RC23925 (W0603-200) March 31, 2006
Computer Science

IBM Research Report

Grexmk: Speeding Up Scripted Builds

Glenn Ammons
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Grexmk: Speeding Up Scripted Builds

Glenn Ammons
IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, New York, USA
ammons@us.ibm.com

ABSTRACT

Slow builds can be extremely costly; for example, one of our customers loses forty percent of their developers' productivity to waiting on builds. Build avoidance—building only what must be built to integrate a change—solves the problem but makes the build unreliable unless the build script captures all dependences of build outputs on sources. Therefore, in practice, integrating even a minor change often requires rebuilding a large software system from scratch.

Our solution can be summarized as "trust but verify". Exploiting the fact that large software systems tend to be composed of loosely coupled parts, we semi-automatically divide a large, all-or-nothing build into many small "mini-builds", without changing the original build. Each mini-build declares its dependences, which are trusted while scheduling mini-builds to integrate a change but verified by executing each mini-build in a sandbox in which only its declared dependences are available.

We implemented these ideas in a tool suite called Grexmk and applied Grexmk to two large all-or-nothing builds. On the first build, we demonstrated linear speedups to six build machines. On the second build, we sped up the average time for an incremental build by a factor of 1.2.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering*; D.3.2 [Programming Languages]: Language Classifications—*Make, specialized application languages*

General Terms: Languages, Performance, Reliability, Verification

Keywords: build avoidance, dynamic analysis, sandboxing

1. INTRODUCTION

Slow builds can be extremely costly; for example, one of our customers loses forty percent of their developers' productivity to waiting on builds. Build avoidance—building only what must be built to integrate a change, as in Make [5]—solves the problem but makes the build unreliable unless the build script captures all dependences of output files on source files. Therefore, in practice, integrating even a minor change often requires rebuilding a large software application from scratch.

Slow builds hurt projects in ways beyond merely forcing developers to wait. On many projects developers cannot test their own changes because the build is too slow for them to run it themselves. Slow builds also limit the usefulness of testing and debugging tools, which cannot run without a known good build.

Nonetheless, a large software project's build, no matter how slow it is, is too important to break, because developers, testers, and even software architects depend on access to up-to-date builds. As a result, development groups fear making large changes to their builds.

This paper describes Grexmk, a tool suite for speeding up scripted, all-or-nothing builds. Grexmk contains tools for dividing large builds into *mini-builds* and tools for executing mini-builds in parallel and incrementally. Importantly, using Grexmk requires neither abandoning the original build nor changing it significantly.

A mini-build is an all-or-nothing build that explicitly lists its output files, its source files, its dependences on other mini-builds, and its build script. Developers either define mini-builds by hand or use Grexmk's tools. The tools exploit the fact that large build scripts tend to be composed of loosely coupled parts. For example, scripts for IBM's Wsbld build tool, which is an extension of Ant [1], explicitly declare components; for these scripts, a developer could use a tool that automatically translates components into mini-builds. If the build script were less transparent, the developer could use another tool, which generates mini-builds by dynamically analyzing a trace of the original build's file accesses.

Grexmk executes each mini-build in a restricted environment in which only the mini-build's declared dependences are available. The slogan is "trust but verify"—if a set of mini-builds cannot be safely executed incrementally or in parallel, a dynamic analysis ensures that the sandboxed build fails instead of producing an unreliable result.

This paper makes these contributions:

- A novel method for executing all-or-nothing builds incrementally and/or in parallel. We do not require significant changes to the original build, which makes our method easier for development groups to adopt.
- A novel method for semi-automatically dividing an all-or-nothing build into mini-builds, by dynamically analyzing a trace of the original build's file accesses.

- Demonstrations that two large, all-or-nothing builds are composed of loosely coupled parts, which can be separated easily into mini-builds.

The outline of this paper is as follows. Section 2 reviews Make-based builds and motivates our approach to translating all-or-nothing builds into builds that can execute incrementally or in parallel. Section 3 defines mini-builds and explains how Grexmk executes them. Section 4 describes two techniques for extracting mini-builds from an all-or-nothing build. Section 5 evaluates Grexmk on two large, all-or-nothing builds. Section 6 reviews related work.

