Studying the Performance of the FX!32 Binary Translation System

Paul J. Drongowski
David Hunter
Alpha Migration Tools
Compaq Computer Corporation
Littleton, MA 01460-1289
paul.drongowski@compaq.com

Morteza Fayyazi
David Kaeli
Northeastern University
Department of Electrical and Computer Engineering
Boston, MA 02115, USA
mfayyazi,kaeli@ece.neu.edu

Jason Casmira
University of Colorado
Department of Computer Science
Boulder, CO
casmira@cs.colorado.edu

Abstract

For an emulation/translation environment to attract a significant user base, the environment must provide reliable and efficient application execution. The issues of correctness and performance are fundamental to this environment, yet many times these goals are in conflict. While maintaining correctness is not optional, optimal performance is difficult to always achieve. In order to detect and remedy performance issues, a range of analysis tools are needed.

In this paper we describe three tools that are used to study the performance of the Compaq FX!32 binary translation system. We discuss these tools and illustrate how they are used to identify various performance or workload issues related to translation, emulation and optimization.

1 Introduction

Compaq's FX!32 is a runtime system which uses binary translation to support the execution of x86 Win32 applications on the Alpha/Windows NT platforms. Translation and execution of applications is transparent (i.e., no special commands are needed to translate and run an x86 application). The user is quite unaware of the underlying technology. This transparency will be further enhanced with the introduction of Windows 2000 as FX!32 technology is being embedded within the Windows 2000 operating environment.

Two key technical goals are critical to the success of a commercial binary translation system: correctness and performance. Users have very little tolerance for software bugs that interfere with their ability to do useful work. Binary translation and its supporting environment introduce an additional layer of complexity and mechanism between the application and the native operating system. This layer must be faithful to the behavior of the target while remaining compatible with the host. The job of insuring compatibility and correctness of the FX!32 system with both the Alpha and x86 architectural definitions is no small task.

When we talk about performance in the context of binary translation, we are referring to both the speed of translation and the speed of the translated application. FX!32 uses off-line, profile-directed binary translation. By performing analysis and optimization off-line, more rigorous code analysis and translation can be performed. Profile information is collected during first execution of the application code and this profile is used during translation to guide code identification and generation. This process feeds information about the dynamic behavior of the application forward into translation. This strategy helps FX!32 to achieve performance comparable to a 200-MHz Pentium or a 200-MHz Pentium Pro when executing translated x86 code on a 500-MHz Alpha host [7].

By performing translation off-line, performance of the translator becomes less important than, say, the execution speed of the application. However, the translator employs compiler-like algorithms for data flow analysis, register allocation, and instruction scheduling which are sensitive to the size of a translation unit (e.g., a subroutine.) Translation time must be kept tolerably short.

Correctness and performance are often competing issues. Although many applications are built using a C compiler and the resulting code reflects a C-flavored runtime architecture, the emulation and translation system must be prepared to deal with ill-behaved programs that do not conform to an ideal, uniform runtime architecture. Decisions are made in favor of correctness and occasionally this bias places constraints on performance.

This paper discusses the tools and methods used to study and improve the performance of the FX!32 emulator/translator and execution of translated code. We will begin by briefly describing the architecture of FX!32. Next, the tools used for performance analysis will be described followed by some example scenarios of those tools in action. The paper concludes with a summary of these tools and some suggestions for future research in performance tooling.

2 Overview of FX!32

Architecturally, FX!32 (see Figure 1) consists of seven main components:

1. Transparency Agent
2. Loader
3. Runtime and Emulator
4. Translator
5. Database
6. Server
7. Manager

The Transparency Agent resides within a Windows NT process and detects any attempt to load and execute an x86-architecture program image. If such an attempt is detected, the FX!32 Loader will be invoked to load the x86-architecture image and an Alpha-architecture, translated image (if one exists.) The Emulator is used to interpret x86 instructions when a corresponding Alpha translation does not exist. The FX!32 Runtime provides *jacketed* interfaces to native Alpha library routines and callbacks.

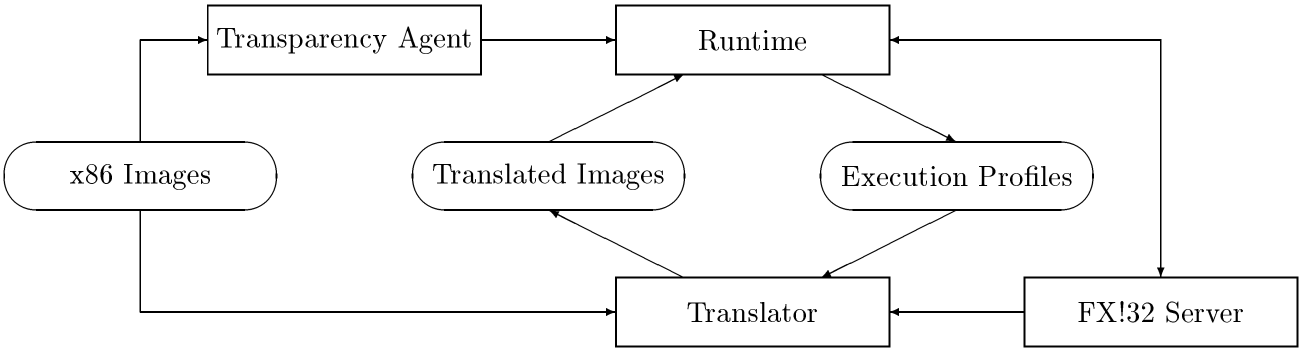

Figure 1: Information flow between FX!32 components

An x86 application being run for the first time is fully emulated. The FX!32 Emulator produces a profile file for each x86 image loaded and invoked by the application. The profile file and the x86 image are submitted to the Translator which produces a native Alpha executable image. The x86 image resides in the directory in which it has been installed while the profile file and translated Alpha image reside in the FX!32 Database. A portion of the Database is stored in the system Registry. The Registry portion of the Database retains information about the images in the application, their location in the file hierarchy, translation status, etc..

