IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems

Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang and Yigal Zemach.

Abstract

IA-32 Execution Layer (IA-32 EL) is a new technology that executes IA-32 applications on Intel® Itanium® processor family systems. Currently, support for IA-32 applications on Itanium-based platforms is achieved using hardware circuitry on the Itanium processors. This capability will be enhanced with IA-32 EL—software that will ship with Itanium-based operating systems and will convert IA-32 instructions into Itanium instructions via dynamic translation. In this paper, we describe aspects of the IA-32 Execution Layer technology, including the general two-phase translation architecture and the usage of a single translator for multiple operating systems. The paper provides details of some of the technical challenges such as precise exception, emulation of FP, MMX™, and Intel® Streaming SIMD Extension instructions, and misalignment handling. Finally, the paper presents some performance results.

1. Introduction

The Intel® Itanium® processor family (IPF) is primarily designed to provide leading performance and capabilities for 64-bit applications and operating systems (OSes). The ability to execute IA-32 applications is needed for flexibility and easy migration from existing IA-32 systems to Intel® Itanium® 2-based solutions. Primary, performance-sensitive applications are encouraged to be ported to Itanium architecture, while secondary, non-performance critical applications and applications or libraries where source code is not available, can continue to execute as IA-32 code.

Currently, IA-32 support on IPF is available through hardware circuitry. IA-32 Execution Layer (IA-32 EL) is a new technology that provides the same capability, executing IA-32 applications on IPF through software. IA-32 EL is dynamic binary translation software that translates IA-32 instructions into Itanium instructions. IA-32 EL runs on both Windows* and Linux* operating systems and can accelerate IA-32 application performance compared to the existing hardware solution. Field tests verified IA-32 EL robustness and performance benefits when compared with the hardware circuitry. IA-32 EL handles all IA-32 user code binaries and does not rely on specific software conventions. IA-32 EL is a software-only solution requiring no special hardware assists. The main challenges were to provide hardware-level quality that correctly executes IA-32 applications, without compromising performance.

IA-32 EL has the following characteristics:

1. Aggressive dynamic information collection during the first translation phase and usage of that information for a second translation phase
2. A single, OS-independent binary for translating IA-32 applications on multiple OSs
3. A mechanism for precise exception handling

This paper is organized as follows: Section 2 describes the general architecture of IA-32 EL. Section 3 describes the IA-32 EL solution for providing one translator that runs on multiple, native Itanium-based OSes. Section 4 explores mechanisms for generating and maintaining a consistent IA-32 state, as required for providing precise exception handling and for enabling a debugger to run on top of the translator. Section 5 focuses on several technological challenges faced in developing IA-32 EL, i.e. floating point, Intel® MMX™ technology, and Intel® Streaming SIMD Extensions (SSE) modeling, and misalignment handling. Section 6 discusses performance and examines benchmark data.

2. Overview

This section describes the general architecture of IA-32 EL. General background on binary translation technology is given toward the end of this paper in the "related work" section. [1, 2, 10, 16, 18]

General Architecture

IA-32 EL is targeted for application-level translation only. Therefore, it runs on top of the native 64-bit operating system, like the FX!32* [6,8] and Hewlett Packard's PA-RISC* translator [23], and unlike the Transmeta* code morphing software [7]. IA-32 EL is loaded to the same user space as the translated application and it operates in the user level only. The application image(s) and data remain unchanged, similar to their layout on the original IA-32 platform.

Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36 2003)
0-7695-2043-X/03 $17.00 © 2003 IEEE

The translator is architected to work on multiple, native Itanium-based OSes. This ability is achieved by separating most of the translation engine and algorithms into an OS-independent module (BTGeneric), which interfaces with a small glue layer that is OS dependent (BTLib). These two components interact via a well-defined API. BTLib is responsible for providing all system services, such as memory allocations.

IA-32 EL is a two-phase dynamic binary translator. It caches translations within the same process, but does not maintain them beyond the translated process. The first phase, cold code translation, is designed to be fast, with minimal optimizations and overhead and uses instrumentation to identify hotspots. The second phase, hot code translation, retranslates and further optimizes those hotspots. Cold code translation is done on a basic-block granularity, with 4-5 IA-32 instructions per block on average. Hot code translation is applied to hot traces on a hyper-block granularity, with about 20 IA-32 instructions per block on average. The entire process is shown later on in Figure 2.

Cold Code Translation

Cold code is generated at basic-block granularity. However, simple analysis is done on neighboring blocks (1-20 basic blocks) for better code generation, as shown in Figure 1. The analysis starts from the current instruction pointer (IP). It includes decoding, building a flow graph, computing the liveness of IA-32 EFlags bits, and tracking floating point (FP) stack changes between blocks.

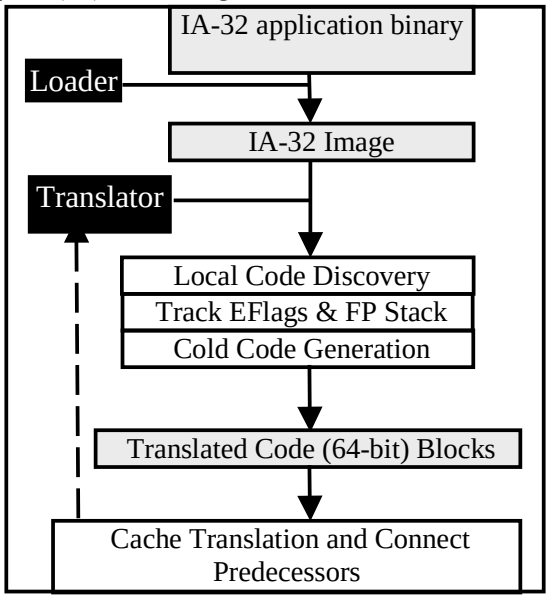

Figure 1-Cold Code Translation

This process enables the translator to eliminate redundant IA-32 EFlags updates and speeds up FP code, as described in chapter 5. The analysis is followed by code generation of few basic blocks only. (Unexecuted blocks are never generated.)

