PNaCl:

Portable *Native Client* Executables

Alan Donovan <adonovan@google.com>
Robert Muth <robertm@google.com>
Brad Chen <bradchen@google.com>
David Sehr <sehr@google.com>

February 22, 2010

Summary

Google's Native Client technology uses software fault isolation (SFI) to enable the execution of untrusted native code inside a web browser, giving web applications greater access to the performance of the client machine while avoiding the security problems with current infrastructure for plugins.

While the operating-system neutrality of Native Client tends to encourage good practices with respect to ISA portability, the burden of building, testing and deploying a program on all supported hardware platforms---currently IA-32, ARM and x86-64---lies with the developer. This arrangement makes it too easy for the developer to fail to support one or more ISAs, and tends to create a barrier for future new ISAs, threatening the portability promise of the Web.

This document describes the design of **PNaCl** (pronounced "pinnacle"), a suite of tools for building, testing, and distributing Native Client programs in an instruction-set neutral format. PNaCl uses the Low-Level Virtual Machine (LLVM) bitcode format to represent ISA-neutral portable executables compiled from code written in a variety of languages including C and C++.

The PNaCl design allows some flexibility in deciding where translation to native machine code occurs. By supporting client-side translation to the client's native instruction set, PNaCl reduces the burden on the developer, enabling support of new instruction sets without recompilation from the source. As PNaCl is layered cleanly on top of current ISA-specific NaCl implementations, the small trusted code base, source language neutrality, and safety properties of the system are preserved.

PNaCl, which will be open-sourced, is still at an early stage of development; we welcome feedback and suggestions on how its design might be improved.

Introduction

The Native Client (NaCl) system allows developers to write components of their web applications in C++ or other languages that compile to object code, to access secure APIs to system services such as graphics and sound hardware, and to execute these components on the client machine without sacrificing the security properties users expect from the Web. Specifically, Native Client programs cannot directly invoke the native operating system, but must use only interfaces that encapsulate system services securely and portably. Note that the same services are accessible to JavaScript programs, albeit with reduced performance.

NaCl modules can thus harness the full power of the client machines' CPUs and other resources, but to do so, a compiled NaCl module must necessarily be tied to a specific instruction set architecture (ISA). This poses a threat to the portability of the web, as application developers must ensure that they build, test and deploy their NaCl modules on all architectures that support NaCl. Currently there are three---ARM, IA-32 and x86_64--- and supporting them requires significant effort plus access to a variety of hardware beyond the means of many developers, but the problem grows only worse as NaCl supports new architectures since each new architecture would require all existing programs to be recompiled. It seems unlikely that developers would tolerate this model; more likely, they would avoid NaCl, or make invalid simplifying assumptions such as "all the world's an x86", potentially leading to fragmentation of the Web.

Throughout this document, we will use the term **Developer** for the person responsible for the design, implementation and testing of the portable executable and **User** for the person interacting with it. The **Server** is the machine from which the NaCl-based web application is served and the **Client** is the user's machine, on which the browser runs.

Goals

The PNaCl project aims to:

1. Provide an ISA-neutral format for compiled NaCl modules supporting a wide variety of target platforms without recompilation from source.
2. Make it easy for NaCl developers to build, test and deploy **portable** executable modules.
3. Support the x86-32, x86-64 and ARM instruction sets initially, but make it straightforward to support other popular general-purpose CPUs in future.
4. Preserve the **security** and **performance** properties of Native Client.

PNaCl allows developers of NaCl modules to use and mix a range of source languages that generate object code (e.g. C, C++, Fortran, Objective C) and to re-use the vast range of existing libraries in those languages with minimal or no modifications.

While preventing developers from writing non-portable programs is not strictly possible, we intend to create an environment that strongly encourages writing portable programs. We will continue to refine the development and testing facilities provided by the system, and expect that a PNaCl module tested on two architectures will work the same way on a third.

Approach

In brief, the approach taken by PNaCl is to adopt the LLVM compiler's intermediate language ("bitcode") as a target platform. Developers compile their source to bitcode and ship the bitcode as a **portable executable**. Clients will include a PNaCl translator that converts the portable executable to a native executable for their machine's ISA, at which point Native Client can validate and execute it.

Figure 1 shows the basic workflow in compiling an executable using LLVM. Compilation is a two-step process. In the first step, the front-end compiler (hereafter just called the "compiler") is invoked on the source files. Syntactic and semantic analysis are performed in the usual way, and the LLVM bitcode files are emitted. As with Native Client, any language that can be compiled to machine code can be supported via this architecture.

