Ninja

Evan Martin

Ninja is a build system similar to Make. As input you describe the commands necessary to process source files into *target* files. Ninja uses these commands to bring targets up to date. Unlike many other build systems, Ninja's main design goal was speed.

I wrote Ninja while working on Google Chrome. I started Ninja as an experiment to find out if Chrome's build could be made faster. To successfully build Chrome, Ninja's other main design goal followed: Ninja needed to be easily embedded within a larger build system.

Ninja has been quietly successful, gradually replacing the other build systems used by Chrome. After Ninja was made public others contributed code to make the popular CMake build system generate Ninja files–now Ninja is also used to develop CMake-based projects like LLVM and ReactOS. Other projects, like TextMate, target Ninja directly from their custom build.

I worked on Chrome from 2007 to 2012, and started Ninja in 2010. There are many factors contributing to the build performance of a project as large as Chrome (today around 40,000 files of C++ code generating an output binary around 90 MB in size). During my time I touched many of them, from distributing compilation across multiple machines to tricks in linking. Ninja primarily targets only one piece–the *front* of a build. This is the wait between starting the build and the time the first compile starts to run. To understand why that is important it is necessary to understand how we thought about performance in Chrome itself.

A Small History of Chrome

Discussion of all of Chrome's goals is out of scope here, but one of the defining goals of the project was speed. Performance is a broad goal that spans all of computer science, and Chrome uses nearly every trick available, from caching to parallelization to just-in-time compilation. Then there was startup speed–how long it the program took to show up on the screen after clicking the icon–which seems a bit frivolous in comparison.

Why care about startup speed? For a browser, a quick startup conveys a feeling of lightness, that doing something on the web is as trival an action as opening a text file. Further, the impact of latency on your happiness and on losing your train of thought is well-studied in human-computer interaction. Latency is especially a focus of web companies like Google or Amazon, who are in a good position to measure and

experiment on the effect of latency–and who have done experiments that show that delays of even milliseconds have measurable effects on how frequently people use the site or make purchases. It's a small frustration that adds up subconsciously.

Chrome's approach to starting quickly was a clever trick by one of the first engineers on Chrome. As soon as they got their skeleton application to the point where it showed a window on the screen, they created a benchmark measuring that speed along with a continuous build that tracked it. Then, in Brett Wilson's words, "a very simple rule: this test can never get any slower."[1] As code was added to Chrome, maintenance of this benchmark demanded extra engineering effort[2]–in some cases work was delayed until it was truly needed, or data used during startup was precomputed–but the primary "trick" to performance, and the one that made the greatest impression on me, was simply to *do less work*.

I joined the Chrome team without any intention of working on build tools. My background and platform of choice was Linux, and I wanted to be the Linux guy. To limit scope the project was initially Windows-only; I took it as my role to help finish the Windows implementation so that I could then make it run on Linux.

When starting work on other platforms, the first hurdle was sorting out the build system. By that point Chrome was already large (complete, in fact–Chrome for Windows was released in 2008 before any ports had started), so efforts to switch even the Visual Studio-based Windows build to a different build system wholesale were conflicting with ongoing development. It felt like replacing the foundation of a building while it was in use.

Members of the Chrome team came up with an incremental solution called *GYP*[3] which could be used to generate, one subcomponent at a time, the Visual Studio build files already used by Chrome in addition to the build files that would be used on other platforms.

The input to GYP is simple: the desired name of the output accompanied by plain text lists of source files, the occasional custom rule like "process each IDL file to generate an additional source file", and some conditional behaviors (e.g., only use certain files on certain platforms). GYP then takes this high-level description and generates platform-native build files.[4]

On the Mac "native build files" meant Xcode project files. On Linux, however, there was no obvious single choice. The initial attempt used the Scons build system, but I was dismayed to discover that a GYP-generated Scons build could take 30 seconds to start while Scons computed which files had changed. I figured that Chrome was roughly the size of the Linux kernel so the approach taken there ought to work. I rolled up my sleeves and wrote the code to make GYP generate plain Makefiles using tricks from the kernel's Makefiles.

Thus I unintentionally began my descent into build system madness. There are many factors that make building software take time, from slow linkers to poor parallelization, and I dug into all of them. The Makefile approach was initially quite fast but as we ported more of Chrome to Linux, increasing the number of files used in the build, it grew slower.[5]

As I worked on the port I found one part of the build process especially frustrating: I would make a change to a single file, run `make`, realize I'd left out a semicolon, run `make` again, and each time the wait would be long enough that I would forget what I was working on. I thought back to how hard we

fought against latency for end users. “How can this be taking so long,” I’d wonder, “there can’t be that much work to do.” As an experiment I started Ninja, to see how simple I could make it.

