No Source

No Problem!

Code?

What if you have to
port a program, but
all you have is a binary?

PETER PHILLIPS and GEORGE PHILLIPS,
DIGITAL ECLIPSE

Typical software development involves one of two processes: the creation of new software to fit particular requirements or the modification (maintenance) of old software to fix problems or fit new requirements. These transformations happen at the source-code level. But what if the problem is not the maintenance of old software but the need to create a functional duplicate of the original? And what if the source code is no longer available?

This exact problem arises when trying to reproduce the original play of old arcade games on modern devices. The game play is so well known that anything short of the original is unacceptable. Often the source code is available, but it may be incomplete and may not cover all of the patches that were added to later production models. In addition, it is too expensive to provide copies of the original hardware.

Providing faithful emulations of video games (and old home computers) is our primary experience with this process. But the same techniques can be applied to other areas. Aging hardware and software can be replaced by new hardware with a completely compatible program.

BRIDGING THE GAP

The general problem can be expressed as “bridging the semantic gap.” You must create a program that precisely maps the meaning of the original program onto a host system. Primarily this means an interpreter of some kind

No Source Code? No Problem!

for the target processor's instruction set, but one must also deal with I/O (input/output) devices. Such an interpreter is known as an *emulator*. (See "The TRS-80: A Simple Emulator" on page 54 of this article.) If the program is automatically converted to a different language, it is called a *translation*.

Why are we tackling the problem at such a low level? Mainly to achieve the highest fidelity possible to the original. Emulation is about mapping semantics. The semantics of hardware are usually well documented either by circuit diagrams or chip-specification documents. Internal layers inside software are usually designed to much looser standards, so it is unlikely that specification documents—if they exist—completely describe the behavior. In fact, the software itself is the most authoritative description available. It is true that chip specifications are not always complete or accurate, but chips are reused and over time the deviations become widely known.

The semantic gap between the target and host

systems is not purely an abstraction. It can be quantified as follows: G = number of host instructions to emulate one target instruction. Given G and some idea of the relative speeds of the host and target system, you can quickly decide if emulation is feasible. The problem here is that the value of G is a function of an actual emulator. A rule of thumb is that G is at least 10, but practically speaking, 10 is lower-bound for systems that are quite similar. Although time is usually the overriding concern, there is an analogue for a semantic gap in terms of storage space.

The semantic space gap is the ratio of the size of the host program to the size of the target program. For an emulator, the host program size is broken into two pieces: the host's representation of the target program and the emulator code and associated tables. Unless the host and target are radically different, there is little to be gained by significant changes to the representation of storage. Thus, if the emulator code is ignored, the semantic space gap is typically exactly one.

EMULATION CARRIES A COST

Intuitively, the value of G really depends on how different the target and host machine are. (See "The TRS-80: A Simple Emulator" on page 54 of this article.) Most modern machines are quite close in basic architecture, using 2's complement arithmetic and register sizes that are multiples of 8 bits. But an emulator has several pieces of overhead to interpret one target instruction:

1. Decode the instruction and dispatch to appropriate code.
2. Perform memory loads and stores—and deal with resulting I/O operations.
3. Perform the core operation (e.g., by adding the two numbers).
4. Update any flags or other side effects. Often this includes updating the virtual time.

These steps can add up to a considerable number of instructions. The decoding step is usually a trade-off between time and space: Larger dispatch tables use memory but make for faster dispatch. The loads and stores present a problem because the target hardware can make choices

about what memory is accessed based on the addressing line signals. Inside a normal program, you cannot distinguish memory accesses, and code is required to map the address into an appropriate array.

An emulator creates a virtual version of the real machine and can be written using any programming language you choose. But if you're trying to minimize the semantic gap, you'll want to pick a language that lets you get as close as possible to the underlying host hardware. Most emulators are written in C as C's semantics are similar to modern hardware. A small bit of assembly language can be used to reduce the gap further.

For instance, most core operations are trivial to code in C. But not entirely: Most microprocessors have a rotate instruction of some kind, which in C can be expressed only as two *shifts* and an *or*. Thus, an instant addition of three is made to the gap. (Unless your C compiler is unusually clever.) This is a case where a bit of inline assembler language can make a big difference.

