12-1-2017

Amake: Cached Builds of Top-Level Targets

Jim Buffenbarger
Boise State University

Amake: Cached Builds of Top-Level Targets

Jim Buffenbarger

Boise State University
Boise, Idaho, USA

Abstract

This paper describes a software-build tool named Amake, an extension of GNU Make. Its additional features solve important problems that have, until now, only been addressed by "high-end" build tools (e.g., ClearCase and Vesta).

With a typical build tool, if a top-level target must be updated, intermediate targets must be built from sources, and then combined to build the top-level target. The enhancements described here allow a top-level target to be fetched from a shared cache, without building, or even fetching its intermediate-target dependencies. Thus, a developer's workspace may need only contain sources and top-level targets. This reduces build time, reduces network traffic, and saves disk space.

Keywords: Software configuration management, Software build system, Software system modeling
2010 MSC: 68N01

1. Introduction

Software configuration management (SCM) is an important field of software engineering. It has two main aspects: build control and version control [1]. A few SCM tools address both aspects (e.g., ClearCase [2] and Vesta [3]), but most address only one or the other (e.g., Make [4] and Subversion [5]). Comprehensive tools have advantages and disadvantages over more focused tools, but their disadvantages prevent widespread adoption.

This paper describes enhancements to the world's most popular build tool, GNU Make [6], which provide many of the advantages of feature-rich comprehensive tools. Briefly, these features include improved correctness and reliability:

- automatic language-independent deduction of file and non-file dependencies (stored in plain-text files that can also be useful to a build engineer)
- comparison of file checksums, rather than timestamps

and improved time and space performance:

- distributed and heterogeneous caching of targets
- top-level builds, which only retrieve wanted targets from the cache

These features are all easily enabled or disabled by defining Makefile variables, either within a Makefile or on the command line. In particular, a developer can control target caching on a build-by-build basis. Furthermore, inadvertently cached targets, or those no longer wanted, can be easily removed from the cache, individually or in bulk.

Some of these features are valuable to any software-development environment, even a single developer using a single workstation, but the cache-oriented features only make sense for a larger-scale environment. Amake's features are intended to support a heterogeneous environment, with multiple developers using their own workstations, connected to a local-area network, with shared resources. This profile fits the laser-printer firmware-development facility described in Section 6.

Created in 1976, Make may seem about as exciting as a fossil, but "there is probably no large software system in the world today that has not been processed by a version or offspring of Make" [7]. Indeed, in 2003, its creator, Stuart Feldman, won the ACM *Software System Award*, because it is such an important tool. Over the years, many variations of Make have been developed (e.g., NMake [8]);

A *build tool* (e.g., Make) processes a *system model* (e.g., a Makefile) to keep a software system's artifacts consistent and updated. The model expresses build steps (e.g., compiler invocations) and dependencies between artifacts. In a Make-like build system, some sort of Makefile specifies *top-level targets*, their *intermediate dependencies*, and (transitively) their *source dependencies*. Intermediate dependencies are targets, too. Source dependencies are not targets, since they are created by people.

A top-level target differs from an intermediate target by being what a developer actually wants, not just a means to an end (e.g., the executable file that is to be distributed to customers). It's what Vesta calls a "shipped derived object." Of course, the set of top-level targets can vary over time, even minute by minute. For example, most build tools let a developer specify *any* target on the command line, thereby making it a top-level target for that build. Therefore, any special treatment of top-level targets must persist only until the next build.

We call a top-level target a *top target*, for short. An intermediate target is also called a *non-source dependency*.

The overall goal of a build tool and a system model is to update a set of top targets from source and intermediate dependencies, while conserving computing resources. Typically, to build a top target, intermediate dependencies must be built first. However, if targets are cached, and an acceptable target has been built anytime in the past, by anyone, only the top target needs to be fetched from the cache. We call this shortcut a *top build*. A build tool that supports top builds requires several kinds of cache interaction: A *cache-get attempt* tries to fetch an acceptable target from the cache. A *cache-top attempt* tries to fetch an acceptable top target from the cache. A *cache-put attempt* tries to store a target in the cache, which always succeeds. We call a successful attempt, of any kind, an *operation*.

Section 2 presents a small example motivating the value of top builds. Section 3 describes other Amake features, which enable top builds. Section 4 explains how top builds work. Section 5 elaborates on the disadvantages of high-end tools, hinted at above. Section 6 highlights our experiences with top builds in two real-world build systems: the Bash command-language interpreter and the Linux operating-system kernel. Finally, Section 7 concludes the paper.

