific x86 code sequence is executed, Code Morphing software interprets the code by decoding the instructions one at a time and then dispatching execution to corresponding VLIW native instruction subroutines. Once the x86 code has been executed several times, Code Morphing software translates the x86 instructions into highly optimized and extremely fast native VLIW instructions, executes the translated code, and caches the native instruction translations for future use. If the same x86 code is required to execute again, the high-performance cached translations are executed immediately and no re-translation is required. For a detailed explanation of the Crusoe processor Code Morphing software technology, see the Transmeta white paper *The Technology Behind Crusoe Processors*.

The flexibility of the software translation approach comes at a price - the processor has to dedicate some of its operating cycles to running the Code Morphing software. These extra operating cycles are cycles that a conventional x86 processor could use to execute application code. To deliver good overall system performance, Code Morphing software has been carefully designed for maximum efficiency and low overhead. Application code developed for use on the Crusoe processor can also benefit from a few simple guidelines that likewise improve code execution efficiency and minimize Code Morphing software overhead.

The guidelines provided below should be followed to maximize the performance/power benefits of Crusoe processors with Code Morphing software.

Instructions and Registers

The Crusoe processor VLIW core incorporates many internal registers, so Code Morphing software attempts to turn repeated loads of the same memory location into on-chip register references, which are faster and more efficient. This technique, for example, works well with temporary variables on the stack. On the other hand, this optimization only works over short spans of code, so it should not be expected to apply to reloading a stack location that was last touched more than, for example, 100 instructions earlier.

Crusoe processors do not suffer from partial register stalls, but using the “high” registers (`ah`, `bh`, `ch`, `dh`) can result in slower execution and should be avoided.

VLIW engines have large instruction words. This off-loads a lot of the work spent in out-of-order execution cores that can be spent elsewhere, but the trade-off is in instruction cache (I-cache) pressure. If there is a choice between using a template that generates thirty different versions of the same piece of code with mildly different parameters and having just one lean routine that does them all fairly quickly, the latter approach will generally prove superior.

Crusoe processor muls and imuls are not single-cycle operations. However, muls and imuls (and idivs) using small immediates, which are commonly used to generate array references by C compilers, are converted into shifts and adds, often leveraging special Crusoe processor instructions that do both a shift and add in the same instruction.

Unconditional branches to fixed addresses eventually become free (after Code Morphing software optimization). Too many conditional branches in an active loop should be avoided. Loops should ideally be limited to three conditional branches or less, with enough non-branching work at either destination of a branch to improve scheduling. Ten to twenty x86 instructions is usually enough.

Expressions should be structured to expose as much parallelism as possible.

Crusoe processors do not feature a write-combiner, but if consecutive 32-bit words are written during PCI writes, this allows the built-in PCI controller (integrated northbridge), to stream them out in burst-mode.

As for most modern processors, working data sets should be sized to fit into the on-chip L1 (64K-byte) and L2 (256K-byte) data caches.

Processor capabilities based on the CPUID string (e.g. “GenuineTMx86”) should never be assumed. The capability bits should be used for this purpose. Current generation Crusoe processors support MMX and conditional `mov` instructions, but don't support 3DNow and SSE instructions.

DMA

DMA operations utilize the Crusoe processor integrated northbridge. Before the processor is put into sleep mode, as many DMA sources as possible should be turned off. This will lower processor power consumption because it minimizes the periodic processor wake-ups required to handle DMA operations.

Long and contiguous DMA reads and writes are more efficient (per byte transferred) compared to many small, non-contiguous DMA accesses, which are less efficient.

A side-benefit of integrating the northbridge into the Crusoe processor is that doing DMA transfers of data already in the L1 or L2 cache is quite fast. This feature can be exploited by processing blocks of data that fit in the cache and then submitting them to the device before they leave the cache.

MMX and Floating-Point Operations

Use of the floating-point stack pointer should be managed carefully. Some applications don't monitor the stack pointer, and as a result they routinely overflow the stack. This can happen, for example, by not cleaning up the stack pointer when exiting functions.

All data references should be aligned. Some compilers have trouble keeping the stack 8-byte aligned, which can slow the system during 8-byte floating-point operands.

Crusoe processor performance is improved if the floating-point control word is set to make floating-point divides and square roots operate on 32-bit floats instead of the full 80-bit internal calculation.

MMX and floating-point operations feature longer latencies than integer and load/store operations. If MMX or floating-point code can be populated with integer work, or if written so the code isn't totally serialized, it should improve code scheduling and efficiency. For best MMX and floating-point performance, four independent operations should be kept going at all times. The `fxch` instruction on Crusoe processors is mostly free due to Code Morphing software optimizations.