When using C, the gap will widen if the target system sets flags (e.g., *carry* or *overflow*) on the result of arithmetic operations. If your emulator is written in assembly language, you may be able to make use of the host system's flag registers. If not, you'll need to add more C code.

And this is just for the processor. We still haven't accounted for any of the work for dealing with other devices in the system. For example, the video game hardware may support movable graphics, called *sprites*, whereas your host system does not.

TRANSLATING BINARY INTO SOURCE CODE

If the differences are particularly large or if the speed difference between the host and target is not large enough, a pure emulator will not be feasible—at least not one that runs in realtime. Possibly the final product requires some modification, such as graphics scaling, to match the host's display. A translation of the target program into source code will be faster than an emulated version and more amenable to modification. Since the translation is automated, unlike a port, the translated version will be as faithful to the original as the emulated version.

The decoding step
is usually a trade-off
between **time**
and space.

No Source Code? No Problem!

While more difficult than writing an emulator, translating the target code is not remarkably difficult when dealing with a set of programs known in advance. Each program may be analyzed to produce a complete set of code paths. The code translator is effectively a reflective version of the emulator. Instead of performing the

machine instructions, it outputs the code to execute the instruction.

Producing the set of all possible code paths may seem daunting or even impossible. Fortunately, a trivial baseline can be established: Assume that every byte in the program is the start of an instruction, then just

The TRS-80: A Simple Emulator

Systems vary a lot in complexity, but the basic tasks of emulator writing can be illustrated by the TRS-80.

The Radio Shack TRS-80 Microcomputer was the first emulator we wrote. A typical system contained Basic in ROM, 16K of RAM, a Z-80 processor, 1K of memory-mapped display, and memory-mapped keyboard. The real system has more pieces, but this is a good start.

MEMORY MAP

\$0000–\$2FFF	Basic ROM
\$3800–\$38FF	Keyboard matrix
\$3C00–\$3FFF	Video memory
\$4000–\$7FFF	RAM

IMPLEMENTING MEMORY

An array of 32,768 unsigned characters will do. Find an old TRS-80 to get a copy of the Basic-ROM and use it to initialize elements 0 to 12,287.

VIDEO DISPLAY

The video memory is organized as 64 columns of 16 rows of text. Character graphics are supported.

Implementation: Mapping each video byte to an ASCII character will be a good first approximation.

THE KEYBOARD

How the keyboard is mapped into \$3800 is slightly more complex (See table 1).

The code is simple: Upon host *key-up* and *key-down* events, *set/reset* the appropriate bits in the memory array. You may defer this work as you won't need an operational keyboard to boot the system."

IMPLEMENTING THE Z-80

This is the most exacting task. There are 255 op codes. Of these, there are four prefix bytes representing extended op codes: \$CB, \$DD, \$ED, \$FD. There are about 630 different prefixes. There are lots of internal patterns to the prefixes that can be exploited to reduce the amount of code to write.

Write the code, put the keyboard and graphics into a nice GUI wrapper, start the Z-80 PC at \$0000, and you'll have an emulator in no time at all.

TABLE 1

Address	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
\$3801	@	A	B	C	D	E	F	G
\$3802	H	I	J	K	L	M	N	O
\$3804	P	Q	R	S	T	U	V	W
\$3808	X	Y	Z	-	-	-	-	-
\$3810	0	1	2	3	4	5	6	7
\$3820	8	9	*	+	<	=	>	?
\$3840	Enter	Clear	Break	Up	Down	Left	Right	Space
\$3880	Shift							

translate all of them, connect them together, and full coverage will be established—except, of course, if the program generates program code. Although this isn't an uncommon practice, especially in older games, only a small amount of code will be analyzed and ported by the programmer.

The brute force approach makes the translated code size bigger than strictly necessary. Code expansion will mainly be due to the translation of program data into host code. The rest of the unused host code will come from instructions that overlap the actual execution paths. The excess translated code is not a problem in itself as it will never be executed.

The extra host-code memory size, however, is likely to be a problem in a game as considerable ROM space will be used for graphics. Because the translated code will be quite obfuscated, any necessary changes may also be an issue. This can be addressed by a more subtle approach that involves tracing all possible execution paths from all known entry points. Only code actually required will be translated, and no data will be translated into code. Yet this approach creates other difficulties.