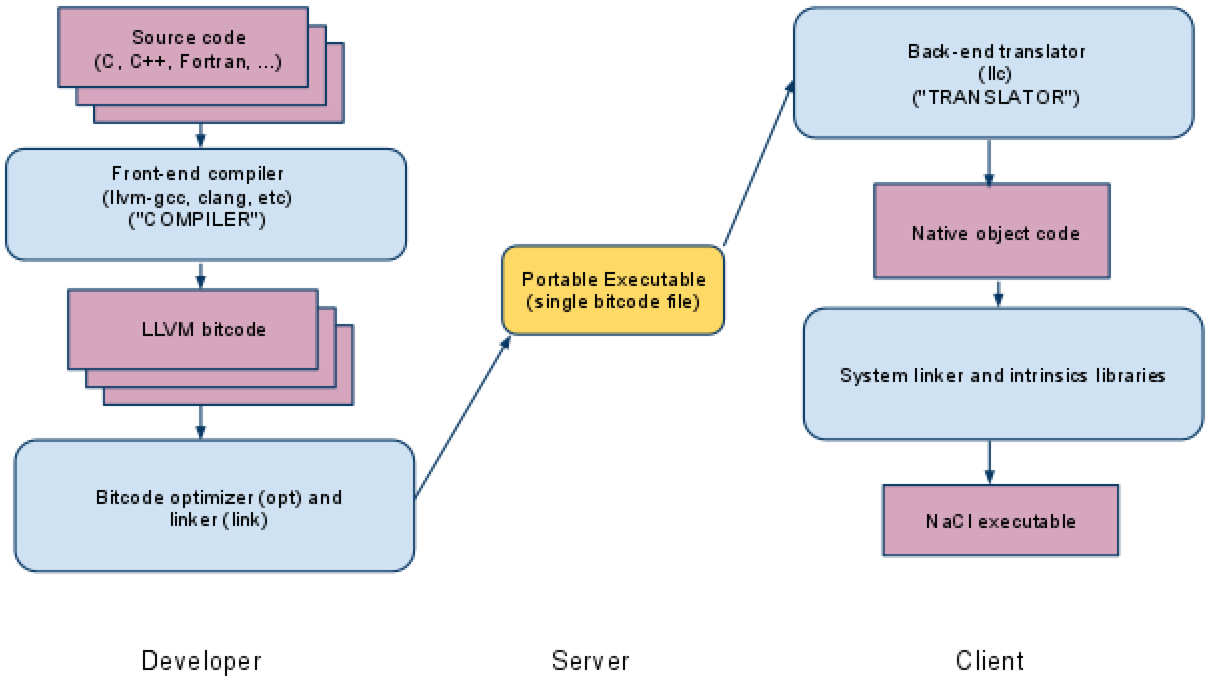

Figure 1. Compilation from source to object code occurs in two steps, unlike a traditional compiler. The intermediate product, an LLVM bitcode file, is distributed. The "traditional" NaCl compilation workflow is also shown.

The bitcode files can then be linked into a single bitcode file with optional optimization passes before and/or after this linking step. (The LLVM toolchain allows great flexibility in the ordering of linking and interprocedural optimization passes, and generates high-quality object code comparable to that of conventional toolchains such as GCC.) The developer deploys the resulting bitcode file which has no external references as portable executable via her web server, together with HTML and other web application components.

The second stage of compilation, which we call "translation", turns this representation into a NaCl executable for the client's instruction set. In the simplest scenario, the translation step occurs on the client machine, although other arrangements may eventually be considered to achieve better code quality or client startup times.

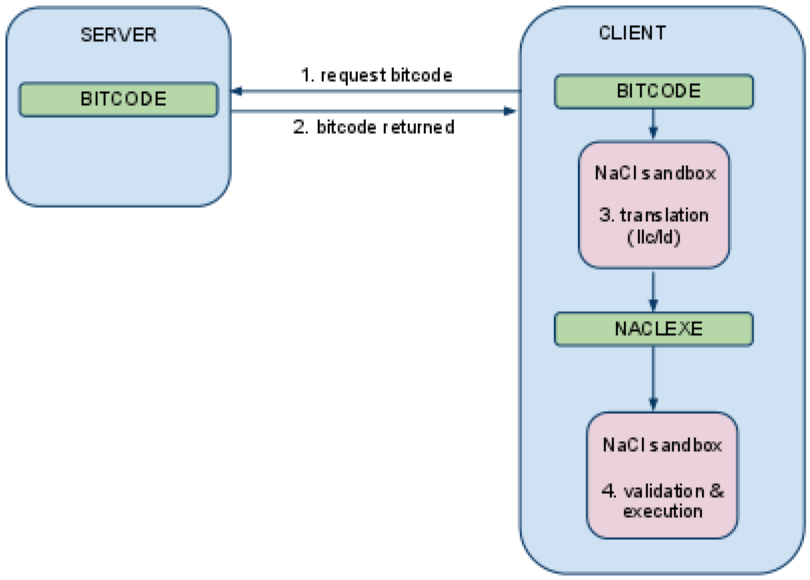

Figure 2. The client requests bitcode from the server then translates it to a native executable for its own architecture. Translation may occur locally (within another NaCl sandbox) or remotely by another entity.

The process of native code generation, assembly and linking, runs inside a Native Client sandbox, so the amount of code added to the client's trusted computing base (TCB) will be very small. By layering PNaCl on top of Native Client, we retain Native Client's strong security properties, and also retain the option of supporting legacy compilers along-side PNaCl.