2. BACKGROUND

The goal of this work is to translate all-or-nothing builds into mini-builds that can execute incrementally or in parallel. This section explains what it means for a build to be able to execute incrementally or in parallel, by reference to "safe" Make-based builds as formalized by Niels Jørgensen [6]. To motivate our work, we use Jørgensen's insights into when incremental builds are safe to explain why all-or-nothing builds are so prevalent.

Jørgensen's main result is a theorem that says, in part,

> If all rules in a well-formed Makefile are complete, fair, and sound, then the following kinds of changes are safely integrated by incremental builds:
>
> - Removing a target file.
> - Modifying a source file.

We need informal definitions of the key ideas in this theorem. In Jørgensen's paper, a *rule* is of the form $Ts : Ds; C$, where Ts is a list of targets (usually build outputs) derived by the rule, Ds is a list of files on which the rule depends, and C is a build command. In Make, only the state of the filesystem matters, so Jørgensen regards build commands as black boxes that alter the filesystem.

A *well-formed Makefile* is a set of rules in which no two rules share a target and there are no cycles of dependent rules. A rule $Ts : Ds; C$ is *complete* iff every target or source file that is read by C is in Ds, the rule is *fair* iff C updates no target that is not in Ts, and the rule is *sound* iff C updates every target in Ts.

Jørgensen's full theorem also characterizes changes to the Makefile that can be safely integrated, but such changes are not important in this paper. Finally, we assume in this paper that any parallel build is equivalent to some valid sequential build (that is, sequential consistency). With this assumption, Jørgensen's results apply to parallel builds, too.

2.1 How all-or-nothing builds arise

We assert that all-or-nothing builds occur whenever the equivalent of a Makefile that meets Jørgensen's conditions is missing. This happens for many reasons. For example, large Make-based builds often divide the build into more manageable parts by invoking Make recursively. As Peter Miller explains [10], this practice leads to incomplete rules, among other problems.

The build problem is acute for Java projects and, especially, for complex Java projects with builds based on the Ant [1] tool. Slow builds of this sort motivated this work and, so far, are the only builds to which we have applied Grexmk. The rest of this section examines these builds in detail.

2.1.1 Building Java programs

Make is rarely used to build Java programs. One reason is performance. Initializing a Java virtual machine (JVM) is time-consuming and a straightforward Makefile would start a JVM for every rule that invokes the Java compiler. This problem can be avoided (see Section 3.2).

A more important reason is correctness. Writing correct Makefiles for Java programs is difficult without an implementation of Make that supports multiple-target rules and few implementations support such rules [1].

Makefiles for Java programs require multiple target rules for two reasons. First, compiling a Java source file may produce more than one class file; all of these class files must be targets of the rule whose build command compiles the source file, or else the rule is unfair. Second, class files can depend on other class files, and the dependence graph over class files can contain cycles. All classes involved in a cycle must be build outputs of the same rule because well-formed Makefiles do not contain cycles.

2.1.2 Ant-based builds

The preceding section argued that typical Make implementations are unsuitable for building Java programs. For small and simple Java projects, the Java compiler's built-in dependence analysis suffices. However, many projects have more complicated requirements, including building archives, running other compilers, predeploying web applications to an application server, and running regression tests.

The most popular build tool for these projects is Ant [1]. Ant buildfiles define "targets" instead of rules. Each target has a name, a list of targets on which it depends, and a list of "tasks", which correspond to build commands in Make. In Ant, targets are *not* associated with files. Instead, Ant targets are similar to "phony targets" in Make: when Ant is invoked with a list of targets to execute, Ant executes each target reachable from the targets in the list in reverse topological order. Consider this example from the Ant manual:

```
<target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="B"/>
<target name="D" depends="C,B,A"/>
```

When the user tells Ant to run target `D`, Ant executes `A`, `B`, `C`, and `D`, in that order, independent of the filesystem's state.

In general, Ant builds cannot run incrementally or in parallel. Ant does define tasks that support incrementality in an ad-hoc fashion. For example, Ant relies on the Java compiler to rebuild class files only when necessary. It is difficult to tell when a buildfile constructed from such ad-hoc tasks permits incremental rebuilds.