The FX!32 Server is the active component that manages work products and invokes the Translator. The Manager is a GUI-based tool which lets the user schedule translations for off-hours and display the status of applications and translations. For a more complete description of FX!32 and its components, the reader is referred to [7, 12].

Profile files contain three kinds of information about the dynamic execution of a program image:

1. addresses that are targets of CALL instructions,
2. source address and target address pairs for indirect jumps and subroutine calls, and
3. addresses of instructions making unaligned references to memory.

The first two kinds of information are used by the Translator to identify code within an x86 image to be translated. Thus, the Translator only translates those portions of the image which have been executed via emulation. Profile information is accumulated by merging execution profile files as they are created. Thus, the portion of an image which is translated will grow with use. For *feature rich* applications, this strategy tends to keep down the size of translated images.

```
...

                        ; Aggressive
ldl    r1, x(r11)       ; Long-word load (aligned address)

...

                        ; Conservative
ldq_u  r1, x(r11)       ; Long-word load (unaligned address)
ldq_u  r2, x+3(r11)
lda    r3, x(r11)
extll  r2, r3, r1
extlh  r2, r3, r2
or     r2, r1, r1
```

Figure 2: Aggressive vs. conservative loads

The third type of profile information is used to identify instructions that make unaligned memory references, and is used to guide the generation of Alpha memory reference instructions. Alpha is a load/store RISC architecture which rewards code for making naturally aligned memory references. A native compiler working from source can align data items for maximum speed through aligned reads and writes. x86-architecture programs can and do make memory references that are not naturally aligned with respect to the Alpha memory model. Further, the FX!32 translator does not attempt to realign application data structures, so the Translator uses an *aggressive* approach to memory references. It will generate Alpha code for the faster aligned access unless a profile record indicates that an x86 instruction has made an unaligned access. The Translator will then produce the *conservative* Alpha code to provide for an unaligned memory reference. Figure 2 illustrates the aggressive (aligned access assumed) and conservative (unaligned access assumed) Alpha code for a 32-bit load from memory [21].

3 Performance Analysis for Binary Translation

Performance analysis requires a fundamental understanding of the system architecture. This knowledge guides the questions to be asked, the information to be collected and the interpretation of performance data. It is also needed to identify and isolate performance problems to system components or perhaps problems at the interface between two components.

Although the number of large-grained architectural components in FX!32 is relatively small, the interactions and relationships between those components are complex. Understanding these interactions and relationships, and using that information for performance analysis is a challenge.

Performance analysis is often applied at different levels of granularity or abstraction. In the case of the Translator, for example, it is important to study the overall execution time and memory requirements in order to detect any translations that take an excessive amount of time, or consume too much memory. Once a problem has been detected, problem resolution is performed hierarchically. In practice, an initial triage is performed to find the likely cause of the problem followed by a push down into the underlying mechanisms.

In a mature system like FX!32, much analysis is devoted to the identification of new opportunities for optimization, and the isolation and repair of existing performance issues. Some key factors in the performance of translated code include:

- the portion of the application which is running translated versus running emulated,
- alignment fix-ups,
- cache conflicts, and
- code quality.

FX!32 runs as part of a standard Windows NT application and in some sense, the analysis of its performance is no different than the analysis of any other complex program. FX!32, however, is an excellent example of a performance-critical program. The sections which follow describe the tools which we have found useful in the analysis of FX!32 and give real-world scenarios for their use.

3.1 Tools

Windows NT supplies a set of basic performance measurement tools such as *Perfmon* to monitor system, process and thread events in real-time [16]. Compaq also has an internal tool, *fix-ups*, which is helpful for isolating the location of Alpha instructions that make unaligned references to memory.

Over several years, Compaq has also invested in the development of several tools for the analysis and modification of programs. These tools include:

- DCPI: Execution profiling and analysis [1],
- PatchWrx: Executable tracing [4, 15], and
- NT-ATOM: Execution-driven simulation [3].

Next we will describe these program analysis tools available for the Compaq Alpha architecture.

3.1.1 DCPI

The Compaq Continuous Profiling Infrastructure (DCPI) is a tool for Alpha platforms that provides continuous profiling of entire systems, including user application code, libraries, drivers and operating system kernel [1]. DCPI samples the Alpha hardware performance counters and collects execution profile information in its own database. The performance event counters on Alpha 21164 processors are extensive, measuring CPU cycles, D-cache misses, I-cache misses, branch mispredicts, CPU stalls, etc.. DCPI provides a set of analytical tools to read, analyze and display profile information. DCPI can, for example:

- summarize the distribution of cycle and cache miss events by image,

- produce a disassembly of an image with cycle and event annotations, and
- suggest reasons for CPU stalls and other code-level performance problems by analyzing event counts with respect to a model of the Alpha hardware.

