Build System Rules and Algorithms

Mike Shal <marfey@gmail.com>

2009

Abstract

This paper will establish a set of rules which any useful build system must follow. The algorithms from *make* and similar build systems will be examined under the context of these rules. Where the algorithms fall short, additional investigation will take place to see how to do better. The outcome of this investigation will suggest that any build system that wishes to be truly scalable must 1) provide a list of file changes since the last build up front, and 2) avoid loading the entire graph. Also, this paper will look into build correctness, and why developers still rely on outmoded concepts like a "clean" build.

1. Introduction

A build system is commonly used during software development to reduce the time spent in the Edit-Compile-Test cycle. The developer will typically work with a fully-built project, then edit a few source files out of many and want to see the result of those changes in the final program. The basic theory behind the build system is that it is much faster to update only the parts of the project that require modification (due to the developer's initial edits) than it is to re-build the entire project. It is well known throughout the industry that the speed and reliability of this process ultimately determines the productivity of the developer:

> A crucial observation here is that you have to run through the loop again and again to write a program, and so it follows that the faster the Edit-Compile-Test loop, the more productive you will be, down to a natural limit of instantaneous compiles.
>
> - Daily Builds Are Your Friend - Joel on Software
> http://www.joelonsoftware.com/articles/fog0000000023.html

There are many variants of build systems, perhaps the most common of which is *make*. Alternatives such as SCons, ANT, waf, A-A-P, OMake, and many

others all attempt to fix various deficiencies in *make*, or are simply re-implementations in different programming languages. However, the core algorithms used by these build systems haven't really evolved since the original directed acyclic graph (DAG) processing ones introduced by *make* 30 years ago.

Originally, the typical usage pattern for *make* involved creating an independent per-directory Makefile to describe the relationships between files in the project. This "recursive make" setup had a number of issues, particularly with regard to speed and reliability. The paper *Recursive Make Considered Harmful*[1] (RMCH) analyzed the recursive make setup to find the root causes of these issues, and suggests using a non-recursive make setup (where all build rules are parsed in a single *make* invocation) to resolve them. This paper helped pave the way for other build systems, where using a "global view" of all dependencies in the system is now the norm.

Although RMCH was a boon for those stuck with poorly implemented recursive Makefiles, it set an unfortunate standard for the next several years that resulted in unscalable build systems. I refer to these as "Alpha" build systems, which are build systems that use a global dependency view and therefore have O(n) updates. A new class of build systems will be presented in this paper, called "Beta" build systems. These systems will scale in $O(\log^2 n)$ time with respect to the project size. (For clarity, $\log^2 n == (\log n) * (\log n)$. The base is omitted.)

Further, RMCH and derivative build systems do not address build correctness as it pertains to file renaming or deletion, which are fairly common and certainly useful operations during software development. While a global dependency view allows for correctness in the example provided in RMCH, that correctness is quickly broken if one of the C files is renamed, or the name of the target executable is changed. The cause of this brokenness will also be examined in this paper. Although this is a seemingly disparate problem from the scalability issue, the solution is actually fundamentally intertwined.

This paper will establish a set of rules for a build system that must be followed in order for it to be useful to a developer. The DAG processing algorithms used by Alpha build systems will be examined under the context of these rules. After investigating the root causes of the poor performance of Alpha build systems, new algorithms will be proposed. The result will be a Beta build system which has theoretical and measurable real-world performance gains, as well as usability improvements that may eventually let the developer forget the build system is even there.

1.1 Definitions

This section provides some definitions for terms used in this paper that are either ambiguous in the software world, or for concepts that don't exist at the time of this writing and thus are made up so a reasonable discussion can follow.

Build System

In this paper, a build system is any piece of software that provides facilities for constructing and parsing the DAG which represents the dependencies among files in a software project. This paper is *not* focused on the aspect of building software related to configuration options as would typically be handled by software such as autoconf or kconfig, even though some of these features have been introduced into build systems billed as replacements for *make* (which itself does not have such features built-in). When the build system is invoked, it will input the DAG and current filesystem state, and output a new DAG and/or files that represent the output of the build.

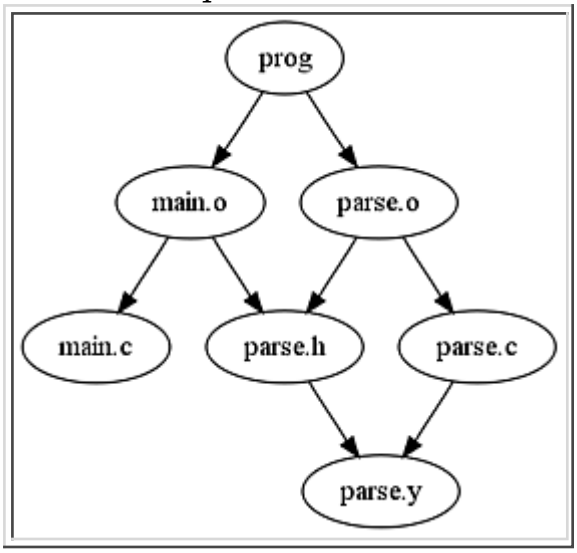
Update

The update operation is the operation of the build system that actually compiles and links files that have changed. This is the "Compile" portion of the Edit-Compile-Test cycle (and optionally the "Test" portion as well, depending on how automated and modular the tests are).

Global DAG

The global DAG is the entire dependency graph of the software system. This is the DAG used in a non-recursive make setup, or in any global "top-down" build system as is common in a *make* replacement. Figure 1 is an example of a global DAG, and is the same example as was used to illustrate the problem in RMCH. Note that the arrows in this graph are actually reversed from the graph in RMCH. Although we will see later in this paper that the arrows actually should go "up", in *make*'s view the arrows go "down". Essentially RMCH displayed how the DAG *should* look, but was mis-representative of how the DAG is processed in *make*.

Figure 1: Global DAG of the example from RMCH



Incomplete DAG

An incomplete DAG is any subgraph of the global DAG that does not have enough information to update a system while maintaining consistency. This is the kind of DAG used by an individual *make* process in a recursive make setup. The use of incomplete DAGs results in the problems described in RMCH. Figures 2 and 3 show two different incomplete DAGs, as might be seen by the two separate Makefiles in a recursive make setup. The greyed-out nodes and links are not part of the DAG - they are only shown to make it obvious what part of the global DAG is missing. An example of one of the issues that comes up from using recursive make is the fact that the main.o->parse.h link is not present in the second incomplete DAG. As described in RMCH, it is possible for *make* to construct parse.o from a new version of parse.h, but link in a main.o which was built from an old version of parse.h.