An automated trace will miss some of the code because of jump tables and other constructs difficult for tracing subroutines. While complex data flow analysis may not be necessary to locate the base address of a jump table, the actual size of the table is unlikely to be expressed in the code. That is, the index into the table generally won't be bounds-checked as the program implicitly ensures that the value is never out of range. Many heuristics can be applied to determine table extent, but a nontrivial number of cases will still require programmer inspection. This kind of work makes a translator more costly than an emulator.

TARGET TRANSLATION LANGUAGES

There are two obvious choices for what language the target program is to be translated into: either assembly language or a high-level language such as C or Java. Output in assembly language lessens the semantic gap,

THE CARRY FLAG

Many processors support a carry flag, the arithmetic carry from the top bit of the register being added. Emulating carry with 8-bit registers in C is relatively easy. The carry from $A + B$ is simply $(A + B) \& (1 \ll 8)$. But what happens when you need the carry from a 32-bit add? For 32-bit integers, $(A + B) \& (1 \ll 32)$ will always be zero. Without resorting to 64-bit arithmetic, you can find the carry by $B > \sim A$. How does it work? $\sim A$ (the complement of A) is the largest possible number you can add to A without a carry: $A + \sim A = (1 \ll 32) - 1$. If B is any larger, then a carry will be generated from the top bit.

and typically results in two or three host-instruction outputs per target-instruction input. But such a translator is more difficult to write and clearly has more limited portability across host machines.

Output in a high-level language (we'll just assume C from now on) will not run as fast but will give us some technological leverage. Compiler optimizations will lessen the need to produce optimized translated code. The extra effort in using program flow tracing to eliminate code paths that are never executed will really start to pay off now. Any translation gains a speed advantage over emulation by removing the instruction *fetch* and *decode* machinery. Translated code based on program flow will preserve the basic blocks of the original. Now the compiler has the opportunity to elide redundant code. For example, the target program may have had to sum three numbers using two *add* instructions. But a compiler could recognize that those two instructions may be replaced by a single *3-operand add*. In addition, the *carry*

flag calculation could be dropped entirely from the first *add* as it is recalculated for the second *add*.

This is not to say that the compiler is some kind of panacea. Data flow analysis can still easily outstrip the compiler. But it will save some work and is likely to be good enough for the job.

It should also be pointed out that it is nothing but absolute foolishness to believe that the code could ever be translated into something resembling normal C (or Java or whatever). At best, the output will look like assembly language code written in C, with an odd syntax that makes it really, really nasty. But the myth of structure improvement is a powerful one; beware its siren call. Although translated code will be functionally equivalent, it will not be maintainable.

Although the translated code will not be especially pleasant, it will be possible to work with it as you would any source code. In particular, new code can be added to paper over translation failures or make any necessary code changes.

No Source Code? No Problem!

Most important, the whole spectrum of performance analysis techniques and tools can be applied to boost execution speed. All the usual lessons apply and they do not need not be reviewed here. But strength reduction of translated loops will be a common theme in many improvements.

For example, consider a loop that is copying memory. In the general case, each byte copied requires a decode of the source and a destination address from target space into host space.

In most cases, the source and destination will be a contiguous block of memory in host space. Instead of translating the addresses inside the loop, the addresses may be translated outside—and the loop itself becomes

a host memory copy using native pointers. The disparity will even be larger for those loops accessing memory-mapped hardware registers where the translation from memory write to device action is typically even more expensive.

Of course, there's no hard dividing line between emulation and translation, so a very efficient strategy would be to have early versions of the translated code revert to emulation when they reach those areas not translated. Such a hybrid could be the end result, with the translation applied only as necessary to especially intensive parts of the program. The downside, however, is that the emulator requires the original target program, exacting extra memory costs. But switching between

Phantasy Star III

As depicted in table 1, the original game ran on a Sega Genesis (an old home console) and was to be emulated on a Game Boy Advance (handheld system).

The CPUs are so close in performance that direct emulation can immediately be ruled out. The larger cartridge size means we can accommodate some code expansion caused by translation. The different endianness of the processors will

make the translation more difficult.

The video systems are similar, but not identical. The tile sizes are the same (8×8 pixels), but differences can be found in the pixel order. A more major obstacle is the screen size, with the television-based Genesis at 320×228 versus the Game Boy's 240×160 dimensions. Some of the television space is unused because of overscan. The difference is large enough that artists must scale many pictures.