2.2 Summary

This section presented the requirements on build systems that aim to build programs incrementally or in parallel and argued that the usual solution, Make, is unsuitable for building Java programs, especially with tools based on Ant. The

[1] An exception is Clearmake [11]. GNU Make [12] supports multiple targets only in pattern rules; as in most Make implementations, other "multiple target" rules are merely shorthand for several single-target rules.

```
#!/bin/sh

echo -n scoo > scoo.txt
echo -n doo > doo.txt
echo -n by > by.txt
cat scoo.txt by.txt > scooby.txt
cat doo.txt by.txt > dooby.txt
cat scooby.txt dooby.txt doo.txt > scoobydoobydoo.txt
```

Figure 1: A silly Bourne shell build script.

```
control:
    Mini-Build:  scooby
    Build-Depends:  scoo, by
    Build-Script:
    ␣#!/bin/sh
    ␣cat scoo.txt by.txt > scooby.txt
sources:
outputs:
    scooby.txt
```

Figure 2: A mini-build specification for the command that creates `scooby.txt` in Figure 1.

next section defines mini-builds, which are our approach to bridging the gap between scripted builds, including Ant builds, and Make.

3. MINI-BUILDS AND MINI-BUILD EXECUTION

This section defines mini-builds and explains how to execute them by translating them into Makefiles.

3.1 Mini-builds

A Grexmk build is defined by a set of *mini-build specifications*. A mini-build specification is a directory that contains three files:

`control` The name, dependences, and build script of the mini-build, formatted to follow this template:

> `Mini-Build:` *mini-build-name*
> `Build-Depends:` *mini-build-name*`*`
> `Build-Script:`
> ␣*line0*
> ...
> ␣*lineN*

`outputs` A list of the files that the mini-build exports to other mini-builds, with one file per line.

`sources` A list of the mini-build's source files.

By convention, a `grexmk` directory at the root of the program's source tree lists all mini-build specifications.

As an example, consider the Bourne shell [7] build script in Figure 1. If one were to create a mini-build for each command in the script, then the mini-build specification for the command that creates `scooby.txt` might be as indicated in Figure 2. Note that the mini-build has no sources.

Also note that the mini-build's build script is almost exactly the same as the original command in Figure 1. In general, when creating mini-builds for an all-or-nothing build, it is important to minimize changes to the original build. Section 4 discusses mini-build creation in detail.

Finally, Jørgensen's results about incremental Make carry over to mini-builds. Each mini-build specification corresponds to a rule. The specification's sources and build-dependences determine the dependences of the rule, the specification's outputs are the targets of the rule, and the specification's build script is the rule's build command. A set of mini-build specifications is well-formed if the corresponding Makefile is well-formed. Thus, Jørgensen's main result can be restated for mini-builds this way:

> If all mini-build specifications that make up a well-formed build are complete, fair, and sound, then the following kinds of changes are safely integrated by incremental builds:
>
> - Removing an output file.
> - Modifying a source file.

3.2 Mini-build execution

Grexmk executes mini-builds by translating them into Makefiles and building them with Make. Like other well-constructed Make-based builds, the builds that Grexmk generates can be executed incrementally and in parallel. This paper makes no contributions to the technology underlying incremental and parallel execution of Make-based builds.

Our novel contribution is the use of sandboxing to detect or avoid unsafe builds dynamically. Each mini-build runs in a restricted environment (the sandbox) in which only the source and output files of its dependences are available initially, instead of running in the master build tree as is the traditional practice. When the build script completes, only its output files are propagated to the build tree.

Sandboxing detects incomplete mini-builds because build scripts with undeclared dependences will fail. Sandboxing eliminates unfair mini-builds because only declared outputs propagate to other mini-builds and to the build tree. Finally, sandboxing detects unsound mini-builds because, if all mini-build specifications are well-formed (a static property), then any declared output that exists after a build script completes must have been created by the script; if an output does not exist, that fact is detected when Grexmk attempts to propagate it to the build tree.

We have experimented with two methods of sandbox construction. The first method relies on a central repository of source and output packages, implemented using the Debian Linux distribution's package management tools [3]. For each mini-build (in fact, each version of each mini-build specification and its source files), the repository holds a source package and, if the mini-build has executed, an output package.

To execute a mini-build on a build machine, Grexmk retrieves the mini-build's source package from the repository and unpacks it in an empty, local directory (the sandbox); retrieves the output packages of each build dependence from the repository and unpacks them in the sandbox; executes the build script; if the build script succeeds, creates a new output package and uploads it to the repository; and destroys the sandbox.

To execute a parallel build, Grexmk generates a Makefile

that submits mini-builds to the OpenPBS batch job scheduler [15], which runs them on a cluster of build machines.