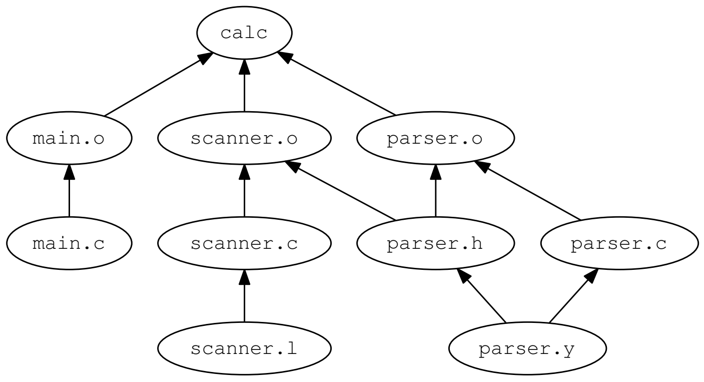

Figure 1: `calc` System Model

2. Motivating Example

Many software-development environments can benefit from top builds. These include any system that compiles source files into files containing some sort of object or intermediate code, which are then linked into some sort of executable file (e.g., a Unix/C system). These do not include some simple interpreted systems (e.g., Java bytecode running on its virtual machine). For demonstration purposes, we choose a system with two levels of intermediate targets: a small arithmetic-expression calculator from The Dragon Book [9], named `calc`. It is a C program, built with help from Flex [10] and Bison[11]. Figure 1 shows its system model, omitting system header files and libraries. Notice that `parser.y` is an immediate dependency of two intermediate targets, which we call *siblings*.

Figure 2 shows a simple GNU Makefile for this system.

Now, imagine that two `calc` developers each begin a new development task, in their own new workspace, starting from the same baseline. For performance,

```
objs:=main.o
objs+=scanner.o parser.o

%.c: %.l
        flex -o $@ $<
%.c: %.y
        bison -o $@ $<
%.o: %.c
        gcc -c $<

calc: $(objs)
        gcc -o $@ $(objs)

scanner.o: parser.c
```

Figure 2: `calc` Makefile

a developer's workspaces are probably stored on their own computer.

With plain Make, the first build in such a workspace, always executes the following set of commands (the order may vary):

```
gcc -c main.c
flex -o scanner.c scanner.l
bison -o parser.c parser.y
gcc -c scanner.c
gcc -c parser.c
gcc -o calc main.o ···
```

The problem is that the first build in another such workspace executes the same commands to produce identical targets. Overall, half the work can be avoided by sharing targets.

With Amake's cache enabled, by adding a few variable definitions to the Makefile, the first developer's build produces:

```
gcc -c main.c
Cache put: main.o
flex -o scanner.c scanner.l
Cache put: scanner.c
bison -o parser.c parser.y
Cache put: parser.c
Cache put: parser.h
Cache put: parser.output
gcc -c scanner.c
Cache put: scanner.o
gcc -c parser.c
Cache put: parser.o
gcc -o calc main.o ···
Cache put: calc
```

and the second developer's entire build produces only:

```
Cache get: main.o
Cache get: scanner.c
Cache get: parser.c
Cache get: parser.h
Cache get: parser.output
Cache get: scanner.o
Cache get: parser.o
Cache get: calc
```

The second developer doesn't execute Flex, Bison, or GCC. Of course, in a real development environment, the first developer's build might also come from cache.

With Amake top builds also enabled, by adding another variable definition to the Makefile, the second developer's build would instead produce only:

```
Cache top: calc
```

The only file created in the workspace is the `calc` executable, the top target.

Alternatively, if the second developer asked Amake to only build what would normally be considered an intermediate target, for example by typing `parser.o` on the command line, a top build would produce only:

```
Cache top: parser.o
```

The dependencies of `parser.o` are not fetched from the cache, because, for this build, `parser.o` is a top target.

3. Amake Features and Implementation

Amake has been evolving for several years, and is described elsewhere [12] [13]. In this section, Amake's features that support top builds are reviewed (e.g., caching). Top builds are the subject of the next section.

Almost all Make variants, including GNU Make, understand only one kind of dependency: one file being newer than another file, according to operating-system timestamps. Furthermore, such dependencies must be specified explicitly in a Makefile. They are often incomplete and/or inaccurate. Consequently, many build systems employ ad-hoc Makefile-generation methods (e.g., scripts, GNU Automake [14], Linux Kbuild [15], or GCC's `-MMD` option [16]) to increase accuracy and reduce the amount of manual labor.

Amake enhances GNU Make with automatic dependency processing. Amake's dependencies are not just files. Its automatic dependency processing detects, records and monitors several kinds of dependency: files accessed, shell-commands executed, programs executed, shared-libraries opened, and environment-variable-values referenced. Furthermore, Amake does not rely on operating-system timestamps: it computes, records, and compares checksums of file content.

The reasoning behind switching from timestamps to checksums deserves a more detailed explanation. Traditional Make implementations compare operating-system timestamps with a *less-than* operation, with a resolution of between one second and one nanosecond, depending on what the operating system and/or filesystem supports, and whether the entire timestamp is compared [6]. For this to work at all, workstations in a distributed development environment must have synchronized clocks. This can be done with common tools like the Network Time Protocol [17], but only to millisecond granularity. However, "undoing" changes to a file (e.g., with an editor or Subversion's `revert` command) must still be done carefully. Timezones and daylight-savings time, should be considered, too. Regarding timestamp comparison, the GNU Make manual warns: "Normally this is not a problem, but in some extreme cases you may need to use tricks like `sleep 1` to work around timestamp truncation bugs." Further: "Commands like `cp -p` and `touch -r` typically do not copy file timestamps to their full resolutions." In contrast, checksum comparison is done with an *equals* operation. This has the added benefit of allowing the metadata to be used as a key for accessing a table or database, as in Section 3.2.1, below. If timestamps were maintained perfectly, one might consider comparing them with an *equals* operation, as well. Note that in both cases we are comparing, with *equals*, metadata of two versions of a single file dependency, rather than comparing, with *less-than*, metadata of a target and one of its file dependencies. A timestamp-equality scheme would still fail to recognize that a file dependency was changed and then changed back to the way it was. For example, a developer might edit/save source file A, realize the change actually belongs in source file B, undo the change to A by moving it to B, and rebuild. Dependencies of A need not be rebuilt, even though a timestamp-equality scheme would do so.