Code generation is accelerated by using prepared translation templates for each IA-32 instruction variant. These templates are patched according to the instruction parameters and environmental factors. The templates are carefully hand optimized to use the best Itanium instruction sequences.

Cold blocks contain instrumentation to collect information that is later used for hot translation. The instrumentation includes a basic-block use counter, an edge counter for blocks ending with conditional or indirect branches, and misalignment detection. This is different than most existing dynamic systems, which perform such instrumentation during interpretation [3, 6, 7, 9, 21, 23]. Examples of instrumentation in translated code are given in [4, 22]. Others [5, 17] suggest using hardware for the same purpose. The advantage of adding the instrumentation in cold blocks is that it provides more precise information for later use, since cold blocks can run longer than interpreted code and still maintain low overhead.

Translated blocks usually jump directly from one block to another. Blocks ending with indirect branches that are not predicted use a fast lookup table to find the branch target. In cases where a block jumps to an address that has not been translated yet, the initial generated code contains a branch to the translator code which is later patched to generate a direct branch between the blocks.

Several variants of cold blocks can exist to handle special cases, including FP exceptions, self-modifying code (SMC), and others. Cold blocks may be recycled due to garbage-collection, unloading of a library, or SMC detection[1].

Hot Code Translation

When the use-counter in a block reaches the heating threshold, the instrumentation code of that block triggers the registration of the block as an optimization candidate by branching out to a special entry in the translator. When enough blocks have registered or one block has registered twice, an optimization session (hot code translation) starts. This algorithm enables evaluation of several hot blocks at once, and thus uses more educated merging and splitting decisions. On the other hand, blocks

[1] Writable page translations include code for detecting possible changes from the code used for translation.

belonging to tight loops will not wait too long due to the second-registration trigger. About 5-10% of the cold blocks reach the heating threshold[2].

The first step in hot block translation is to select a trace of IA-32 basic blocks that compose a hyper block – single entry, multiple exits. This selection is based on the use and edge counter information collected during the cold code run of those blocks. Predication can be used to include both sides of `if…then…` and `if…then…else…` structures as part of the linear trace, according to profitability estimations[3]. If a loop is identified, it may be unrolled. Only about 6% of the hot blocks suffer from a premature exit (with no special penalty).

Next, the original IA-32 code is decoded and analyzed again. Decoded information is not maintained from the cold translation. Here, unlike the fast encoding using binary templates during cold-code generation, each instruction generates associated Intermediate Language data structures (ILs). The ILs represent the target machine instructions and are contained in a linked-list. The precompiled binary templates and the IL-generation are derived from the same template source code. These templates are written in a special language for easier maintenance and validation.

During the IL generation phase, the translator does the following optimizations and preparations:

1. Adds misalignment avoidance code to memory accesses that were detected by instrumentation as being prone to misalignment (see section 5).
2. Tracks IA-32 addresses and their values. Eliminates redundant compound address expressions typical in IA-32 code, such as `[offset + base + index * scale]`.
3. Tracks information about values in registers and uses it for simplifying the translation.
4. Eliminates EFlags generation using techniques similar to those used in cold-code generation.
5. Analyzes FP stack flow and Intel® Streaming SIMD Extensions (SSE) format conversions. See more in section 5 below.
6. Performs other FP optimizations, such as register allocation and FXCHG elimination (see chapter 5 for more details).

The translator scans the resulting IL list to build a data-dependency graph. It removes dead ILs and marks those ILs needed for side-exits only as "sideway ILs". The translator computes weights and attaches them to individual ILs to signify the relative importance of scheduling them early. Peepholes, using dependency information, eliminate additional

