

apmake: A Reliable Parallel Build Manager

Derrick Coetzee

Anand Bhaskar

George Necula

University of California, Berkeley

{dcoetzee, bhaskar, necula}@eecs.berkeley.edu

Abstract

Build systems such as *make* support incremental and parallel building, but these features are unreliable in the presence of incomplete dependency information. We describe a system that automatically augments a build system to provide parallel and incremental building while guaranteeing the same final output as a clean, serial build. Each build task is run inside a transaction that isolates its effects from concurrently running build tasks, and the results of build tasks are cached for later reuse. By dynamically monitoring all filesystem accesses, all file-based dependencies between build tasks can be reliably identified. In experiments on three small builds on a quad-core machine, an initial build using our system took between 54% to 219% as long as a clean, serial build, while an incremental build (with no files changed) using our system took between 22% to 71% as long as a clean, serial build.

1 Motivation

Large software projects often reach thousands of files and millions of lines of source code. Build automation systems, or *build systems* for short, are responsible for automating the execution of build tools such as compilers in order to process all the source code and produce the final, executable output. The time required to execute a build is a critical factor in a number of software engineering metrics such as: developer cycle time, frequency of continuous integration testing, throughput of check-in verification systems, and time to ship a critical patch. Yet a 2003 survey showed that more than half of the 30 surveyed commercial projects had a clean, serial build time of 5-10 hours [10]. This motivates the development of builds that can run faster than a clean build.

To address this need, many existing build systems provide two features: *parallel builds*, in which multiple build tasks are executed simultaneously, and *incremental*

builds, in which results of previous builds are reused and only a subset of build tasks are run, based on what build inputs have changed. In both types of builds, the developer must explicitly specify *dependencies* for each build task, describing other build tasks which must run before it. For example, in a C project, C source files must be compiled into object files before the object files can be linked into an executable binary. If even one dependency is omitted, the soundness of both parallel and incremental builds is compromised: build tasks may be run out of order, leading to incorrect reuse of out-of-date results, build failure due to missing results, and race conditions due to concurrent access to files. Whether a failure occurs, and which failure occurs, depends on which input files have changed and the build schedule selected by the build system. As a consequence, “most organizations run their builds completely sequentially or with only a small speedup, in order to keep the process as reliable as possible” [10].

Incomplete dependencies arise naturally whenever a developer changes the code or the build scripts and introduces a new dependency, but fails to correctly update the dependency information in the build configuration. As a simple example, consider the build described by this makefile:

```
all: generated.h foo

generated.h: config
    gen config -o generated.h

foo: foo.c
    gcc foo.c -o foo
```

Here, a tool called `gen` is run to generate the header file `generated.h` from a file `config`; then the binary `foo` is compiled from the C source file `foo.c`. Now suppose the developer modified `foo.c` to include the header file `generated.h`, and also modified `config`. A clean, serial build will still produce the expected result, since

`generated.h` is listed before `foo` in the `all` target, but an incremental or parallel build may run the `gcc` action before, or simultaneously with, the `gen` action, leading to incorrect output or build failure.

Another problematic scenario is when a large build is formed by composing a number of existing builds for various components. Some part of one component’s build may depend on some output of another component’s build, but simply specifying dependencies at the level of components does not expose enough parallelism. Achieving fine-grained parallelism generally requires merging component build systems into a single unified build system, which can be a costly endeavor.

2 Goal

Our high-level goal is to develop a build framework with three properties: it provides incremental and parallel builds, it reliably produces correct results without manual maintenance, and it provides a low-cost migration path for existing projects. In particular, it should produce correct results even in the presence of incomplete or missing dependencies.

Defining this goal precisely requires us to define what we mean by “correct”. In general, the specification of a correct build depends on developer intentions and is not tractable to infer. Instead, we seek a specification that is easy for a developer to create and debug, without additional training. We specify that **a correct build should produce the same output as a clean, serial build**. Clean, serial builds are easy to implement, and because they are deterministic and repeatable, issues are easy to reproduce and fix. This implies that in order to be of benefit, our system should build more quickly than a clean, serial build.

