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 any 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 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 70% to 200% as long as a clean, serial build, while an incremental build (with no files changed) using our system took between 14% to 73% 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, sequential build time of 5-10 hours. [13] 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, "[m]ost organizations run their builds completely sequentially or with only a small speedup, in order to keep the process as reliable as possible." [13]

Incomplete dependencies arise naturally whenever a developer change introduces a new dependency, but fails to correctly update the dependency information in the build system. 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 gener-

[*] This paper was submitted in December 2010 as a class report for CS 262a, taught by Prof. Eric Brewer at the University of California, Berkeley, in Fall 2010. This is not a peer-reviewed work.

ated.h, and also modified *config*. A 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; we call this the *composition problem*. One component's build may depend on the output of another component's build, but coarse-grained dependencies between components do not expose enough parallelism. Achieving fine-grained parallelism generally requires merging component build systems into a single unified build system, which could be a costly endeavor.

2 Goal

Our goal is to create a *build manager* tool that can be deployed in any existing build environment with little to no configuration or migration cost, and provides parallel, incremental builds that are guaranteed to be correct and reliable. In particular, it should produce correct results even in the presence of incomplete or missing dependencies.

More carefully defining this goal 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. The most obvious such specification is that **a correct build should produce the same output as a clean, serial build**. Because clean, serial builds are procedural, they are easy for developers to understand; because they are deterministic and repeatable, issues are easy to reproduce and fix. This implies of course that in order to be of benefit, our system should build more quickly than a clean, serial build.

3 Our solution

To motivate our solution, we consider an example build. Suppose that the clean, serial build consists of running three tasks, foo, bar, and baz, in that order. Each task reads and writes a series of files. We begin by optimistically executing all three tasks concurrently. The read and write operations are executed in some serial order in real time, expressed by the circled numbers in Figure 1.

The first two reads, labelled 1 and 2, should read the initial versions of the files that were present before the build started; these might be input files, for example. Next, bar writes to C, and both foo and baz read C; foo should see the initial version of C, while baz should see the version written by bar.

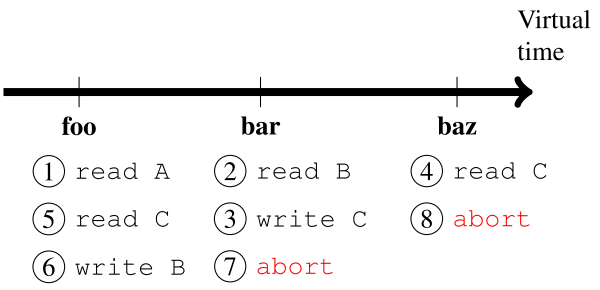

Figure 1: Example build demonstrating abort and cascading abort.

Next, foo writes to B. Because bar occurs later in the clean, serial build, it should have read the version of B written by foo, but instead read the initial version, potentially resulting in incorrect behavior. To cope with this, we will abort the task bar, undoing all of its effects. In particular, this undoes the write to C which was already read by baz. Because bar influenced the behavior of baz, baz must also be aborted (cascading abort). Finally, bar and baz are restarted and bar will read the correct version of B.

The concurrency control described above is based on multiversion timestamp concurrency control. [1] In standard terminology, the tasks above are being run inside a transaction, and an ordering of events that causes some transaction to read the wrong version is termed *physically unrealizable behavior*. We assign virtual times to transactions based on the position at which they occur in the clean, serial build. This guarantees that if they complete without aborting, the result will be equivalent to that of the clean, serial build, as desired. The most important difference is that traditional timestamp concurrency control is only concerned with obtaining a result equivalent to *some* serial order, whereas 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.

3.1 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. For example, if a transaction reads a file, we will record the file contents that were made accessible to it at that time; if a transaction writes a file, we will record the data written; if a transaction tests for a file's existence, we record whether it exists at that time; and so on. This information is stored in a persistent cache. Later, if the same build task is re-run, we will replay all of its virtual filesystem actions. If at any point the result of an action disagrees with the results returned before, for example because the contents of a file it read has changed, it is a cache miss. In this case, any changes made during replay are discarded and the process is run normally. If the replay completes without any disagreement, it is a cache hit; the changes made 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.

