Stability of the PNaCl bitcode ABI

Document version: 0.1 (June 17th, 2013)

Introduction

The goal of this document is to define the PNaCl stable bitcode ABI for the first release of Chrome in which PNaCl is enabled for the open web. This will be the first release in which PNaCl starts guaranteeing ABI stability.

What is meant by ABI stability

ABI stability for PNaCl bitcode means the guarantee of *backwards compatibility* - described by the following scenario:

- A developer creates a PNaCl application and places it online. The application is a .pexe generated with the toolchain of Chrome release M.
- At some point in the future, a user wants to run the .pexe with Chrome release N (with N >= M).
- Even though the PNaCl translator distributed with Chrome has likely changed between releases M and N, we guarantee that the .pexe continues working in the new release.

Conversely, the related issue of *forward compatibility* is avoided by explicitly refusing to load newer bitcode versions in older Chrome translators, as follows:

- As before, a user generates a .pexe with Chrome release M and places it online.
- A user tries to run the .pexe with Chrome release K with K < M. Note that this only happens if the user didn't update Chrome, which is unlikely.
- The PNaCl translator reads the bitcode version from the .pexe and sees it's a newer version than it supports. The translator refuses to load the .pexe and emits an error message.

The scope of this document

This document aims to define the requirement for PNaCl bitcode ABI compatibility, and lay out the steps we plan to undertake to reach a stable ABI. The goal here is not to provide a reference documentation of the ABI, but rather to facilitate discussion within the team and write down our collective thoughts on the subject.

PNaCl changes not related to bitcode stability

PNaCl also has differences from standard LLVM targets which are unrelated to bitcode stability. These are generally beyond the scope of this document, but to summarize, they fall into 2 categories: 1) those related to portability across architectures, and 2) those related to Native Client and its security sandboxing model.

For example, the architecture is defined as “le32” indicating that the architecture is a generic little-endian machine with 32-bit pointers. The target triple is `le32-unknown-nacl`, and the compiler #defines `__pnacl__` when the architecture is le32 and `__native_client__` when the OS is NaCl. The data model is ILP32, which means that ints, pointers, and longs are all 32 bits wide.

Additionally, clang’s ABI lowering for calling conventions is different from that used by hardware architectures (because they are different from each other, even for the same C types). There are a few additional restrictions on PNaCl programs such as that they may only use IEEE floating-point types.

The use of Native Client to securely sandbox PNaCl code also imposes some restrictions on programs: however the details of the sandbox implementation are subject to change and should not generally be exposed to developers.

Goals

The main and obvious goal of PNaCl bitcode ABI stability is to support backwards compatibility of PNaCl bitcode. As the section above describes, it's essential to allow existing bitcode placed online to run on newer versions of Chrome. This is similar to any standard-abiding web platform application (HTML+CSS+JS) that is expected to keep working in newer browsers.

An additional goal is to define a smaller and simpler subset of the LLVM IR for PNaCl bitcode. A smaller and simpler subset is easier to test and validate. Furthermore, if an alternative fast code generator for PNaCl is ever considered, such a subset will be easier to implement.

Initially, the goal of this document is to help deciding which subset of the LLVM IR will be part of the PNaCl bitcode ABI in the first public release.

Challenges for bitcode ABI stability

As the introduction above explains, we aim to define a stable ABI that will enable a "new" translator to read "old" bitcode successfully and compile it to native code that is validated and executed on the user's machine. Further, we want to benefit from advances in LLVM technology and use progressively newer LLVM releases for future versions of the PNaCl translator.
On the other hand, our usage of LLVM IR as the portable representation for PNaCl implies that we're dealing with a changing compiler IR that doesn't have stability as one of its declared goals.

As a more concrete example: assuming we "freeze" the bitcode ABI on LLVM release 3.3 (which is due sometime in June 2013), in a future PNaCl translator release we'll want to use the up-to-date LLVM, say 4.0. There's a very high probability that LLVM IR will change between 3.3 and 4.0 to the extent that a bitcode reader from LLVM 4.0 won't be able to read IR generated by the front-end of LLVM 3.3.