Sections 3.1 and 3.2 will describe at a high level on how reliable incremental and parallel builds are achieved. Section 3.3 will describe implementation details and how we achieve low-cost migration.

3 Our solution

For the purposes of this work we define a build as a sequence of tasks which communicate exclusively through reads and writes to shared state. In the case of a typical *make*-based build the tasks would be processes and the state would be the filesystem; a monolithic build system that runs all build steps inside a single process, like Java Ant, might define a task as a function call and the shared state in terms of global variables. In either case the concepts are the same.

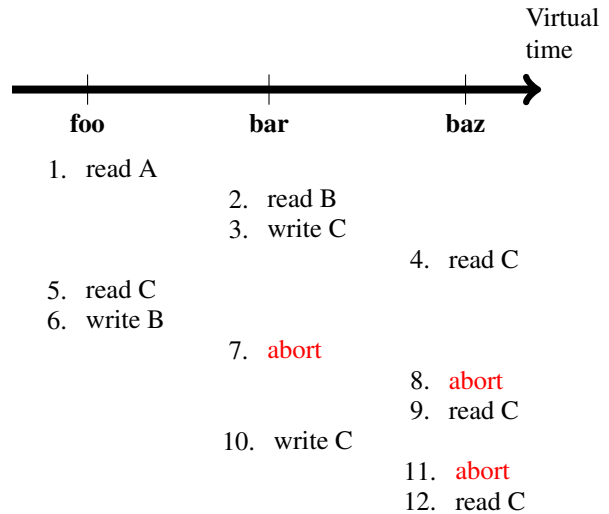


Figure 1: Example build demonstrating abort and cascading abort.

3.1 Parallel builds

We begin by describing how parallel builds are achieved. Suppose that the clean, serial build consists of running three tasks, `foo`, `bar`, and `baz`, in that order. The position at which each task is run in the clean, serial build is called its *virtual time*; for example `foo` occurs at an earlier virtual time than `bar`. In a serial build, virtual time and real time, the order in which operations actually occur, are identical, but in a parallel build they may differ.

The concurrency control used by our system is based on multiversion timestamp concurrency control [1]. Each task is run inside a transaction, isolating its effects from those of concurrently running tasks and allowing its effects to be rolled back if the transaction is aborted. Timestamp concurrency control provides the guarantee that the final state will be the same as running the transactions in order by their virtual time; in our case this is the order of the clean, serial build, as desired.

Returning to our example, we begin by optimistically executing all three tasks concurrently, as shown in Figure 1. The horizontal axis indicates virtual time, while the vertical axis indicates the order read and write operations are executed in in real time.

The first two reads (1 and 2) read the initial versions of the resources A and B that were present before the build started; these might be input files, for example. Next, `bar` writes to C, and both `baz` and `foo` read C (4 and 5); `foo` should see the initial version of C, while `baz` should see the version written by `bar` at (3).

Next at (6), `foo` writes B. Because `bar` occurs later in virtual time, it should have read at (2) the version of B

written by foo at (6), but instead it read the initial version, potentially resulting in incorrect behavior. An ordering of events that causes some transaction to read the wrong version of a resource is termed *physically unrealizable behavior*. To cope with this, we abort the task bar, undoing all of its effects. In particular, this undoes the write to C at (3) which was already read by baz at (4). Because bar influenced the behavior of baz, baz must also be aborted (*cascading abort*).

After bar and baz are restarted, baz reads C at (9); this reads the initial version of C, since the write at (3) was undone by the abort. At (10) bar writes to C, causing baz to abort and restart again. Finally baz reads the correct version of C and all tasks complete correctly.

While traditional timestamp concurrency control is only concerned with obtaining a result equivalent to *some* serial order, we are concerned with enforcing a *particular* serial order. This is why in the event of a conflict, standard timestamp concurrency control aborts the writer, whereas we abort the reader (if the writer were restarted with the same timestamp, the same conflict would re-occur).

To cope with the need for different processes to see different versions of the same file, each transaction is run inside a *virtual filesystem* in which it sees the effects of transactions with earlier (or the same) virtual timestamps, but not transactions with later virtual timestamps. The initial virtual filesystem is based on the real underlying filesystem, allowing input files to be read.