A build task is identified by its command-line and environment, so a single task may have many distinct cache entries. If there are multiple entries, they are all evaluated as above, and the first to produce a 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.

Reuse of results across builds is used to achieve incremental builds. As described in section 3.5, reuse of results during a single build is also essential for implementing hierarchical tasks.

3.2 Pessimistic concurrency control using action intervals

The concurrency control described so far is purely optimistic. It leads to unacceptably slow behavior due to a large number of restarts, and it does not take advantage of known information about dependencies. To deal with this we introduce the concept of *action intervals*: each action issued to the transaction manager is replaced by a "begin" and an "end" pair, one issued some time before the transaction performs the action and one issued some time afterwards. If a transaction has issued a "begin" but not an "end" for a particular action, that action is said to be *outstanding*.

Whenever a transaction issues a "begin" for an action, we check to see whether that action would conflict with any outstanding action (cause an abort). If so, the transaction issuing the "begin" is suspended until this is no longer the case. See Figure 2 for an example.

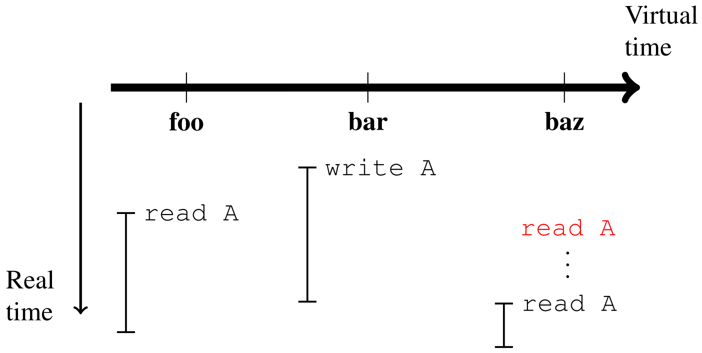

Figure 2: Example demonstrating pessimistic concurrency control using action intervals. The intervals next to an action denote the "begin" and "end" of the action. The read on A by *baz* falls within the action interval of the write on A by *bar*, and causes *baz* to be suspended till the end of the interval. However, *foo* isn't suspended since it occurs before *bar* in virtual time.

To implement pessimistic concurrency control using this mechanism, at the beginning of each process we make a prediction about what actions that process will take, and speculatively issue a "begin" for each of them. The prediction need not be either complete or correct, and can be based on several sources, including:

- Actions performed by a build tool run with the same command line in the past. The cache used for incremental building already tracks all the information needed to do this. We call this an *approximate cache hit*.
- Dependency information obtained by parsing build configuration files, such as makefiles.
- Dependency information extracted by application-specific tools such as "gcc –M".

In addition to enabling pessimistic concurrency control, action intervals allow multiple similar actions — such as reads or writes to the same file — to be consolidated, improving performance. It suffices to issue a "begin" when a file is opened and an "end" when it is closed.

3.3 Extended transaction actions

The basic timestamp concurrency control of [1] supports only reads and writes. While sufficient in theory, in practice, the use of such a limited action set effectively eliminates any opportunity for parallelism due to two common scenarios.

The first scenario is that most builds create or delete many files in the same directory. Creating or deleting

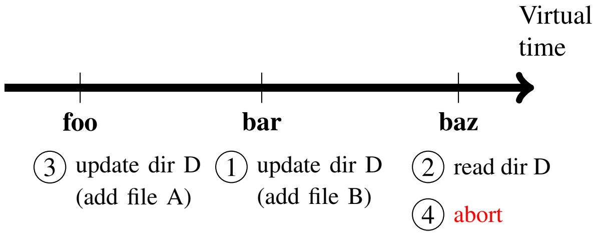

Figure 3: Example build demonstrating *update directory* action; *baz* is aborted but *bar* is not.

a file causes an update to the directory; it is important to track these updates, because a later task may wish to enumerate the contents of the directory. Unfortunately, if this update is implemented as a read followed by a write, no two processes that create or delete files in the same directory can run concurrently without aborting.