DCPI has proven to be a trustworthy, low-overhead, tool for the isolation of coarse grain performance problems as as well as code-level problems.

3.1.2 PatchWrx

PatchWrx is a static binary-rewriting instrumentation tool for capturing full instruction and data address traces on the Compaq Alpha platform running Windows NT [15]. The toolset modifies the binary image prior to execution. PatchWrx captures execution of the operating system and DLL's, as well as the application. Traces are generated as an application is run. Trace records are captured when the modified executable encounters a modified instruction at run time.

PatchWrx modifies a binary image by identifying all branches, loads and stores in the executable image, and then *patches* these instructions, turning them into unconditional branches into a *patch section*. For branches, if the current instruction is either a BSR or JSR, then a BSR is used to branch to the patch section (this is necessary in order to preserve the return address register field). Otherwise, a BR instruction is used to effect the branch to the patch section. The patch section is additional code generated by PatchWrx, that is appended to the end of the executable. The patch section issues a *PALcall*[1] (the type of call depends on the type of instruction that was patched), collecting the trace log entries which are later stored in the trace buffer. After a trace entry is stored to the buffer by the PALcode, the PALcode returns to the patch section at an address where a copy of the original instruction resides. The original instruction is executed and control branches back to the branch target, if the original instruction was a taken branch, or to the next sequential instruction, if the original instruction was a not-taken branch or a load/store.

When we need detailed instruction-level information about FX!32, we can obtain traces using PatchWrx. We can generate a complete instruction trace that can then be used as input to an architectural evaluation that is considering the runtime performance of FX!32.

3.1.3 NT-ATOM

NT-ATOM is a framework based upon Compaq's original *TRU64 Unix ATOM* tool [11] and operates under the Windows NT for Alpha operating system. NT-ATOM leverages previous efforts of another Compaq project called SPIKE [10], to handle the manipulation of Windows NT images. A more complete description of NT-ATOM, its interaction with SPIKE, and the variances between it and the original ATOM can be found in [3].

Like its predecessor, NT-ATOM allows the instrumentation of executables and dynamic linked libraries (DLL) using a straight-forward application programming interface (API). Instrumentation can

[1] The Privileged Architecture Library (PAL) facility of the Alpha is a microcode-like set of routines that are specific to a particular Alpha operating system implementation.

be performed at a program, procedure, basic block or instruction granularity. New procedure invocations are introduced at each of these specified instrumentation points in the original program. When the program is executed, the introduced procedures pass various data from the executing program (e.g., pc address, register values, registers used, opcodes, etc.) to an analysis procedure.

Using the API's provided with NT-ATOM, a variety of different tools have been built to study the performance of applications. Significant in these tools are those that allow the capture of application-level instruction and data traces. These dynamic traces are then used to facilitate execution-driven simulations. The normal distribution contains several sample tools for analyzing the dynamic behavior of applications such as: instruction execution profiles, branch prediction accuracy, cache simulation and procedure execution profiles.

4 Studying Binary Translation

Each of these tools has specific strengths that we utilize to study FX!32 performance. The following scenarios show how some of these tools can be used in practice.

4.1 Alignment fix-ups

Compaq measures and tracks the effectiveness of FX!32 on certain benchmarks such as the Byte CPU benchmark (Bytecpu) and Bapco Sysmark32, and on key customer-oriented applications such as Adobe Photoshop. This provides us with a historical perspective on performance and a set of reference points for execution speed (typically, overall execution time or a performance index) and guides proactive improvement of FX!32 performance. The reference points also provide a baseline for the detection of infrequent and isolated degradations in performance. Any change in performance is immediately investigated.

Some changes, such as fix-ups of unaligned memory references, are due to dynamic x86 program behavior. As noted previously, the FX!32 Emulator profiles x86 instructions which make unaligned memory references. For those instructions, the Translator will generate *conservative* Alpha code sequences for unaligned memory access. If the profile does not indicate a misalignment, the Translator assumes data alignment, and generates the *aggressive* Alpha code for an aligned access (Figure 2.)

When an Alpha load-aligned or store-aligned instruction (aggressive case) references memory with an unaligned address, the operating system is invoked to *fix-up* and complete the memory reference. The fix-up operation is 200 to 300 times more expensive than a load or store.

Unfortunately, the behavior of some x86 programs is inconsistent from run to run. Many programs allocate data structures dynamically. On subsequent runs, a dynamically allocated data structure may or may not be aligned. If aggressive, aligned memory access instructions have been generated, the translated program may perform inconsistently. In the case of a program like Bytecpu, alignment fix-ups may occur after translation. If the fix-ups occur in middle of a tight inner loop, the performance penalty is substantial.

The presence of alignment fix-ups is quickly revealed by Perfmon which can monitor this event type. The Compaq fix-ups tool uses sampling to close-in on and identify specific Alpha instructions which are causing the fix-ups. Although the fix-ups tool can identify the source of the problem, it cannot feed the results of its analysis back into the translation process.

4.2 Translated Versus Emulated Execution

A key determinant of performance is the portion of the application code which executes translated versus the portion that is emulated. x86 instructions that must be emulated (some instructions can not be translated), execute at a lower rate due to fetch/decode overhead associated with emulation. These emulated portions are not able to take advantage of the optimizer provided by FX!32.

To see some of the effects of translation, we will look at performance data after translation. DCPI provides a cross-image summary of sample counts taken during a measurement period (epoch.) Table 1 is a DCPI summary of FX!32 executing the Sysmark32 Excel benchmark test.