Checksums are never computed "across the network," only on a developer's workspace files. Nevertheless, computing them may seem to incur serious performance overhead, compared to simply using operating-system timestamps. To address this, we developed a "checksum-cache daemon," named `ccd`, which can be used by Amake, by defining a Makefile variable. A `ccd` instance is a local-host, single-process, single-threaded, TCP-based daemon. It caches computed checksums, and recomputes them when alerted by the Linux Inotify library [18]. Timing analysis of thousands of consecutive, up-to-date, top builds of the Bash distribution, described in Section 6, indicate a speedup of about 80% using a `ccd`. We also investigated using the checksums automatically computed by the (now default on Fedora) BTRFS filesystem [19], but they are block, rather than file, oriented.

Amake stores a target's computed dependency information in several files that each have the target's name as their base name. Filename extensions distinguish the `.cmd`, `.dep`, `.sib`, and `.top` files. A Makefile is expected to explicitly include `.dep` files:

```
sinclude *.dep
```

The others are processed implicitly.

3.1. Dependencies

Particular kinds of dependency processing can be enabled or disabled by appropriate Makefile-variable definitions.

3.1.1. Files Accessed

A target file is typically built from other files. For example, an object file might be built from a C source file and all the C header files it includes. If any of the source or header files change, the object file must be updated.

Traditional variants of Make, like GNU Make, require this kind of dependency to be detected and recorded manually. Some tools can help automate this process, but only for particular kinds of target. Traditional monitoring of this kind of dependency consists of comparing time-of-last-modification attributes of the files involved. Time-based monitoring is susceptible to problems caused by computers disagreeing about the current time and version-control operations that change a file's time attributes.

Amake uses a shared library and the POSIX [20] `LD_PRELOAD` facility [21] to "wrap" the Standard C Library. For example, when a program (e.g., a compiler) calls a library function to open a file, the call is intercepted by the wrapper library, which records the name of the opened file, and then forwards the call to the real function. The names of files referenced while building a target are stored in the target's `.dep` file.

A target's `.dep` file also contains the checksum of each referenced file, computed at the time the target was built. Monitoring consists of comparing checksums, rather than time attributes.

3.1.2. Shell-Commands Executed

A target file is typically built by executing a sequence of shell commands. For example, an object file might be built by executing a command composed of a compiler's name and its command-line options. If the command(s) to execute change, the target must be updated.

Many Make-based build systems ignore this kind of dependency. Others try to make every target depend on the entire Makefile. Neither method is satisfactory.

Amake records the sequence of commands to execute, to build a target, in the target's `.cmd` file. Monitoring consists of textually comparing command sequences. For performance, however, sequence checksums are compared.

3.1.3. Programs Executed

Even if the sequence of commands used to build a target has not changed, if a program executed by those commands changes, the target must be updated. Many such programs (e.g., a compiler) rarely change, but homegrown tools (e.g., scripts) can change frequently. Sometimes, the program does not change, but a different program with the same name is executed (e.g., because the value of the `PATH` environment variable is different).

Many Make-based build systems also ignore this kind of dependency. Others might try to record it manually, but only for homegrown tools. Again, neither method is satisfactory. Debugging problems caused by missing such a dependency can be very difficult.

Amake uses its wrapper library to detect and record when one program calls a Standard C Library function to execute another program (i.e., one of the `exec()` family). The names of programs executed while building a target are also stored in the target's `.dep` file. It also contains the checksum of each executed program's file, computed at the time the target was built. Again, analysis consists of comparing checksums, rather than time attributes.

3.1.4. Shared-Libraries Opened

Even if the commands and programs used to build a target have not changed, if a shared library containing code executed by one of those programs changes, the target must be updated. This is rare, except for complex homegrown tools. This kind of dependency is typically ignored.

Amake also uses its wrapper library to detect and record shared libraries linked to an executed program. Such dependencies and checksums are also stored in a target's `.dep` file.

3.1.5. Environment-Variable-Values Referenced

A set of commands, or even a program, that builds a target may change its behavior according to the values of environment variables. For example, GCC queries the value of many environment variables. If the value of a critical variable changes, the target must be updated. This problem is nefarious: environment-variable values can be inherited from a user's login session. Yet, this kind of dependency is typically ignored.

An Amake Makefile can specify the names of critical environment variables. The values of these variables are stored in a target's `.cmd` file, as variable assignments. The assignments occur after *all* environment variables have been unset (by the `-i` command-line option to `/bin/env`). Likewise, subprocesses that built targets have "sanitized" environments.

3.2. Architecture

Figure 3 illustrates the architecture of an executing Amake system. A roundtangle is a process or library; bold denotes multiplicity. Solid arcs denote execution and/or data transfer. Dotted arcs denote pipe or socket communication.