3.2 Incremental builds

To implement incremental builds, we dynamically record for each transaction all actions performed in the virtual filesystem by that transaction and their results. Examples of actions include reads, writes, deleting a file, and testing a file for existence. If the action is a *read* action, its result is recorded (e.g. the file contents that were read). If the action is a *write* action, the requested changes are recorded (e.g. the bytes that were written to the file). This information is stored in a persistent cache.

Later, if the same build task is re-run, we replay all of its actions in order from the cache, but using the current contents of the filesystem. For example, if before it read “foo.c”, it will read “foo.c” again, but any changes made to “foo.c” since the cache entry was created will change the result of the read. If at any point the result of a *read* action differs from the result observed during the cached execution, this indicates a cache miss; any changes made during replay are discarded and the process is run normally. If the replay completes and all results of *read* actions matched, it is a cache hit; the changes made by *write* actions during the replay are retained and the process is not run. This scheme is flexible enough to cope

with a wide variety of filesystem actions, while remaining conceptually simple.

When a process first begins, the only information available is its command-line and environment; any cache entry matching on these values is evaluated as described above, and the first to produce a cache hit is used. Entries are never removed from the cache, but in practice it may be useful to prune cache entries that are not likely to be reused. By persisting and reusing cache entries across builds, incremental building is achieved.

An important assumption for the above procedure to be valid is that all sources of nondeterminism are captured by the list of actions in the cache. There are some situations where this is not true: where a process participates in inter-process communication, multithreaded processes, and other exceptions. In these cases the process is not eligible for caching.

For larger builds, it’s useful to cache larger sections of the build than single tool invocations. To achieve this, the actions of parent processes (processes which spawn other processes) are cached together with those of their descendants. If there is a cache hit for the parent process, none of its descendants need to be re-run.

3.3 Implementation

Now that we’ve described how reliable serial and parallel builds operate conceptually, we describe how existing builds can be augmented to use them with minimal migration effort. Our primary focus was on *make*-based builds, for which each build task is run in a separate process and communication between tasks is via the filesystem, command-line arguments, and the environment. This isolation makes it relatively easy to dynamically trace dependencies through system call interception using *ptrace*. We created a build monitor tool called *ap-make* (for *automatic parallel make*) that does this. We run a clean, serial build using *make*, but trace all system calls it makes and any processes it forks. By modifying the system call number and return value, we can selectively modify the behavior of a subset of system calls. In particular, we modify *wait4()* so that any attempt by *make* to wait on a child is skipped. This effectively creates parallelism between tasks without the need to parse the build configuration file (in our case, the *makefile*).

In the event of an abort, we kill the associated process and re-execute it with the same arguments with which it was started. By emulating *getppid()*, the process cannot observe that it is being re-executed.

The use of *ptrace* is also convenient since it allows us to suspend processes immediately before any system call; this allows us to suspend processes that are likely to conflict and so avoid unnecessary restarts. These predictions are based in turn on predictions regarding which

files will be read and written by each process; typically these predictions are supplied by previous builds.

The standard *ptrace* implementation has high overhead because it intercepts every system call (both before and after) and only allows the monitored process’s memory to be updated a single word at a time. Each of these events involves a context switch. To increase performance, a number of small kernel modifications were made to *ptrace* to mitigate these issues. In all, less than 100 lines of code were added.

In our implementation of the incremental build cache, rather than storing the complete contents of files which are read, we store only their Fowler-Noll-Vo (FNV) hash (a hash chosen for its efficiency). [9] An alternative, used for example by *make*, is to assume that file contents match if the timestamp matches. This is not reliable in general but can be more efficient.

Alternatives were considered for intercepting operations. A custom filesystem, as used in Vesta, [6] can efficiently interpose on all file operations; this could be implemented by stacking a new filesystem on top of the existing one (using a kernel filesystem, a user-mode filesystem, or a network filesystem server) and running build tools using *chroot* inside it. However, *chroot* requires super-user access, and such a system cannot (by itself) interpose on the *wait4()* calls needed to force children to run in parallel.