Cycles	% Cumulative	dmisses	imisses	Image
4913764	40.9%	380397	461276	EXCEL.OPT
1961649	57.3%	178329	297983	win32k.sys
1436843	69.2%	103842	177213	ntoskrnl.exe
695633	75.0%	39039	49348	mga.dll
611960	80.1%	2367	4459	hal.dll
408758	83.5%	69384	45325	wx86cpu.dll
326307	86.3%	21719	35684	ntdll.dll
278700	88.6%	28840	49635	GDI32.dll
221205	90.4%	17176	7949	rasdd.dll
192651	92.0%	16989	29623	Ntfs.sys
185204	93.6%	35126	3648	dcpisvc.exe
150015	94.8%	11351	20066	jacket.dll
147607	96.1%	10371	13203	KERNEL32.dll
139328	97.2%	14534	17779	USER32.dll
115245	98.2%	8208	18563	MSVCRT.dll
99084	99.0%	7368	13899	MSO95.OPT
22399	99.2%	1509	2771	RPCRT4.dll
13625	99.3%	1248	1413	MSTEST40.DLL
11050	99.4%	383	1008	ANALYSIS32.OPT
7639	99.5%	439	510	tcpip.sys
6881	99.5%	563	939	SHELL32.dll
6633	99.6%	627	507	dec_malmd_ns.dll
5210	99.6%	356	444	fx32agnt.dll
5207	99.7%	563	299	loader.dll

Table 1: DCPI summary for the Sysmark32 EXCEL benchmark

Table 1 characterizes the Sysmark32 Excel workload. EXCEL.OPT is the translation of EXCEL.EXE. Even though EXCEL.OPT has the most activity (roughly 40%), the display/user interface load is also quite high (roughly 27%.) This situation demonstrates the importance of fast access to native system libraries via *jacketing* as well as high quality translated code. We can quickly assess the degree to which improvements in translated code alone can affect overall performance.

The FX!32 Emulator resides in wx86cpu.dll. DCPI is able to provide a routine-by-routine breakdown of cycle samples within wx86cpu.dll (Table 2.) Although the number of cycles attributed to

Emulator service	Samples
Emulation	36,486
Control	196,266
String support	152,159
FP support	23,847
Total	408,758

Table 2: Breakdown for Excel Emulator services

wx86cpu.dll appears high (408,758), the number of cycles due to emulation is much smaller (36,486.) The other cycles are attributed to runtime services provided by wx86cpu.dll such as string operations, high-complexity floating point operations, etc., which are executed on behalf of translated code. This indicates that a relatively insignificant amount of emulation is being performed. Knowledge of the architecture and internal mechanism of the Emulator helps make this assessment. We have also instrumented the Emulator to measure key events such as transfer of control from Translated code into the Emulator. Instrumentation provides information about *why* such transfers are occurring as well as frequency of occurrence.

4.3 Cache Conflicts in FX!32

Ideally, emulation and translated execution would have the same memory address trace as the application program. If the original application was optimized for procedure placement (to reduce instruction cache collisions) or data layout (to reduce data cache collisions), emulation and translated execution would benefit from optimizations embodied in the original application program assuming compatible cache behavior between host architectures – an assumption that is not likely to hold.

Unfortunately, this is not the case in FX!32. The Emulator maintains and uses its own data structures for bookkeeping (such as a table of x86 to Alpha address mappings) and translated code calls runtime services in the Emulator. Introduction of extra memory references and procedure calls into the memory trace disturbs the locality of the original x86 program.

The Alpha 21164 processor has separate, direct-mapped I- and D-caches. Each cache is 8Kbytes with a 32 byte line size. Direct-mapped caches are particularly susceptible to cache line collisions. Occasionally, cache conflicts arise between bookkeeping operations and x86 application program operations. Such collisions can have a dramatic impact on performance especially if the conflicts occur within an inner loop.

Again, DCPI is an invaluable aid in finding cache line conflicts. DCPI samples the hardware cache miss counters and annotates the Alpha disassembly with the sample counts on a per instruction basis. Cache line conflicts show up as instructions with relatively high misses.

Figure 3 shows a code fragment which is suffering a performance loss due to D-cache collisions. The instruction at 0x0050022e4 shows a high D-cache sample count and the resulting high cycles per instruction (CPI) value computed by DCPI. Often, collisions occur in pairs (or other multiples.) In this case, the conflict is occurring with a critical Emulator service. The conflict was resolved by using

Cycles	Dmiss	CPI in cycles	Address	Instruction
11775	45		0x0050022a8	`stl ra, 4(sp)`
41	0		0x0050022ac	`and s5, #3, t1`
1062	8		0x0050022b0	`ldq t4, 0(fp)`
0	0		0x0050022b4	`bne t1, 050022c8`
2275	75	2.0	0x0050022b8	`ldl t3, 0(s5)`
2384	223	2.1	0x0050022bc	`ldah t3, -1(t3)`
1072	217	1.0	0x0050022c0	`lda t3, 393c(t3)`
1066	1	1.0	0x0050022c4	`beq t3, 05002380`
1060	3		0x0050022c8	`beq t11, 05002300`
33	0	0.0	0x0050022cc	`ldq t0, 0(t11)`
2146	441	2.0	0x0050022d0	`subl t0, s5, at`
1055	4	1.0	0x0050022d4	`bne at, 05002300`
1047	2	1.0	0x0050022d8	`srl t0, #20, t10`
1049	2	1.0	0x0050022dc	`beq t10, 05002300`
1150	24	1.0	0x0050022e0	`ldq t0, 0(t10)`
42269	2854	38.6	0x0050022e4	`ldl t1, 1584(t4)`
52	12	0.0	0x0050022e8	`subl t0, s5, at`
2121	192	1.9	0x0050022ec	`bis at, t1, at`
1041	4	0.9	0x0050022f0	`bne at, 05002300`
0	0		0x0050022f4	`sra t0, #20, t2`
992	2		0x0050022f8	`bic t2, #1, t2`
0	0		0x0050022fc	`bne t2, 050027a0`

