OBLIQUE PERSPECTIVE

Recycling Programs: Software Rehosting Part 2

Will Business Strategies Stand in the Way of Technology?

By John Wharton, Applications Research

Last October this column reviewed two techniques for executing software on non-native CPUs (see µPR 10/2/91, p.17). In Part 2 we examine some other possible benefits of binary recompilation technology, and discuss non-technical factors that may delay its success.

With its founding members dropping like flies, it's just a matter of time before ACE bites the dust. But for all the smoke and fury it caused, and all the product plans left dead in its wake, the ACE initiative had one positive effect on the industry. It raised public awareness of the problems of microprocessor proliferation, and spurred interest in techniques for recycling existing programs.

The concept of binary software rehosting isn't new. For years Insignia Solutions has offered a program called SoftPC that lets RISC-based UNIX workstations run standard x86 applications, and Hunter Systems has had tools to help application vendors port existing programs to new platforms. Lately, though, it seems interest in this technology has reached critical mass.

Apple's PowerBook ads are actively promoting a Mac version of SoftPC with the promise of full DOS compatibility. At this month's World-Wide Developers' Conference, Apple repeatedly assured attendees that the PowerPC operating system would have software emulation of the 68K built-in, for compatibility with current Mac applications. Apple also announced an alliance with Echo Logic, an AT&T spin-off that's aggressively developing binary rehosting tools. (Details on this project will appear in the next issue.)

Sun recently created a whole new subsidiary called SunSelect to pursue hardware and software tools for running non-SPARC applications on their workstations. At the SPARCstation 10 roll-out (see p. 11), Sun demonstrated its own x86 emulation program called SunPC, as well as Liken, a new Macintosh emulator developed by Accelerated Systems.

DEC's strategy for Alpha counts heavily on recompilation techniques to convert existing VAX applications to the new architecture. One of HaL Computer's primary markets is thought to be current users of various IBM business systems. Software to convert their existing applications to run on HaL hardware would help these efforts considerably.

Conventional Wisdom

Any vendor's phonograph can play any vendor's LP, any cassette deck can play any cassette, and any CD player plays any audio CD. So why, novice users ask, can't any vendor's software run on any desktop system? A very good question indeed.

Currently, different host platforms can "play" the same word processor or spreadsheet only if its developer offers a different version for each. But relatively few programs are currently available for more than one host, and many may never be. Not all compilers are available immediately for the newest CPU architectures, and assembly language programs and custom utilities are seldom rewritten. Obsolete hardware and boutique system designs may be deemed not worth pursuing. Shareware suppliers fall by the wayside, and even legitimate vendors go bust.

But even when programs *are* ported to multiple hosts, the problems encountered are many. Different software versions generally have different capabilities, with different user interfaces and different command semantics. Data files may use different formats, with different fonts, characters, and formatting rules. And even when each version starts with the very same source code, compiler differences can introduce new bugs and change the subtle ways in which commands interact with each other.

So in order to run the widest assortment of software, serious small-system users are forced to amass multiple systems: Macs to run Mac software, PCs for DOS, and maybe a workstation or two for UNIX—one each at work, maybe one each at home, and perhaps a notebook version of each for the road. With ARC systems planned, and PowerPCs on the way, the problem is becoming unmanageable. Who can afford to replace so many boxes, as each gets out of date? When I travel, how many notebooks can I fit in my briefcase?

And can anyone afford to scrap *any* old system? I've got several old ISIS-II (remember ISIS?), CP/M, and CP/M-86 boxes squirreled away in my office, on call for the times I need to exhume old source files or retrieve tax records for an audit. I would much rather invest my limited funds in one top-of-the-line desk-top, notebook, or whatever, use it to run the software I need for the bulk of my work, and yet reserve the option to run other systems' software as occasional needs arise.