This incompatibility in IR between LLVM releases manifests in three ways:

- Semantic changes in IR. For example: type system rewrite in 3.0, attribute changes in 3.3, possible future removal of IR instructions.
- Changes in binary LLVM bitcode encoding.
- "Silent" change in bitcode usage between passes that occur before and after pexe creation. As PNaCl uses a newer LLVM, the pexe will have been created with an older version which might emit bitcode that now looks odd to newer LLVM, hindering optimizations or exposing bugs that do not exist in upstream LLVM (where the bitcode never goes out of sync).

Approaches to maintaining backwards compatibility of bitcode

This discussion knowingly ignores the option where we fork LLVM from some given version and maintain our own “stable” version of it for PNaCl. In other words, it’s assumed that the LLVM backend (code generation and emission) keeps evolving in the PNaCl translator.

Approach #1: a single supported bitcode version

We define a stable bitcode in the first public release of PNaCl. This is the bitcode that all subsequent releases will generate.

Pros:

- Only a single version of the bitcode reader is required. Each version of the translator will have to convert this stable bitcode to its up-to-date in-memory representation for consumption in the backend.
- Provides forward-compatibility to some extent. Since future PNaCl front-ends will continue generating the stable bitcode, older translators will be able to read them as well.

Cons:

- If future versions of LLVM add new features to the IR that we want to provide as part of PNaCl, we will have to backport upstream’s BitcodeReader/BitcodeWriter changes to PNaCl’s copy of the bitcode reader/writer. This might get harder if upstream’s bitcode reader/writer diverges.
- Requires more maintenance of the PNaCl frontend, assuming that we want to keep following the advances in bug fixes in Clang. Newer versions of Clang will generate newer IR, which we’ll have to “cut down” to the stable subset. Note that some changes (such as rewrites or removal of instructions) may be non-trivial.

Approach #2: evolving the stable bitcode over time, while maintaining backwards compatibility

As with approach #1, we define a stable bitcode in the first public release of PNaCl. However, in future releases we may change the bitcode ABI, as long as we ensure that newer translators can still read older bitcode formats.

Pros:

- We can enjoy advances in LLVM in the future. For example, new features that get added to the IR and aid optimizations.
- Easier to follow upstream Clang in the PNaCl front-end. If Clang changes the bitcode it generates in significant ways, we can incorporate these changes into the new stable bitcode ABI.

Cons:

- This approach requires maintaining N bitcode readers: one per supported bitcode version. The translator will have to recognize the bitcode version and apply the appropriate reader that knows how to translate that version of the bitcode into the new translator in-memory structures.
    - We will have to maintain N readers. If LLVM's C++ API for building in-memory IR changes, we will have to backport upstream changes to N bitcode readers.
    - This will increase the size of the PNaCl translator.
- Forward compatibility is impossible in this approach since older translators won't be able to read new .pexes. This isn't a big issue if we assume PNaCl auto-updates like Chrome does. This can be mitigated with an SDK option to target older PNaCl versions.

Approach #3: hybrid between #1 and #2

Since there is currently a large unknown with regards to the changes LLVM and Clang may undergo in the near and far future, it may make sense to start with approach #1, keeping the option to switch to #2 at some point.
It's crucial that we guarantee backwards compatibility by always being able to read and translate the bitcode generated by the first public release of PNaCl. Whether we'll allow newer versions of this bitcode to also be generated and read by future toolchains is an implementation detail that should be transparent to developers and users.

Steps towards a stable bitcode ABI

The PNaCl portable bitcode is based on LLVM IR. For the sake of defining a stable bitcode ABI, a number of simplifications and restrictions have to be applied to LLVM IR in order to synthesize a subset that can remain stable over time.

The optimizers will generally work best if they are allowed to operate on the current version of the IR in its full generality; therefore the general plan is to run Clang and the target-independent optimizations first, and then transform their output into the stable subset in a final phase called ABI legalization.

On the translator side, a step called “ABI verification” is performed. This verifies that the downloaded .pexe conforms to the PNaCl bitcode ABI. The verifier is the “ultimate source of truth” for what is and what isn’t part of the bitcode ABI. A snapshot of its current implementation is available online:
http://git.chromium.org/gitweb/?p=native_client/pnacl-llvm.git;a=tree;f=lib/Analysis/NaCl

The following sections describe the concrete ABI legalization steps we are considering. It also attempts to provide reasons in favor and against these transformations. An implicit argument against all transformations is the engineering effort required to implement and maintain them (which includes reviewing and testing with each LLVM merge). On the other hand, an implicit argument in favor of any simplification is reduced exposure to changes in the bitcode semantics upstream; only changes in parts of the IR that are accepted as stable would require using one of the backward compatibility approaches above. The other benefit of simpler IR is that it may allow simplification of the translator. This could allow the translator to be smaller and/or faster, and also have a reduced testing and attack surface. Even though we validate and run the translator itself in a NaCl sandbox, fewer supported IR features means fewer potentially exploitable bugs that may break one of the security layers.