Figure 2: Incomplete DAG as viewed by the "main" Makefile

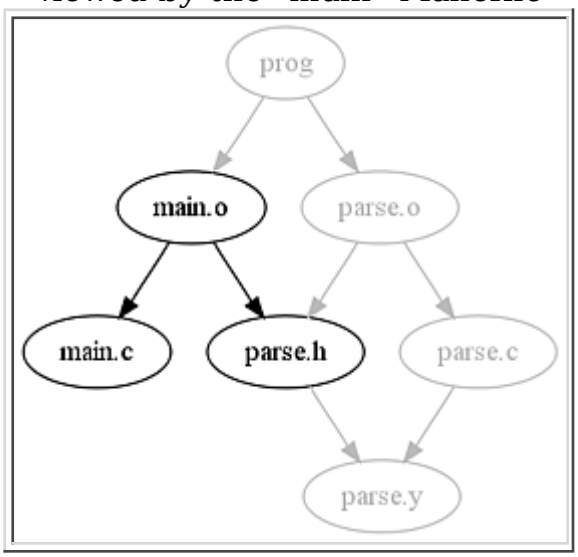
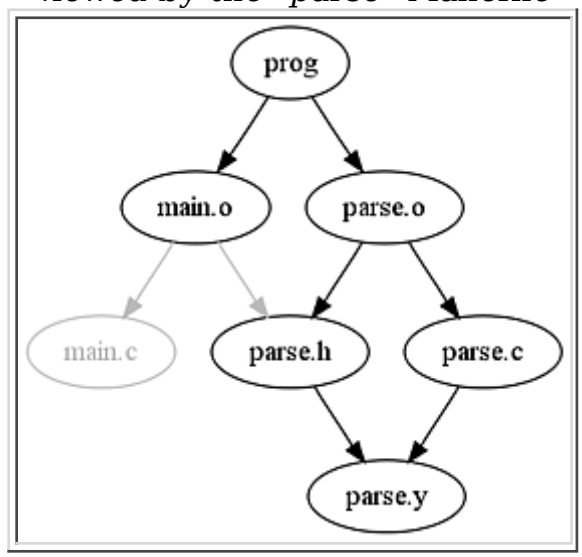


Figure 3: Incomplete DAG as viewed by the "parse" Makefile



Partial DAG

A partial DAG is any subgraph of the global DAG that (unlike an incomplete DAG) has enough information to update a system while maintaining consistency. Further, a *minimal* partial DAG is the smallest partial DAG in the set of partial DAGs that include a particular subset of root nodes. Figures 4 and 5 show the partial DAGs for the root nodes main.c and parse.y, respectively. Note that both of these examples are minimal partial DAGs. A partial DAG that includes both source files would actually consist of the entire global DAG.

Figure 4: Partial DAG including main.c

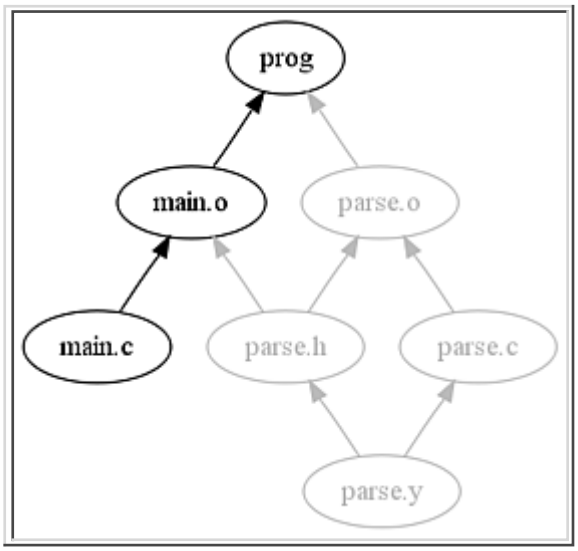
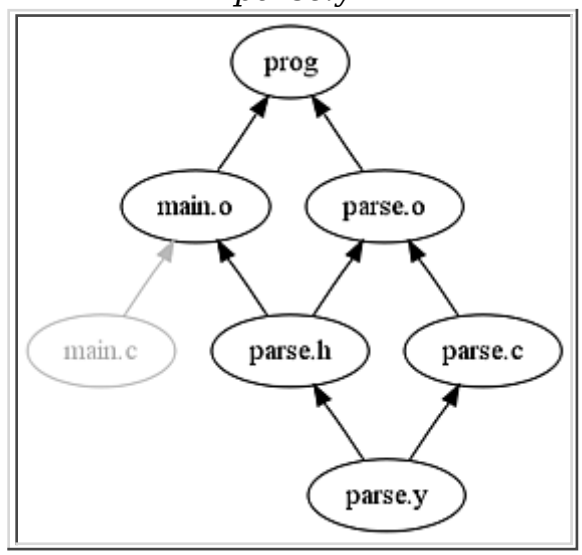


Figure 5: Partial DAG including parse.y



Alpha Build System

An Alpha build system is any build system that uses a global view of dependencies in order to update a software project. Specifically, this includes *make*, as well as its aforementioned derivatives like SCons, ANT, and waf, among others. Note that *make* is considered an Alpha build system independent of whether a non-recursive or recursive style setup is used. In a non-recursive setup, the global view of the entire project is standard. In a recursive setup, one can imagine it as a set of independent Alpha build system executions, each of which has its own "global" view of an incomplete DAG.

All Alpha build systems run in O(n) time, where n is the number of files in the project.

Beta Build System

A Beta build system is any build system that uses a partial DAG to update a software project. In particular, a Beta build system is marked by its use of the Beta Update Algorithm, which is described later in this paper.

Beta build systems are capable of running in $O(\log^2 n)$ time, where n is the number of files in the project.

System State *X*

The system state includes the contents and permissions of files and directories. Other file metadata such as timestamps or owner information is *not* included in the system state definition. For example, suppose two developers download and build a "Hello, World" project at different times. Both developers may have a directory with the files *hello.c*, *hello.o*, and a *hello* executable. Even though the files will have different owners and timestamps, both developers will have the "Hello, World" project in a particular system state, called *X*. If one of the developers changes the source file to print a different message, his project will now be in system state X' to indicate that the project is changed and needs updating. After an update, his project will be in system state *Y*. Similarly, adding a new file to the project would result in a new and different system state.

Note that any software project that includes build timestamps into the final product is inherently not reproducible, since the current time is (by definition) ever-changing. Use of timestamps or other file metadata in such a manner is discouraged.

Up-to-date System

A system is up-to-date if it is in state *X*, and doing an update with a correct build system either performs no changes, or the changes are innocuous, so that the system is still in state *X*. Note that if a broken build tool has a system in state *X'* and incorrectly leaves it in state *X'* when it should go to state *Y*, then the system is *not* up-to-date. That is, we're looking at the actual system state here, not just the output of a specific program like *make* when it claims that a target is up-to-date.

2. Build System Rules

First, I will list the rules so that they can be easily referenced, then follow with the detailed rationale. Any build system must abide by the following rules:

1. Scalability: Given an already up-to-date system, applying a change to the system and building it should take time that is proportionate to the changes required to bring the system up to date.
2. Correctness: Given an up-to-date system in state X, applying change A and updating the system may result in state Y. Undoing change A and performing an update must return the system to state X.
3. Usability: There must not be different ways of building the system that the user must remember in order to accommodate rules 1) or 2).

2.1. Scalability

> Fast iteration is more than just about time and speed. It's also about how you feel about the code and what you dare do with it.
>
> - The Quest for the Perfect Build System
> http://gamesfromwithin.com/?p=42

Rule 1) is necessary because updating an already built system is the most common operation done during development, and any time spent waiting for the program to update is time wasted. Note this rule mentions that the time should be proportionate to *the changes required to bring the system up to date*. In particular, build systems that take time proportionate to the size of the entire project are not scalable. Projects grow over time, yet the size of the changes done in a single development cycle is fairly constant. That is, even though a project may have thousands of files, a developer will generally only focus on a particular algorithm, or on the interaction between a few modules.