To overcome this issue, we define a new *update directory* action which adds or removes a single file from a directory, as shown in Figure 3. Update-update pairs can be executed out of order, but update-read pairs cannot; unlike writes, updates do not follow the Thomas write rule [16] (i.e. they cannot mask one another's effects). At the time a directory is read, we combine all relevant updates to obtain the result. Additional actions are issued for the file itself, so that any conflict over adding/removing the same file is still detected.

The second scenario is similar and involves the use of log files and/or the standard output. If all or most build actions append to some common stream, and this is treated as a read followed by a write, opportunities for parallelism will be lost. Instead, we define an *append* action. Appends can be exchanged with one another (since they are write-only) but not with reads. At the time of a read, all relevant appends are concatenated to obtain the result.

Rather than treating these new actions as special cases, we generalized the concept of read-write conflicts and the Thomas write rule to a *conflict matrix* and a *mask matrix*, analogous to the use of matrices in describing conflicts in locking (see e.g. [4]). In the conflict matrix, the entry in row i, column j is set to 1 if and only if performing action i at a later real time but earlier virtual time than action j, on the same file, leads to a physically unrealizable behavior (for example, the entry (*write*, *read*) would be set to 1). In the mask matrix, the entry in row i, column j is set to 1 if and only if action j *masks* or hides the effects of an action i occurring earlier in virtual time on the same file (e.g. (*write*,*write*) would be 1). Figure 4 shows our complete matrices.

Conflict matrix

	R	W	E	C	D	U	A
R	0	0	0	0	0	0	0
W	1	0	0	0	0	×	0
E	0	0	0	0	0	0	0
C	0	0	1	0	0	0	0
D	×	×	1	0	0	×	×
U	1	×	0	0	0	0	×
A	1	0	0	0	0	×	0

Mask matrix

	R	W	E	C	D	U	A
R	1	1	1	1	1	1	1
W	0	1	0	0	1	×	0
E	1	1	1	1	1	1	1
C	1	1	0	1	1	1	1
D	×	×	0	1	1	×	×
U	0	×	0	0	1	0	0
A	0	1	0	0	1	0	0

Figure 4: Conflict and mask matrices, where R, W, E, C, D, U, A denote read, write, exists, create, delete, update directory, and append, respectively. Read and write can only be performed on existing files; create and delete create/delete a file or do nothing if it already does/does not exist. Read-only actions have no effects, so their effects are masked vacuously by any action. Irrelevant values are marked with an ×.

3.4 Implementation

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 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 emulate a subset of system calls ourselves. In particular, we emulate *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 description file (e.g. 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. Additionally, other transactions may be aborted, as described in section 3.5. 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, which is used to implement the pessimistic concurrency control described in section 3.2. We issue "begin" actions before each relevant system call, and (implicitly) perform all "end" actions when the process terminates. For pessimistic concurrency control, we also wrote an application-specific wrapper for *gcc* to predict, in advance of being executed, that *gcc* will create and write to its output file.

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: a ptrace option was added allowing all system call arguments to be retrieved at once; tracing of some system calls not of interest to us was disabled; and a system call was added to allow us to open a file on behalf of another process (the use of this call obviates the need to emulate read and write calls, which are particularly expensive). 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). [12]

Our ptrace-based implementation is not amenable to build systems such as *Ant*, which use class-loading to load all build tools into a single process. We cannot reliably identify which task is performing each file access, and we have no means of aborting and restarting transactions associated with tasks. However, our method applies to such systems as well, and could be used, for example, to construct a drop-in replacement for *Ant* with relatively minor modifications.

Alternatives were considered for intercepting operations. A custom filesystem, as used in Vesta, [7] 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.

Another promising alternative is binary rewriting of all system calls at load-time using a library, thereby eliminating frequent context switching by caching some state in-process and communicating with the monitor via shared memory only when necessary. A preliminary *apmake* system based on Jockey [14] used this approach, but was not efficient in practice because typical builds spawn a large number of very short-lived processes for operations such as "rm" and "mv". Even when all the necessary rewrites are cached, the cost of loading the library alone significantly increased overall build-time. A hybrid approach may be possible in which *ptrace* is used to trace short-lived tasks while binary rewriting is used to trace long-lived tasks.