4 Experiments

The primary goal of our experiments was to determine the benefit *apmake* could provide compared to a clean, serial build using *make*. Secondary goals include: measuring scalability with the size of the build, measuring scalability with the number of processors, measuring how fast *apmake* performs a clean, serial build (a measure of overhead), and measuring performance compared to parallel and incremental builds using standard build tools (“the price of safety”).

Experiments were performed on an Intel Core i7-720QM 1.60 GHz quad core hyperthreaded CPU with 8 GB of RAM and 6 MB of L3 cache. Our primary test case so far was CircleMUD [3], a 41,000-line C-based multiplayer game server (version 3.1). This test case was chosen because it has an unusually simple build system for a project of its size, consisting merely of compiling and linking a number of binaries from C source files. Figure 2 shows our relative advantage to a clean, serial build on this test case during the initial (non-incremental) build. Our tool has no knowledge about the dependencies during this build. Cost is high on a single processor due to the overhead of *ptrace* monitoring, process restarts, and other factors. Additionally, the *apmake* monitor process, which must serially process all events of all pro-

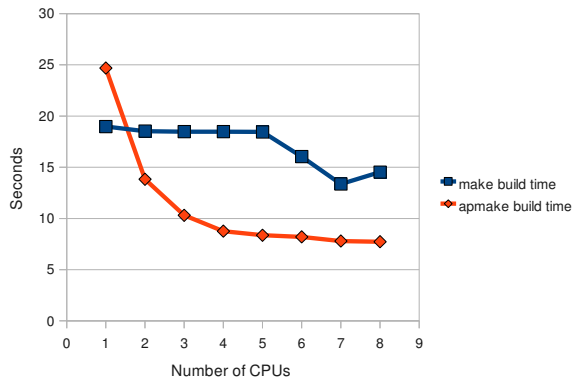


Figure 2: Performance of initial (non-incremental) CircleMUD test build under both clean, serial *make* and under *apmake*, as the number of processors is varied using the `maxcpus` kernel option. All values are averages of 10 trials. Average standard deviation for *make* and *apmake* were 0.32 and 0.37 sec, respectively.

cesses, is a bottleneck, leading to limited scalability. The average number of restarts per build in the 8 processor case was 7.3 (in the case of CircleMUD, only one build task can be restarted, one of the link steps).

In incremental builds on 8 processors in which no files were changed, *apmake* required 2.1 sec. An incremental *make* build in this case requires < 0.1 sec (but does not provide the same guarantee of safety in the presence of incomplete dependencies). If a single source file was changed (affecting only a compile step and a link step), *apmake*’s time increases to 2.8 sec, while *make*’s time increases to 0.98 sec. A clean, parallel build of CircleMUD using “`make -j 8`” took 4.8 sec. This is about 38% faster than the initial run of *apmake* (but does not provide the same guarantee of safety in the presence of missing dependencies).

To test generalization to other code bases, we also tested *apmake* on the source code of *make* itself (version 3.81, about 32000 lines of code) and *flex*, a lexer generator tool (version 2.5.35, about 26000 lines of code). For each test we did a clean, serial build using *make*, an initial run of *apmake* with no knowledge of dependencies, and an incremental run of *apmake* with no files changed, as shown in Figure 3. Performance was slower in the initial *apmake* build for *make* and *flex* due to the overhead of *apmake* (e.g. overhead of *ptrace* and transaction processing). Incremental *apmake* times were superior to the clean, serial build in all cases.

To measure the overhead incurred by the monitor, we take two different approaches: in one, we disable all but one processor, eliminating any opportunity for parallelism. Building CircleMUD with this approach takes 23.9 seconds, 63% higher than the clean, serial with

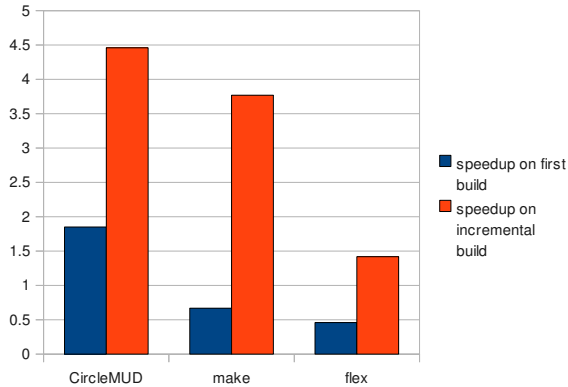


Figure 3: Speedup of *apmake* on all tests. First speedup is for initial run, without any dependency information, and second speedup is for an incremental run of *apmake* with no files changed. Values calculated from average runtimes over 10 trials.