The basic idea is that the lower-left part of the figure performs dependency detection, while the right half of the figure performs caching. For caching, an Amake process starts and communicates with a per-build target-cache daemon (`tcd`), which communicates with the single and shared target-cache-index daemon (`tid`). A `tcd` executes with the developer's permissions, and accesses the

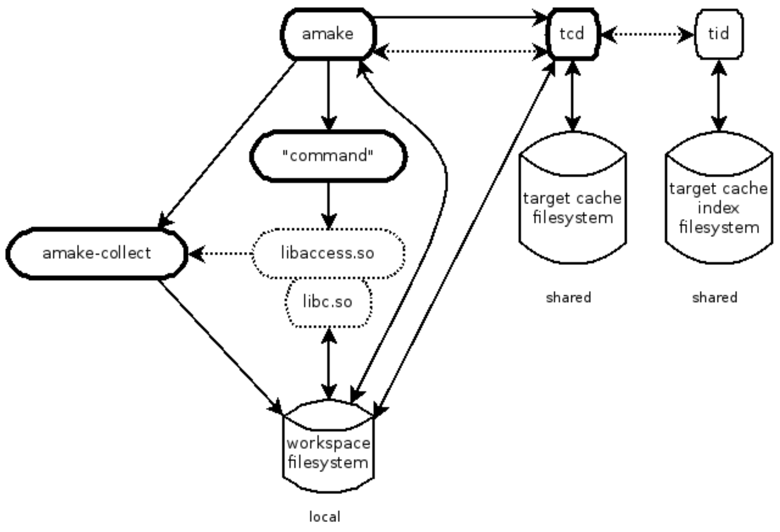

Figure 3: Amake Architecture

developer's local workspace and the shared cache filesystem. The one-and-only `tid` executes with elevated database-access permissions, and performs index lookups, helping a `tcd` find suitable targets in the cache filesystem.

A `tcd` services three kinds of request:

- *cache get*: Inputs include the current working directory, target name, architecture identifier (e.g., `Linux_x86_64_4.7.7-200.fc24.x86_64`), and command-line checksum. If successful, a suitable target is stored in the developer's workspace.
- *cache top*: This is similar to a *cache get*.
- *cache put*: Inputs include the current working directory, target name, architecture identifier, target checksum, and command-line checksum. The target is stored in the cache.

The `tid` services two kinds of request:

- *get*: Inputs include the target name, architecture identifier, target checksum, and command-line checksum. Output is a set of cache-filesystem paths.

- *put*: Inputs include the target name, architecture identifier, target checksum, command-line checksum, and cache-filesystem path. There is no output.

A `tid` request does not need the current working directory, since the `tid` does not access developer workspaces. Indeed, this is one reason why the `tid` is separate from the `tcd` instances. Other reasons are: mutual-exclusion requirements, permissions requirements, and improved modularity.

Figure 3 is further described in the following subsections.

3.2.1. Target-Cache-Index Daemon (TID)

The target-cache-index daemon is assumed to be perpetually executing. This is a traditional multi-threaded daemon process, which listens to a well-known TCP/IP port and services requests from other processes. It can be the special-purpose light-weight `tid` program, based on GNU DBM [22] and distributed with Amake. Or, it can be a general-purpose SQL server (e.g., MySQL's `sqld` [23]). For discussion, we'll assume the `tid` is used. In any case, the `tid`, and thus its database, is shared by all developers within the environment. It provides metadata for the actual target cache.

3.2.2. Amake Process

In Figure 3, a developer has started a build, by executing Amake, from a directory within a workspace. As before, the workspace is expected to be local to the developer's computer.

An Amake process accesses a developer's workspace, for many kinds of files:

- One or more Makefiles, of course, are stored in the workspace.
- Source files used to build a target are stored in the workspace. They, as well as the Makefiles, are likely put there by a version-control system (e.g., Subversion).
- Amake records the commands that build a target, along with relevant environment-variable information, in a `.cmd` file, which is stored in the

workspace.

- Target files (e.g., object files, libraries, and executables), both top and intermediate, are stored in the workspace. They may have been actually built in the workspace, or they may have come from the target cache.

- Some homegrown tools (e.g., scripts) may reside in the workspace, much like source files. Other homegrown and third-party tools (e.g., compiled ones) comprise source files and target files, which may reside in the workspace. Indeed, entire compiler toolchains may fall into this latter category.

3.2.3. Target-Cache Daemon (TCD)

When Amake decides that a top or intermediate target either does not exist in the workspace, or that it exists and must be updated, a target-cache daemon is contacted. That `tcd` begins servicing a cache-get or cache-top attempt, by making a similar request to the `tid`.

The `tid` consults its database and replies with a possibly empty set of regular Unix paths, to cache entries containing candidate targets. The `tcd` then iterates over those paths, looking for the first cache entry with dependencies that match, according to checksums, those in the developer's workspace. Notice that checksums are never *computed* on files over the network, only on files in the developer's local workspace.

If a matching cache entry is found, the `tcd` copies it to the workspace. Siblings and metadata (e.g., the target's `.dep` file) are also copied. If the cache and workspace happen to be in the same filesystem, hard links are created, rather than copies.