The repository is a bottleneck with this method, which is mitigated by optimizations that are implemented by the Debian tools. In particular, each build machine caches packages that have already been retrieved from the repository.

In the second method, sandboxes are populated by creating symbolic links to the master build tree instead of by unpacking packages. This method requires that the master build tree be accessible to each build machine, presumably via a network file system. We used this method on a customer's build; because of licensing restrictions on OpenPBS, we do not evaluate the performance of this method on parallell builds in Section 5.

Finally, Section 2 remarked that a straightforward Makefile for Java would start a JVM for every rule that invokes the Java compiler, which would be slow. With both sandboxing methods, our implementation uses Mark Lindner's Jolt [9] to avoid that overhead. Jolt is a JVM daemon that allows multiple build scripts to reuse a JVM.

4. CREATING MINI-BUILDS

This section describes Grexmk's tools for dividing all-or-nothing builds into mini-builds. Sometimes, a build's specification permits a direct translation into mini-builds; Grexmk includes a tool that translates specifications for Wsbld (IBM's Ant-based build tool) into mini-builds. To handle more difficult builds, Grexmk uses a dynamic analysis to create mini-builds from a user-written specification and a trace of a build's file accesses.

4.1 Creating mini-builds from another specification

Many builds are composed of smaller parts ("projects", "components", or "modules"), where the build lists specifications of the parts and the supposed dependences between them. Sometimes, these specifications can be translated automatically into mini-build specifications.

For example, builds that use IBM's Wsbld build tool are made up of components. A component is defined by two XML files: `component.xml` describes how the component fits into the overall build, including a list of its outputs and dependences on other components; `build.xml` contains the component's build script, in a format that extends the format of Ant buildfiles.

Grexmk includes a simple program that translates Wsbld component specifications into mini-builds. The mini-builds are faithful to the original component specifications; in fact, each mini-build's build script invokes the build commands in the corresponding `build.xml` file. An important consequence is that the Grexmk build is faithful to the original build, except that the Grexmk build uses sandboxes to detect errors that prevent incremental and/or parallel builds. Section 5 presents our experience in building a large Wsbld-based build with Grexmk.

4.2 Creating mini-builds from a trace

This section describes a Grexmk tool, DivideTrace, for creating mini-builds when the original build lacks an adequate specification of its parts. DivideTrace is a dynamic analysis that takes two inputs: a trace and a user-written *division specification*.

The first input is a trace of one sequential execution of an all-or-nothing build. One trace is adequate because, by design, builds are repeatable. The trace interleaves a trace of the build's commands with a trace of their interactions with the file system. That is, we assume that all of the effects that matter are recorded in the file system, which is a reasonable assumption for most builds.

Most build tools can be configured to print a trace of the commands that they execute. If not, the build can be annotated to print the appropriate output. For example, `echo` statements could be inserted before and after each command in the build in Figure 1:

```
...
echo "Building scooby.txt"
cat scoo.txt by.txt > scooby.txt
echo "Done building scooby.txt"
...
```

Grexmk uses the Strace [13] system-call tracer to capture the interactions of build commands with the file system. The output of Strace is reduced to file accesses (namely, reads and writes) and interleaved with the build command trace to form a single trace. For example, here is part of the trace generated by the build in Figure 1:

```
...
Building scooby.txt
read(scoo.txt)
read(by.txt)
write(scooby.txt)
Done building scooby.txt
...
```

The second input to DivideTrace is a user-written division specification. The division specification tells DivideTrace how to divide a trace of the entire execution into a sequence of traces of mini-build "executions". We require this input from the user because

1. Automatically inferring mini-build scripts is impossible when build scripts are programs in a general-purpose programming language such as Ant.

2. Mini-build scripts must be simple, because the original build is too important to break and because developers will not be confident that complicated scripts are correct. Often, with an appropriate division, the mini-build scripts are just "calls" into the original build script; by contrast, a bad division might require rewriting portions of the original script.

3. An automatic division must be overly conservative because, in principle, values from one interval of the trace could affect a following interval without going through the file system.

Figure 3 shows pseudocode for a division specification of the build in Figure 1 [2]. The specification contains two declarations. The first declaration instructs DivideTrace to ignore all file accesses from the beginning of the trace up to the first line that contains `"Building"`. The second declaration tells

[2]In our implementation, there is no concrete syntax for division specifications; instead, the developer writes Perl [16] programs that construct abstract syntax trees and invoke the DivideTrace evaluator.

```
ignore:
    start-line:  0
    stop-regexp:  /Building/
multibracketed:
    name:  start.$1
    start-regexp:  /Building (.*)/
    stop-regexp:  /Done building/
    export-regexp:  /scoobydoobydoo.txt/
```

Figure 3: Division specification for Figure 1's build.