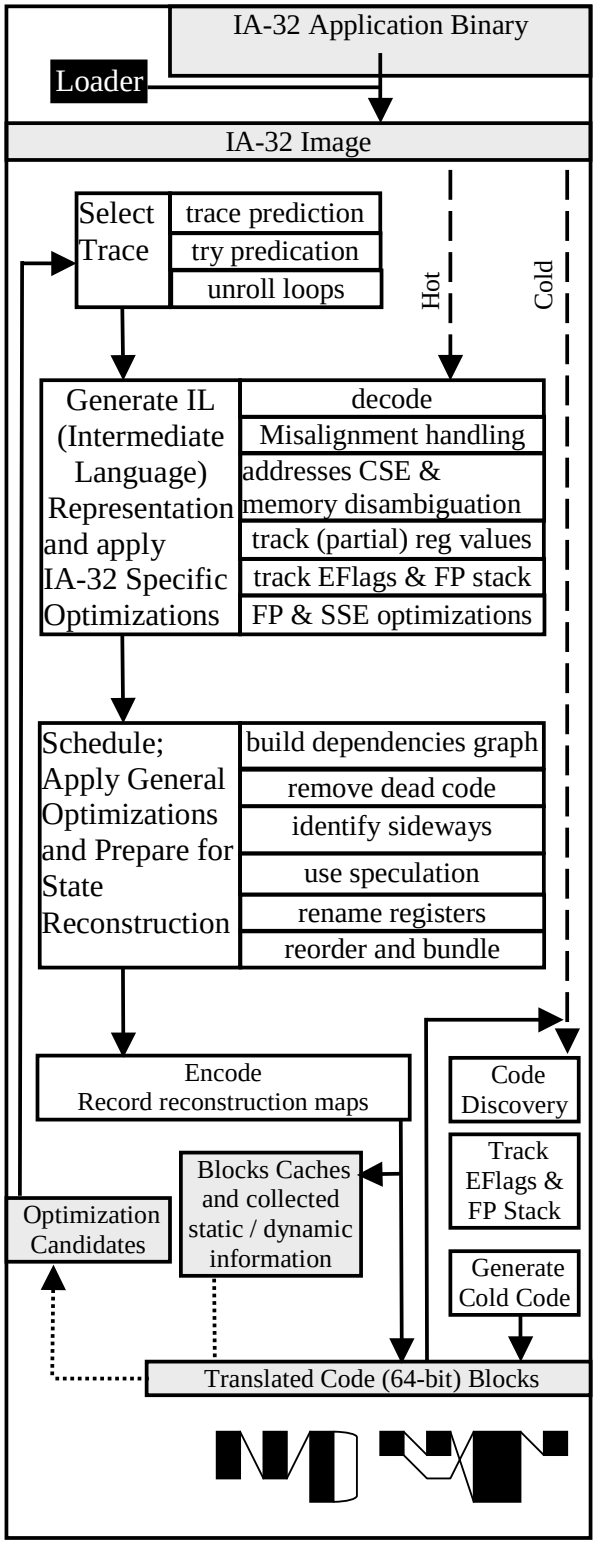

Figure 2 - Overall Code Translation

[2] It heavily depends on the application workload.
[3] See [10] for similar considerations

instructions.
Next, the scheduler reorders the instructions in the hot block. ILs are ordered and bundled according to architectural and microarchitectural limitations. For better reordering, the translator uses:

- Register renaming[4] for anti-dependencies elimination and for atomicity (state recovery) support.
- Control and data speculation based on hardware speculation mechanism and software techniques, as detailed in Section 4.

The translator builds recovery information that is used in case of exception or interruption. Sideways ILs are scheduled outside the main block unless they can be incorporated efficiently in one of the bundles of the main trace.
Finally, the encoded block with its information is placed in the translation cache and connected to its predecessors.
Overall, hot code translation overhead per IA-32 instruction is about 20 times more than cold-code translation overhead per instruction.

3. Interaction with the Operating System (OS)

As an application-level binary translator, IA-32 EL runs above the 64-bit OS in the application program's virtual space and privilege level. Currently IA-32 EL supports Windows and Linux OSes for Itanium-based platforms. The configuration capabilities of these OSes allow assigning IA-32 EL as the execution vehicle for 32-bit applications, which makes it completely transparent for the end user. Once loaded and initialized, IA-32 EL gets control from the OS in order to run the 32-bit code of the application within the same virtual address space. IA-32 EL uses the native OS for multiple functions, such as acquiring system resources (memory, synchronization objects, etc.), executing 32-bit system calls issued by the IA-32 application, signal handling, exceptions and other system notifications.
To simplify re-hosting of IA-32 EL on multiple OS platforms and to reduce its validation cost, IA-32 EL was architected as two components: a major OS-independent component called BTGeneric and a thin (about 1% of the IA-32 EL image) OS abstraction layer called BTLib.
The OS interface is made via the OS abstraction layer only, which is the only OS-specific component of IA-32 EL. In other words, any system request from a BTGeneric component goes through an OS-generic interface to BTLib, which in turn passes it to the underlying OS. Each OS requires its own implementation of BTLib to be supported by IA-32 EL.
BTGeneric is implemented in a separate binary module. The same binary module format is used for all platforms. BTLib loads the BTGeneric module at application launch time.
The interface between BTLib and BTGeneric (BTOS API) is defined on the binary level and excludes any compiler and OS dependences. This is a bi-directional protocol, implemented partly by BTLib and partly by BTGeneric. For example, as shown in Figure 3, when the translation engine needs a memory block to store translated code, it calls a BTOS API function for memory allocation. This function, implemented by BTLib, redirects the memory allocation request to the corresponding OS function. On the other hand, when an exception occurs in the translated code, the OS calls an exception handler in BTLib, which in turn calls a BTOS API function implemented by BTGeneric and requests a consistent IA-32 state corresponding to the faulty instruction.
Special attention was devoted to versioning control between the two components: BTLib and BTGeneric. Taking into account future modifications and extensions to BTOS API, as well as backward compatibility issues, IA-32 EL uses its proprietary protocol to ensure that BTLib and BTGeneric versions match each other.

4. Precise IA-32 State Support without Performance Penalty

A single IA-32 instruction is usually represented in the translated code by more than one Itanium instruction. However, if an exception occurs, it becomes necessary to regenerate a consistent and precise IA-32 state for the point of exception, based on the actual Itanium architecture state at that point. This IA-32 state is required for at least two reasons:

- For proper exception/unwinding handling by the OS.
- When execution resumes from the start of the IA-32 instruction, some or all of IA-32 state may have been changed by the exception handler.

The problem is more complex for optimized (hot) code. In hot code, instructions originating from the translation of different IA-32 instructions are inter-mixed. Other works [14,15] focus on lazy state reconstruction using different techniques. This

[4] IA-32 Execution Layer allocates the entire 96-register stack. The translated code operates in the same frame except for very rare function calls.