Table 3: DCPI profile with a D-cache conflict (blank entries denote where DCPI did not produce a cycle count.

a better algorithm in the Emulator service – an algorithm which reduced ancillary memory traffic. The effect of the fix is shown in the DCPI profile in Figure 4.

4.4 The Predictability of Branches

Whenever possible, direct calls to translated Alpha routines are made from translated code. In certain cases, such as calls across image boundaries, a direct call cannot be generated since the Alpha entry address in the target image is not known at translation time. This mapping must be made at runtime using a special x86 to Alpha address mapping table included in the target image. The FX!32 Emulator and Runtime components provide the look-up service. Most recently used mappings are kept by the Emulator in a software cache. When translated code requests a look-up, the Emulator first checks its cache for a mapping (we will refer to this check as *Branch 1*), and transfers control to the Alpha target address if the cache contents are not stale (i.e., invalid) (we will refer to this check as *Branch 2*) and the target is known (*we will refer to this check as Branch 3*). The Emulator also checks the x86 target address to detect a call to a jacket, a so-called *native call* operation, and will transfer control as quickly as possible to the jacket (we will refer to this check as *Branch 4*). Otherwise, a more time consuming look-up is performed by the Runtime which updates the Emulator cache as a side-effect.

Using NT-ATOM, we have evaluated the predictability of these four branches in the `wx86cpu.dll` image that handle calls across image boundaries. We use the built-in branch tool provided by NT-

Cycles	Dmiss	CPI in cycles	Address	Instruction
11552	54		0x0050022a8	`stl ra, 4(sp)`
63	1		0x0050022ac	`and s5, #3, t1`
978	3		0x0050022b0	`ldq t4, 0(fp)`
0	0		0x0050022b4	`bne t1, 050022c8`
2260	48	2.0	0x0050022b8	`ldl t3, 0(s5)`
2230	234	2.0	0x0050022bc	`ldah t3, -1(t3)`
1160	146	1.0	0x0050022c0	`lda t3, 393c(t3)`
1082	6	1.0	0x0050022c4	`beq t3, 05002380`
1064	2		0x0050022c8	`beq t11, 05002300`
20	2	0.0	0x0050022cc	`ldq t0, 0(t11)`
2244	359	2.1	0x0050022d0	`subl t0, s5, at`
1025	4	1.0	0x0050022d4	`bne at, 05002300`
1032	3	1.0	0x0050022d8	`srl t0, #20, t10`
981	6	1.0	0x0050022dc	`beq t10, 05002300`
1072	39	1.0	0x0050022e0	`ldq t0, 0(t10)`
285	43	0.3	0x0050022e4	`ldl t1, 1584(t4)`
2164	171	2.1	0x0050022e8	`subl t0, s5, at`
1084	209	1.0	0x0050022ec	`bis at, t1, at`
987	4	0.9	0x0050022f0	`bne at, 05002300`
0	0		0x0050022f4	`sra t0, #20, t2`
1002	2		0x0050022f8	`bic t2, #1, t2`
0	0		0x0050022fc	`bne t2, 050027a0`

Table 4: DCPI profile without a D-cache conflict (blank entries denote where DCPI did not produce a cycle count.

ATOM. A direct-mapped set of 2-bit counters is simulated. No aliasing occurs in the branch prediction table. Each counter is initialized to be weakly taken.

We use two different applications to evaluate branch behavior. The first application used is the Winmag benchmark. This benchmark captures Excel creating and filling a typical summary spreadsheet, creating graphs from the data and displaying multiple views of these graphs. This benchmark is predominately CPU and video intensive. The second application used is the CorelDraw benchmark which is part of the Bapco benchmark suite.

Table 5 provides frequency counts for the four branches of interest in the `wx86cpu.dlll` image. We are specifically interested in assessing the predictability of these four conditional branches.

As we can see, Branch 4 is the most frequently executed of the four branches, and is frequently predicted correctly. When taken, this branch transfers control to a jacketed Alpha API routine, which provides for more efficient dispatch. Branch 2 is always predicted correctly, and possibly could be optimized out of the code. Branches 1 and 3 possess less predictable behavior. Branch 1 is taken when the entry in the translation cache does not change (i.e., exhibits high temporal locality), so we can see that there are multiple entries vying for the same translation cache space.

Branch number	Winmag frequency/prediction accuracy	Bapco frequency/prediction accuracy
1	6.10K/82.4%	41.2K/69.9%
2	5.16K/100%	22.0K/100%
3	5.16K/95.0%	22.0K/99.2%
4	1.63M/99.9%	458K/99.0%

Table 5: Frequency counts and BTB prediction rates for four branches in `wx86cpu.dll` after translation has been run.