The `tcd` is a non-traditional daemon program:

- It is a short-lived daemon. Amake starts one, as necessary, for a particular build. Thus, multiple `tcd` processes may be executing.

- It executes locally on a developer's computer.

- It listens for requests over a named socket.
- It services requests by forking child processes.
- It exits, after a period of idle time (e.g., one minute), after its children exit.

3.2.4. Target Build

When Amake decides that a target does not exist in the workspace, or that it must be updated, and the cache-get attempt fails to obtain an acceptable target, the target must be built in the workspace.

To collect automatically detected dependencies, Amake then forks and executes a program named `amake-collect`, which, during the target's build, reads this information from a pipe. The other end of the pipe is discussed below. Eventually, `amake-collect` stores this information in the developer's workspace (e.g., in the target's `.dep` file). Currently, `amake-collect` is a Perl script.

A target is updated by executing a sequence of commands. Amake, like other flavors of Make, executes each such command in a newly forked child process. Unlike other flavors of Make, Amake carefully controls the environment variables available in the child process. This helps prevent the command from behaving differently for different users, due to differences in login environments. For example, GCC is dramatically affected by the existence and values of several environment variables (e.g., `GCC_EXEC_PREFIX`).

Environment "scrubbing" is needed in two places: (1) the `.cmd` file and (2) between the `fork()` that creates the child process and the `exec()` that executes the command. This turns out to be trickier than one might imagine. A completely empty environment is insufficient. Some environment-variable values must be set or preserved. Such variables fall into three categories: (1) those needed for command execution (e.g., `PATH` and `LD_PRELOAD`), (2) those needed by recursively executed Amake processes (e.g., `MAKEFLAGS`), and (3) those set in the Makefile to control the command's behavior (e.g., GCC variables). The names of variables in this last category must be listed in a Makefile variable, so

Amake knows to preserve their values in child processes.

To accomplish automatic dependency detection, Amake sets the subshell's `LD_PRELOAD` environment variable to `libaccess.so`, thereby intercepting calls to relevant library functions in `libc.so` (e.g., `open()`). The intercepting functions write this information to the pipe being read by `amake-collect`. Interprocess semaphores provide synchronization.

After the target is built, a cache-put operation is performed.

3.3. Target Cache

When Amake determines that a target must be updated, it checks its cache for a suitable, previously built target, before otherwise rebuilding the target and putting it in the cache. The cache has two main parts: an index database containing target-dependency information and an network-mounted directory containing the targets themselves. Each entry in the former points to a subdirectory in the latter. The subdirectories are carefully structured to allow concurrent access, with host-name and process-number components. A target cache is intended to be shared across an entire development environment, supporting multiuser, multihost, and heterogeneous development.

Caching only targets is not enough. Often, a single Makefile rule builds multiple files, other than just its target. For example, in Figure 1, Bison creates a `.h` file as well a `.c` file, from a single `.y` file. Recall that such a file is called a sibling; it is stored in the cache along with its target.

Of course, GNU Make's parallel-build facility can work with the target cache.

4. Top-Build Implementation

Top builds are Amake's latest feature, but they are only possible because Amake caches targets. Strictly speaking, a top build requires caching of only top targets, but, since one of the last build's intermediate targets can be one of the next build's top targets, all targets should be cached. Automatic dependency analysis is not required for top builds, but it increases their accuracy.

At first, Amake's cache subsystem seemed sufficient for implementing top builds. However, closer consideration revealed that workspace-oriented processing is required prior to cache-oriented processing. In particular, if the workspace already contains an instance of the top target, it must be tested for acceptability, according to its source dependencies, before a cache-top attempt or processing its non-source dependencies. If it is acceptable, its dependencies can be immediately pruned. Fortunately, workspace-oriented processing and cache-oriented processing were similar enough to be unified, albeit with careful parameterization. Thus, by factoring existing code out of the cache subsystem, into a new "acceptability" module, shared between the cache subsystem and the main subsystem, very little new code needed to be developed.

The rest of the solution was to create another kind of per-target dependency file, called a `.top` file. Like a `.dep` file, a `.top` file expresses a target's dependencies, but only its source dependencies. A `.top` file contains the leaves of the transitive closure of the target's direct dependencies, computed from `.dep` files. With this careful construction of `.top` files, a cache-top operation is simply a cache-get operation with a `.top` file, instead of a `.dep` file.

If a target has only source dependencies, its `.top` and `.dep` files are identical.

A target's `.top` file is created just after the target's `.dep` file has been created. Recall that a `.dep` file is created after a target is updated, so it can contain the dependencies accessed during the update. Figure 4 summarizes the overall target-update logic.

The abovementioned acceptability module takes two inputs:

- the path to a workspace target or cached target
- the path to a `.top` or `.dep` file (respectively)

It determines acceptability of the target, by comparing the dependencies named in the `.top` or `.dep` file, with the actual files in the workspace (i.e., via checksums).