Programming model

PNaCl should make it easy for developers to write their application once and be confident that it will execute, with identical behavior, on every platform supported by Native Client. While programmers are still required to follow the usual disciplines of source portability, e.g. avoidance of inline assembler code, their task is simplified by the PNaCl programming model which constrains much of the variation between typical computing platforms, such as the relative sizes of integer datatypes.

We have already ported numerous libraries to this ABI and our experience so far suggests that libraries that are already portable to a number of platforms require little effort to port to Native Client. Once the details of the ABI are fully specified, we intend to provide recipes to help programmers port code quickly, for example, by making effective use of compiler flags to detect non-portable features.

The PNaCl two-stage compilation approach has ramifications for the programming model at the level of both bitcode and source code. We will explore these in turn in this section.

Bitcode portability

For a PNaCl program to run properly and predictably on all targets, a number of invariants must be true of the bitcode regardless of its origin and source language. In effect, these invariants comprise the ABI of the "PNaCl machine", the contract between the writers of

PNaCl bitcode (whether man or machine) and the bitcode translator for each architecture. For now, this section gives an brief flavor of the invariants; they will be specified formally in a future document.

First and foremost, portable programs must be well-formed LLVM bitcode files according to the LLVM Bitcode Format specification.

Address Space. The PNaCl machine consists of a flat 32-bit address space (even when implemented on 64-bit hardware). The minimum addressible unit (byte) is 8 bits. Initially only the lower 1GB of the address space will be available. The stack starts at the upper limit, grows downward, and is not executable.

Data types. The addresses of all objects are 32 bits wide. Signed and unsigned integers are represented using 2's complement, and may be of width 8, 16, 32 and 64 bits. Floating point operations follow the IEEE 754 standard; both single (32 bit) and double (64 bits) precision are supported. All datatypes are naturally aligned. LLVM intrinsics may be used to manipulate vectors.

Integer byte-order. Multibyte integers will be encoded least-significant byte first as all our current target platforms are little endian and many other platforms can be switched to little endian, sometimes even on a process by process basis.

Concurrency and memory model. All memory load and store operations within a single thread appear sequentially consistent. No assumptions may be made about the values observed from a load from a memory location stored by another thread other than those guaranteed by the specifications of the LLVM atomics operations. Programmers must ensure that all access to shared state in a multi-threaded program is guarded by locks and free from data races.

Runtime system. Entry points to the runtime system (NaCl Service Runtime) for memory management thread operations, IPC, etc. are provided by trampolines at well-known locations in the address space. Higher level interfaces to these services and other libraries will be provided in bitcode form.

Target-specific assembly. The LLVM bitcode language includes an `asm` operator that represents a string of input to the target machine's assembler. Since this is inherently non-portable, PNaCl forbids its use and no PNaCl compiler will make use of it: bitcode files must be free of target-specific inline assembler expressions and module-level inline assembly. (N.B., these are distinct from *source*-level `asm` directives as described below.)

Source code portability

The rules for writing portable source code using PNaCl extend those of Native Client, which in turn extend conventional good practices for writing programs independent of any particular operating system or hardware platform. The source code rules are of course language-specific, and for brevity we will consider here only the issues for the C language. Regardless of which source language is used, programmers should follow standard practices for portability in that language, such as avoiding compiler-specific constructs and adhering to language standards where applicable.

Data model. The compiler will use the ILP32 model, i.e. sizeof(int) = sizeof(long) = sizeof(void*) = 4, i.e. 32 bits. sizeof(long long) = 8, i.e. 64 bits.

Conditional compilation. The compilation model of the C language exposes many details of the target machine as early as preprocessing, allowing programmers to select between different implementations of the same function based on the target architecture. For example, intrinsics for vector manipulation are typically defined using target-specific inline assembly. This "early binding" technique should not be used with PNaCl since it requires knowledge of the target platform that tends to introduce portability defects.

PNaCl programs should avoid the use of conditional compilation to abstract over architecture-specific differences, and should use C code plus LLVM intrinsics to express the behavior so as to be portable to all PNaCl targets. While this would be less efficient with a conventional toolchain since it defeats opportunities for optimization, LLVM's use of link-time interprocedural optimization can often completely eliminate the extra cost. Consider an example portable program of the form:

```
if (has_feature_x()) impl1(); else impl2();
```

where `has_feature_x()` is an intrinsic provided at translation time that returns a constant. The optimizer will perform constant propagation, dead-code elimination and inlining, and the resulting object code will be as good as if the body of `impl1` or `impl2` had been selected as early as preprocessing time.

Inline asm expressions. GCC inline asm expressions are permitted in C source code, but the target machine for such expressions is the PNaCl machine, i.e. LLVM bitcode, not the assembly language of any particular CPU. This feature may be useful for accessing LLVM intrinsics, such as low-level routines for portable vector manipulation.

Feedback

PNaCl is still at an early stage of development, and we welcome feedback and suggestions. Please join the discussion at **`native-client-discuss@googlegroups.com`**.

Related Links

- http://code.google.com/p/nativeclient/
- http://llvm.org/
- http://llvm.org/docs/BitCodeFormat.html