These changes only require modifications to perhaps a handful of files. If you can go through a cycle in a few seconds when the project is small, why should those same changes require several minutes or hours when the project is large?

2.2. Correctness

Rule 2) is necessary in order to be sure that the program used in testing is actually representative of the source files. There should be no concern about the side effects of moving around functions, files, or directories. Undoing any of these changes and performing an update must get the source tree back into a previous state so the developer is free to try another approach. Further, all developers in the team should be able to update their source trees without worrying that cruft from a previous build will put their system in a bad state. This must be a normal operation directly supported by the build system -- not a special case that needs to be handled manually by the developer.

> Renaming is the killer app of distributed version control
>
> - Mark Shuttleworth
> http://www.markshuttleworth.com/archives/123

What good is a renaming capability in version control if the build system can't handle it?

2.3. Usability

Finally, the developer shouldn't have to worry about the scope of the source changes. It should be enough to tell the build system, "I'm done editing -- make it work." If the developer is left wondering, "Can I do a fast build in just this subdirectory or do I have to build from the top for this change?", or "Should I do a 'clean' build just to be sure?", then this indicates poor usability. As a real-life example of how the poor usability from *make* can affect a user, see this FAQ from the FreeBSD Handbook:

> 24.7.14.1. Do I need to re-make the world for every change?
>
> There is no easy answer to this one, as it depends on the nature of the change. For example, if you just ran CVSup, and it has shown the following files as being updated:
>
> ```
> src/games/cribbage/instr.c
> ```

```
src/games/sail/pl_main.c
src/release/sysinstall/config.c
src/release/sysinstall/media.c
src/share/mk/bsd.port.mk
```

it probably is not worth rebuilding the entire world. You could just go to the appropriate sub-directories and make all install, and that's about it. But if something major changed, for example src/lib /libc/stdlib then you should either re-make the world, or at least those parts of it that are statically linked (as well as anything else you might have added that is statically linked).

At the end of the day, it is your call. You might be happy re-making the world every fortnight say, and let changes accumulate over that fortnight. Or you might want to re-make just those things that have changed, and be confident you can spot all the dependencies.

- FreeBSD Handbook: "Rebuilding world"
http://www.freebsd.org/doc/en/books/handbook/makeworld.html

There needs to be an "easy answer". There should only be one command to update the system, and you should be able to run it anytime, regardless of what you changed or how you changed it. Let the build system decide if only one subdirectory needs to be updated, or which statically linked binaries need to be built, or which stale files need to be removed. It should not be your responsibility to keep track of the whole system.

2.4. Examples of Build System Rule Violations

Here we will take a brief look at how a build system rule violation might manifest itself for a *make* or *make*-like build system.

You can tell if your build system is violating the first rule (scalability) if:

- Updates were very quick when you first started a project, but now that there are thousands of files spanning many directories, making small changes seems to take forever (even though the build system only executes a few commands when it finally decides what to do).
- Because of the previous issue, perhaps you sometimes perform a build in a particular subdirectory to save time and you "know" that it will be sufficient to bring the system up-to-date for the changes you have made. This is actually a violation of rule 3), since running "cd foo; make" is different from just "make" -- i.e. you have to keep track of your build system in your head. However, the reason you would do this is because as a developer, you can't stand it when rule 1) is broken.

The second rule (correctness) is violated because *make* does not get rid of stale targets. For example, consider this Makefile:

```
program: foo.c
	gcc $< -o $@
```

Now, you decide you don't like the name "program" because it's too generic. So you change the target to "hello_world":

```
hello_world: foo.c
	gcc $< -o $@
```

On second thought, "program" was a pretty cool name:

```
program: foo.c
	gcc $< -o $@
```

At this point, you may run *make* and expect everything to be perfectly up-to-date. And while "program" certainly would be up-to-date, you would still have "hello_world" lying around. No matter, just "rm hello_world", right?

But, what if this *make* process isn't executed by a developer? Now it's a process run automatically when someone checks-in a file. And instead of a stale program, you left libfoo.so lying around, and some other program links to an old library that isn't actually built anymore. Since the build still succeeds, this problem goes unnoticed for some time. Then a new developer wants to build your project, checks it out, and finds that it doesn't work because libfoo.so doesn't exist for him.

Obviously, that is unacceptable. So, to actually check that the poor developer doesn't find a broken build, you end up doing this on your automated build machine:

```
rm -rf foo
revision_control checkout foo
cd foo; make
```

Look familiar? Or maybe your build system builds everything in a separate directory. This is merely a minor performance improvement:

```
rm -rf build
revision_control update
make
```

If you have to do something like this to get a correct build, it should tell you something: you do not have a true build system. You have a glorified shell script.

Also note that this clearly violates rule 3). As a developer, you usually run

"make" if all you do is change a C file. But, if you do something entirely unheard of like change a library name, you have to do "rm -rf build; make". You better not do something really silly like rename a file, or dare I say it -- a directory! If you address rule 3) by changing your build procedures to always do "rm -rf build; make", or "make clean all", or some other such nonsense, then you're seriously in breach of rule 1).

We have seen some ways that violations of the first two rules manifest themselves in defective build systems. Developers may recognize this, and then modify their behavior by running the build system in different ways to try to force it into exhibiting the desired behavior. As a result, developers may choose to violate rule 3) in order to try to make up for a lack of scalability or correctness.

Now we will look more in depth into why existing build systems -- which are grouped here as "Alpha" build systems -- fail to provide any of these basic capabilities.

3. Alpha Build System Algorithms

The Alpha build system algorithm is very simple:

1. **Build DAG**: Construct a global DAG from dependency files.
2. **Update**: Walk through the DAG to see which targets are out of date and run the commands to update them.

The various build systems vary somewhat in their specific implementation. For example, *make* uses timestamps in step 2 while SCons uses file signatures, and ANT uses XML files instead of Makefiles in step 1. However, the basic algorithm is still the same. Both steps 1 and 2 in the algorithm presented here are linear time operations. It is actually more useful to investigate step 2 first to see how Alpha build systems determine which targets are out of date, and then use information from that to come back and re-visit step 1.

3.1. Alpha Update Algorithm

Let us look at how an Alpha build system determines that a target is out-of-date. At this stage of the build process, the system already has a global DAG loaded in memory. A very simplified version of the Alpha update algorithm is presented here -- it recursively updates files based on their dependencies. This example shows it comparing timestamps, so it is very similar to how *make* works. You can see how a file must be updated if any of its dependencies have just been updated, or if a dependency is newer than the target. This function would initially be invoked with the default target (such

as "all", for example).

```
update_file(f)
        need_update = false
        foreach dependency d {
                if(update_file(d) == UPDATED ||
                   timestamp(d) newer than timestamp(f))
                        need_update = true
        }
        if(need_update) {
                perform command to update f
                return UPDATED
        }
        return PASS
```

Although this algorithm is vastly simplified (for example, a real implementation would not re-process a file that has already been checked), we can still examine it to estimate its runtime performance. The update_file() function does a simple depth-first search on the graph. Assuming the initial target connects all of the subgraphs in the DAG, this means the entire DAG must be parsed. Further, the timestamp() function is called for every file in the build tree. Although stat() is generally very fast, it still requires non-zero time. For thousands or millions of files it can no longer be considered an insignificant amount of time to read all of those directory entries. Ultimately the scalability of this update algorithm is limited to O(n), where n is the size of the DAG. Consider the RMCH parse example. Here, the global DAG is already loaded from a non-recursive Makefile, and the user has changed the main.c file. The dotted outlines indicate nodes that have been visited by *make*.