Note: regarding Challenges for bitcode ABI stability, this section will focus on the semantic aspects of LLVM IR, ignoring the issue of binary encoding.

Stripping LLVM metadata

Description: LLVM metadata describes constructs like debug information and optimization hints (for example TBAA and branch weights). By design, metadata is inherently unstable and has undergone considerable semantic and structural changes in the past. Metadata should be stripped from .pexes and not be part of the stable ABI.

Note: we may consider adding back specific kinds of metadata in the future in order to support some forms of debug information (especially to reconstruct stack traces to give developers better user crash dumps) and/or aid certain optimizations.

Issue: https://code.google.com/p/nativeclient/issues/detail?id=3348

Pros:

- The inherent instability of metadata makes it a very improbable candidate for inclusion into the stable ABI.
- Reduction of .pexe size.
- Reduction of the test surface of the translator - we can avoid testing its handling of metadata.

Cons:

- For debug metadata, this forces a slightly more complex flow of development. We want debug metadata to be stripped from .pexes by default, even if the developer initially compiled his C/C++ code with `-g`.
- For optimization hints, this may cause the translator to miss some optimizations. This is, however, questionable: IR-level optimizations are done before the .pexe is generated, and in practice we didn't observe difference in the translation result with and without TBAA metadata.
- No stack traces on crash.
- Developer-side incompatibilities are still a problem (may need to do clean rebuilds after toolchain switch, and there is no clear error message when there are incompatibilities).

Rewriting vararg calls

Description: Functions that accept variable arguments and calls to them should be replaced with a scheme where the arguments are explicitly packed in a buffer (allocated by the caller). A pointer to this buffer will be passed into the function instead of the variable argument list. `va_*` instructions and intrinsics within the function will be rewritten to read the arguments from the explicit buffer.

Issue: https://code.google.com/p/nativeclient/issues/detail?id=3338

Pros:

- vararg calls are tricky to implement and aren't fully supported by the LLVM backend for all architectures.
- This transformation simplifies the ABI while being safe because there are no vararg calls from PNaCl bitcode to native code.
- This could provide some performance improvements:
    - "opt" can sometimes better optimise explicit buffer-passing than varargs calls.
    - On x86-64, a varargs function no longer has to spill all arguments to the stack.

Cons:

- Produced bitcode will be larger due to the explicit argument packing and unpacking. However, the effect isn't likely to be large due to the low number of vararg functions and calls in typical code.
- Calling a varargs function without it being declared with a C function prototype will produce code that doesn't work.

Expanding constant expressions

Description: LLVM supports the concept of constant expressions (expressions involving other constants) that can be used as arguments to instructions. Constant expressions should be expanded to an explicit sequence of equivalent LLVM instructions.
For handling of constexprs in the context of global initializers, see Restrict usage of structs, arrays and pointers in the stable bitcode.

Issue: https://code.google.com/p/nativeclient/issues/detail?id=3337

Pros:

- Simplifies the bitcode.
    - Each LLVM instruction will represent a small unit of computation, rather than the arbitrarily many computations that are introduced by nested constant expressions.

Cons:

- Increased bitcode size. Initial measurements show that this change has slightly increased the size of gzipped bitcode. See the issue for more details.
- ~~Impact on performance~~: we haven't noticed a measurable impact on performance.

Strip fastcc-related attributes

Description: the fastcc calling convention in LLVM is target-dependent, devised to speed up internal function calls by passing arguments in registers even on architectures where the standard calling convention does not do so (e.g. 32-bit x86). Calling convention specifications (and a related `inreg` parameter attribute) should be stripped from the bitcode.
Instead, we can mark all compatible `internal` functions fastcc explicitly in the translator for x86-32.