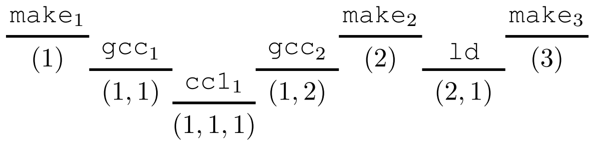

Figure 5: Example of hierarchical tasks; *make* spawns *gcc* and *ld*, and *gcc* spawns *cc1*. A transaction is assigned to each process segment, with timestamps (shown below each segment) in lexicographic order from left to right.

3.5 Hierarchical tasks

If the clean, serial build consisted of a simple sequence of processes being executed in order, it would suffice to assign a single transaction to each process. However, in real builds such as those using *make*, there is a hierarchy of processes, with child processes spawned by parents. In such a setting, no assignment of transactions to processes yields an ordering matching that of the clean, serial build. Instead, parent processes must be split at each point where a child process is spawned to form *process segments*. If a transaction is assigned to each segment, these can then be assigned times to match the order of the clean, serial build. An example is shown in Figure 5.

In the sequence model, each new process is dynamically assigned a fresh virtual timestamp greater than all those assigned so far. In the case of hierarchical tasks, timestamps are represented as finite sequences of integers, ordered lexicographically and beginning with (1). When a process with timestamp $(x_1, x_2, ...x_i)$ spawns a new child, the child receives timestamp $(x_1, x_2, ..., x_i, 1)$ and the new segment of the parent receives timestamp $(x_1, x_2, ..., x_i + 1)$.

Just as before, when a transaction is aborted and restarted, it maintains its old timestamp, and as a result, its position in the transaction tree. Since we restart processes by re-executing them, individual process segments cannot be restarted independently. Instead, if any segment of a process is aborted, all its segments are aborted, allowing the process to be restarted from the beginning. When a non-leaf process is aborted and restarted, it may not spawn the same sequence of children, so in order to ensure uniqueness of timestamps, we must abort (and not restart) all existing descendants of a transaction before aborting and restarting it.

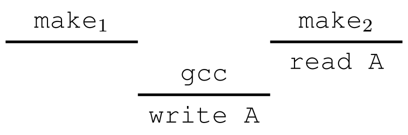

Figure 6: The progress problem with hierarchical tasks. Suppose *make* reads A immediately after forking *gcc*. If *gcc* is aborted as soon as *make* is aborted, then *make* is restarted, the same failure will re-occur.

Caching results of hierarchical tasks is problematic, since the behavior of a later segment of a process may depend on what actions were taken by the process's descendants. There is also no straightforward way to implement a cache hit for a process segment (which should cause the process to "skip" to the next child spawn point). We instead adopted the approach of caching task *subtrees*; the actions of a task and all its descendants are concatenated into a single list. If there is later a cache hit for the task subtree, the full list is replayed, replicating the entire subtree's effects without needing to re-execute any of the processes in that subtree.

In the case where a transaction conflicts with one of its ancestors in the tree, there is a risk of failing to make progress, as showin Figure 6. To resolve this, we take advantage of the cache used for incremental builds by allowing all descendants of a process to terminate before it is restarted. This ensures that their actions will be entered in the cache and are available for reuse during the current build. In the example in the figure, the results of *gcc* would be cached, and after *make* is restarted, *gcc* would produce a cache hit, and so its write to A would occur immediately upon being spawned, eliminating the opportunity for a conflict.

4 Experiments

In our experiments, our primary goal was to determine the benefit our system 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 or 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 was CircleMUD,[3] a 41,000-line C-based multiplayer game server (version 3.1). Its build consists merely of compiling and linking a number of binaries from C source files. Figure 7 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 processes, is a bottleneck, leading to limited scalability. The average number of restarts per build in the 8 processor case was 7.3.