Figure 6: Make invoked with target "prog"

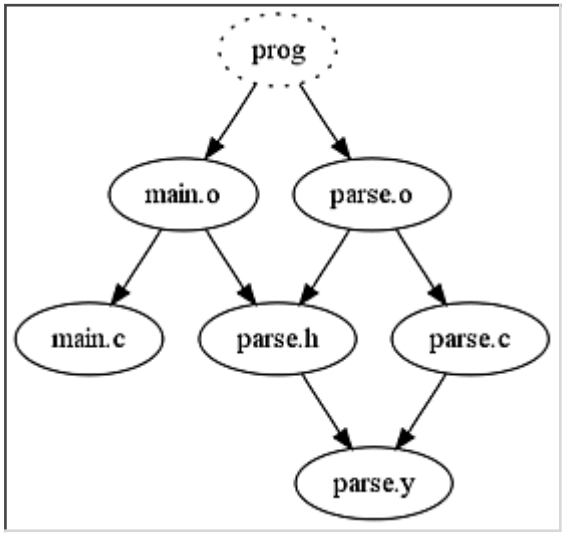


Figure 7: To update prog, must visit main.o

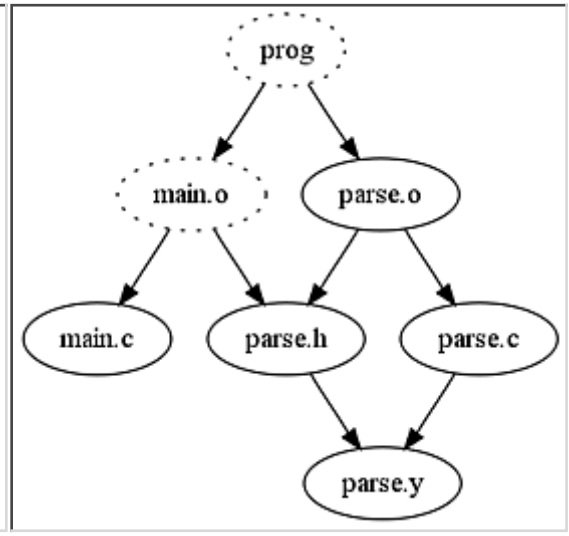


Figure 8: To update main.o, must visit main.c

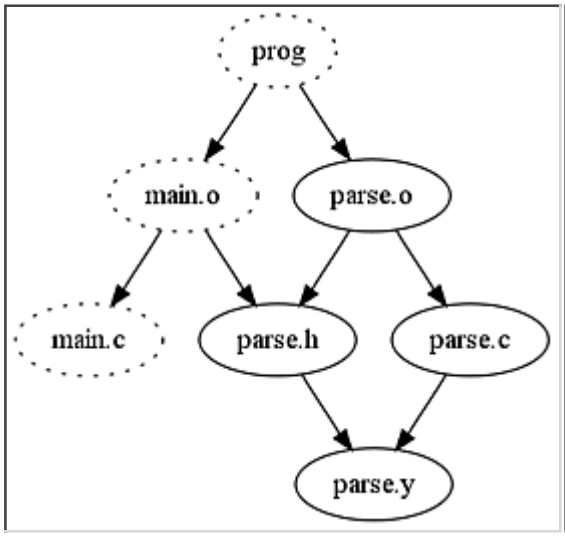


Figure 9: To update main.o, must also visit parse.h

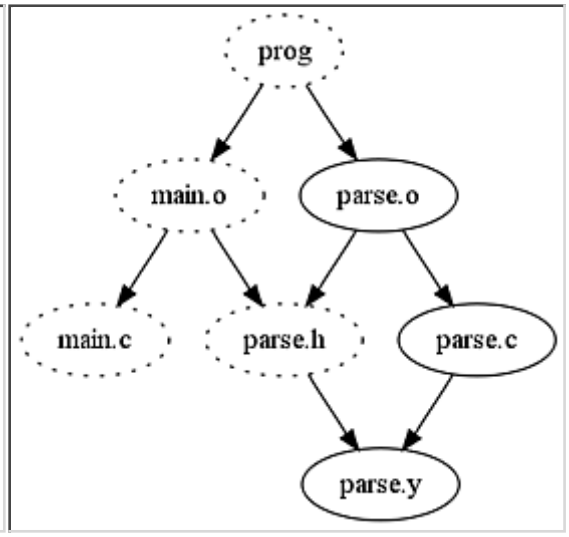


Figure 10: To update parse.h, must visit parse.y

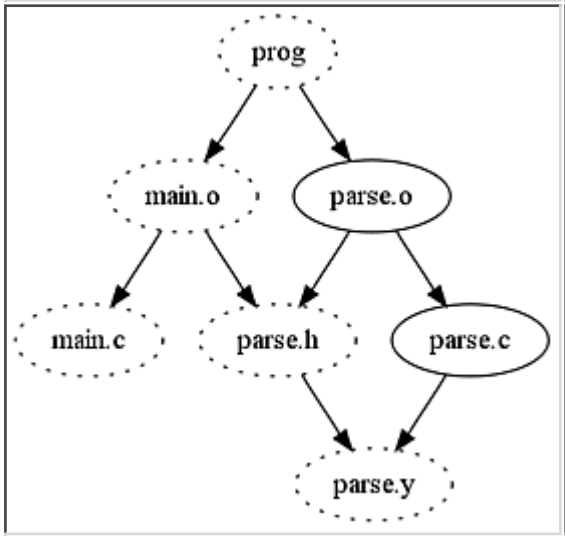


Figure 11: To update prog, must also visit parse.o

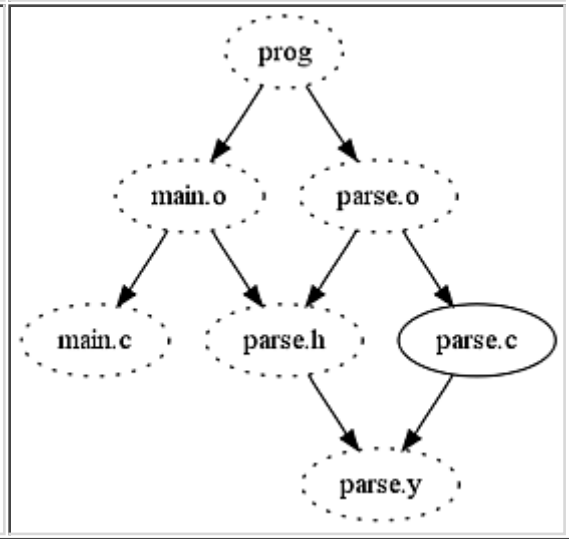
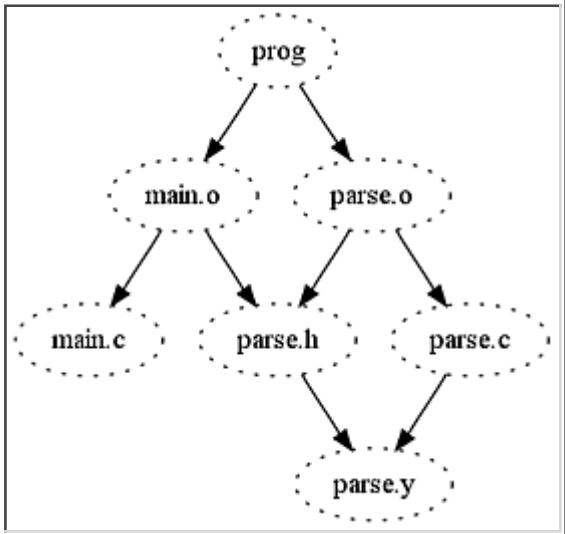