Issue: https://code.google.com/p/nativeclient/issues/detail?id=2346

Pros:

- We don't gain from having this hint applied in the front-end. We can simplify the ABI by not supporting it, and instead mark the calling convention internally for architectures where it makes sense (x86-32 in this case).

Strip system-specific function attributes

Description: strip some system-specific function attributes, like `naked` and `alignstack`.

Issue: https://code.google.com/p/nativeclient/issues/detail?id=3415

Pros:

- Such attributes don't make sense in a stable platform-independent ABI.

Disallow inline assembly

Description: Since inline assembly is inherently non-portable, we should reject bitcode with inline assembly in it. It will not be part of the stable ABI.

Issue: https://code.google.com/p/nativeclient/issues/detail?id=3126, https://code.google.com/p/nativeclient/issues/detail?id=2345

Pros:

- Inline assembly is non-portable.

Cons:

- No inline assembly, even if the original code has preprocessor macros for many target platforms.

Only allow whitelisted intrinsics and external symbols

Description: Only intrinsic functions that are known to exist (and correctly implemented by the backend) across all platforms should be callable from bitcode. Also, only external symbols that we know are provided by the NaCl runtime should be allowed.

Initial whitelist for intrinsics:

- `llvm.nacl.read.tp` for thread pointer.
- `llvm.memcpy.*.*`, `llvm.memset.*.*`, `llvm.memmove.*.*` (but need an implementation for it and need to stop exporting them).
- `llvm.trap`
- `llvm.bswap.{i16,i32,i64}`
- `llvm.sqrt.{f32,f64}` (errno checking done by library, domain assumed good).

Initial whitelist for external symbols:

- `_start`
- `setjmp`, `longjmp`. Will use specially named intrinsics that get translated to native implementations by the translator.
- `__sync_*` for atomics. This is still being experimented with, see Memory Model, Atomics, Fences. The main issue is that not all platforms necessarily provide atomic versions of all of these, but in practice they may all be lowered into single instructions by specific backends.

Intrinsics disallowed:

- `llvm.eh.*`
- `llvm.vacopy, llvm.vaend, llvm.vastart`
- `llvm.returnaddress, llvm.frameaddress`
- Transcendentals: `llvm.sin.*`, `llvm.cos.*`, `llvm.log*`, `llvm.exp*`, `llvm.pow*`
    - Disallow for now, and rely on consistent library implementation. Explore later with fast-math flags.
- `llvm.flt.rounds`: Disallow for now and expand to “1”. Enable in the future after adding an `llvm.flt.set.rounds` intrinsic.
- `llvm.expect`: Disallow for now. Have frontend normalize these to branch weight metadata. Optimize the IR using branch weight metadata, then strip the metadata.
- Many others.

Intrinsics, which seem harmless TBD:

- Other gcc `__builtin_*` intrinsics:
    - `llvm.prefetch`
    - `llvm.ctlz.i32, llvm.cttz.i32, llvm.ctpop.i32`

Issue: https://code.google.com/p/nativeclient/issues/detail?id=3378

Pros:

- For a stable ABI it's important to define exactly what external symbols are allowed to appear in the bitcode. These symbols will have to be provided by future releases of the runtime, or lowered in the translator; therefore, they must be carefully restricted.
- This allows us to strictly provide only those intrinsics that are supported in a cross-platform way by the translator.

Restrict bitcode integer and floating-point types

Description: Restrict the bitcode integer types supported by the stable ABI to `i1, i8, i16, i32` and `i64`. Also, restrict the floating-point types to `f32, f64`.

Issue: https://code.google.com/p/nativeclient/issues/detail?id=3360

Pros:

- Simplifies the scope of bitcode ABI. Since our current aim is for C and C++ on mainstream 32 or 64-bit CPUs, these sizes seem to be sufficient.
- Can have a positive effect on translation time. For the CPUs we target, type legalization that runs during instruction selection would have to reduce types to the ones mentioned above anyway.

Cons:

- Produced code may be potentially slower due to missing an optimization. However, this is questionable since we haven't yet seen an example of it happening.

Restrict integer arguments and return values to at least `i32`

Description: LLVM currently allows function return types such as `zeroext i8` and `signext i8`. This complicates the language the PNaCl translator has to handle. We should expand such arguments and return values to always be `i32`, and handle the sign/zero extensions explicitly in IR code.