4.5 Improving Code Quality

An important analytical activity is the identification of "hot spots" in translated code. New opportunities for optimization are often discovered by assessing the quality of frequently executed code. The first step is to identify the hottest routines out of potentially several thousand routines in an image. The most frequently executed code within a hot routine is then examined and analyzed via DCPI.

The Translator directly generates an Alpha binary image with only one procedure descriptor and very little symbolic information. Without this information, DCPI has trouble breaking down its profile information by routine. Fortunately, images produced by the Translator contain a special section, the x2a table, which provides a mapping from x86 routine entry addresses to the corresponding Alpha entry addresses in translated code. A version of DCPI was modified to use the x2a table to summarize profile information by routine. We have also written and used PERL scripts to accomplish the same ends by postprocessing DCPI output.

Here is an example taken from a hot inner loop. In Figure 6, performance is being lost due to the scheduling of the Alpha load instruction just before the instruction which uses the result of the load (instruction at address 0x06151ef48.) The Alpha 21164 processor issues the consuming instruction speculatively while the D-cache is being evaluated for the load. If the D-cache misses, the consuming instruction is brought back to the issue point and is replayed into the pipeline with a loss of cycles. To correct this problem and avoid a potential replay, an instruction is scheduled between the load and the consuming instruction as shown in Figure 7 (at address 0x0202031ac.) These kinds of code quality problems are reported to the Translator development team for remedial action.

Cycles	CPI in cycles	Address	Instruction
23219	3.1	0x06151ef48	`ldl s1, 10(s4)`
15356	2.0	0x06151ef4c	`cmpult s1, t6, s2`

Table 6: Performance loss due to load/use replay

DCPI provides an Alpha-specific analysis tool with knowledge about instruction slotting and scheduling embedded within it. The tool suggests possible causes for high CPI instructions.

Cycles	CPI in cycles	Address	Instruction
7582	1.0	0x0202031ac	`ldl t0, 10(s4)`
7616	1.0	0x0202031b0	`stl s5, 40(s4)`
7619	1.0	0x0202031b4	`cmpult t0, s5, t0`

Table 7: No loss due to load/use replay

The current version of the FX!32 Translator has been modified to produce an assembler language listing of a translated image. After assembly, the resulting image can be run as part of the application. This feature allows us to try new ideas for optimization by hand and is helpful in assessing the effect of new optimizations on performance. Optimizations can be tried on a broader scale by using NT-ATOM to rewrite the binary image.

4.6 Implementing MMX: A case study

Versions of FX!32 up to and including v1.4 did not include support for MMX instructions. We decided to assess MMX support for v1.5, possibly delaying full implementation until v1.6. Our approach took the following steps:

- Assess the benefit of MMX on x86 architecture
- Identify key MMX operations to guide implementation decisions
- Develop emulation routines for MMX
- Add code generation for MMX to the Translator
- Measure performance, evaluate the implementation, and iterate

Our first step was to measure the benefit provided by MMX for x86 applications. The study concentrated on specific Photoshop operations since an Alpha-native version of Photoshop was not commercially available and we knew that certain operations relied heavily on MMX. The FACET operation, in particular, gains the most benefit from MMX, cutting FACET execution time roughly in half. With this information in hand, we selected FACET as the focal point for study during implementation. It became a defacto benchmark for performance improvements.

We began implementation with Alpha code templates for each MMX instruction. These templates used a uniform convention to pass arguments and results via specific Alpha integer registers. In this form, the templates were readily incorporated into Emulator routines and were handy references for implementation of the code generation routines in the Translator. The code templates were tuned for the 21164 and 21264 Alpha processors. Certain MMX-flavored operations, like 64-bit bit-wise logical operations, are standard Alpha ISA features. However, the 21264 also includes the multimedia extensions (MVI) for signed and unsigned value saturation and (un)packing. We expected to receive an extra performance boost from MVI on 21264.

One early point for debate was the physical location of *x86-architected* MMX register values. The Alpha architecture provides 32 64-bit integer registers and 32 64-bit floating point registers. The 21164 processor does not provide instructions to transfer values between register sets; a value is transferred by writing it to memory from the source register and then reading the value from memory into the destination register. Unfortunately, the 21164 processor behavior also imposes an execution penalty (replay stall) on a read operation following a write to the same location – exactly the instruction sequence needed to move data between the register sets. This situation is corrected in the 21264 processor with instructions (ITOF/FTOI) to perform the side-to-side transfers. The trade-off between representation in integer registers, floating point registers and memory had the following nature:

- If the MMX values were stored in floating point registers, they would need to be moved to the integer side for execution. This transfer would very likely be penalized on the 21164.
- If the MMX values were stored in integer registers, fewer integer registers would be freely available in translated code (i.e., a smaller pool for the register allocator.) Furthermore, eight free integer registers could not be made available easily in the Emulator.
- If the MMX values were stored in memory, load/store traffic would be higher and access would potentially be slower due to D-cache misses.

We decided to represent the MMX values in the Alpha floating point registers since delivery of 21264-based machines was beginning and future, planned memory optimization would be able to remove some back-to-back write-read operations to the same location, thereby alleviating the penalty on 21164. Taking the decision this way did not reduce the pool of registers available for allocation in translated code. We also believed that intermediate results could be carried in integer registers without shuffling by postponing updates to x86-architected registers until necessary.