Figure 12: To update parse.o, must visit parse.c (parse.h already visited)



Note that even though main.c was reached by the third iteration, the entire graph must be parsed to find any other incident links on main.c (the modified file). On the face of the problem, it may seem impossible to do better than linear time. After all, how could the build system know that traversing down the entire "parse" side of the tree is unnecessary? Clearly each dependency must be checked to see if the target needs to be updated. The problem is that an Alpha build system tries to answer the wrong question. Another way to look at the update_file() function is as if it provides an answer to the question "Do I need to update this target?" So it starts by asking, "Do I need to update 'prog'?", which can only be answered by asking "Do I need to update main.o?", "Do I need to update parse.o?", and so on through the DAG. There is no way to answer these questions in better than linear time.

The root of the problem is that an Alpha build system has no *a priori* information about what file modifications have taken place. The only way they can find out what has changed is by looking at each file in turn in order to compare timestamps or file signatures. Before looking into alternatives, let us consider *why* a file has changed in the first place. A file changes on the system because the developer performs some action -- for example, updating from source control, saving the file in an editor, creating or deleting a new file in the filesystem, etc. All of these actions could necessitate an update in the build system. However, all of these actions are 1) performed in a userspace program on a file or set of files; and 2) the file modifications end up going through the filesystem layer of the kernel. Let us assume for the time being that either the userspace program or the kernel saves a list of these changes for the build system. The list contains information like "files X

and Y have been modified; file Z was removed".

3.2. Alpha DAG Building Algorithm

Even with this *a priori* information, most Alpha build systems cannot make efficient use of it to perform the first step (constructing the DAG) in less than linear time. The issue here is one of dependency storage -- that is, the representation of the DAG on the disk. For example, in *make* this takes the form of several Makefiles along with separate dependency files that were generated from "gcc -MMD" (historically, *makedepend*) or something similar. The algorithm of reading Makefiles can look something like this:

```
read_makefile(m)
       foreach line in m {
              if(line is include statement) {
                     read_makefile(include file)
              }
       }
```

This ignores the majority of actually parsing the Makefiles for targets and commands -- here we are just considering the act of reading in the Makefiles to look at dependency information. The read_makefile() function will be called with the default (or top-level) Makefile. This will cause *make* to read in all sub-Makefiles and generated dependency files, which is again an O(n) operation. A non-recursive make setup may have this information split up in several user-defined Makefiles, as well as automatically generated dependency (.d) files from gcc. In the parser example, the .d files only have redundant information. In a real-world project they would add information regarding the automatically detected header file dependencies, but that doesn't apply here since the only header file is generated. Here are the build fragments of one possible non-recursive make setup for the parser:

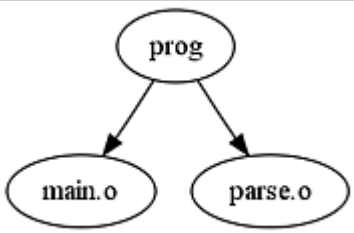
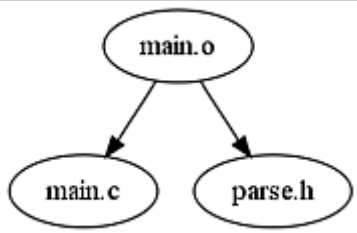
Figure 13: Top-level Makefile with linker information	*Figure 14: "main" Makefile fragment (also main.d)*
	

Figure 15: "parse" Makefile fragment

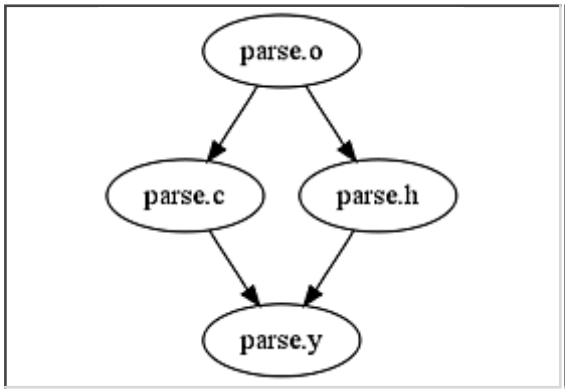
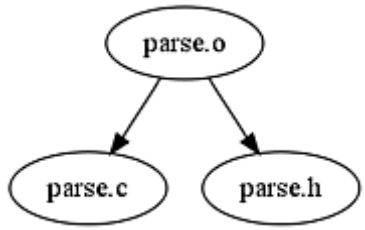


Figure 16: parse.d



These fragments are all loaded to generate the global DAG. Suppose we wanted to use the fact that "main.c has been modified" to try to short-circuit this read/include loop in order to pull in only specific fragments. Unfortunately, *any* of the fragments here may list main.c as a dependency. We may use domain-specific knowledge to conclude that it will be contained in main.d and/or a specific Makefile, but this quickly grows impractical. Consider parse.h - any fragment anywhere in the system could rely on this header. There is no way to know in advance which set of these Makefiles we need to include. Therefore, any Alpha build system that relies on information generated from "gcc -MMD" or similar must necessarily execute in linear time. In order to do better, a build system must have a more well-defined structure for keeping track of dependencies.

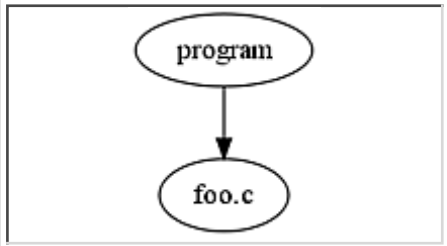
3.3. Alpha Data Persistence

Both the update and DAG building algorithms reveal why Alpha build systems are limited to O(n) scalability, but neither address build correctness. For that we need to look at data persistence. Consider the simple Makefile example from before, where the program name was changed. This Makefile...

```
program: foo.c
	gcc $< -o $@
```

...was successfully parsed and executed, resulting in the system state shown in Figure 17:

Figure 17: System State A



The Makefile is then changed and the build executes again:

```
hello_world: foo.c
	gcc $< -o $@
```

This Makefile also parsed and executed successfully. According to the Makefile, the file system should now be in state *B* (Figure 18). However, in reality the system is in state *C* (Figure 19).