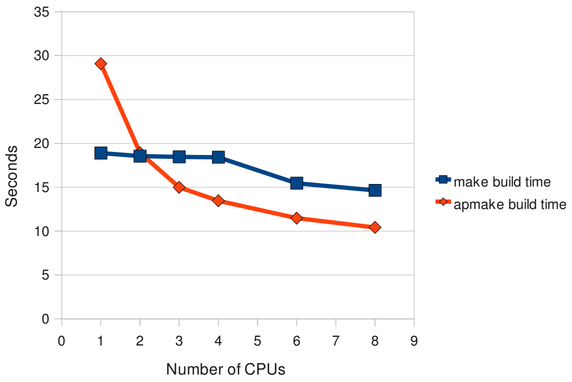

Figure 7: Performance of initial (non-incremental) CircleMUD test build under both clean, serial *make* and under *apmake*, as the number of processors is varied using the `maxcpus` kernel option. All values are averages of 3 trials. Average standard deviation for *make* and *apmake* were 0.32 and 0.85 sec, respectively.

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). 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 twice as fast as the initial run of *apmake* (but does not provide the same guarantee of safety).

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 8. 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

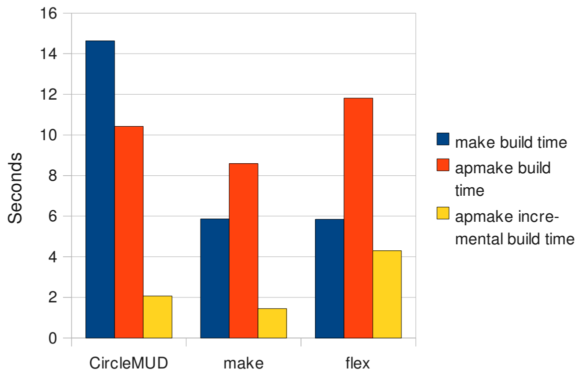

Figure 8: Performance of all tests under clean, serial *make*, initial run of *apmake*, and incremental run of *apmake* with no files changed. All values are averages of 3 trials. Average standard deviation for *make*, *apmake* initial runs, and *apmake* incremental runs were 0.09, 0.21, and 0.12 sec, respectively.

clean, serial build in all cases.

When *apmake* is modified to run the build sequentially (by obeying all *wait4()* system calls, but still running on 8 processors), there are no restarts and the time for building CircleMUD is 24.4 sec. This is better than the time on a single processor (29.0 sec) but still 67% higher than the sequential build time. This gives an impression of the overhead incurred by the monitor.

When more dependency information is available, performance improves. To demonstrate this, we added a simple *build wrapper* for *gcc* which examines its command line, and if it sees "-o filename" it predicts a write on that file.

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, [7] a configuration management system created by Compaq research. 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, [8] 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*, [17] based on *compilercache*. [15] 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. [13] 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, nor is it agnostic to the choice of build system; it parses the build configuration file to determine the initial dependency graph. 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 heavyweight 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. [10] 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. [6]

Because we use transactions to control and roll back modifications to a filesystem, a natural question is whether transactional filesystems [5] 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 of earlier tasks. Second, this model cannot support hierarchical tasks, 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

6.1 Ensuring soundness

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 by far is to merely determine whether a file exists. We treat these calls as *exists* actions, and fabricate default information for the remaining fields. This works only as long as the applications being traced do not use the information in a meaningful way.

Calls normally used to create effects, such as *unlink()*, can be used to read information by examining the return code (e.g. *unlink()* returns EISDIR only if the given path refers to a directory). Many file-related calls can be used to infer permissions information by checking for an EACCES return. Mounting and unmounting can affect interpretation of all paths in the affected subtree. Opening a file for writing on a read-only filesystem should return an error. We ignore all of these.

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, even if the read was (say) only looking for files matching a particular pattern. The result is that programs that use *getdents()* are restarted frequently. Some programs (most notably *make*) cache filesystem contents so that they can perform common operations without making system calls; this defeats attempts to analyze dependencies at the system call level. Presently we resolve this (unsoundly) by ignoring conflicts with *make*.