make alone. In the other approach, we disable our modifications to the *wait4()* system call, which causes *make* (executing serially) to wait normally for each child process to complete before executing the next one. Building CircleMUD with this approach on 8 processors requires 17.6 sec (average of 10 runs, stdev 0.19 sec). This is still 20% higher than with *make* alone.

The largest example we attempted to run *apmake* on was the Linux kernel with the *allnoconfig* configuration. Although we were able to complete a build using *apmake*, undiagnosed errors in the system caused it to produce an incorrect result, and so results are omitted.

5 Related work

The problem of making incremental builds reliable was one of the primary goals of Vesta, [6] a configuration management system created by Compaq/Digital Systems Research Center. The most important mechanism for implementing this is the *runtool cache*, which is similar to and formed the basis for *apmake*'s cache. Instead of system call interception, it detects file accesses by build tasks via a custom filesystem. Vesta could also cache larger sections of the build [7], enabling them to achieve incremental builds that take time proportional to the size of the changes rather than the size of the source tree. Vesta also placed heavy emphasis on *repeatable* builds, the ability to reproduce any build and the sources used to build it. However the additional benefits of Vesta come at a cost: Vesta requires a custom filesystem, version control, and build system, and provides no support for migrating from existing ones. Additionally, the system is built around a client-server model with a shared cache,

which is effective for large teams but incurs overhead for small projects. Finally, Vesta provides no support for parallel or distributed building, and would require substantial design changes to support these.

A more practical, but more limited system that uses caching to speed up builds is *ccache*, [12] based on *compiler-cache*. [11] It caches results of invocations of standard compiler tools like *gcc*, but does not generalize to other tools.

The problem of automatically parallelizing builds, and in particular distributing existing builds across clusters of build servers, was the focus of technology patented by Electric Cloud, Inc [10]. Like our system, Electric Cloud optimistically runs build tasks (which they call *jobs*) in parallel, and if a conflict is detected the output of the task is deleted and the task is re-executed. Conflicts are used to augment the build configuration file in subsequent builds, just as we avoid conflicts in subsequent builds using data in the cache. Their method of looking up file versions in response to file reads is also similar to ours. However Electric Cloud does not use a cache or support incremental builds. The authors claim Electric Cloud could be used in combination with Vesta, but Vesta does not explicitly describe dependencies in its build configuration file, so it's not clear how this would be done.

Electric Cloud is designed to cope with a number of issues specific to distributed builds such as efficient distribution of sources, clock synchronization, and node failure which can be ignored in a single-node setting. Although large manycore servers are rapidly becoming more economical than they were in 2003, distributed builds remain valuable for very large builds. However, as was the case for Vesta, the use of build servers is heavy-weight and impractical for small builds, limiting the ability to scale down.

Our virtual filesystem implementation, which transparently redirects processes to read and writes files in a different location without their knowledge, can be compared to *file virtualization* in Windows Vista [8]. The primary use of file virtualization is to implement the Virtual Store, part of User Access Control: when a legacy application attempts to write a file to a location that requires administrator privileges to access, rather than ask the user to elevate the application's privilege, the file is written at a private location in their user directory. File virtualization is an operating system feature that does not require user mode support, but also does not maintain multiple versions. A less direct comparison may also be made to stackable filesystems, which implement filesystems with additional features (such as, in our case, versioning and rollback) on top of simpler underlying filesystems [5].