Figure 18: System State B

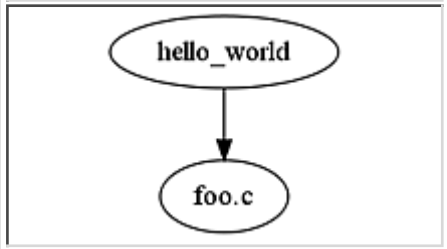
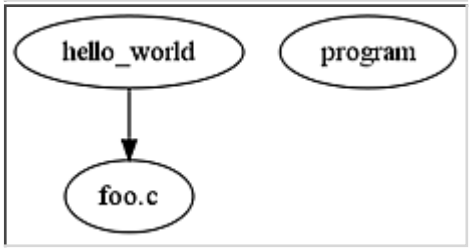


Figure 19: System State C



This is problematic -- the user may run some test cases which try to execute "program" and succeed because the file is still there. Pleased that the test cases pass, the developer checks in the changes. A new developer may come along and check out the project, only to find that the same test fails because "program" is never generated. This scenario is caused by the fact that the build system was supposed to generate system state *B*, but the actual result was state *C*. Ideally, the build system would remove the old target and go to state *B*. With the old file gone, the first developer would find the test case that tries to run "program" fails. This test could then be fixed before checking in the changes.

The root cause of this issue is that the user-generated dependency file (the Makefile) is the only location of the information that "program" was originally generated by the build system. When the user removes that bit of information from the Makefile, the build system therefore has no way of knowing that "program" needs to be removed in order to attain the correct system state. In short, *make* has no data persistence algorithm. In order to maintain correctness, it would need an algorithm like the following:

```
pre_update_file(newDAG, oldDAG)
        foreach non-leaf target in oldDAG that is not in newDAG {
                remove target from filesystem
        }

post_update_file(newDAG)
        save newDAG as oldDAG
```

The pre_update_file() function would be called after *make* parses in the new DAG (represented by Makefiles and auto-generated dependency files), but before it calls update_file() with the default target. This would compare the list of files that will be generated by the new DAG to the ones that were generated from a previous build, and remove those that no longer apply. Then after the build is complete, the new DAG will be saved in post_update_file() so that it can be used to compare against in the next build execution.

Although some Alpha build systems other than *make* may save the old DAG (for example, recent versions of SCons can save the DAG in a dblite database), I have not found any that remove old targets in order to achieve correctness.

4. Beta Build System Algorithms

In this section, we will describe the update and DAG building algorithms that are required for any Beta build system. Some suggestions for handling data persistence in a Beta build system are also given; however, there may be many alternative solutions. The Beta build system algorithm is very similar to an Alpha build system:

1. **Input file change list**: Required *a priori* information for Beta build systems.
2. **Build partial DAG**: Build a partial DAG based on the file change list and the complete (global) disk-based DAG.
3. **Update**: Execute commands necessary to bring all the nodes in the partial DAG up-to-date.

This build system algorithm may seem counter to the conclusions reached in *Recursive Make Considered Harmful*. Specifically, that paper warns that build errors can occur when the build system uses an incomplete DAG. The distinguishing factor in a Beta build system is that the partial DAG built in step 2 is still *correct*. That is, all nodes that must be visited will still be visited, even though much of the on-disk DAG can be ignored for small updates. Now we will look at each of the steps in more detail.

4.1. Beta Input File Change List Algorithm

The Beta build system does not need to do much here -- this merely serves as a placeholder to note that it is required in order to achieve any algorithmic improvements. For example, this could be as simple as reading the contents of a file, or selecting rows from a database with a flag set. The file change list may distinguish between the types of changes, such as "file X was modified, file Y was deleted, and file Z is new". For now, we will just consider the case where a file has been modified. It doesn't matter to the build system how this list is generated -- some possibilities include adding this capability into an IDE, using a filesystem monitor, or providing the information in the filesystem itself. The following algorithm shows a Beta build system inputting the file change list:

```
input_file_change_list(input, change_list)
        foreach file_changed in input {
                add file_changed to change_list
        }
```

This algorithm does a foreach over the number of files changed. That is, it is O(n) where n is the number of changes to the filesystem.

4.2. Beta Partial-DAG Building Algorithm

At this stage, we have the beginning workings of a DAG in memory, and need to finish the partial DAG based on the contents of the complete on-disk DAG. The following algorithm accomplishes this goal:

```
build_partial_DAG(DAG, change_list)
        foreach file_changed in change_list {
                add_node(DAG, file_changed)
        }

add_node(DAG, node)
        add node to DAG
        dependency_list = get_dependencies(node)
        foreach dependency d in dependency_list {
                if d is not in DAG {
                        add_node(DAG, d)
                }
                add link (node -> d) to DAG
        }
```

This algorithm is a bit more difficult to analyze in the general case because it depends on the complexity of the graph. At the very least, it is O(n) with respect to the file change list, since *build_partial_DAG()* has a *foreach* loop over the *change_list* variable. Also, all of the dependencies of a node are loaded into the DAG (and subsequently processed themselves). Therefore, this

algorithm is also O(n) with respect to the size of the partial DAG that is loaded. Finally, the complexity of the *get_dependencies()* function must be taken into account. This function operates on the complete disk-based DAG. Clearly, the node-to-dependency mapping must be stored in a structure that allows quick searching time (such as a tree). An unsorted list would quickly relegate this algorithm to O(n) across the project size.

It is worth noting that the *get_dependencies()* function answers a different question than the algorithms found in Alpha build systems. Specifically, it can answer "What files need to be updated, given that this file has changed?" For example, the file change list may contain "main.c was modified". In *add_node()*, *get_dependencies(main.c)* will return "main.o". In turn when main.o is added to the DAG, *get_dependencies(main.o)* will return "prog". In contrast, Alpha build systems generally have an equivalent kind of function, but instead *get_dependencies(main.o)* would return "main.c, parse.h". Essentially, Beta dependencies are in the opposite direction of Alpha dependencies for purposes of the build and update algorithms (though in practice, both directions are actually useful). Graphically, here is how the construction of the partial DAG looks given the input of main.c:

Figure 20: Disk-based DAG.

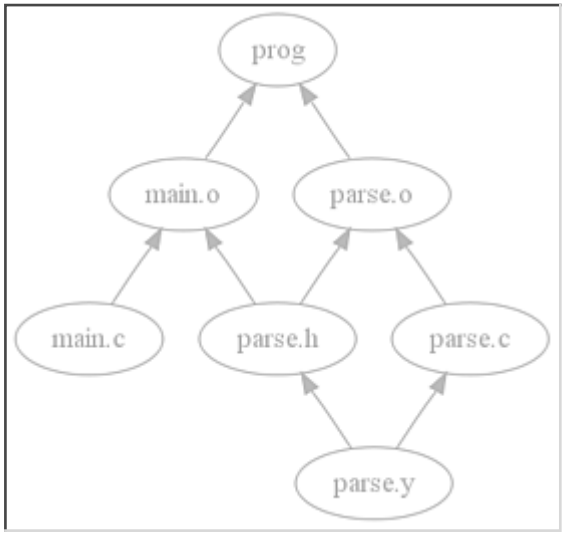


Figure 21: Input file change list.