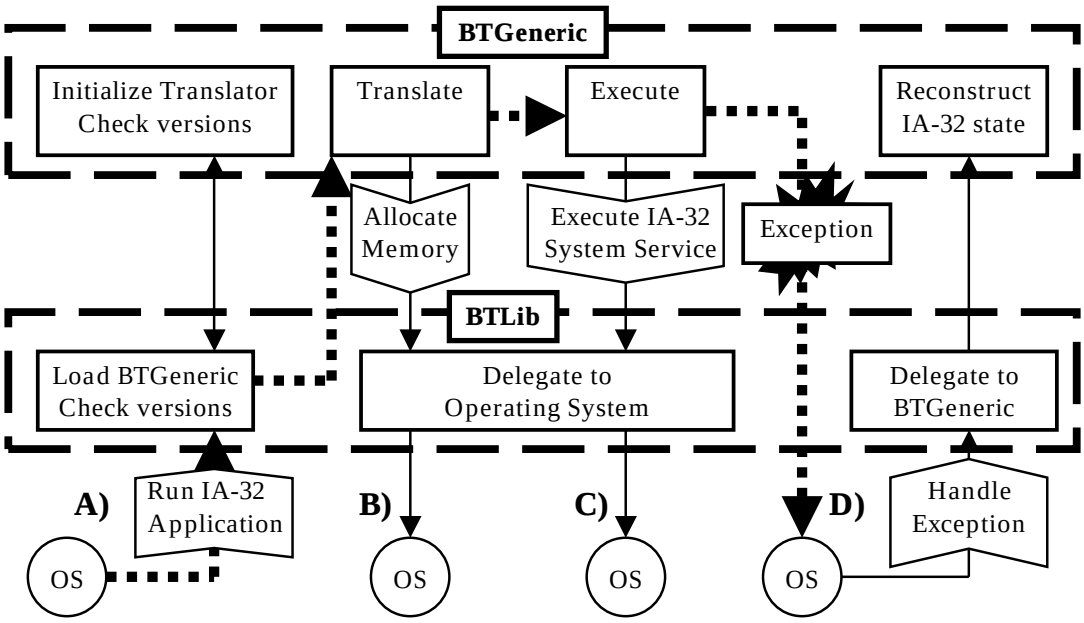

Figure 3: Interaction of IA-32 EL with the Operating System during A) process initialization, B) translation, C) execution of an IA-32 system call, D) exception handling.

section describes how IA-32 EL maintains atomicity and precise exceptions in cold and hot code.

Exception Handling

During execution of the translated code, all exceptions are native (IPF) code exceptions. These exceptions are different than in the IA-32 code: the IP is different; the exception code can be different; and the registers involved are 64-bit registers. Once an exception is raised in the translated code, the OS calls IA-32 EL before passing the exception notification to the IA-32 application itself. IA-32 EL filters out exceptions that refer to the translated code and converts the Itanium architecture state information into the corresponding IA-32 state. The resulting state is then used to simulate the corresponding IA-32 application exception handler. As part of this first-time handling, exception code may be modified by the handler to match the exception that should have occurred in the IA-32 code. In some cases, the exception must be nullified, or prevented from further escalation to the IA-32 application exception handler, because no IA-32 exception should have occurred at this point in the original code.

One such example is where the original code is running with masked FP exceptions and the translated code requires unmasking the exceptions to support SSE unmasked exceptions[5].

IA-32 State Reconstruction for Cold Code

Cold code translation needs to guarantee that, on any exception, the original IA-32 state can be generated correctly. To achieve that, each IA-32 instruction is translated in such a way that its IA-32 state change happens only after executing the last Itanium instruction that can fault e.g. memory and floating-point instructions. Consider the pseudo-code in Table-1.

In addition, at the beginning of every such sequence, IA-32 EL saves the IA-32 IP and some additional information into a dedicated 64-bit register, called the "IA-32 state register". Upon receiving an exception and locating its source in the cold code, the IA-32 state register is used to map to the IA-32 IP. IA-32 state information is readily available in their "canonic" locations, since no IA-32 state update occurs until the last potentially faulty instruction has been executed. Note that this process is not needed for non-faulty IA-32 instructions. The overhead of IA-32 state register updates is negligible both in terms of time and code

[5] IA-32 supports separate masking for FP and SSE code, unlike Itanium architecture.

	Correct	Incorrect
Push eax	add r.addr = -4, r.esp;; st4 [r.addr] = r.eax mov r.esp = r.addr	add r.esp = -4, r.esp;; st4 [r.esp]= r.eax

Table 1. In the correct code, r.esp is updated after the memory store is updated. In the incorrect code, r.esp is updated before the memory store.

size.

IA-32 State Reconstruction for Hot Code

Reconstructing the IA-32 state in hot code is a significantly bigger challenge for two reasons:

- In hot code, Itanium instructions originating from different IA-32 instructions are usually inter-mixed, and the IA-32 state (registers) is often represented by other registers (for example, in case of register renaming). As a result, exceptions in hot code may appear in an incorrect order and redundant exceptions may occur.
- Hot blocks are composed of several IA-32 basic blocks and may contain branches, loops, and predicated `if then else` code sequences.