Issue: https://code.google.com/p/nativeclient/issues/detail?id=3342

Pros:

- Simplifies the scope of bitcode ABI.
- Removes potential non-portable behaviour, to prevent a pexe from accidentally behaving differently on x86-32 and x86-64 (for example).

Fixed TLS layout for PNaCl

Description: Currently, TLS variables are part of the bitcode ABI. The backend computes the actual layout of TLS variables, which is target dependent. We can compute the TLS layout in advance and encode it in the bitcode. This will leave only the intrinsic that obtains the thread pointer in the ABI.

Issue: https://code.google.com/p/nativeclient/issues/detail?id=2837

Pros:

- Simplifies the ABI.
- Avoids a target dependency and thus simplifies the user threading library (removing the need to support multiple TLS variable layouts).
- Removes the need for an ELF linker to allocate TLS variable layout.
- Allows the code generator to use shorter instruction sequences for accessing thread-local variables, because thread pointer offsets are known at code gen time.

Cons:

- Limits interoperability with TLS-using native code that's not built with PNaCl, but this is currently out of scope.

Restrict usage of structs, arrays and pointers in the stable bitcode

Description: Struct types add expressiveness to LLVM IR and aid in compiling from C/C++, but play no important role in code generation (after all the mid-level IR passes have run). Since structs can be arbitrarily nested and complex, removing them from the stable bitcode ABI can help simplify it. Similar simplification can be applied to other derived types like arrays and pointers. This also allows to remove the `getelementptr` instruction from the stable bitcode.

Issues:

- https://code.google.com/p/nativeclient/issues/detail?id=3343
- https://code.google.com/p/nativeclient/issues/detail?id=3113 (handling of global initializers)

Concrete proposal details

Struct types and arrays are removed from the stable bitcode ABI. They will instead be represented by pointers. Structs in registers and usages of `extractvalue/insertvalue` will be expanded to be `alloca`'d instead, and can then be replaced by pointers. Efficient copies of structs can be replaced by `llvm.memcpy` calls.

Pointers are replaced by `i32` (valid for PNaCl which explicitly sets pointer size to 32 bits in the ABI). Some restricted uses of pointers will be allowed in order to preserve existing semantics of LLVM instructions. For example:

- `alloca` instructions return pointers
- `load` and `store` instructions require pointers
- `llvm.memcpy` intrinsics require pointers

The `ptrtoint` and `inttoptr` instructions will be employed at conversion points to bridge between addresses contained in `i32` values and pointers for the instructions/intrinsics listed above.

Handling `getelementptr`

`getelementptr` will be expanded to a sequence of equivalent arithmetic instructions acting on `i32` instead of pointers. `getelementptr` is a complex instruction and it's worthwhile excluding it from the stable bitcode ABI.
Note: some measurements indicate that programs compiled with fast isel on x86-64 have performance regressions after GEP removal. This is in accordance to expectations, since fast isel has a special optimization for folding GEPs into addressing modes.

Handling global initializers

A problem with removing the derived types like structs and arrays is handling global initializers, which can't be easily expanded into code.

For example, this C code:

```
struct S {
  struct Pair {
    char c1;
    char c2;
  } p;
  int *ptr;
};
extern int global;
struct S x[] = { { { 1, 2, }, &global },
                 { { 3, 4, }, &global + 1 } };
```

In LLVM this currently gets represented as:

```
@x = global [2 x %struct.S]
  [%struct.S { %struct.Pair { i8 1, i8 2 }, i32* @global },
   %struct.S { %struct.Pair { i8 3, i8 4 },
               i32* getelementptr (i32* @global, i64 1) }], align 16
```

There are two problems here:

- We want to get rid of arbitrarily nested derived types like `struct.S`
- The reference to global within the initializer of `x` requires a non-trivial constant expression, which we're also interested in expanding out

The currently leading proposal is to get rid of such initializers in the stable bitcode (wire format) while still using a restricted form of structures and pointers in the bitcode reader and writer in order to keep inter-operating with the existing LLVM backend facilities.

In the bitcode the above initializer will be encoded as two arrays:

- An array of bytes
- An array of 32-bit "relocations" to globals; a relocation is a pair `(offest_into_bytearray, global_var)`.

