An Unconventional Proposal: Using the x86 Architecture As — The Ubiquitous Virtual Standard Architecture —

Jochen Liedtke Nayeem Islam Trent Jaeger Vsevolod Panteleenko
Yoonho Park

Thomas J. Watson Research Center
IBM
Hawthorne, NY 10532
{jochen,nayeem,jaegert,vvp,yoonho}@us.ibm.com

1 The Problem

There are 100+ million computers in the world. Even smaller organizations have easily 100+ machines; 10,000+ are typical for medium-sized organizations like a university or a bank. Current network technology is so ubiqitious and so powerful that we increasingly use these crowds of computers as one "technical being" instead of thinking of them as single machines. Consequently, we try to support distributed applications, not only by moving data around but also by remote execution of downloaded/uploaded code (applets, servlets) and even dynamically migrating active objects, i.e., curently executing programs (agents, load distribution).

Unfortunately, not all of these 100+ million machines are compatible with each other. Currently, in the workstation/PC/NC segment, we see about 7 different hardware architectures: x86, PowerPC, Alpha, Mips, Sparc, PA-Risc, 68K. Some architectures are likely to disappear over time, e.g. 68K; however, new ones will show up (perhaps Intel's IA-64). Heterogeneity will probably remain a problem over the next decade.

More or less compatible OS APIs and tools, in particular compilers, help to move a source program from an *x*-machine to a *y*-machine. Moving a compiled program is harder; moving a currently executing program (migrating) between *x* and *y* is the hardest.

One approach to move compiled programs between architectures is based on architecture-independent intermediate representations for compiled programs, e.g. Java bytecodes [10] or slim binaries [6]. However, it does not seem likely that in the near future all compilers will use a single intermediate language. (The language community has dreamed about the UNCOL (unitary compiler language) approach for nearly 40 years [15]. The UNCOL idea is to have a single language-independent code generation interface and thus architecture-independent compilers.) So the mentioned approach is restricted to certain programming languages and does not (yet?) give us general mobility for compiled programs.

To solve the inter-architecture mobility problem for portable agents and for load distribution, we must be able to migrate a currently executing program with all its data, including temporary stack and heap data. A first approach to this problem is the ubiquitous interpreter. For example, the Aglets system [9] uses a JVM; Ara [11] uses Tcl. A second approach [14] is based on generating special (native) code that permits migration at certain synchronization points ("bus stops") (Per architecture, a native-code version was generated when the source was compiled.). Similar to the techniques mentioned in the previous paragraph, both solutions suffer from the fact that they are specific to a single language.

2 The Vision

We envision a ubiquitous virtual hardware architecture that is available on any real hardware architecture.

- *Freedom of Movement.*
  When a module or program *P* is compiled for the *x*-architecture, the binary should execute

also on every other architecture. There should be no need for the source or an intermediate representation of *P*.

- *Instant Migration.*
When a program *P* executes on one machine, it should be possible to migrate it to another machine *at any point during its execution* and to any other hardware architecture.

- *Free Lunch.*
And, of course, performance should not be compromised: Any architecture-dependent and -independent optimization should be applicable to the program *P* on the original hardware architecture *x* without that *P* loses its mobility. Although one cannot expect comparable effects of *x*-dependent optimizations when running on the architecture *y*, *P*'s performance should nevertheless be reasonably good on *y*.

We propose to use the x86 architecture [8] for this ubiquitous virtual standard architecture. Since only application programs will migrate, there is no need to include privileged instructions, page tables, etc. in the ubiquitous architecture. V86 and perhaps even MMX might also be dropped.

Remark: Although the ubiquitous architecture masks out the basic hardware heterogeneity, automatic load distribution, cluster computing, mobile agents, etc. need in addition higher-level support, e.g. a distributed or clustered OS and a homogeneous API for mobile agents.

3 Why x86?

Why do we propose to use an existing real architecture and not an artifical architecture (like Elate's VP [16]) that can be easily and efficiently emulated on every real architecture? Why do we propose x86 and not another Risc architecture?

1. Using an existing real hardware architecture has the benefit that all (or most) required tools (compilers for all languages, linkers, libraries and standard software packages) already exist. The cost of porting them to a new architecture (and preserving effficiency) would be horrenduous.

2. Currently, 90+% of all workstations, PCs and NCs are x86-based. So any program compiled for the ubiquitous virtual standard architecture will automatically perform (and even perform optimally) on 90% of all machines without requiring an emulation system to be built.

3. Emulating the x86 architecture by binary translation on any $\approx$32-register processor is simpler and more efficient than the other way round. Probably, it is even better than emulating one 32-register architecture on another 32-register architecture. Furthermore, there is already some experience in translating x86 binaries on the fly to other architectures, e.g. Digital's FX!32 [3].