In order to support precise IA-32 state restoration and at the same time aggressively reorder instructions to produce optimized hot code, IA-32 EL uses *commit points*. As opposed to a faulty instruction, a commit point is not an Itanium instruction, but a "barrier" enabling the translator to generate a consistent IA-32 state. The translator then does the following:

1. Associates several faulty points in the code with a single commit point.
2. Limits reordering between instructions belonging to different commit points.
3. Requires that, within a single IA-32 instruction translation, the state update occurs after the last faulty IPF instruction, similar to cold code case.

To minimize the impact of these limitations, IA-32 EL associates as many faulty instructions as possible to a single commit point[6]. The first commit point is usually set at the beginning of the block, and is replaced only when the translator encounters an irreversible faulty instruction (memory store or branch), or can no longer preserve IA-32 state elements. The translator copies the original IA-32 state changes into backup registers.

Using commit points makes it possible to aggressively reorder the instructions, because providing a consistent IA-32 state is required only at the very last faulty instruction in a group that refers to the same commit point. If an exception happens on any instruction that is located earlier, the translator ignores (nullifies) it and rolls forward until the last instruction in the group is reached.

[6] The translator sets one commit point per 10 native instructions on average.

5. IA-32 Specific Optimizations

IA-32 architecture has some unique characteristics that require special handling in order to achieve high performance, especially in areas where the target Itanium architecture differs significantly from IA-32 architecture.

IA-32 FP, Intel® Streaming SIMD Extensions, and MMX™ Technology Emulation

The IA-32 architecture and the Itanium architecture have a different floating point and MMX technology model. IA-32 FP instructions refer to eight 80-bit registers organized in an FP stack (see Figure 4), and the SSE instructions refer to eight 128-bit XMM registers. The Itanium architecture supports both FP and SSE instructions using a flat register file of 128 82-bit registers. The IA-32 FP stack may contain empty or valid entries represented in a TAG register. All addressing of the FPU data registers is relative to the register on the top of the stack. The register number of the current top-of-stack register is stored in the TOP of Stack (TOS) field in the FPU status word. Load operations decrement TOS by one and load a value into the new top-of-stack register [ST(0)]. Store operations store the value from the current ST(0) register into memory and optionally increment TOS by one. Most operations access ST(0) as a source

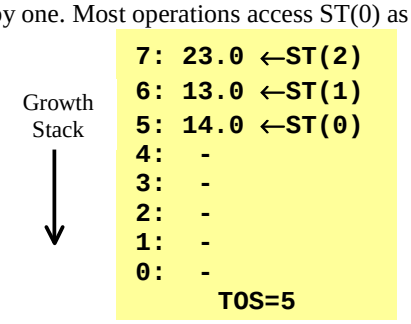

Figure 4. IA-32 FPU Stack

and destination operand. As a result, ST(X) is arotating register that is determined by TOS value.In addition, the eight IA-32 64-bit MMX registers are aliased to the significands of the eight FP stack registers in their initial positions. On Itanium processors, MMX instructions operate on integer registers. The differences between the architectures pose specific optimization challenges to the translator. This section focuses on three specific challenges: emulating the FP-stack, aliasing FP to MMX registers, and handling different XMM data formats. For more details, see [12, 13].

FP stack emulation is challenging for two reasons. First, mapping a rotating stack into static registers can result in many move operations whenever the TOS changes. An alternative approach [6] of modeling the FP-stack in memory has a high overhead because of the memory accesses. The second difficulty is the need to check the TAG word for each FP register access, in order to raise the appropriate FP-stack fault in case of a read or write from an invalid entry.

The FP aliasing to MMX registers is difficult because the implementation of native register aliasing requires moving a value from an FP to an integer register (which models the corresponding MMX register) following every FP operation, and from an integer to FP register following every MMX instruction. Theses moves are extremely expensive, so another solution is required.

The IA-32 instructions dealing with XMM registers may use four different data formats: packed-integer, packed-single, packed-double, and scalar. The corresponding representations in pairs of Itanium processor FP registers require conversions when moving between formats. Forcing each block to convert its eight XMM simulated registers into some arbitrary “canonical” format at the entry and exit of the block is very expensive. So a low overhead method is also needed in this case.

IA-32 EL uses the following scheme to speed up such cases: Some speculative assumptions (listed below) are made in the body of each block, which enable aggressive optimizations. At the block’s head, a check is done to validate the assumptions made when the block was translated. If the check fails, the translator jumps to a correction code. Status updates at the end of the block enable the next blocks to carry their checks.

For the stack emulation problem, IA-32 EL uses speculative assumptions that the TOS remains constant for all entrances to the same block, and that no stack exceptions occur. The block body translation benefits, since the mapping to Itanium processor FP registers is fixed, with no rotations or memory overhead. The block head compares the actual TOS to the speculated one, and compares the TAG values to those required to keep stack operations correct. Static analysis at translation time marks stack entries that can be either empty or valid.

When mismatches are detected, these recovery actions are taken:

- On TOS mismatch, rotate register values.
- On TAG mismatch, rebuild a special block to catch the right stack fault.

The speculation success rate observed in this case is excellent (99-100%). Compiled code in most cases maintains the same TOS and TAG at the entrance of a block.

For the FP to MMX register aliasing problem, the speculative assumptions are as follows:

1. If the block contains MMX instructions, the last executed FP or MMX instruction prior to entering the block was an MMX instruction.
2. If the block contains FP instructions, the last executed FP or MMX instruction prior to entering the block was an FP instruction.

Hence no integer-to-FP or FP-to-integer moves occur in the block itself. A single Boolean value check at the block head is enough to detect mismatches.

When the check fails, the recovery code copies FP values to MMX registers or vice versa, and toggles the Boolean value.