For the example above, this is:

```
Byte array:
```

```
  [1, 2, 0, 0,  // c1, c2, padding
   0, 0, 0, 0,  // addend 0: @global's address will be added
   3, 4, 0, 0,  // c1, c2, padding
   4, 0, 0, 0]  // addend 4: @global's address with be added
Relocations:
  [(4, @global),
   (12, @global)]
```

The LLVM IR that will be produced by bitcode reader for this encoded data is:

```
%flattened_data = type <{ [4 x i8], i32, [4 x i8], i32 }>
@x_flattened_as_struct = global %flattened_data <{
  [4 x i8] c"\01\02\00\00",
  i32 ptrtoint (i32* @global to i32),
  [4 x i8] c"\03\04\00\00",
  i32 add (i32 ptrtoint (i32* @global to i32), i32 4)
}>
```

This re-introduces structs and a restricted usage of constant expressions into LLVM IR, but we are not concerned about IR inside the backend. The goal is to not have these constructs “on the wire”.

For producing such initializers, the PNaCl front-end will include a pass that rewrites global initializers into the “flattened” form shown above, and the bitcode writer will translate it to the byte array + relocations form.

Define `__{init|fini}_array_{start|end}` at bitcode creation time

Description: LLVM defines the special global variables `llvm.global_ctors` and `llvm.global_dtors` to list global constructors and destructors. On the binary executable level, the special symbols `__init_array_start/end` and `__fini_array_start/end` are expected instead by the runtime. We should convert the former to the latter during linking instead of in the translator, so that the special globals variables don't make it part of the ABI.

Issue: https://code.google.com/p/nativeclient/issues/detail?id=3018

Pros:

- Smaller scope of stable bitcode ABI.
- It is an obstacle to removing the translator's use of the binutils linker (ld).

C++ exception handling ABI

Not supported in first release. We'll require code compiled with `-fno-exceptions`. ABI checker makes sure that landingpad and invoke instructions are not being used in the bitcode.

This issue tracks disallowing invoke/landingpad:
https://code.google.com/p/nativeclient/issues/detail?id=3377

Issues for adding exception handling in a later release:
https://code.google.com/p/nativeclient/issues/detail?id=3118
https://code.google.com/p/nativeclient/issues/detail?id=2798

Untrusted Fault Handling

Not supported in first release.

Memory Model, Atomics, Fences

Syscalls

One option for atomic and fences is to expose all primitives through NaCl syscalls. The below solution can still generate code that goes through syscalls if needed, but is cleaner in that it doesn't expose this implementation detail in the pexe. Simply using syscalls also doesn't specify a memory model for pexes.

C11/C++11

We can reuse a subset of the C11/C++11 memory model, atomics and fences, as represented in LLVM's IR. Our goal is for PNaCl to provide a portable memory model and atomic operations for a single program and its threads. This implies that we don't need to support device memory or cross-program communications through these primitives. In the future, supporting other languages or architectures where either C++'s or LLVM's representation are insufficient may involve adding more intrinsics or external functions to what we expose.

One point to note is that C++11 also defines limited threads, if we use its memory model and atomics then we should also allow users to use its thread library. There will be some work in supporting the latest runtime libraries to make this happen. For now we expose pthreads, but a future release will support full C11/C++11 runtime.

http://clang.llvm.org/docs/LanguageExtensions.html#c11-atomic-builtins
http://gcc.gnu.org/wiki/Atomic/GCCMM/LIbrary

LLVM and GCC also support instructions that aren't strictly in the standard (nand, min, max, ...) as well as sizes which PNaCl doesn't support (128-bit integral values). We'll avoid supporting these for the first release: the representation we choose for C11/C++11 primitives is likely to evolve portably with LLVM releases, but non-standard features may not be as forward-compatible.

Memory Ordering

C11/C++11 offer 6 different memory orderings, and it's unclear if LLVM's implementation is correct for all backends of interest, and if user code will exhibit portable behavior (e.g. it may work on x86 but be incorrect on ARM). One option for a first release would be to change all orderings to sequential consistency in the .pexe, and consider relaxing this requirement in future releases. This would allow code to compile properly, leave older .pexes in a stable state, and allow us to broaden what's accepted in a later release.

Memory accesses which aren't explicitly marked as atomic may be reordered, split, fused, or elided. They are emitted by the target machine as-if they were executing on a single thread,

unobservable by other threads. This is a source of undefined behavior.