The acceptability module is called early in a target's processing, with a workspace `.top` file, to determine if the workspace target is already acceptable,

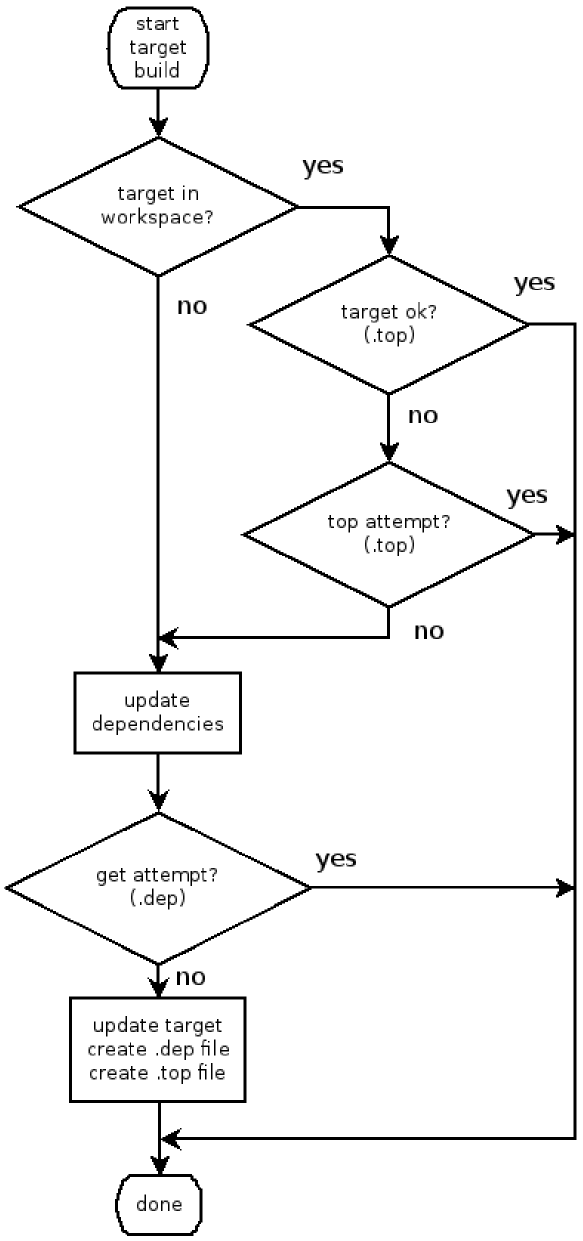

Figure 4: Target-Update Logic

```
calc: /usr/bin/flex
calc: /usr/bin/bison
calc: /usr/bin/m4
calc: /usr/bin/gcc
calc: /usr/bin/as
calc: /usr/bin/ld
⋮
calc: /usr/include/stdio.h
calc: /usr/include/stdlib.h
⋮
calc: /lib64/libc.so.6
⋮
calc: scanner.l
calc: parser.h
calc: parser.y
calc: main.c
```

Figure 5: `calc.top`

thus preventing an unnecessary cache access and further dependency processing. If it is unacceptable, a cache-top attempt is made.

The module may also be called later in a target's processing, in a cache-get operation, with a cached `.dep` file, to determine if a cached target is acceptable for the workspace.

Figure 5 shows `calc.top`, an elided `.top` file, for the `calc` target.

5. Other Tools

Make no mistake: ClearCase and Vesta are outstanding software-engineering tools. Indeed, they are more like environments or systems than mere tools. Both provide remote development, automatic dependency analysis, derived-object sharing, parallel building, and distributed building. Vesta provides what we call top builds; ClearCase does not. (In many ways, Vesta is better than ClearCase.) Once you understand their features, or have come to rely upon them, other SCM tools seem rather primitive.

Nevertheless, ClearCase and Vesta have their drawbacks. Both tools are very complex and tightly coupled to their respective version-control subsystems. While Amake is undeniably more complex than GNU Make, it is a standalone build tool, which can be paired with any version-control tool.

ClearCase and Vesta are also tightly coupled to the operating system. Originally, ClearCase required a customized kernel, but now, like Vesta, it employs a special filesystem, implemented as a loadable kernel module. In contrast, Amake simply requires the POSIX shared-library `LD_PRELOAD` facility [21], to intercept certain library calls made during a target update (e.g., `open()`).

However, if the (optional) checksum-cache daemon is deployed, to speed-up Amake's checksum comparisons, the Linux-specific Inotify API is required. Sadly, this is not a POSIX feature (e.g., it is not implemented in BSD Unix).

While Vesta is an open-source project, ClearCase is an expensive commercial product. Its build tool, ClearMake, is network intensive and rather slow, being designed when processors were slow and networks were fast. It can require a team of specially trained system administrators. To be fair, Vesta and Amake probably require an on-site specialist, too.

6. Real-World Experiences

An early version of Amake is being used to build laser-printer firmware at Hewlett-Packard's development facility in Boise, Idaho. This version uses timestamps rather than checksums, and does not have a target cache.

In addition to a thorough test suite, Amake is used to build itself. This is another way of saying Amake can build GNU Make [6], version 4.1, after small changes to three stock source files, and the addition of about twenty new source files. GNU Make's stock build system follows the typical GNU Automake style. After the typical `configure` step, and a bootstrapping build, a few Amake-enabling variable definitions are added to the stock Makefile. In addition, an Amake variable is defined to allow several configuration files to be built without command-dependency checking. That's it. Amake, with a couple of command-line arguments, then builds Amake.