The speculation success rate observed in this case was also very close to 100%: MMX instructions and FP instructions are usually not mixed within the same computation area.

For the multiple SSE formats problem, hot blocks try to adjust their input/output formats to each other, according to the order in which they were generated. The speculative assumption is that the format set by the previous block is the same as the one used by the current block, hence no further conversions are required in the beginning of the block. The block head compares the required formats for all XMM registers with the current run-time status. If the check fails, the code exits the block to perform the relevant conversions. Again, the speculation success rate for this case is fairly good - only less than 0.2% operations required conversion as observed in the worst case among SPEC2000 benchmarks.

FXCHG elimination is an optimization specific to the FP stack. The common IA-32 operations on FP stack are limited to the stack top. That forces the IA-32 compilers to do a lot of fxchg operations - swap two stack values. The limitation does not exist in the IPF register file; so IA-32 EL can handle the

fxchg as implicit renaming instead of generating copying instructions.

Handling Misaligned Data Accesses

The penalty for misaligned data accesses on the IA-32 architecture is very low. Consequently, many IA-32 applications contain a significant number of misaligned data accesses. On the other hand, the penalty for misaligned data accesses on the Itanium architecture is very high, especially in cases where misalignment is handled by the OS rather than the hardware – such handling takes on the order of several thousand cycles. Taken together, these two facts lead to the conclusion that execution of an IA-32 application on IA-32 EL, without avoiding data misalignments, can cause a significant performance cost. The same conclusion was drawn for FX!32 [8]. On some applications, data misalignment detection and avoidance performed by IA-32 EL contributed to a significant speedup. For example one workload that initially took 1236 seconds to complete, completed after 133 seconds when adding misalignment detection and avoidance. This section describes the data misalignment detection and avoidance mechanism used by IA-32 EL.

A simple method for detecting and avoiding data misalignments is to generate each data access with a test of the data address. In the case that the address is misaligned with respect to the access size, the access is done in parts. The following code shows an example of a two-byte load using this method:

```
// test bit0 to see if address is
// 2byte aligned. Predicates p.mis
// and p.al set appropriately.
// Will use p.mis and p.al to predicate
// the following instructions
tbit p.mis,p.al     = r.addr, 0
// 2 byte load if aligned
(p.al)  ld2 r.val   = [r.addr]
// if misaligned load each byte separately
(p.mis) ld1 r.val   = [r.addr]
(p.mis) add r.addrH = 1, r.addr
(p.mis) ld1 r.valH  = [r.addrH]
// combine the separately loaded bytes
(p.mis) dep r.val   = r.valH, r.val, 8, 8
```

While this method avoids the misalignment penalty, it incurs significant overhead. Note that the FALSE predicated instructions do consume cycles. A method that is low in overhead and high in coverage was needed.

The method used in IA-32 EL consists of three stages:

1. Initially in cold blocks, all instructions that may have a misaligned access are lightly instrumented so that, if there is a misaligned access in the block, it branches out to the translator and the block is regenerated. Note that the instrumentation in this stage does not provide information on which specific access was misaligned.

2. Regenerated cold blocks detect and avoid misalignments. They are more heavily instrumented to provide detailed misalignment information: which instructions had misaligned accesses and the type of misalignment. (For example, for 8-byte accesses, the translator indicates if the misalignment was of 1-byte or 4-byte granularity.) This enables a shorter misalignment avoidance sequence in hot blocks. The avoidance is achieved by generating possibly misaligned data accesses in the manner described in the 2-byte load example above.

3. During hot code generation, the information from cold code is examined for each of the cold blocks that make up the hot block. Each instruction that is marked as misaligned is generated to detect and avoid misalignment, much in the manner described in the 2-byte load example above, but with some enhancements as follows:

   a. The addresses for which misalignment detections were generated in the hot block are tracked. If the current address of an access needs misalignment detection, and the address is equivalent to, with regard to misalignment, an address for which detection has already been done, the result of the earlier detection is used. (In the 2-byte load example, IA-32 EL uses the predicates that were set in the previous t-bit instruction.)

   b. The sequence of code that implements the access when the address is misaligned may be quite long. In this case, the scheduler moves all or part of these instructions outside the translated block code. (They will be branched to if needed and, after their execution, a branch will be done back into the block.) This is similar to how the scheduler handles sideway instructions.

The mechanism described so far does not handle behavior changes that occur after the hot block is generated. Such changes are observed on some applications. To cope with misalignments that appear only after the optimized code is generated, the following actions occur in stage 3: Each instruction for which no misalignment has been observed, but is empirically considered to have significant danger of incurring misaligned access later on, is instrumented. The instrumentation is very light in this case. If a misalignment is detected, the block branches into the translator. IA-32 EL identifies the hot block in which the misalignment occurred. The identified block is discarded and information is recorded to specify that, when the hot block is regenerated, all such instructions

should be generated with misalignment detection and avoidance.

6. Performance Results

This section presents the performance results of IA-32 EL translator. The measurements were performed on a 1GHz Itanium® 2 processor with 3MB L3 cache and 2GB RAM. The results in this section refer to SPEC CPU2000* benchmarks [20] and Sysmark* 2002. The IA-32 binaries were compiled using the Intel® C++ Compiler 6.0 [19].