DivideTrace to create a mini-build for each trace interval that begins with a line that matches the regular expression `"Building (.*)"` and ends just before a line that contains `"Done building"`. The name of the mini-build will be the substring of the first line that matched `(.*)` in the regular expression.

In general, a specification is a list of declarations. Each declaration is of one of four kinds: a *bracketed* declaration matches at most one trace interval, where a match corresponds to one mini-build; a *multi-bracketed* declaration matches zero, one, or more intervals, where each match corresponds to a mini-build; an *ignore* declaration matches intervals that ought to be ignored; and a *prebuilt* declaration matches no intervals and corresponds to a mini-build with a trivial build script.

Each declaration has a list of properties (that is, name-value pairs). Table 1 lists the most commonly used properties, their meanings, and the declarations in which each property is allowed.

Given a specification and a trace, DivideTrace evaluates the specification on the trace as follows:

Divide the trace In one pass, DivideTrace splits the trace into disjoint intervals and assigns each file access to its enclosing interval. Accesses that belong to intervals that match ignored declarations are discarded. Every interval is matched to exactly one declaration. Intervals that match bracketed and multi-bracketed declarations are assigned names by evaluating the value of their `name` property; the expression can inspect the text of the interval, in order to construct different names for multi-bracketed declarations.

DivideTrace reports an error if the trace cannot be divided unambiguously into intervals; if intervals cannot be matched unambiguously to declarations; if any file access lies outside of all intervals; or if names collide between two intervals, between two prebuilt declarations, or between an interval and a prebuilt declaration.

Assign files Every accessed file is assigned either to an interval or to a prebuilt declaration.

A file F belongs to a prebuilt declaration P iff F is in the `outputs` list of P.

Otherwise, F belongs to an interval I iff F is written only in I and not read before it is written, or F is never written and read only in I.

DivideTrace reports an error if a file cannot be assigned, if a file could be assigned to more than one interval or prebuilt declaration, or if a file that belongs to a prebuilt declaration is written.

Categorize files Every assigned file is categorized as a prebuilt file, a source file, an output file, or a temporary file.

A file F is a prebuilt file iff F belongs to a prebuilt declaration.

Otherwise:

- F is a source file iff F is never written.
- F is an output file iff F is written within its interval; and F is read after its interval, or F matches the `output-regexp` of its interval's declaration.
- Otherwise, F is a temporary file.

Note that every assigned file is categorized. DivideTrace reports an error if any interval lacks output files or if any prebuilt declaration lacks prebuilt files.

Stop or create mini-builds If any errors were reported, DivideTrace stops. Otherwise, DivideTrace creates a mini-build specification for each interval and prebuilt declaration, with these specification files:

`control` The `Mini-Build` field is the name of the interval or prebuilt declaration.

For prebuilt declarations, the `Build-Depends` field is empty. For an interval I, the name `N` is added to I's `Build-Depends` field iff I reads a file that belongs to J and J is named `N`.

If an interval's declaration has a `build-script` property, the `Build-Script` field is the result of evaluating the property's expression; as with `name` properties, the expression can inspect the text of the interval.

`outputs` For intervals, all output files; for prebuilt declarations, all prebuilt files.

`sources` For intervals, all source files; for prebuilt declarations, all prebuilt files.

The resulting mini-builds are not guaranteed to be equivalent to the original build. However, Grexmk guarantees (via sandboxing and Jørgensen's results for Make) that all executions of the mini-builds (incremental, parallel, or sequential) are equivalent. That is, the developer only needs to satisfy himself that *some* dependence-respecting execution of all mini-builds is correct.

5. EVALUATION

This section evaluates Grexmk on two large, all-or-nothing builds. Most experiments were run on a machine with 1GB of RAM and a single 1.66GHz Intel Pentium 4 processor, running Debian Linux. Parallel builds were run on up to six machines, where three machines were "slow" and three were "fast". Fast machines have 1.66GHz Intel Pentium 4 processors, between 1GB and 2GB of RAM, and a fast connection to the repository (11MB/s over http, 0.2ms pings); slow machines have 0.9GHz to 1.66GHz Intel Pentium 4 processors, between 250MB and 1GB of RAM, and a slow connection to the repository (3.5MB/s over http, 10ms pings).