Architecturally-neutral distribution formats are one way to let program files run on each of many hosts, but the early efforts traded generality for some amount of run-time efficiency. If there's a software package I need daily, I'd like to think the version I use is finely tuned for my machine. Moreover, the ANDF option works only among vendors who publish in that format, and does nothing to salvage existing code. It would be far better, I think, if there were some way to execute existing software on new platforms, so the venerable WordStars of old would run on the fastest CPUs of tomorrow. Software emulation is flexible but slow. More promising, I think, would be binary code recompilation.

Binary Recompilation Concepts

Here's how it works. Binary recompilers are a special class of translation program, like conventional compilers, except that they read executable binary files as their input instead of ASCII source code. The recompiler interprets this code as a specialized, very baroque language, and emits an executable program file targeted for a different CPU and OS.

On the surface, the process seems inefficient. Blindly expanding each source instruction to one or more target instructions would produce longer and slower programs. But by searching for "basic blocks" in the input—instruction sequences guaranteed always to execute together—the translation can be made more efficient.

A series of i860, 88K, or R4000 instructions used to compute an address, load data, compute a sum, and store the results could, in theory, be replaced by a single 486 instruction—shorter, denser, and faster than the original, and on a CPU with a somewhat less clouded future. And conventional peephole and global optimization techniques can further refine the process.

There's still a problem, though. While many programs yield readily to the up-front control-flow analysis needed to parse a raw binary file, certain others do not. Self-modifying code, stack-resident routines, dynamic linking, and convoluted computed branch paths can all confound an input parser. When such constructs are encountered at run-time, recompiled programs must either reinvoke the translation phase or fall back on interpretive execution.

Thus the most general recompilation systems are a hybrid of two technologies. One approach is simply to emulate the original program at first, interpreting old instructions while simultaneously profiling their execution. As program segments and subroutines that execute repeatedly are identified, equivalent blocks of native code can be patched in—analogous to caching program loops for faster execution.

A different approach, followed by DEC's Alpha recompiler, generates an all new program based on static control-flow analysis. If run-time execution departs from the anticipated control paths, the system reverts to conventional emulation until control reenters a recompiled execution thread, and logs information about the unforeseen execution state to disk. Later attempts to recompile the same program "learn" from the log file in order to produce a more complete version.

Silver Linings

Recompiled programs can be huge, thanks to code expansion, pointer-mapping tables, and run-time support routines and interpreters. But disk space keeps growing, memory keeps getting closer to free, and paging makes sure everything fits. Self-modifying code schemes continue to fall out of vogue, and developers use compiled languages almost universally now, which recompile more efficiently than assembly code.

Still other factors help ameliorate recompilation inefficiencies. As sophisticated OSs and GUIs prevail, CPUs spend less time within the application itself, and more time within the system software. OS code is already optimized to run flat-out on the host, so the overall impact of any slow-down in the recompiled application is reduced.

And now even CPU architects are jumping into the fray. IBM's RS/6000 included a number of features that accelerated the execution of non-native code—a fact that probably wasn't lost on Apple when the PowerPC was developed. DEC admits that several provisions of Alpha were intended primarily to help VAX emulation.

As a result, DEC says converted VAX applications can run up to one-third as fast as those compiled directly to Alpha. Echo Logic claims an overall execution efficiency for 68K code within a factor of two, and is aiming for 80%. Assuming one's primary machine is inherently much faster than whatever other odd systems one may have laying around, the difference in effective performance may be a wash.

And it's worth noting that future compilers might eliminate entirely the need for run-time emulation, by foregoing self-modifying code algorithms, for example, and by including "hint" records that identify non-obvious branch destinations in their object files. Such hints would be optional, and existing programs would still run, so I'm not advocating an architecturally-neutral format here—just one that's somewhat less architecturally hostile.

Blue-Sky Possibilities

Recompilation technology may have other uses that have yet to be discovered. In essence, the first stage of the recompilation process extracts the underlying logic from an otherwise inscrutable program, and it may reveal structural errors in the original. In early work with the Alpha recompiler, DEC found new bugs in pro-

grams that had gone undetected for years.