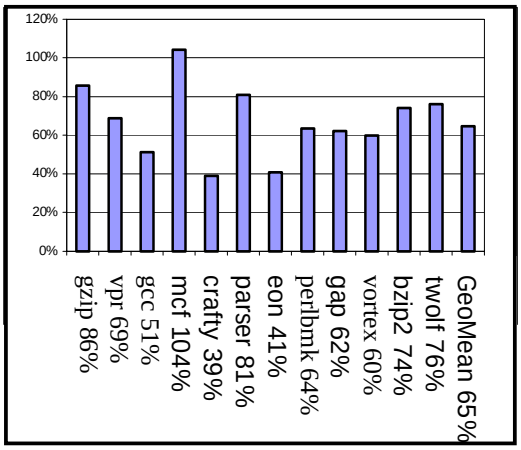

Figure 5 - SPEC CPU2000* int scores for IA-32 EL compared to native Itanium® processor performance. Native performance is equal to 100%.

Figure 5 shows the relative SPEC CPU2000 score compared to native execution (higher is better) of highly optimized binaries generated by the Intel compiler for the Itanium 2 processor [19]. IA-32 EL reaches performance level of 65% of the native performance on integer benchmarks. On mcf, performance is slightly higher than native performance due to the much smaller data footprint of the IA-32 version that use 32 bit data items as opposed to 64 bit data items used by the native version.

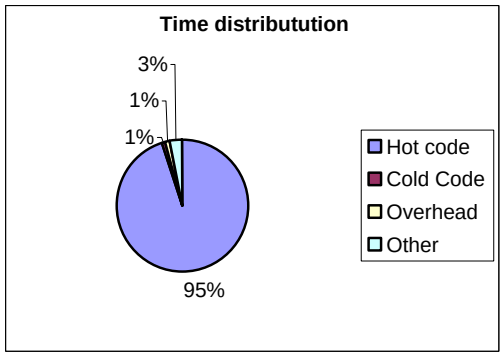

Figure 6 - Execution time distribution for translated SPEC CPU2000* applications.

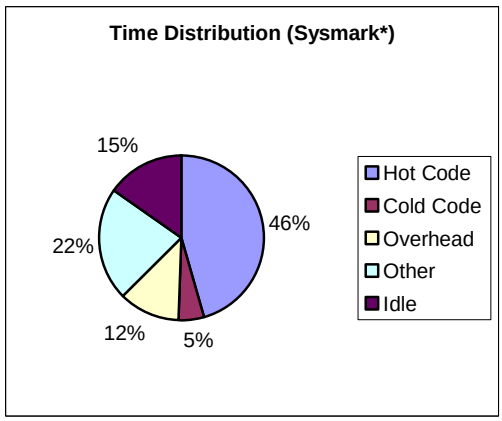

Figure 7 – Same for Sysmark 2000*.

Figure 6 shows the average (and typical) distribution of execution time for SPEC CPU2000 applications. Note that hot trace selection was accurate, accounting for 95% of the execution time. The speedup is the result of running relatively efficient cold code, generated with minimal overhead, for long period of time with instrumentation. Translators using interpretation in the first phase need to move to hot code generation much earlier, and thus potentially collect less representative data. The accuracy of the instrumentation is especially critical for misalignment elimination. On such benchmarks, the hot code performance is 3X better than cold code, providing another indication of the accuracy of the hot traces selection and of the high potential of hot code optimizations.

Figure 7 shows the average execution time distribution of Sysmark 2000 applications. The profile of the applications in this benchmark is different than SPEC CPU200 applications. The Sysmark 2000 applications are much bigger and their execution is spread more evenly. As a result

only 45% of the executed code is hot vs. 5% cold. The amount of translated code has an impact on the overhead. Notice that in this benchmark the applications spend a significant amount of time in the OS kernel and drivers. This code is not translated, executes natively on the machine, and is shared between translated and native code. Spending a significant amount of time in the kernel and drivers is a typical behavior of many Windows applications. Running that code natively on the Itanium architecture contributes to the performance. The idle time is large – 15% on average. This provides an opportunity for future work on utilizing this time and reducing the translation overhead.

Figure 8 compares IA-32 EL performance with the performance on an IA-32 platform. The result of the floating-point benchmarks should be especially noted, taking into account the floating point modeling challenges described. The excellent, native floating-point performance of the Itanium processor family is a key contributor, together with the floating-point model and optimizations done in IA-32 EL.

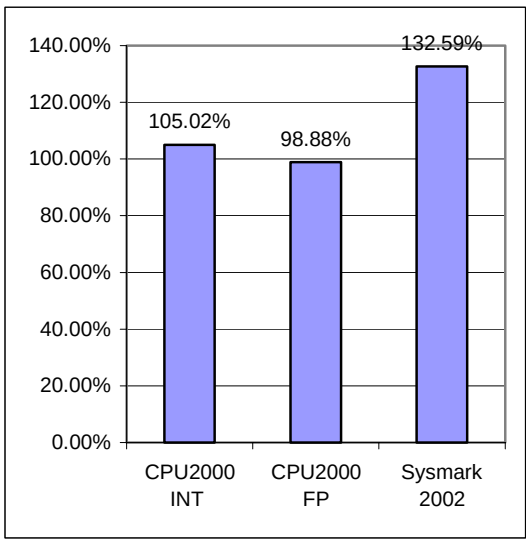

Figure 8 - Relative performance of IA-32 EL running on a 1.5GHZ Itanium® 2 processor compared to a 1.6GHZ Xeon™ processor (higher is better). Sysmark* 2002 numbers refer to the Internet Content Creation part of the benchmark.