At first glance, it looks surprising that the x86 architecture is technically the best choice. However, x86 has substantially fewer registers than any other architecture. Translating instructions from an 8-register instruction set into a 32-register host architecture is relatively simple. 8 host registers are mapped to the 8 x86 registers. The remaining host registers can be used to emulate other x86 resources or to improve the performance of the emulation (by using temporary registers, etc.) For the opposite case, most of the 32 virtual registers had to be implemented as main memory locations.[1] For the problems of emulating "32" registers of a processor on the 32 registers of a different processor , e.g. "32" 32, see Sites *et al.* [13].

Another nice feature of x86 is the absence of branch delay slots. Emulating an architecture with branch delay slots on an architecture where branches are not delayed is hard.

4 Technical Details

4.1 On-the-Fly Binary Translation

Binary translation, e.g. as described in [2, 3, 4, 13] is the basis for architecture emulation. For the ubiquitous architecture, x86 instructions must be transparently translated on the fly, i.e., when accessed in memory. This gives us independence of load

[1] Besides severe performance implications, this also raises the problem how to ensure that the pseudo-register memory locations are not also used as x86 memory. If, however, the x86 architecture is emulated on 32 real registers, all emulator-internal variables can be held either in registers (and are thus inaccessable for x86 code) or in read-only pages that are allocated dynamically.

file formats, permits direct code generation (without generating a load file), and even enables page-based (partial, on-demand) migration of code and data within a clustered system.

Single-Instruction Translation translates each individual instruction of the source in isolation to the host architecture, independent of its instruction context. This technique is fast since no context analysis is required. After decoding the x86 instruction, an equivalent sequence of host instructions can be generated from a template. On a 200 MHz processor, typically less than 0.2 sec/MB are required. However, the technique does not allow the host code to be optimized. On the other hand, branch targets (labels) are simple. They never require retranslation of the target code as long as they branch to addresses that are mapped to the beginning of an already translated x86 instruction.

Multiple-Instruction Translation analyzes entire basic blocks or even sets of basic blocks to generate better optimized host code. As [3] shows, the costs of such context-sensitive translations are not negligible. Fortunately, multiple-instruction translation can be used like a hot-spot compiler, transparently optimizing only the dynamically detected hot spots of the code.

4.2 Segments

Emulating x86 segments on other architectures is costly: for any memory access, the effective address has to be checked against the segment limit and the segment base has to be added. Depending on the host architecture, 1 to 3 additional cycles per memory access are required. In addition, the code becomes larger and needs more instruction-cache space. This effect is probably even worse than the additional execution cycles.

Fortunately, many systems use only a flat memory model on the x86. Nevertheless, segments should be included in the ubiquitous architecture. However, they must be for free as long as they are not really used. Any translator should start with generating fast flat-memory code (without code for adding the segment base and checking the segment limit). As soon as a non-flat segment is loaded into a segment register, the already translated code must be invalidated and the translator must generate slower segment-sensitive code from then on.

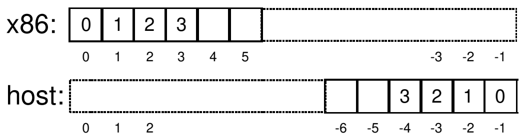
4.3 Little Endians

Fortunately, one's-complement machines are no longer relevant. However, the world is still divided in little-endian and big-endian machines. There is no doubt that the ubiquitous architecture must be little endian (the 90+% argument). Although most non-x86 processors can specify the byte ordering at boot time, the ubiquitous architecture needs little endian semantics even when the processor is booted with big endian ordering. The PowerPC architecture offers load/store instructions operating in the "reverse" ordering. So we can easily generate little endian code, even if the machine runs in big endian mode.

Otherwise, the entire x86 address space can be mapped "inverted" to the host architecture. (The memory is not typed and programs can freely mix byte access and word access on the same address. So there is no way to reverse only the "words" in memory and leave the "bytes" untouched.)

i) When loading the x86 address space into the host, the entire byte ordering is inverted. The byte at x86 address 0 is loaded into host address -1, 1 to -2, and so on; in general x86 address a is mapped to $-a-1$. (This includes the original x86 code. Fortunately, translating instructions backwards performs as well as forwards.)

ii) The translator then generates host code in such a way that any word access to x86 address a operates on host address $-a-4$; byte access is mapped to $-a-1$, etc. For dynamically calculated addresses, this requires typically one additional instruction per memory access.



Loading the word from x86 address 0, host address -4, in big-endian ordering now loads the correct value 0x3210 into the register. When accessing a byte at x86 address 0, host address -1, the least significant byte is correctly accessed.

4.4 Floating Point

All relevant floating point units claim IEEE-754 compliance. The memory representation of 32-

bit and 64-bit floating-point values is basically the same on all these machines. Nevertheless, adding three or more values can result in dramatically different sums on different 754-compliant processors. Internally, the x86 architecture uses 80-bit floating-point registers. Adders and multipliers operate also 80 bits wide. The 80-bit values are rounded to 64-bit or 32-bit values only when they are written to memory. In general, processors that have only 64-bit floating-point registers will produce less precise, i.e. different, results.