Volatile

The standard mandates that volatile accesses execute in program order (but are not fences, so other memory operations can reorder around them), are not necessarily atomic, and can't be elided/fused. Everything else is very target-specific.

The C++ standards committee had a paper about volatile, and decided to leave it unspecified for reasons that PNaCl doesn't encounter:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2016.html

For PNaCl, we can therefore let the optimizer have its way with volatiles on the developer's machine, but change all volatile accesses to intrinsics/external functions with a specified memory ordering at .pexe creation time. This would isolate a .pexe from future changes in the translator. We could use relaxed ordering (which has stronger semantics than the standard mandates), though the more conservative approach may be to transform all of them to sequentially consistent (even stronger). This eases our support of legacy code, and combined with builtin fences these programs can do meaningful cross-thread communication without changing code.

Pros:

- "Porting" to PNaCl means using C11/C++11 or the `__sync_*` primitives, which is not wasted work. It's therefore easier to adopt PNaCl, and a lot of existing code will "just work".
- The generated code will be faster than going through NaCl mutex syscalls.
- Regular NaCl has access to these atomic instructions, PNaCl doesn't.
- `volatile` in code "just works" on all target platforms.

Cons:

- 64-bit atomics support may have degraded performance on some MIPS targets (pre-MIPS3?), compared to x86-32, x86-64 and ARMv7. This means that on some platforms where there is no instruction then the implementation must use a lock instead.
- The above restriction means that atomics may perform much slower on different platforms. This is really a limitation of platforms, not PNaCl, but it means that timing can be very different from one platform to another, leaking some of the platform-independent abstraction.
- Accessing the same memory location without atomics can still have potentially racy behavior, which is also part of the C11/C++11 standard (all such accesses must be through an atomic variable, anything else is undefined).
- Atomic accesses are only guaranteed to work on aligned addresses. This is as specified by the C11/C++11 standard, and is often a limitation of the platform too. We could mitigate this with tools like LLVM's sanitizers.
- Ignores multi-program communication as well as devices. So do NaCl and C11/C++11.

- The “fast” variants of C++11’s atomics don’t really make sense in the context of PNaCl because their size must be chosen before the target platform is known.

Undefined Behavior

C and C++ undefined behavior allows efficient mapping of the source language onto hardware, but leads to different behavior on different platforms.

PNaCl exposes undefined behavior in the following ways:

- The Clang front-end and optimizations that occur on the developer's machine determine what behavior will occur, and it will be specified deterministically in the pexe. All targets will observe the same behavior. In some cases, recompiling with a newer PNaCl SDK version will either:
    - Reliably emit the same behavior in the resulting pexe.
    - Change the behavior that gets specified in the pexe.
- The behavior specified in the pexe relies on IR, runtime or CPU architecture vagaries.
    - In some cases, the behavior using the same PNaCl translator version on different architectures will produce different behavior.
    - Sometimes runtime parameters determine the behavior, e.g. memory allocation determines which out-of-bounds accesses crash versus returning garbage.
    - In some cases, different versions of the PNaCl translator (i.e. after a Chrome update) will compile the code differently and cause different behavior.

Specification

Our goal is that a single pexe should work reliably in the same manner on all architectures, irrespective of runtime parameters and through Chrome updates. We don't believe that this goal is fully attainable for a first release, we've therefore specified as much as we could and will continue improving the situation in subsequent releases.

One interesting solution could be to offer good support for LLVM's sanitizer tools (including UBSan) at development time, so that developers can test their code against undefined behavior. Shipping code would then still get good performance, and diverging behavior would be rare.

Note that none of these issues are vulnerabilities in PNaCl and Chrome: the NaCl sandboxing still constrains the code through Software Fault Isolation.

Well defined for PNaCl bitcode:

- Dynamic initialization order dependencies → the order is deterministic in the pexe.
- Bool which isn't 0/1 → the IR instruction sequence is deterministic in the pexe.
- Out-of-range enum → the backing integer type and IR instruction sequence is deterministic in the pexe.
- Reaching end-of-value-returning-function without returning a value → reduces to "ret i32 undef" in IR.
- Aggressive optimizations based on type-based alias analysis → TBAA optimizations are

done before stable bitcode is generated and their metadata is stripped from the pexe, behavior is therefore deterministic in the pexe.
- Operator and subexpression evaluation order in the same expression (e.g. function parameter passing, or pre-increment) → the order is defined in the pexe.
- Signed integer overflow → two's complement integer arithmetic is assumed.
- Atomic access to a non-atomic memory location (not declared as std::atomic) → atomics all lower to the same compatible intrinsics or external functions, the behavior is therefore deterministic in the pexe.
- Integer divide by zero → always raises a fault (through hardware on x86, and through integer divide emulation routine or explicit checks on ARM).