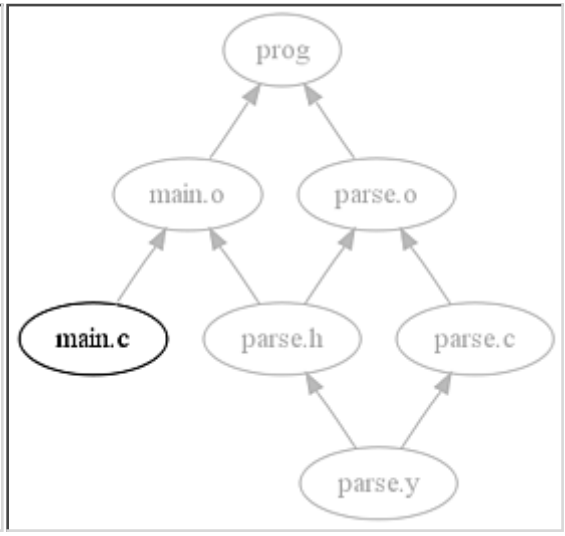
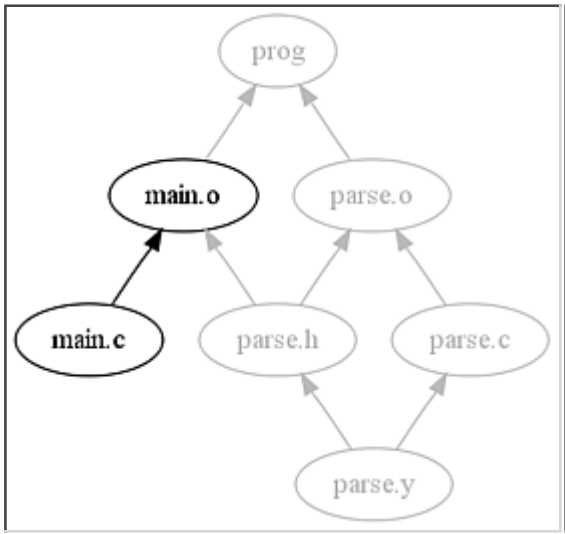
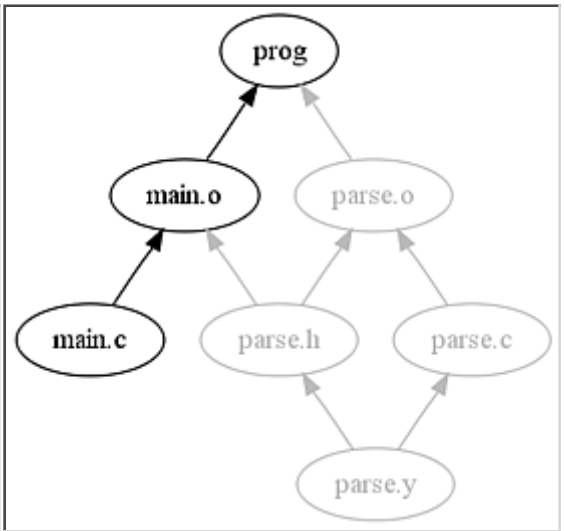


Figure 22: get_dependencies(main.c) returns main.o	*Figure 23: get_dependencies(main.o) returns prog*
	

At this point, the DAG is complete because the last node, "prog", has no outgoing links. The greyed-out nodes corresponding to the on-disk DAG in Figure 23 are never even considered by the build system. This allows the Beta build system to be truly scalable, since the "parse" side of the graph could be arbitrarily large. As long as the "main.c -> main.o -> prog" relationship remains consistent, the only additional time that the Beta partial-DAG building algorithm requires as the project size grows is solely dependent on the search algorithm used by the on-disk DAG storage.

4.3. Beta Update Algorithm

The update algorithm is very simple for a Beta build system. This is because of a few properties of the partial DAG built in the previous step:

- The partial DAG contains *every* file that needs to be updated in the system.
- The partial DAG contains *none* of the files that do *not* need to be updated in the system.

That is, in the set of all correct partial DAGs, the partial DAG formed by the Beta Partial-DAG Building Algorithm is minimal. Note that this is only true if we assume that the on-disk DAG is accurate (i.e. there are no missing or extraneous links). This is a safe assumption to make, since it is fairly trivial to monitor a program's file accesses during execution to find the exact file-level dependencies. As a result of having a minimal DAG, the update algorithm in a Beta build system can merely walk the graph and update every file.

```
update(DAG)
      file_list = topological_sort(DAG)
      foreach file in file_list {
              perform command to update file
      }
```

Using the main.c partial DAG from the previous example, the system can be updated quite easily:

Figure 24: Start at the root of the tree (main.c)	*Figure 25: Process main.c's dependencies (compile main.o)*	*Figure 26: Process main.o's dependencies (link prog)*
		

Topological sort algorithms are linear with respect to the size of the DAG (nodes + edges), and then the foreach loop over the file_list also executes in O(n) time. That is, the update algorithm is linear with respect to the size of the partial DAG.

4.5. Beta Scalability

The algorithms above scale linearly with respect to the input file list and the size of the partial DAG. However, this doesn't provide much insight into how it compares to an Alpha build system that scales linearly with respect to the global DAG. Here we will make a few assumptions about the typical developer behavior and software project complexity in order to derive a reasonable and comparable answer.

First, let's look at the size of the input file list. Generally, most of a developer's time is spent doing small update cycles. The developer would only build the entire project once, and then make small changes to see how it affects the system. If we assume a developer is implementing the behavior of a specific subsystem, or perhaps the interaction of two subsystems, they will only edit a handful of files at a time before re-building. As a point of reference, consider how many editor windows you have open while developing. In some cases the developer may change something that affects all files (renaming some

well-used function, for example), but these types of changes are rare and require a full re-build anyway. For the typical case of editing a few files, we can assume that the input file list is a constant O(1) size with respect to the project size. That is, regardless of how big the project is, the developer is focused on a small part of it.

Now we will consider the size of the partial DAG. In some pathological cases (such as a DAG with one root node that links to all other nodes, or a DAG with a single linear chain), the partial DAG will be the same as the complete DAG. Fortunately, most DAGs used in software development are tree-like structures with a high branching factor. Therefore, one can expect the complexity of the partial DAG to be O(log n) with respect to the size of the entire project.

Finally, if we assume nodes are stored in a tree structure that has O(log n) searching time, then we have the following algorithmic complexity for Beta build systems given a standard update operation:

Algorithm	Complexity
input_file_change_list	O(1)
build_partial_DAG	$O(\log^2 n)$
update	O(log n)

The two log factors in the build_partial_DAG algorithm come from the size of the partial DAG (log n), and the time to execute get_dependencies (also log n), which is done for every node in the partial DAG. These three algorithms represent the three main steps to a Beta build system update -- run sequentially, the total complexity is therefore $O(\log^2 n)$. This is a significant improvement over the Alpha build system update of O(n).

Because the partial DAG is minimal, no build system can use a smaller DAG and still be correct. Also, each node in the partial DAG must be visited in order to build the graph; therefore, the natural lower bound on building the partial DAG is O(n). As has been shown, both the Beta DAG-building and update algorithms are O(n) with respect to the partial DAG. Therefore, Beta build system updates are asymptotically optimal among all build systems that use file-based dependencies.

4.6. Beta Data Persistence

One of the failings with some Alpha build systems is that there was no DAG that persisted across build invocations. For Beta build systems, it is necessary to have a specific DAG structure with a good search algorithm in order to satisfy the constraints imposed by the update procedures. This DAG structure

must be separate from the "userspace" DAG elements that a developer designs in the build-system (for example, the notion that a .c file becomes a .o file is written by the developer, but the .h file that is included may be added to the DAG automatically when the file is compiled). This requirement of DAG persistence helps, but by no means solves the correctness rule.