Fortunately, very few programs will be affected by the difference in precision. However, numerically instable algorithms and some highly x86-optimized numeric algorithms will suffer when they run on a less-precise processor.

To avoid this incompatibility, Java first decided that always the least-precise arithmetic has to be implemented. However, as Coonen [5] showed, this results in very significant performance degradation since the floating-point values have to be written to memory (and loaded back) after each floating-point operation. The effect of artificially reducing the precision on an x86 processor can easily be illustrated by the loop of a vector multiplication ($s := s + a_i \times b_i$):

```
loop:  fld   [r1]          fld   [r1]
       fmul  [r2]          fmul  [r2]
       add   r1,delta      add   r1,delta
       add   r2,delta      add   r2,delta
       fadd                fst   [temp]
       dec   r3            fadd  [temp]
       jnz   loop          fst   [temp]
                           fld   [temp]
                           dec   r3
                           jnz   loop
```

The left column is normal x86 code and uses the 80-bit register stack of the processor. The right column reduces the effective precision after the multiplaction and after the addition. On a Pentium, the execution time (without memory stalls) for the left code is 7 cycles, for the right code 14 cycles per iteration.

For pragmatic reasons, we propose therefore that the precision has to be at least 64 bit *but can be arbitrarily increased* on any implementation of the ubiquitous architecture. This enables fast emulation on all machines, but opens the door for some incompatibilities. However, the incompatibility is somehow restricted: If a numeric algorithm is recompiled for another processor and then still works properly, then the original x86 binary will work as well on the ubiquitous x86 architecture on the said processor.

4.5 Page Tables, Privileged Instructions, and V86 Mode

Fortunately, all these difficult-to-implement features can be ignored. The are not required for a ubiquitous 32-bit user architecture.

5 Ubiquitous APIs?

Having a ubiquitous standard API for OS services, Window sytems, etc., is another problem that has to be solved for enabling unlimited dynamic migration of software. Fortunately, the ubiquitous-API problem is technically independent from the ubiquitous-architecture problem. In fact, most available API's are implemented on nearly all hardware architectures. This principle orthogonality permits us to combine any ubiquitous architecture with arbitrary ubiquitous APIs.

However, the hard problem is to find a generally accepted API. Posix is standardized but not used; Unix is used but not standardized; NT is de-facto standard but ever changing; the Java APIs are (not yet?) generally accepted.

Fortunately, the ubiquitous-API problem is not of the all-or-nothing type but is open for stepwise solutions. Once we have a ubiquitous architecture, we can e.g. realise a ubiquitous API for applets, then for games, etc. Finally, of course, we must end up with general, OS-type, ubiquitous APIs. To circumvent the heterogeneity problem of APIs, one might even optimistically aim at offering multiple APIs that could be used alternatively per migrated object.

6 Prospects

The existence of a ubiquitous virtual architecture would make life easier. If this architecture should become reality in the near future, the x86 architecture should be used. Whether it can become reality, basically depends on two performance questions: How fast can we translate binaries? How fast will they execute on the host architecture? The engineering challenge is to find out what we can achieve.

References

[1] *PowerPC 603 Risc Microprocessor User's Manual*. IBM, Motorola, 1994.

[2] R. Bedicheck. Some efficient architecture simulation techniques. In *Usenix Winter Conference*, pages 53–63, January 1990.

[3] A. Chernoff and R. Hookway. Digital FX!32 — Running 32-bit applications on Alpha NT. In *The Usenix Windows NT Workshop*, pages 9–15, Seattle, WA, August 1997.

[4] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report UWCSE 93–06–06, University of Washington, Seattle, WA, 1993.

[5] J. T. Coonen. A note on Java numerics. e.g. http://www.math.chalmers.se / thomas /D++ /Laborationer /Lab1 /Coonen.html, December 1997.

[6] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[7] J Heinrich. *MIPS R4000 Risc Microprocessor User's Manual*. Mips Technologies Inc., 1994.

[8] Intel Corp. *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual*, 1993.

[9] D. B. Lange and M. Oshima. Programming mobile agents in java – with the java aglet api (the aglet cookbook). http:// www.trl.ibm.co.jp /aglets/ aglet-book/ index.html, 1997.

[10] T. Linholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[11] H. Peine and T. Stolpmann. The architecture of the ara platform for mobile agents. In *First International Workshop on Mobile Agents*, pages 9–15, Berlin, August 1997. Springer Verlag. Lecture Notes in Computer Science No. 1219.

[12] R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Equipment Corp., 1992.

[13] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Digital Technical Journal*, 4(4):137–152, 1992.

[14] B. Steensgaard and E. Jul. Object and native code thread mobility among heterogeneous computers. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 68–78, Copper Mountain Resort, CO, December 1995.

[15] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel. The problem of programming communication with changing machines: a proposed solution. *Communications of the ACM*, 1(8), August 1959.

[16] Tao Systems. Elate's translation process. http:// www.tao.uk /2/ tao/ elate/ translation.html, 1998.