The FX!32 Translator uses profile information to find x86 routines to be translated. This profile information is produced by the Emulator. Thus, an x86 routine must be executed by the Emulator before it can be translated. Development work on the Emulator portion was started early so that the MMX templates could be tested and profile data would be available to drive the Translator during its development. The first rule of FX!32 performance engineering is *run translated* with the second rule being *make translated code execute as fast as possible.* Much effort has been expended to assure that as much application code as possible can execute translated, including extra instrumentation in the Emulator to determine how often and why translated code is returning to interpreted execution.

Since we had not yet committed to MMX implementation in FX!32 release v1.5, we elected to use a simple, subroutine-oriented style for the first implementation. In this style, the templates were translated to *dual entry* subroutines which can be called from either the Emulator or translated Alpha code, and the Emulator and translated code pass arguments and results via the uniform convention in the templates. This style was already in use for string and high complexity floating point instructions (at the interface between the Emulator and translated code.) Considerable implementation work was still needed in the Emulator to fetch arguments and store results, but this approach was still a relatively fast way to a first implementation. If performance was acceptable, this first version could be included in v1.5, with an enhanced implementation in v1.6.

In order to execute a translated MMX operation in the first cut, translated code loads MMX values into the (integer) argument registers, calls the appropriate MMX routine in the Emulator, and unloads the result (if necessary) from the result register. Unfortunately, this overhead proved to be much too high to be practical. FACET execution time was worse than non-MMX execution. Although this result was disappointing, we were able to use DCPI to identify the most frequently used MMX instructions in FACET as candidates for in-line generation. We were very pleased with the instruction pairing in the templates, getting near perfect dual-issue and maximum parallelism.

We then began to iterate the *develop, measure, and evaluate* process. In-line generation of the MMX logical instructions was implemented first. These MMX instructions had the highest overhead-to-real-work ratio since several instructions were required to move arguments/results and call the appropriate subroutine in the Emulator, while the *real* work was performed by a single Alpha bit-wise logic instruction. DCPI analysis showed a significant improvement. If overhead could be reduced for the other instructions, execution on 21164 could possibly beat non-MMX execution.

Version	21164	21264
Baseline (no MMX)	12.44 sec	9.67 sec
MMX in subroutines	27.14 sec	19.62 sec
MMX in-line	14.70 sec	3.77 sec
Pentium II	6.24 sec	6.24 sec

Table 8: Photoshop FACET execution time

Encouraged by this analysis, the remaining code generators were implemented with all MMX instructions generated in-line. Performance on 21264 with MVI and ITOF/FTOI was outstanding (Tableref̃facet.) However, performance on 21164 continued to lag non-MMX execution. DCPI revealed the presence of lost cycles due to back-to-back store/load penalties. Since elimination of these store/load operations depended upon optimization which could not be completed in the v1.5 delivery time frame, we decided to ship v1.5 FX!32 with MMX enabled on 21264 hosts, but not on 21164 hosts. This was unfortunate since MMX sped Emulator performance on both platforms (6:22 without MMX versus 4:05 with MMX.) Platforms used for testing were:

- **21164**: Personal Workstation 500a (500MHz 21164, 128Mb RAM)
- **21264**: Prototype AlphaPC 264DP (500MHz 21264, 512Mb RAM)
- **Pentium II**: Celebris GL-2 (266MHz PII, 128Mb RAM)

DCPI proved to be an invaluable tool for evaluating and improving code quality. It also helped us to make the key decision to keep going with our implementation, leading us to deliver MMX support to our 21264 customers earlier than we had originally planned.

4.7 Tracing FX!32

To demonstrate how PatchWrx can be used to study system-level performance during binary translation, we have captured traces during emulation of the sample x86 program, arm2.exe. This program is an openGL graphics-intensive application of a 3D solid model robot arm performing rotation motion simulation.

In this scenario we are interested in determining the amount of time spent in FX!32, system dynamic link libraries, and the operating system. In our first scenario, we perform full emulation since this is the first time the arm2.exe binary is run. In our second scenario, we capture a translated version of the arm2.exe executable, excluding the cached translations in the trace. We are limiting our focus to FX!32 and the native system services used by the application (we could have chosen to also instrument the translated image, though this may have limited the length of our trace).

When FX!32 emulates arm2.exe, we find that it spends 98% of the execution (measured in instructions executed) in the hal, s2 and openGL DLL's. The application is dominated by graphics and this drowns out any overhead due to emulation. The average basic block length for this trace is 5.7 instructions, and the conditional branch correct prediction rate is 96.8% (using a 2-level gshare predictor). Running the same executable after translation, we see that execution spends more of its time in the same three DLL's (99%), has a longer average basic block length (5.8 instructions), and has a higher correct branch prediction rate (97.2%). All this makes sense since even less of the execution appears in the emulator and related services.

This analysis helps illustrates an important issue in jacket strategy – the choice of interfaces to jacket. In the minimal approach, only the operating system interface is jacketed and other support libraries (e.g., USER32.DLL) execute as x86 code. Only one set of jackets must be designed, implemented, tested and maintained. However, full advantage is not taken of native Alpha support libraries. The FX!32 approach jackets support libraries as well as the operating system interface making full use of native Alpha libraries. This performance advantage comes at the the cost of higher development and maintenance effort. Cost is reduced in part through automated generation of jacket code from interface descriptions. In the arm2.exe example, FX!32 jackets the openGL DLLs yielding full native performance.

5 Conclusions

DCPI, PatchWrx and NT-ATOM have proven to be effective tools to analyze and improve FX!32 performance. Some limitations and opportunities for future work should be noted, however.