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. Examples of calls that might be useful in such a library:

- path_exists(path): does a file/directory exist at the given path? returns no on error
- file_size(path): gets size of a file in bytes; returns zero on error
- unlink(path): unlinks a file if it exists and is a file; returns no result
- cache(directory): caches the contents of the given directory; returns no result
- find_files(directory, pattern): finds all files in a directory matching a given pattern

Note that partial functions (such as *file_size()*) are converted into total functions that always produce some meaningful result. This limits the amount of information that can be inferred.

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.

6.2 Change detection and minimal builds

Although *apmake*'s incremental builds are faster than clean, serial builds, they still (like *make*) require time proportional to the size of the build in order to scan all input files looking for changes. To avoid this, two measures are needed: the ability to cache larger sections of the build, such as modules, and the ability to reliably identify build inputs which have changed since the previous build (change detection). Change detection is handled by *make* through timestamps, while Vesta handles it through version control integration. File notification systems like *dnotify* and *inotify* can track all changes to a directory, but this faces two difficulties: the notification service must be running at all times, and these systems do not scale effectively to tracking the large and dynamic set of directories accessed by a typical build.

A promising alternative is operating system support for obtaining comprehensive lists of changes to a volume during a time interval. Often such a feature is straightforward to provide as an extension to existing journalling requirements, and facilitates other applications like efficient backup. NTFS, for example, provides *Change Journals* (also known as *USN Journals*) for this purpose. [11] We are not aware of an existing build system that exploits this feature, or of a similar feature for UNIX filesystems. Similarly, it would be useful if the filesystem were to track a hash of each file's contents over time. Such a system could be efficiently implemented, for example, using Merkle trees. [9]

The other requirement is to cache larger sections of the build. Our existing implementation can effectively cache larger sections of builds that invoke subtasks to build modules (e.g. systems using recursive *make*) but would not be effective for builds in which the task tree is shallow and has high fan-out. In this case, it may be useful to automatically identify clusters of tasks that can be cached as a unit.

7 Acknowledgments

The authors wish to thank: the Compaq research team, for providing a fundamentally new design in the space of build systems; David Wagner, for advice regarding coping with concurrent accesses to files and for suggesting other faculty; Maria Welborn, for advice regarding system call virtualization via system call rewriting; Ras Bodik for providing assistance with funding; Ras Bodik, Koushik Sen, and others for providing feedback regarding presentation; Eric Brewer, for providing suggestions regarding file operation interception and feedback on evaluation; and software developers at Microsoft and Berkeley for providing feedback regarding their experiences with build systems.

8 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] Gray, J. N., Lorie, R. A., Putzolu, G. R., and Traiger, I. L. Readings in database systems (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998, ch. Granularity of locks and degrees of consistency in a shared data base, pp. 175–193.

[5] Haskin, R., Malachi, Y., and Chan, G. Recovery management in quicksilver. *ACM Trans. Comput. Syst. 6* (February 1988), 82–108.

[6] Heidemann, J. S., and Popek, G. J. File-system development with stackable layers. *ACM Trans. Comput. Syst. 12* (February 1994), 58–89.

[7] 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.

[8] Heydon, A., Levin, R., and Yu, Y. Caching function calls using precise dependencies. *SIGPLAN Not. 35* (May 2000), 311–320.

[9] Merkle, R. C. U.S. Patent #4,309,569: method of providing digital signatures, Filed 1979 September 5, issued 1982 January 5.

[10] Microsoft. New UAC Technologies for Windows Vista. `http://msdn.microsoft.com/en-us/library/bb756960.aspx`, 2007.

[11] Microsoft. About Volume Management: Change Journals. `http://msdn.microsoft.com/en-us/library/aa363798%28v=vs.85%29.aspx`, 2010.

[12] Noll, L. C. Fowler / Noll / Vo (FNV) Hash. `http://www.isthe.com/chongo/tech/comp/fnv/index.html`, 1991–2009.

[13] 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.

[14] Saito, Y. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging* (New York, NY, USA, 2005), AADEBUG'05, ACM, pp. 69–76.

[15] Thiele, E. compilercache. `http://www.erikyyy.de/compilercache/`, 2001.

[16] Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4* (June 1979), 180–209.

[17] Tridgell, A., Rosdahl, J., et al. ccache — a fast C/C++ compiler cache. `http://ccache.samba.org/`, 2002–2010.