The sound system presents the largest gap. FM synthesis uses relatively few parameters to create a large variety of sound effects, which must be replaced with precomputed samples on the host system. Fortunately, we have enough additional cartridge space for the necessary samples.

In this case, we had access to relatively accurate source code. A tool was written to compare a disassembly of the shipped ROM with the source to identify a few differences that were backported. Translating from the source made it a bit easier to deal with the scaled graphics. The translator (a Perl script) converted the assembler source to C macros, which expanded to the implementation of each 68000 instruction.

Finally, there was extensive testing to ensure that the finished version of PSIII played identically to the original.

TABLE 1

	Genesis	Game Boy
CPU	16 bit 8MHz 68000	32 bit 16MHz ARM7
RAM	64 Kbyte	256 K
Video Memory	64 Kbyte	96 K
Video I/O	DMA	DMA and Direct access
Video Type	Tiles and Sprites	Tiles and Sprites
Word Order	Big endian	Little endian
Cartridge	1 Mbyte	8 MB
Sound	FM synthesis	PCM samples

Overview of the target and host systems

emulator and translated code is about as expensive as a fast context switch.

LOOKING TO THE FUTURE OF BINARY TRANSLATIONS

What does the future hold for binary translations? The creation of binaries that require translation is slowing down. Even in the game industry, coding baselines have moved into C and C++ by and large. Such games can be ported in the traditional manner. Mind you, source code does get lost and—although the language may be directly usable—the source code for support libraries, middleware, and tools may have never been available. Here we can see translation becoming part of a suite of porting techniques, or a forensic tool that assists us in the reconstruction of original code.

Emulation itself will remain a relatively inexpensive way to bring software to a new platform, especially in the game industry. The increasing complexity of target platforms will make the emulators more difficult to write and, in all likelihood, more difficult

to get running fast. Advances in computer architecture are the double-edged swords of emulation. For example, multimedia instruction-set extensions are often quite useful in crossing the semantic gap between target and host graphics systems, but a few years down the road an emulator programmer will be faced with emulating those complex instructions. Despite the addition of those 2D and 3D instruction extensions, there is faint hope of instruction additions that will directly benefit emulator writers. The micro-parallelism inherent in individual instructions and bus decoding will continue to limit the extent to which overhead can be reduced.

Some hope, however, does rest with the increasing prevalence of emulators in the mainstream such as Sun's Java Virtual Machine and Microsoft's .NET VM. There is some market pressure for native processors to run these VMs faster, so writers of other emulators and translators stand to benefit from this. But don't hold your breath for long; increasing support for virtual machines apparently is not the main interest of chip manufacturers

whose different instruction sets engender developer loyalty.

More radical architectures such as MIT's Oxygen processor, or (what has been disclosed of) Sony's PlayStation 3 architecture, present opportunities for faster execution of emulators and translated code. The various parallel subsystems of a target platform such as address decoding, instruction dispatching, instruction execution, and so on map much better onto a set of many small processors than a single, monolithic processor. Instead of stopping and waiting for data or address translation, the process could be pipelined across the processors in

much the same way the real hardware accomplishes the task.

Field programmable gate array (FPGA) technology holds the greatest promise for fast emulators. Reconfigurable hardware is clearly of use in trying to emulate an existing target hardware platform. What remains to be seen, however, is if FPGA features will migrate into mainstream processors. Even a limited ability to construct special-purpose instructions could help. Dispatch-

ing and data translation are common, but expensive, elements in emulators and translators that are generally quite simple at the gate and wire level. Q

LOVE IT, HATE IT? LET US KNOW:

queue-ed@acm.org or www.acmqueue.org/forums

GEORGE PHILLIPS received his master's degree from the University of British Columbia. He is currently working at Digital Eclipse (<http://www.digitaleclipse.com>) as a programmer of both original and emulated video games. He also worked as a Unix system administrator and on a commercial Web crawler.

PETER PHILLIPS received his bachelor's degree in computer science from the University of British Columbia. He has 15 years of experience in software development and system administration and is currently employed as a developer at Digital Eclipse. He is working on 3D PC title and emulation projects.

© 2003 ACM 1542-7730/03/0900 \$5.00.