Amake can also build Bash [24], version 4.3. Bash is a shell, or command-language interpreter, for an operating system. Bash's stock build system also follows the GNU Automake style. Compared to GNU Make, Bash is a more complex system to build. It has subsystems, homegrown tools, a Bison grammar, and static libraries.

After the typical `configure` step, Amake builds Bash by simply "prepending" a small set of Amake-enabling variable definitions to the distributed Makefile, as command-line options. The distributed Makefile needs no modification.

For large-scale testing, we have been using Amake to build a version of the Linux kernel (viz., `linux-2.6.27.38`). This is not the best example for top builds, because there are many top targets, but it is a realistic example. The kernel's Kbuild-based build process was modified for Amake builds, mainly because Amake performs the previously ad-hoc dependency processing automatically. In particular, Kbuild uses a phony target, named `FORCE`, to force rebuilds of many targets, rather than determining whether they are already up to date. In addition, Kbuild's scheme to compare previous and current build commands requires building many other targets, coincidentally named `.cmd` files. Kbuild must build its own tools, too.

Comparing GNU Make and Amake build times for the Linux kernel (or any other system) is tricky. There is more than one way to configure the target cache, as well as multiple use cases. Below, we consider building on a single commodity-class workstation, with the cache on its local disk. The checksum-cache daemon, described earlier, in Section 3, is not used.

First, consider the common use case of building in a new workspace, containing source code but no target files.

Starting with new workspaces, and an empty cache, GNU Make requires 21:40 (MM:SS), compared to Amake requiring 36:09 (67% slower). Of course, GNU Make uses the Kbuild machinery; Amake does not. However, Amake has to populate the cache, with many many cache-put operations.

Again starting with new workspaces, but with the previously populated cache, an Amake top build requires 02:39, compared to an Amake non-top build requiring 04:45 (79% slower). Even though there are many top targets, the top build requires 5408 fewer cache operations ($(16034-10626)/16034 = 34\%$). The GNU Make build, from the previous paragraph, is 718% slower than the Amake top build.

Now, consider another use case: building in an up-to-date workspace. This is similar to the very common use case of building in an almost up-to-date workspace. This is what a developer might do after editing a source file.

For a Linux-kernel up-to-date build, GNU Make can actually be slower than Amake, because of the ad-hoc Kbuild processing, described earlier, which Amake avoids. We see this when comparing a GNU Make build requiring 00:22 with an Amake non-top build requiring only 00:15, where GNU Make is 47% slower. However, an Amake top build requires 01:15, which is 241% slower than GNU Make.

This last anomaly is due to the atypically large number of top targets in a Linux build. They share many source dependencies, and there are many more source dependencies than intermediate dependencies. The top build checks those source dependencies repeatedly, for each top target. The non-top build avoids this duplicated work, by only checking source dependencies once, when considering intermediate targets.

7. Future Work and Conclusion

Amake is free software. It can be obtained from:

`http://posixamake.sourceforge.net`

What Amake needs most is more users. The most appropriate kind of development environment is one with many developers, a deep dependency structure, and a long from-scratch build time.

The Hewlett-Packard installation mentioned above is just such an environment. That lab has about a hundred developers, maintaining a C-based embedded system, with thousands of source files, and many layers of build steps. The firmware requires a couple of hours to build from scratch. It includes a real-time operating system, and many homegrown tools that generate C code (e.g., for fonts). Developers have their own workspaces on their own computers. Many of them are quality-assurance personnel, who never change source files; they only build baselines. This use case is a perfect candidate for top builds. As yet, however, their solution is to `tar` and check-in built workspaces, via Subversion, so they can check-out "built" systems for testing.

We wish we could report on real-world, large-scale, industrial, use of Amake's caching features. Such experiences would undoubtedly expose, and allow fixing of, latent bugs. More interestingly, though, it would allow developer workflows to be established and documented. For example, a developer would likely want to enable/disable caching and/or top builds in certain circumstances. If a developer's changes are expected to be of no interest to other developers, the subsequent build could be performed with caching disabled, by defining a variable on the Amake command line. Or, if a workspace needs intermediate targets, perhaps for specialized debugging or distribution, a top build could be disabled, in the same way. On the other hand, an integration-team build, destined for validation and labeling, would certainly be performed with caching enabled, to benefit other developers when they rebase their workspaces.

Another fact of life for a large-scale target cache, whether for Amake or other systems, is that of pollution. A variety of bugs can result in the build system successfully producing a corrupt target, which is then put into the cache, and subsequently fetched into multiple developers' workspaces. The effect is that of a virus. For example, a script might simply return the wrong exit code, after producing a truncated target, and the build system has no way of detecting the problem. Clearly, workflows for large-scale cache maintenance could be established and documented.

Google has a complex build system, not a single tool, that implements many of Amake's features [25]. Their system also has features that we may want to add to Amake. Their "functional build model" is straight from Vesta's philosophy that a build step is the evaluation of a referentially transparent function.

They use “digests” rather than checksums and `.cmd` files, but with the same effect. They use a custom filesystem, for caching source files on developer clients and targets on build servers. Curiously, they try to hide the cache, by recording target-build `stdout/stderr` output, and replaying it during a cache-get operation. We will investigate adding this feature to Amake, but it might not be such a good idea. Questions include:

- Should the output of a target’s build commands include that of a subtarget upon which it depends? Should the subtarget’s output be separate? Should it be duplicated?