Binary translation generally operates on images which have been stripped of symbolic information, and obviously, the original source code for the application is not available. This makes code analysis and debugging quite difficult. Tools to enhance program understanding, even just def-use information, would be a real benefit.

Translated images contain very little symbolic information and separate procedure descriptors are not created for translation units (routines.) DCPI assumes that source is available and uses procedure descriptors, etc., to organize its results. As this information is not available for translated images, DCPI produces a large volume of unorganized information for translated images. Results must be either browsed (tedious) or filtered using PERL scripts to be useful. Again, better visualization and program understanding tools would be helpful.

Profiling via DCPI and direct instrumentation of the code provide different, yet complementary, forms of performance information. Tools to integrate and interpret performance information from multiple tools would provide greater insight into the cause of problems.

Finally, in this process, problems are manually identified, reported to the development team, and corrected. The time spent in this loop is long. If profile information was collected about translated code, too, during execution and fed back to the Translator, the time around the loop would be shortened considerably. Problems which only appear at runtime, such as alignment fix-ups, could be automatically corrected without manual intervention. Optimization can be focused on inner loops as the famous 90-10 rule of program behavior (i.e., 90% of the time is spent in 10% if the code) has not been repealed.

References

[1] J.M. Anderson, et al., "Continuous Profiling: Where Have All the Cycles Gone?," *ACM Transactions on Computer Systems*, Vol. 15, No. 4, November 1997, pp. 357-390.

[2] P. Argade, D. Charles, and C. Taylor, "A Technique for Monitoring Run Time Dynamics of an Operating System and a Microprocessor Executing User Applications," *ACM SIGPLAN Notices*, Vol. 29, No.11, Nov 1994, pp. 122-131. Vol. 18, No. 2, Jun. 1990, pp. 270-279.

[3] E.B. Betts, D.P. Hunter and S.L. Smith, "Moving ATOM to Windows NT for Alpha," URL `www.research.digital.com/SRC/publications/src-papers.html`, Compaq Computer Corporation, January 1999.

[4] J. Casmira, D.P. Hunter and D.R. Kaeli, "Tracing and Characterization of Windows NT-based System Workloads," *Digital Technical Journal*, Vol. 10, No. 1, December 1998, pp. 6-21.

[5] J. Casmira, D.R. Kaeli, and D. Hunter, "Operating System Impact on Trace-Driven Simulation," *Proc. of the 31st Annual Simulation Symposium,* Boston, MA, April 1998, pp. 76-82.

[6] B. Chen, D. Wall, and A. Borg, "Software Methods for System Address Tracing: Implementation and Validation," DEC WRL Research Report, 94/6, 1994.

[7] A. Chernoff, et al., "FX!32: A Profile-Directed Binary Translator," *IEEE Micro*, Vol. 18, No. 2, pp. 56-64.

[8] A. Chernoff and R. Hookway, "Running 32-bit x86 Applications on Alpha NT," *Proc. USENIX Windows NT Workshops*, Usenix Association, Sunset Beach, CA, 1997, pp. 17-23.

[9] R.F. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," *Proc. of ACM Sigmetrics*, May 1994, pp. 128-137.

[10] R. Cohn, et al., "Spike: An Optimizer for Alpha/NT Executables," *Proc. USENIX Windows NT Workshop*, August 11-13, 1997.

[11] A. Eustace and A. Srivastava, "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools," *Proc. Winter 1995 USENIX Conference*, New Orleans, LA, January 1995.

[12] R. Hookway and M. Herdeg, "Digital FX!32: Combining Emulation and Binary Translation," *Digital Technical Journal,* Vol. 9, No. 1, 1997, pp. 3-12.

[13] J. Larus, "Abstract Execution: A Technique for Efficiently Tracing Programs", *Univ. of Wisconsin-Madison Technical Report*, 1990.

[14] D. Lee, P. Crowley, J.-L. Baer, T. Anderson, and B. Bershad, "Execution Characteristics of Desktop Applications on Windows NT," *To appear in the Proc. of the 25th International Symposium on Computer Architecture*, Barcelona, Spain, June 1998.

[15] S. Perl and R. Sites, "Studies of Windows NT Performance Using Dynamic Execution Traces," *2nd USENIX Symposium on Operating System Design and Implementation,*, pp. 169-183, 1996.

[16] J. Richter, "Custom Performance Monitoring for Your Windows NT Applications," *Microsoft Systems Journal,* August 1998.

[17] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy and B. Bershad, "Instrumentation and Optimization of Win32/Intel Executables Using Etch," *Proc. of USENIX Windows NT Workshop,* August 1997, pp. 1-8.

[18] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Transactions on Modeling and Simulation*, Vol. 7, No. 1, January 1997, pp. 78-103.

[19] R.L. Sites, et al., "Binary Translation," *Communications of the ACM*, Vol. 36, No. 2, February 1993, pp. 69-81.

[20] A. Srivastava and A, Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *ACM SIGPLAN Notices*, Vol. 29, No. 6, June 1994, pp. 196-205.

[21] *Alpha AXP Architecture Handbook*, Digital Equipment Corporation, No. EC-QD2KA-TE, October 1994.

[22] L.S. Wilson, C.A. Neth and M.J. Rickabaugh, "Delivering Binary Object Modification Tools for Program Analysis and Optimization," *Digital Technical Journal*, Vol. 8, No. 1, 1996.