Related Work

Several recent publications, e.g. [4, 7], included a thorough review of other works related to dynamic binary translations: dynamic translators, dynamic optimizers, and related developments in hardware. A general classification of binary translators can be found at [2].

Other products in the field are Transmeta CMS [7], FX!32 [6], HP Aries [23], and the various Java and MRTE JIT translators; additional research frameworks are Dynamo [3,4] and DAISY [9].

Transmeta CMS is a dynamic translator that emulates IA-32 on a VLIW HW that is not exposed. It runs beyond the IA-32 OS and hence emulates the entire architecture with full computability. FX!32 is a dynamic-static hybrid translator from IA-32 to Alpha architecture, with a lower level of compatibility (e.g., FP double precision emulation). HP Aries translates PA-RISC binaries to IPF, above HPUX* OS. Its hot-spots are optimized at the basic-block level.

Optimizers like Dynamo, which translate to the same ISA, try to generate code that outperforms the originally compiled code. Translating to the same ISA, Dynamo can "bail-out" to native execution whenever the optimization turns out as ineffective. "Bailing out" is inapplicable for binary translators.

Conclusions

This paper presents the underlying technology of the IA-32 Execution Layer, a dynamic binary translation from IA-32 to IPF. Emphasis was given to some of the key features of the technology that contribute to its robustness and high performance. These features include precise exception implementation in the software, OS-independent architecture, floating point, MMX technology, and SSE modeling, and misalignment handling.

Acknowledgment

Thanks to all IA-32 EL team members, current and past ones, who took IA-32 EL from vision to reality. Special thanks to Ronny Ronen for his numerous comments and review of the paper. Thanks to Evelyn Duesterwald for her initial feedback on the draft. Special thanks to Laura Cane and Maggie Auerbach for reviewing and bringing the paper to its current shape.

References

[1] Eric R. Altman, Kemal Ebcioglu, Michael Gschwind and Sumedh Sathaye, *"Advances and Future Challenges in Binary Translation and Optimization"*, Proceedings of the IEEE Special Issue on Microprocessor Architecture and Compiler Technology, November 2001.

[2] Eric R. Altman, David Kaeli, and Yaron Sheffer, *"Welcome to the opportunities of Binary Translation"*, IEEE Computer 33(3), March 2000.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banjeria, *"DYNAMO: A Transparent Dynamic Optimization System"*, Programming Language Design and Implementation, June 2000.

[4] Derek Bruening, Timothy Garnett, and Saman Amarasinghe, *"An Infarstructure for Adaptive Dynamic Optimization"* in the Proceedings of the International Symposium on Code Generation and Optimization, 2003.

[5] Howard Chen, Wei Chung Hsu, Jiwei Lu, Pen Chung Yew and Dong Yuan Chen, *"Dynamic Trace Selection Using Performance Monitoring Hardware Sampling"* in the Proceedings of the International Symposium on Code Generation and Optimization, 2003.

[6] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavall and John Yates *"FX!32: A Profile-Directed Binary Translator"*. IEEE Micro(18), March/April 1998.

[7] Dehnert, J.C.; Grant, B.K.; Banning, J.P.; Johnson, R.; Kistler, T.; Klaiber, A. and Mattson, J., *"The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges"* in the Proceedings of the International Symposium on Code Generation and Optimization, 2003.

[8] Paul J. Drongowski, David Hunter, Morteza Fayyazi, David Kaeli, *"Studying the Performance of the FX!32 Binary Translation System*", in the Proceedings of the 1st Workshop on Binary Translation , Newport Beach, CA, Oct. 1999.

[9] Kemal Ebcioglu and Erik R. Altman **"***DAISY: Dynamic Compilation for 100% Architectural Compatibility"*, Proceedings of the 24th Annual Symposium on Computer Architecture, June 1997.

[10] Kemal Ebcioglu, Erik R. Altman**,** Michael Gschwind and Sumedh Sathaye, **"***Dynamic Binary Translation and Optimization"*, IEEE Transactions on Computers 50(6), June 2001.

[11] Kim M. Hazelwood and Thomas M. Conte, *"A Lightweight Algorithm for Dynamic If-Conversion During Dynamic Optimization"*, International Conference on Parallel Architectures and Compilation Techniques, October 2000.

[12] Intel Corporation,*"Intel IA-32 Architecture Software Developer's Manual"*, Vol. 1-3 2003.

[13] Intel Corporation,*"Intel IA-64 Architecture Software Developer's Manual"*, Vol. 1-4, January 2000.

[14] Michael Gschwind and Eric R. Altman,*"Optimizing and Precise Exceptions in Dynamic Compilation"*, Second Workshop on Binary Translation Held in PACT 2000.

[15] Michael Gschwind and Eric R. Altman, *"Precise Exception Semantics in Dynamic Compilation"*, in the Proceedings of the Symposium on Compiler Constructions, April 2002.

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

[17] S.J. Patel and S.S. Lurnetta. *"RePlay: A Hardware Framework for Dynamic Program Optimization"*, Technical Report CRHC-99-16, University of Illinois, December 1999.

[18] R.L. Sites, A. Chernoff, Kirk, M. Marks, and S. Robinson, "*Binary Translation*," Comm. ACM 36 (2), Feb. 1993.

[19] Intel compilers http://www.intel.com/software/products/compilers/

[20] SPEC CPU2000 http://www.specbench.org/osg/cpu2000

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

[22] David Ung and Cristina Cifuentes, "*Optimizing Hot Paths in a Dynamic Binary Translator"*, Second Workshop on Binary Translation Held in PACT 2000, October 2000.

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