So far, this paper has assumed that the on-disk DAG has already been constructed. I have no proof for the optimal on-disk DAG construction algorithm, so this is left as an exercise for the reader. Instead, I will offer some considerations here, and then describe an actual implementation of a Beta build system so one can see how data persistence can be used to achieve correctness.

Imagine a large project with many directories and files. One of these directories contains main.c, and another directory has parse.y. Ideally, we would have some build fragment (such as a per-directory Makefile) that describes how to build these files. Other (perhaps unrelated) modules have their own build fragments. Now suppose that main.c is getting too large, and we wish to break part of it off as a separate utils.c file in the same directory. This has some repercussions on the immediate directory, since we need to add a new node to the DAG (utils.o). It also has an effect on the fragment used to build parse.y and link the final executable. It would clearly be sub-optimal to have to re-parse every build fragment in the project, since many of them are unrelated to this particular module. We would prefer to only parse the two build fragments that are necessary.

Assuming there is an easy way to parse only the necessary build fragments, it should be trivial to compare the list of output files for a particular fragment against a previous build. For example, the parse build fragment outputs parse.h, parse.c, parse.o, and prog. If the executable name is changed from "prog" to "hello_world", then the new list of outputs is parse.h, parse.c, parse.o, and hello_world. The build system can see that prog was in the old list but not in the new list, and therefore remove it from the filesystem. Assuming a logarithmic directory structure, this means file deletion and renaming can be handled in logarithmic time.

4.7. An Example Beta Build System: tup

Tup is a build system that uses the Beta update algorithm to perform quick updates, and is also able to delete stale target files from the filesystem when necessary. I believe it is a build system that actually satisfies all three rules given in this paper (and where it doesn't, there is probably a bug).

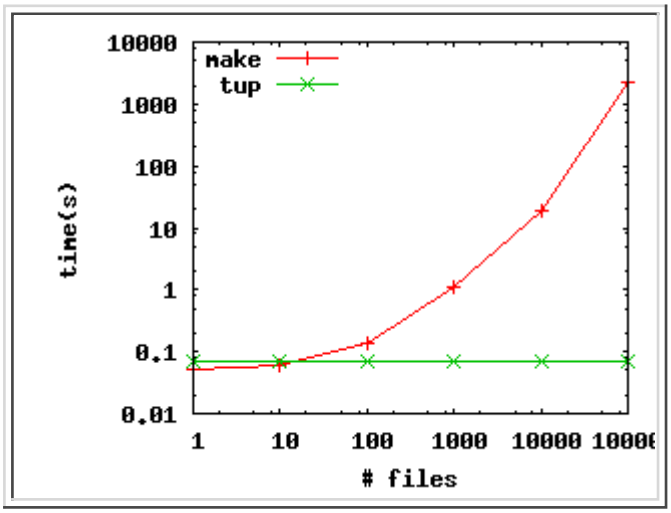
Tup stores its disk-based DAG in an SQLite database. The nodes are indexed so

that they can be retrieved in O(log n) time (this satisfies the searching requirement imposed by the *get_dependencies()* algorithm). In addition to the standard file-based nodes, tup also has a node in the DAG for each command used to generate those files. This is done for a few reasons -- first, it allows easier handling of commands that produce multiple outputs (such as a parse.y file which ends up creating a parse.c and parse.h file). Second, it is easy to correctly re-issue commands that have changed (such as by adding options to a gcc command-line). Finally, when a command node is removed from the DAG, the outgoing links point directly to the files that need to be deleted in order to satisfy the correctness rule.

Aside from the Beta update algorithm, I do not believe tup's design choices described here are necessarily ideal -- there may be better alternatives. However, it does use per-directory build fragments. As mentioned previously, this allows tup to handle file additions and deletions (and by extension, re-naming) in logarithmic time. This can serve as a baseline to measure other possible implementations of Beta build systems. Tup also shows that satisfying all three rules is possible in practice.

For an example of tup's scalability, several test projects were constructed with sizes of 1, 10, 100, 1000, 10000, and 100,000 C files. These projects were built separately with *make* and with tup. When a single C file is changed in the 100,000 file case, *make* can take over half an hour to figure out which file needs to be compiled and linked. In contrast, tup takes a near-constant 0.07s to cherry-pick the correct C file.

Figure 27: Make vs. tup single C file build times



Re-visiting the simple data persistence example from earlier, one can see that tup is able to automatically remove a stale file when a target name is changed. This results in the correct system state without having to manually do a clean

build or fresh checkout:

```
$ cat Tupfile
: foo.c |> gcc %f -o %o |> program

$ ls
foo.c  program  Tupfile

$ vi Tupfile
(change "program" to "hello_world")

$ cat Tupfile
: foo.c |> gcc %f -o %o |> hello_world

$ tup upd
Parsing Tupfiles
[ ] 0/1 (  0%) 1: .
[=] 1/1 (100%)
Deleting 1 file
[ ] 0/1 (  0%) 15: program
[=] 1/1 (100%)
Executing Commands
[ ] 0/1 (  0%) 21: gcc foo.c -o hello_world
[=] 1/1 (100%)

$ ls
foo.c  hello_world  Tupfile
```

More details specifically about tup and its design choices can be found at http://gittup.org/tup/. There is more in-depth information about the performance comparison against the non-recursive make implementation, as well as a comparison to an oracle machine implementation. These comparisons are included in the source tree if you wish to perform your own tests.

5. Summary

This paper has presented a set of rules that can be used to measure build systems with regard to scalability, correctness, and usability. It has shown how the classical ("Alpha") build systems of today do not satisfy any of these rules, and scale in O(n) time. New algorithms have been proposed that any future ("Beta") build system must use. These algorithms scale in $O(\log^2 n)$ time, which satisfies the first rule (scalability). Suggestions have also been provided for addressing the second rule (correctness) particularly in regard to deleting or renaming files. The third rule (usability) can therefore be satisfied by ensuring that the developer does not have to perform special-case commands in order to see results quickly, or obtain a correct build. Specifically, the following key factors must be addressed by the build system:

- Contrary to *Recursive Make Considered Harmful*, the update algorithm cannot rely on a global DAG and instead must use a partial DAG.
- A file change list must be maintained and provided to the build system update operation.
- Secondary storage for the graph must be used that is separate from the user-defined build input.
- File/directory deletion and renaming must be treated as first class operations.

Even with all of these factors in place, there is still much to consider when designing a build system. The update algorithm presented here is but one piece of the puzzle. The designer must also take into account issues such as new file creation, individual build fragment parsing, and build configuration. I believe each of these operations can be done in logarithmic time.

With a truly scalable and correct build system, developers can utilize short Edit-Compile-Test cycles to rapidly create new solutions and algorithms, investigate issues, or restructure the source tree. Eventually, we may be able to leave the issues of Alpha build systems in the past and focus on our actual software projects, no matter how large they may become.

6. References

[1] Miller, P.A. (1998), *Recursive Make Considered Harmful*, AUUGN Journal of AUUG Inc., 19(1), pp. 14-25.