Property	Meaning	Allowed in
`start-line N`	Begin an interval at line number `N` of the trace.	`bracketed, multi-bracketed, ignore`
`stop-line N`	End a interval at line number `N - 1` of the trace.	`bracketed, multi-bracketed, ignore`
`start-regexp R`	Begin each interval at every line that matches R.	`bracketed, multi-bracketed, ignore`
`stop-regexp R`	If a interval has begun for this declaration, end the interval just before the next line that matches R.	`bracketed, multi-bracketed, ignore`
`name E`	For each interval, E evaluates to the name of the corresponding mini-build.	`bracketed, multi-bracketed, prebuilt`
`build-script E`	For each interval, E evaluates to the build script of the corresponding mini-build.	`bracketed, multi-bracketed`
`output-regexp R`	For each interval, files in the interval that match R are outputs of the corresponding mini-build.	`bracketed, multi-bracketed`
`outputs FS`	The files FS are outputs of the prebuilt mini-build.	`prebuilt`

Table 1: Properties of division-specification declarations.

Build machines	Time (minutes)	Speedup
No Grexmk, one fast	124	1.0
One fast	167	.75
Three fast	59	2.1
Three slow	83	1.5
All six	37	3.4

Table 2: Execution times and speedups for building the IBM product in parallel with Grexmk. The first paragraph of this section describes "fast" and "slow" machines.

Build machines	Machine-minutes		
	Build	Validation	Overhead
One fast	124	13	29
Three fast	129	20	27
Three slow	167	45	37
All six	145	31	44

Table 3: Machine times for Grexmk builds, divided into build, validation, and overhead time.

5.1 IBM product build

This build is a subset of a Wsbld-based IBM product build [3]. The subset consists of over 1GB of source divided among 510 components and the original, all-or-nothing build completes in 124 minutes. There are two kinds of components in the build: binary components are precompiled prerequisites with trivial build scripts, while source components contain the product's source code and have non-trivial build scripts. There are 140 binary components, whose "source" size totals 691MB; the remaining 352MB of source is divided among 370 source components.

To apply Grexmk, we defined one mini-build to hold all of the Wsbld component specifications and automatically translated each Wsbld component specification into a mini-build specification, for a total of 511 mini-builds. By executing the mini-builds in a sandbox, we found one error in the original build: one Wsbld specification was missing a dependence. Note that we fixed the error before running the experiments described here.

Table 2 shows execution times and speedups for the original build and for four Grexmk builds that used the package-based execution method of Section 3.2. Grexmk's sequential overhead is 25%, but Grexmk scales well: the speedup from one fast machine to three fast machines is $2.1/.75 = 2.8$, or almost linear; also, the speedup on all six machines is 3.4, compared to a linear speedup of $2.1 + 1.5 = 3.6$.

Table 3 divides the machine-time (in machine-minutes) spent in each Grexmk configuration into build, validation, and overhead time. Build time is the machine-time spent running Wsbld. Validation time is the machine-time spent setting up sandboxes; validation time could be eliminated at the expense of safety. Finally, overhead time is the machine-time spent scheduling jobs, transferring files over the network, and so forth. The data shows that building parallelizes almost perfectly, while validation and overhead rise only slightly as the number of build machines increases; for example, if validation and overhead parallelized perfectly, their combined machine-time on all six machines would be 65 minutes, while the actual time was 75 minutes.

[3]The full build runs only on Windows, but Grexmk does not run on Windows

5.2 Customer build

This is a customer's build, scripted with Ant. The original, all-or-nothing build completes in 11.7 minutes. Although this build is significantly faster than the IBM build, the customer executes thousands of builds per week and estimates that developers spend forty percent of their time waiting on builds. Like the IBM build, the customer's build is divided between binary prerequisites and source code: the build has 1445 files of binary prerequisites, whose size totals 191MB, and 29893 files of source code, whose size totals 429MB.

We used DivideTrace to divide this build into 414 mini-builds. Our division specification was 384 lines of Perl code, including whitespace. Many of these lines were devoted to lists of directories in the original build: we listed 57 directories of precompiled prerequisites and 119 lines of other projects, one per line, for a total of 176 lines.