- Command output could automatically be stored in a file, and treated as a target’s sibling, whose content may (or may not) be displayed during a cache-get operation. Alternately, if the output is *that* important, should a developer just be required to explicitly redirect it to a file, from within a Makefile, so it would automatically be a target’s sibling?

- Would such output be too confusing to a developer? It might contain information about the original builder’s environment (e.g., hostname, username, date/time, process identifiers, or influences from explicitly ignored file dependencies). The timing and interleaving of output lines might be confusing, especially for parallel builds. The effect of of sending a `SIGSTOP` or `SIGINT`, typically bound to `Ctrl-Z` and `Ctrl-C` (respectively), might be surprising.

- How would top builds and recursive builds look? For a top build, would the developer see *all* output, perhaps with duplication? Or, for example, would the developer just see a big `gcc -o` command, even though none of the object files would be in the workspace.

GNU Make is a mature system, with infrequent releases (until recently), but porting Amake’s changes to new versions is easy. Only nine changes to GNU Make’s distributed source code are required. The recent upgrades to version 4.0, and then 4.1, each took only thirty minutes.

References

[1] W. Tichy, Tools for software configuration management, in: Proceedings of the International Workshop on Software Version and Configuration Control, Teubner-Verlag, 1988, pp. 1–20.

[2] Atria Software, Inc., ClearCase User's Manual, `http://publib.boulder.ibm.com/ infocenter/cchelp/v7r0m1` (1996).

[3] A. Heydon, R. Levin, T. Mann, Y. Yu, The Vesta Approach to Software Configuration Management, Tech. Rep. 168, Compaq Systems Research Center, `http://www.vestasys.org` (2001).

[4] S. Feldman, Make: A program for maintaining computer programs, Software — Practice and Experience 9 (4).

[5] B. Collins-Sussman, B. Fitzpatrick, C. Pilato, Version Control with Subversion, `http://svnbook.red-bean.com` (2014).

[6] P. Smith, GNU Make, Free Software Foundation, Inc., `http://www.gnu.org/software/make` (2016).

[7] Association for Computing Machinery, 2003 Software System Award: Stuart Feldman, [Online; accessed 3-November-2014] (2014). URL `http://awards.acm.org/award_winners/feldman_1240498.cfm`

[8] G. Fowler, The fourth generation Make, in: USENIX Portland 1985 Summer Conference Proceedings, USENIX, 1985, pp. 159–174.

[9] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.

[10] V. Paxson, W. Estes, J. Millaway, The Flex Manual, `https://www.gnu.org/software/bison/manual` (2012).

[11] C. Donnelly, R. Stallman, Bison: The Yacc-compatible Parser Generator, `https://www.gnu.org/software/bison/manual` (2013).

[12] J. Buffenbarger, Amake: GNU Make with Automatic Dependency Analysis and Target Caching, in: British Computer Society Configuration Management Specialist Group Conference (BCS CMSG), 2012.

[13] J. Buffenbarger, Adding Automatic Dependency Processing to Makefile-Based Build Systems with Amake, in: International Workshop on Release Engineering (RELENG 2013), 2013.

[14] D. MacKenzie, GNU Automake, Free Software Foundation, Inc., `http://www.gnu.org/software/automake` (2014).

[15] B. Kaindl, kbuild - linux kernel build, [Online; accessed 19-October-2012] (2012).
URL `http://www.sf.net/projects/kbuild`

[16] R. Stallman, the GCC Developer Community, Using the GNU Compiler Collection, Free Software Foundation, Inc., `http://gcc.gnu.org/onlinedocs` (2010).

[17] Wikipedia, Network Time Protocol, [Online; accessed 9-March-2017] (2017).
URL `https://en.wikipedia.org/wiki/Network_Time_Protocol`

[18] Wikipedia, Inotify — Wikipedia, the free encyclopedia, [Online; accessed 19-January-2015] (2014).
URL `http://en.wikipedia.org/wiki/Inotify`

[19] Wikipedia, BTRFS — Wikipedia, the free encyclopedia, [Online; accessed 19-January-2015] (2014).
URL `http://en.wikipedia.org/wiki/Btrfs`

[20] Wikipedia, POSIX — Wikipedia, the free encyclopedia, [Online; accessed 23-February-2012] (2012).
URL `http://en.wikipedia.org/wiki/POSIX`

[21] Wikipedia, Dynamic linker — Wikipedia, the free encyclopedia, [Online; accessed 23-February-2012] (2012).
URL `http://en.wikipedia.org/wiki/Dynamic_linker`

[22] P. Nelson, J. Downs, S. Poznyakoff, GNU DBM, Free Software Foundation, Inc., `http://www.gnu.org/software/gdbm` (2011).

[23] D. Axmark, M. Widenius, MySQL Reference Manual, Oracle, Inc., `http://dev.mysql.com/doc` (2012).

[24] C. Ramey, The GNU Bash Reference Manual, Free Software Foundation, Inc., `http://www.gnu.org/software/bash` (2016).

[25] N. York, Tools for Continuous Integration at Google Scale, [Online; accessed 19-January-2015] (2011).
URL `https://www.youtube.com/watch?v=b52aXZ2yi08`