Sometimes it may even make sense to analyze a program and "recompile" it back to the same CPU. Existing programs could then be enhanced by reorganizing code for superscalar machines, or by rescheduling instructions for a new execution pipeline. As increasingly arcane bugs surface in increasingly complex microprocessors, the technology might be used to determine whether old programs contained a potential hazard, and if so, to rework them to be safe. This might have given Intel a more graceful way to recover when bugs were found in the first production 486s.

Ditto for new system hardware and OSs. Apple's Mac IIci, IIfx, and Quadra each caused certain existing programs to break, as did the move from System 6 to System 7. In the future, automatic installation utilities might scan existing applications and use recompilation technology to fix newly-failing code.

Recompilation technology may also provide a way to soak up the surplus execution cycles that will someday come free with multiprocessing systems like the SPARCstation 10. An incremental recompilation process goes through many stages—interpretive emulation, execution profiling, code generation, and final native execution, for example—that are largely independent. Phoenix Technologies once proposed a quite elegant scheme for partitioning such tasks among loosely-coupled CPUs in multiprocessor systems.

...and Darkening Clouds

So recompilation technology shows plenty of promise. It would be good for individual computer users since it simplifies both hardware and software purchasing decisions. It would be good for large corporate buyers, since it preserves investments in old equipment and facilitates large heterogeneous networks in which any node could run any application.

It would be good for system vendors since it ensures immediate availability of key applications on new CPU architectures. It would be good for OS vendors, since it expands the market for their products. It would be good for application vendors since it increases the installed base of potential buyers for any given program release. And it would be good for microprocessor vendors since it relaxes the need for absolute interchip compatibility.

So if everyone comes out ahead, shouldn't everyone be happy? Not quite. Every silver lining has its cloud. For recompilation technology to succeed it needs the support of the largest, best-established software vendors. These are the people with the best knowledge of how the programming interfaces of the original OSs really work. These are the people who could integrate recompilation capabilities most seamlessly into OS products of the future.

But the largest, best-established software vendors also have vested interests in maintaining the status quo. The benefits of recompilation technology basically lower the barrier to entry of smaller, third-party vendors. Once you've cleared the hurdles, though, it's in your best business interest to keep the entry barriers to others high. If you can afford to maintain different versions of a program for each target platform, you may prefer that they *not* be interchangeable, in order to stimulate repeat sales and to sustain a different pricing structure for every market segment.

If so, then software vendors could simply refuse to provide the support hooks that would make recompilation most straightforward. Application vendors might threaten to sue the vendors of third-party recompilers; since the files they produce would clearly be derived from original copyrighted works, it could be argued that the use of a recompiler violates the original software licensing terms. Some licenses specifically preclude disassembly or reverse-engineering. The prosecution in such cases would benefit greatly should the recent Sega-Accolade decision (see p. 16) stand up in court.

And application vendors could design barriers to code recompilation into the products they ship. Since recompiled programs mimic exactly the logical structure of the original, existing copy-protection schemes should work equally well to prevent unauthorized recompilations. Copy-protected software whose execution is keyed to a unique Ethernet node ID number, for example, would fail to work on a new host. Software that reads encrypted data values from a separate add-in hardware adapter, or that required the periodic reinsertion of the original distribution diskette, would be similarly recompilation-protected.

And if all else failed, application vendors could simply sabotage third-party recompilation efforts. Programs could be peppered with exactly the sorts of structures that make recompilation inefficient: spurious instances of self-modification, Byzantine computed branches, and gratuitous references to hard-to-mimic status flags. While these tricks would not prevent a sufficiently adept recompiler from producing properly executing code, they could make the rehosted version so painfully slow as to not appeal to end users.

Wrap-Up

So it appears the technology problems that currently prevent software recycling are tractable, and can likely be brought under control. Less tangible business issues, though, may cause trouble. A conspiracy of large OS and application vendors may yet keep this technology out of the hands of end users.

Oliver Stone, are you listening? ♦