To get the best floating-point performance from Crusoe processors, all exceptions should be masked in the FSR, and `denorm` operands should be avoided. They should not be "accidentally" used as memory operands to floating point operations, e.g. as might occur in "`fmul dword ptr[eax]`".

Strategies to Avoid

Timing should not be done using artificial processor benchmarks. Code Morphing software will reduce some artificial benchmarks into trivial code sequences. For instance, Code Morphing software often turns a sequence of unconditional branches into a single stream of code, eliminating the branch, so simple branching benchmarks appear to be getting zero-cycle branches. A walk-through code snippet of a real application engine will yield much more realistic performance timing results. Also, timing-sensitive code should be avoided. If a power-saving state triggers in the middle of a timing benchmark, the user experience for the rest of the application will be thrown off.

In general, self-modifying code should not be used with Crusoe processors. If absolutely necessary, self-modifying code should be limited to only patching immediates or address offsets.

It is very difficult for Code Morphing software to anticipate correct jump destinations. In general, indirect jumps can be very expensive in modern processors, and this holds true for Code Morphing software as well. However, commonly executed function pointers, like those used in virtual methods of C++ classes, assuming they rarely change, are eventually optimized away.

Crusoe processors, like most modern processors, execute real- or SMI-mode code poorly. This code should be avoided when possible and minimized where necessary.

In most cases, critical loops should not be less than about ten instructions, although the exact number depends on which instructions are used. Code Morphing software de-constructs x86 operations into so-called "atoms", and those atoms are scheduled into "molecules" (VLIW instructions). The longer the loop, the better the scheduler can do, while at the same time, I-cache pressure can also become an issue. Loop-unrolling, however, should not be over-used either.

Writable data and x86 instructions should not be mixed on the same 4K page. When data is written to a memory page, Crusoe processors assume that any translations associated with code on that page should be invalidated until proven otherwise.

Excessively complex instructions should not be used in the middle of an active loop. Crusoe processors send those instructions to less efficient “out-of-line” code, which breaks-up translation scheduling. The biggest offender is the `rep` prefix on `mov` instructions, but other examples include `cpuid`, `cmpxchng`, `loadcs`, `wrmsr`, `rdmsr`, `pusha`, `rdtsc`, `bts`, `bsf`, `bsr`, etc. `rep mov` instructions are the best way to move large amounts of data, but are not a good way to move small chunks of data because they become dynamically generated “out-of-line” code that doesn't schedule with the other atoms in the translation to VLIW.

Do not use segmentation. Run in “flat mode”: segment base zero, limit 0xffffffff, all data segments readable and writable.

Other Guidelines

During the first few iterations through application code, Crusoe processors execute more slowly than in the steady state. Code Morphing software learns how executed code behaves and optimizes it for native VLIW execution. This should be considered when trying to measure the processor speed, or when interfacing with devices that have real-time constraints.

The adaptive learning behavior of Crusoe processors with Code Morphing software can cause x86 application execution time to vary more than expected. Very few applications are so sensitive that this matters, but turning off the L2 or L1 cache on a conventional processor will often mimic these pauses and help catch race conditions.

If an application is idle while waiting for system events, e.g. waiting for a keypress, disk interrupt, or the mouse button to be released, a HALT instruction should be used while waiting for an interrupt. Applications should not be designed to just sit in a loop and poll for some system activity. Applications that rely on loop polling are extremely inefficient and unnecessarily waste energy, heating up the processor with no performance benefit.

Application background activities that are not essential should be disabled, by default, in battery-operated mode. Examples include spell-checking and grammar-checking functions on word processors. If these functions are made to run in background mode by default, they generate a lot of processor activity and use a large amount of power, without the user being aware of the energy penalty involved. If these functions are disabled by default and only run as-needed, the application will be much more energy efficient and “battery-friendly”.

Applications should be made “benchmark-capable”. The game applications that were most effective in the development of Crusoe processors and Code Morphing software were Doom and Quake, because they both have built-in benchmarking modes that allow saving results to a text file. Doom and Quake are also available for Linux, allowing extensive automated testing of each Code Morphing software and Crusoe processor hardware revision using Transmeta’s automated Linux-based test farm. If an application can be easily benchmarked and works well on a portable computer, this applications can be used in the development and enhancement of existing and next-generation Crusoe processors and Code Morphing software.

Property of: Transmeta Corporation
3940 Freedom Circle
Santa Clara, CA 95054 USA
(408) 919-3000

http://www.transmeta.com