The Design of Ninja

At a high level any build system performs three main tasks. It will (1) load and analyze build goals, (2) figure out which steps need to run in order to achieve those goals, and (3) execute those steps.

To make startup in step (1) fast, Ninja needed to do a minimal amount of work while loading the build files. Build systems are typically used by humans, which means they provide a convenient, high-level syntax for expressing build goals. It also means that when it comes time to actually build the project the build system must process the instructions further: for example, at some point Visual Studio must concretely decide based on the build configuration where the output files must go, or which files must be compiled with a C++ or C compiler.

Because of this, GYP’s work in generating Visual Studio files was effectively limited to translating lists of source files into the Visual Studio syntax and leaving Visual Studio to do the bulk of the work. With Ninja I saw the opportunity to do as much work as possible in GYP. In a sense, when GYP generates Ninja build files, it does all of the above computation once. GYP then saves a snapshot of that intermediate state into a format that Ninja can quickly load for each subsequent build.

Ninja’s build file language is therefore simple to the point of being inconvenient for humans to write. There are no conditionals or rules based on file extensions. Instead, the format is just a list of which exact paths produce which exact outputs. These files can be loaded quickly, requiring almost no interpretation.

This minimalist design counterintuitively leads to greater flexibility. Because Ninja lacks higher-level knowledge of common build concepts like an output directory or current configuration, Ninja is simple to plug into larger systems (e.g., CMake, as we later found) that have different opinions about how builds should be organized. For example, Ninja is agnostic as to whether build outputs (e.g., object files) are placed alongside the source files (considered poor hygiene by some) or in a separate build output directory (considered hard to understand by others). Long after releasing Ninja I finally thought of the right metaphor: whereas other build systems are compilers, Ninja is an assembler.

What Ninja Does

If Ninja pushes most of the work to the build file generator, what is there left to do? The above ideology is nice in principle but real world needs are always more complicated. Ninja grew (and lost) features over the course of its development. At every point, the important question was always “can we do less?” Here is a brief overview of how it works.

A human needs to debug the files when the build rules are wrong, so `.ninja` build files are plain text, similar to Makefiles, and they support a few abstractions to make them more readable.

The first abstraction is the “rule”, which represents a single tool’s command-line invocation. A rule is then shared between different build steps. Here is an example of the Ninja syntax for declaring a rule named “compile” that runs the gcc compiler along with two `build` statements that make use of it for specific files.

```
rule compile
  command = gcc -Wall -c $in -o $out
build out/foo.o: compile src/foo.c
build out/bar.o: compile src/bar.c
```

The second abstraction is the variable. In the example above, these are the dollar-sign-prefixed identifiers (`$in` and `$out`). Variables can represent both the inputs and outputs of a command and can be used to make short names for long strings. Here is an extended compile definition that makes use of a variable for compiler flags:

```
cflags = -Wall
rule compile
  command = gcc $cflags -c $in -o $out
```

Variable values used in a rule can be shadowed in the scope of a single `build` block by indenting their new definition. Continuing the above example, the value of `cflags` can be adjusted for a single file:

```
build out/file_with_extra_flags.o: compile src/baz.c
  cflags = -Wall -Wextra
```

Rules behave almost like functions and variables behave like arguments. These two simple features are dangerously close to a programming language–the opposite of the “do no work” goal. But they have the important benefit of reducing repeated strings which is not only useful for humans but also for computers, reducing the quantity of text to be parsed.

The build file, once parsed, describes a graph of dependencies: the final output binary depends on linking a number of objects, each of which is the result of compiling sources. Specifically it is a bipartite graph, where “nodes” (input files) point to “edges” (build commands) which point to nodes (output files)[6]. The build process then traverses this graph.

Given a target output to build, Ninja first walks up the graph to identify the state of each edge’s input files: that is, whether or not the input files exist and what their modification times are. Ninja then computes a *plan*. The plan is the set of edges that need to be executed in order to bring the final target up to date, according to the modification times of the intermediate files. Finally, the plan is executed, walking down the graph and checking off edges as they are executed and successfully completed.

Once these pieces were in place I could establish a baseline benchmark for Chrome: the time to run Ninja again after successfully completing a build. That is the time to load the build files, examine the built state, and determine there was no work to do. The time it took for this benchmark to run was just under a second. This was my new startup benchmark metric. However, as Chrome grew, Ninja had to keep getting faster to keep that metric from regressing.

Optimizing Ninja

The initial implementation of Ninja was careful to arrange the data structures in order to allow a fast build, but wasn't particularly clever in terms of optimizations. Once the program worked, I reasoned, a profiler could reveal which pieces mattered.[7]

Over the years, profiling pointed at different pieces of the program. Sometimes the worst offender was a single hot function that could be micro-optimized. At other times, it suggested something more broad like being careful not to allocate or copy memory except when necessary. There were also cases where a better representation or data structure had the most impact. What follows is a walk through the Ninja implementation and some of the more interesting stories about its performance.