Will not fix:
- Null pointer/reference has behavior determined by the NaCl sandbox:
    - Raises a segmentation fault in the bottom 64KiB bytes on all platforms, and on some sandboxes there are further non-writable pages after the initial 64KiB.
    - Negative offsets aren't handled consistently on all platforms: x86-64 and ARM will wrap around to the stack (because they mask the address), whereas x86-32 will fault (because of segmentation).
- Accessing uninitialized/free'd memory (including out-of-bounds array access):
    - Might cause a segmentation fault or not, depending on where memory is allocated and how it gets reclaimed.
    - Added complexity because of the NaCl sandboxing: some of the load/stores might be forced back into sandbox range, or eliminated entirely if they fall out of the sandbox.
- Executing non-program data (jumping to an address obtained from a non-function pointer is undefined, can only do `void(*)()` → `intptr_t` → `void(*)()`.
    - Just-In-Time code generation is supported by NaCl, but will not be supported by PNaCl's first release. It will not be possible to mark code as executable in the first release.
    - Offering full JIT capabilities would reduce PNaCl's ability to change the sandboxing model. It would also require a "jump to JIT code" syscall (to guarantee a calling convention), and means that JITs aren't portable.
    - PNaCl could offer "portable" JIT capabilities where the code hands PNaCl some form of LLVM IR, which PNaCl then JITs.
- Out-of-scope variable usage → will produce unknown data, mostly dependent on stack and memory allocation.
- Data races: any two operations that conflict (target overlapping memory), at least one of which is a store or atomic read-modify-write, and at least one of which is not atomic → this will be very dependent on processor and execution sequence.

Potentially can fix:
- Shift by greater-than-or-equal to left-hand-side's bit-width or negative.
    - Some of the behavior will be specified in the pexe depending on constant

propagation and integer type of variables.
    - There is still some architecture specific behavior.
    - PNaCl could force-mask the right-hand-side to bitwidth-1, which could become a no-op on some architectures while ensuring all architectures behave similarly. Regular optimizations could also be applied, removing redundant masks.
- Using a virtual pointer of the wrong type, or of an unallocated object.
    - Will produce wrong results which will depend on what data is treated as a vftable.
    - We could add runtime checks for this, and elide them when types are provably correct.
- Some unaligned load/store (also, see atomics above) → could force everything to `align 1`, performance cost should be measured. The frontend could also be more pessimistic when it sees dubious casts.
- Reaching "unreachable" code.
    - LLVM provides an IR instruction called "unreachable" whose effect will be undefined. We could change this to always trap, as the `llvm.trap` intrinsic does.
- Zero or negative-sized variable-length array (and alloca) → can insert checks with -fsanitize=vla-bound.

Floating point [Unsorted, will not all be resolved for first launch]:

- Partial IEEE754 implementation:
    - Float cast overflow.
    - Float divide by zero.
    - Different rounding modes.
        - Could support switching modes (the 4 modes exposed by C99 FLT_ROUNDS macros).
    - Default denormal behavior → we could mandate flush-to-zero, and give an API to enable denormals in a future release.
    - Fast-math optimizations:
        - We'll strip fast-math from the pexe for now, but could enable it at a later date, potentially at a perf-function granularity.
        - Flush-to-zero → on or off by default? Provide an API?
        - Canonical NaN, if any → probably hard to guarantee.
        - Ignore NaN to commute operands → only valid with fast-math.
        - NaN signaling → fault.
        - Passing NaNs to STL functions (the math is defined, but the function implementation isn't, e.g. std::min/std::max) → will be well-defined in the pexe.
        - Fused-multiply-add have higher precision and often execute faster → we could disallow them in the pexe, and only generate them in the backend if fast-math was specified.
        - Transcendentals → we could ship with emulation routines whose behavior is well-known, and use the hardware if fast-math is provided.