Because we use transactions to control and roll back

modifications to a filesystem, a natural question is whether transactional filesystems [4] could be leveraged to implement an *apmake*-like system. If the transactions are committed in order, it will ensure that physically unrealizable behavior does not occur. Such an approach has two important limitations: first, transactions cannot see effects of earlier transactions that have not yet committed. Although this helps to avoid cascading abort, it also makes aborts at commit time likely for any task with a dependency on earlier tasks. Second, this model cannot support hierarchical tasks (where each process may spawn children), since unrealizable behavior is detected only at commit time, at which time all earlier transactions have already been committed. Therefore if one process segment is aborted, it may not be possible to abort earlier process segments.

6 Future work

The current monitor implementation faces performance limitations in its design: it must handle a large number of system calls on a single thread (for example the CircleMUD test handles about 75000 calls over less than 10 seconds). Each system call requires substantial processing to implement the virtual filesystem and conflict detection functionality, and the monitored process cannot make progress while its system calls are being processed. If the monitor saturates the processor, multiple build tasks may queue up waiting for their system calls to be handled. An open question is to find a design that avoids these limitations.

In order for our cache subsystem to be sound, it must capture all possible sources of nondeterminism in a program, similar to replay systems like ReVirt. [2] We succeed in capturing many of these, including the results of most file-related system calls, the contents of “/dev/random”, and the current directory. Others prove difficult to handle efficiently and are ignored, including the system time, network access, reading data through a pipe, reading shared memory, and CPU performance counters.

Even for ordinary file-related system calls, we were forced to compromise on soundness in order to achieve reasonable efficiency, because many system calls return more information than is typically used by the application. For example, the *stat()* system call returns (among other things) the inode number, user ID, size, and last accessed time of a file. Despite this, its most common use is to merely determine whether a file exists. Calls normally used to create effects, such as *unlink()*, can be used to read information by examining the return code. The *getdents()* system call, which reads the contents of a directory, is also problematic, since reading a directory conflicts with any update to that directory.

A promising future direction for coping with many of these problems is to create a library that provides a *narrow interface* for interacting with the system — that is, it exposes as little information as possible to the caller. Build tools, once ported to use this library, will naturally be more likely to yield cache hits, even in a completely sound system. The library could interact directly with the monitor process over IPC, avoiding the need for system call interception. Additionally, by porting standard system libraries to run on top of this narrow-interface layer, we can facilitate gradual migration of tools.

7 Availability

The *apmake* prototype used to produce the results in this work can be downloaded with source code under a BSD License from: <http://www.cs.berkeley.edu/~dcoetzee/apmake/>

References

- [1] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv.* 13 (June 1981), 185–221.
- [2] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.* 36 (December 2002), 211–224.
- [3] ELSON, J. Circlemud. <http://www.circlemud.org/>, 1994–2006.
- [4] HASKIN, R., MALACHI, Y., AND CHAN, G. Recovery management in quicksilver. *ACM Trans. Comput. Syst.* 6 (February 1988), 82–108.
- [5] HEIDEMANN, J. S., AND POPEK, G. J. File-system development with stackable layers. *ACM Trans. Comput. Syst.* 12 (February 1994), 58–89.
- [6] HEYDON, A., LEVIN, R., MANN, T., AND YU, Y. The vesta approach to software configuration management, March 2001. Compaq Systems Research Center Research Report 168.
- [7] HEYDON, A., LEVIN, R., AND YU, Y. Caching function calls using precise dependencies. *SIGPLAN Not.* 35 (May 2000), 311–320.
- [8] MICROSOFT. New UAC Technologies for Windows Vista. <http://msdn.microsoft.com/en-us/library/bb756960.aspx>, 2007.
- [9] NOLL, L. C. Fowler / Noll / Vo (FNV) Hash. <http://www.isthe.com/chongo/tech/comp/fnv/index.html>, 1991–2009.
- [10] OUSTERHOUT, J., DELMAS, S., GRAHAM-CUMMING, J., MELSKI, J. E., MUZAFFAR, U., AND STANTON, S. U.S. Patent #7,676,788: architecture and method for executing program builds, Filed 2003 March 25, issued 2010 March 9.
- [11] THIELE, E. compilercache. <http://www.erikyyy.de/compilercache/>, 2001.
- [12] TRIDGELL, A., ROSDAHL, J., ET AL. ccache — a fast C/C++ compiler cache. <http://ccache.samba.org/>, 2002–2010.