Parsing

Initially Ninja used a hand-written lexer and a recursive descent parser. The syntax was simple enough, I thought. It turns out that for a large enough project like Chrome[8], simply parsing the build files (named with the extension `.ninja`) can take a surprising amount of time.

The original function to analyze a single character soon appeared in profiles:

```
static bool IsIdentifierCharacter(char c) {
  return
    ('a' <= c && c <= 'z') ||
    ('A' <= c && c <= 'Z') ||
    // and so on...
}
```

A simple fix–at the time saving 200 ms–was to replace the function with a 256-entry lookup table that could be indexed by the input character. Such a thing is trivial to generate using Python code like:

```
cs = set()
for c in string.ascii_letters + string.digits + r'+,-./\_$':
    cs.add(ord(c))
for i in range(256):
    print '%d,' % (i in cs),
```

This trick kept Ninja fast for quite a while. Eventually we moved to something more principled: `re2c`, the lexer generator used by PHP. It can generate more complex lookup tables and trees of unintelligible code. For example:

```
if (yych <= 'b') {
    if (yych == '`') goto yy24;
    if (yych <= 'a') goto yy21;
    // and so on...
```

It remains an open question as to whether treating the input as text in the first place is a good idea. Perhaps we will eventually require Ninja's input to be generated in some machine-friendly format that would let us avoid parsing for the most part.

Canonicalization

Ninja avoids using strings to identify paths. Instead, Ninja maps each path it encounters to a unique `Node` object and the `Node` object is used in the rest of the code. Reusing this object ensures that a given path is only ever checked on disk once, and the result of that check (i.e., the modification time) can be reused in other code.

The pointer to the `Node` object serves as a unique identity for that path. To test whether two `Node`s refer to the same path it is sufficient to compare pointers rather than perform a more costly string comparison. For example, as Ninja walks up the graph of inputs to a build, it keeps a stack of dependendent `Node`s to check for dependency loops: if A depends on B depends on C depends on A, the build can't proceed. This stack, representing files, can be implemented as a simple array of pointers, and pointer equality can be used to check for duplicates.

To always use the same `Node` for a single file, Ninja must reliably map every possible name for a file into the same Node object. This requires a *canonicalization* pass on all paths mentioned in input files, which transforms a path like `foo/../bar.h` into just `bar.h`. Initially Ninja simply required all paths to be provided in canonical form but that ends up not working for a few reasons. One is that user-specified paths (e.g., the command-line `ninja ./bar.h`) are reasonably expected to work correctly. Another is that variables may combine to make non-canonical paths. Finally, the dependency information emitted by gcc may be non-canonical.

Thus most of what Ninja ends up doing is path processing, so canonicalizing paths is another hot point in profiles. The original implementation was written for clarity, not performance, so standard optimization techniques–like removing a double loop or avoiding memory allocation–helped considerably.

The Build Log

Often micro-optimizations like the above are less impactful than structural optimizations where you change the algorithm or approach. This was the case with Ninja's *build log*.

One part of the Linux kernel build system tracks the commands used to generate outputs. Consider a motivating example: you compile an input `foo.c` into an output `foo.o`, and then change the build file such that it should be rebuilt with different compilation flags. For the build system to know that it needs to rebuild the output, it must either note that `foo.o` depends on the build files themselves (which, depending on the organization of the project, might mean that a change to the build files would cause the entire project to rebuild), or record the commands used to generate each output and compare them for each build.

The kernel (and consequently the Chrome Makefiles and Ninja) takes the latter approach. While building, Ninja writes out a build log that records the full commands used to generate each output.[9] Then for each subsequent build, Ninja loads the previous build log and compares the new build's commands to the build log's commands to detect changes. This, like loading build files or path canonicalization, was another hot point in profiles.

After making a few smaller optimizations Nico Weber, a prolific contributor to Ninja, implemented a new format for the build log. Rather than recording commands, which are frequently very long and take a lot of time to parse, Ninja instead records a hash of the command. In subsequent builds, Ninja compares

the hash of the command that is about to be run to the logged hash. If the two hashes differ, the output is out of date. This approach was very successful. Using hashes reduced the size of the build log dramatically–from 200 MB to less than 2 MB on Mac OS X–and made it over 20 times faster to load.

Dependency Files

There is an additional store of metadata that must be recorded and used across builds. To correctly build C/C++ code a build system must accomodate dependencies between header files. Suppose `foo.c` contains the line `#include "bar.h"` and `bar.h` itself includes the line `#include "baz.h"`. All three of those files (`foo.c`, `bar.h`, `baz.h`) then affect the result of compilation. For example, changes to `baz.h` should still trigger a rebuild of `foo.o`.

Some build systems use a “header scanner” to extract these dependencies at build time, but this approach can be slow and is difficult to make exactly correct in the presence of `#ifdef` directives. Another alternative is to require the build files to correctly report all dependencies, including headers, but this is cumbersome for developers: every time you add or remove an `#include` statement you need to modify or regenerate the build.

A better approach relies on the fact that at compile time gcc (and Microsoft’s Visual Studio) can output which headers were used to build the output. This information, much like the command used to generate an output, can be recorded and reloaded by the build system so that the dependencies can be tracked exactly. For a first-time build, before there is any output, all files will be compiled so no header dependency is necessary. After the first compilation, modifications to any files used by an output (including modifications that add or remove additional dependencies) will cause a rebuild, keeping the dependency information up-to-date.

When compiling, gcc writes header dependencies in the format of a Makefile. Ninja then includes a parser for the (simplified subset) Makefile syntax and loads all of this dependency information at the next build. Loading this data is a major bottleneck. On a recent Chrome build, the dependency information produced by gcc sums to 90 MB of Makefiles, all of which reference paths which must be canonicalized before use.

Much like with the other parsing work, using `re2c` and avoiding copies where possible helped with performance. However, much like how work was shifted to GYP, this parsing work can be pushed to a time other than the critical path of startup. Our most recent work on Ninja (at the time of this writing, the feature is complete but not yet released) has been to make this processing happen eagerly during the build.

Once Ninja has started executing build commands, all of the performance-critical work has been completed and Ninja is mostly idle as it waits for the commands it executes to complete. In this new approach for header dependencies, Ninja uses this time to process the Makefiles emitted by gcc as they are written, canonicalizing paths and processing the dependencies into a quickly deserializable binary format. On the next build Ninja only needs to load this file. The impact is dramatic, particularly on Windows. (This is discussed further later in this chapter.)

The “dependency log” needs to store thousands of paths and dependencies between those paths. Loading this log and adding to it needs to be fast. Appending to this log should be safe, even in the event of an interruption such as a cancelled build.

After considering many database-like approaches I finally came up with a trivial implementation: the file is a sequence of records and each record is either a path or a list of dependencies. Each path written to the file is assigned a sequential integer identifier. Dependencies are then lists of integers. To add dependencies to the file, Ninja first writes new records for each path that doesn't yet have an identifier and then writes the dependency record using those identifiers. When loading the file on a subsequent run Ninja can then use a simple array to map identifiers to their Node pointers.

Executing a Build

Performance-wise the process of executing the commands judged necessary according to the dependencies discussed above is relatively uninteresting because the bulk of the work that needs to be done is performed in those commands (i.e., in the compilers, linkers, etc.), not in Ninja itself[10].

Ninja runs build commands in parallel by default, based on the number of available CPUs on the system. Since commands running simultaneously could have their outputs interleave, Ninja buffers all output from a command until that command completes before printing its output. The resulting output appears as if the commands were run serially.[11]

This control over command output allows Ninja to carefully control its total output. During the build Ninja displays a single line of status while running; if the build completes successfully the total printed output of Ninja is a single line.[12] This doesn't make Ninja run any quicker but it makes Ninja *feel* fast, which is almost as important to the original goal as real speed is.

Supporting Windows

I wrote Ninja for Linux. Nico (mentioned previously) did the work to make it function on Mac OS X. As Ninja became more widely used people started asking about Windows support.

At a superficial level supporting Windows wasn't too hard. There were some straightforward changes like making the path separator a backslash or changing the Ninja syntax to allow colons in a path (like `c:\foo.txt`). Once those changes were in place the larger problems surfaced. Ninja was designed with behavioral assumptions from Linux; Windows is different in small but important ways.

For example, Windows has a relatively low limit on the length of a command, an issue that comes up when constructing the command passed to a final link step that may mention most files in the project. The Windows solution for this is "response" files, and only Ninja (not the generator program in front of Ninja) is equipped to manage these.

A more important performance problem is that file operations on Windows are slow and Ninja works with a lot of files. Visual Studio's compiler emits header dependencies by simply printing them while compiling, so Ninja on Windows currently includes a tool that wraps the compiler to make it produce the gcc-style Makefile dependency list required by Ninja. This large number of files, already a bottleneck on Linux, is much worse on Windows where opening a file is much more costly. The aforementioned new approach to parsing dependencies at build time fits perfectly on Windows, allowing us to drop the intermediate tool entirely: Ninja is already buffering the command's output, so it can parse the dependencies directly from that buffer, sidestepping the intermediate on-disk Makefile used with gcc.

Getting the modification time of a file– `GetFileAttributesEx()` on Windows[13] and `stat()` on non-Windows platforms–seems to be about 100 times slower on Windows than it is on Linux.[14]