Table 4 lists execution times for several variations of the customer's build. All builds were run on one "fast" machine, with sandboxes built with symbolic links, as described in Section 3.2. For incremental builds, a list of forty files was randomly generated and the build was repeated forty-one times, changing one file from the list between each build. This estimates the time to integrate a developer's change, assuming that all source files are equally likely to change.

The original build is well-tuned and runs almost twice as fast as a full build with Grexmk, whose overhead is mostly

Kind	Time (minutes)
Original, full	11.7
Grexmk, full	21.4
Grexmk, full, no sandbox	15.8
Incremental	$m = 9.5$, $s = 5.72$
Incremental, no sandbox	$m = 4.2$, $s = .97$

Table 4: Execution times for the customer's build in various setups. For incremental builds, the table reports the mean and standard deviation over forty-one runs (see the text).

due to sandboxing. On average, Grexmk achieves a speedup of 1.2 on incremental builds. One problem is that the end of the build gathers output files from many mini-builds into one large archive. This step takes about four minutes in the original build; if this bottleneck is ignored, incremental builds achieve a speedup of about 1.4; more importantly, optimizing the bottleneck increases the build's parallelism. Note that the customer's product is installed by unpacking this archive.

6. RELATED WORK

Make [5] was the first tool to speed up builds via build avoidance. Make requires a *Makefile*, which declares which output files depend on which source files and describes how to build output files from source files. Essentially, Section 4 describes two techniques for generating Makefiles from all-or-nothing, scripted builds.

Merijn de Jonge has proposed another technique [2] for dividing large builds into components. However, his technique assumes that every component corresponds to a subtree of the original source tree and only works if the original build script is written in a special language. In our experience, customers are reluctant to rewrite their builds to attain speedups. In particular, de Jonge's technique does not apply to the builds discussed in Section 5.

Grexmk detects missing dependence declarations by running each mini-build's script in a sandbox, in which only the declared dependences are available. Electric Cloud [4], a Make variant for executing builds in parallel, and ClearMake, a Make variant included with the ClearCase [11] configuration manager, also detect missing dependences. While Grexmk's sandboxes are simply directory trees in a normal file system, Electric Cloud and ClearMake use special file systems to implement dependence-detection. Another difference is that missing declarations result in a failed build in Grexmk, while ClearMake and Electric Cloud can repair the build by executing (or reexecuting) build scripts in an order that obeys true dependences.

Our approach assumes that the original build is difficult to analyze statically. By contrast, smart recompilation [14] analyzes the program statically to reduce the amount of code that must be recompiled to integrate a change. Giovanni Lagorio's doctoral dissertation [8] explains some of the issues in applying this sort of approach to Java.

In the experiments in Section 5, the original build scripts were written in Ant [1] or a derivative of Ant. Grexmk also applies to build scripts written in other languages. For example, as Peter Miller explains [10], invoking Make recursively can cause all-or-nothing builds. Grexmk could be used to divide recursive Make-based builds into mini-builds.

7. CONCLUSION AND OPEN QUESTIONS

Grexmk is a tool suite for reengineering all-or-nothing builds so that they can be executed incrementally or in parallel. The suite uses one dynamic analysis to divide a build into mini-builds and another dynamic analysis to verify that a set of mini-builds executes safely. We showed that this technique works well on two real-world builds, but questions remain.

The larger goal of this work was to make it easier for developers to change large programs, but slow builds are not the only obstacles to integrating changes. A change is not fully integrated until the program is deployed, configured, and tested with the change. Suppose that these phases, as written, are also too slow—can they be reengineered to avoid deploying, configuring, and testing parts of the program that are unaffected by a change? What analyses would be necessary?

To use DivideTrace (see Section 4.2), the developer must write a division specification. With better analyses, could this work be avoided for common cases?

Finally, we believe that mini-builds are easier to maintain and optimize than are monolithic builds; will experience prove us right?

8. ACKNOWLEDGEMENTS

Chet Murthy and Robert Yates made significant contributions to this work.

9. REFERENCES

[1] Apache Ant 1.6.5 manual. `http://ant.apache.org/manual`.

[2] M. de Jonge. Decoupling source trees into build-level components. In J. Bosch and C. Krueger, editors, *Proceedings of the Eighth International Conference on Software Reuse*, volume 3107 of *LNCS*, pages 215–231. Springer-Verlag, July 2004.

[3] Debian—the universal operating system. `http://www.debian.org`.

[4] Electric Cloud. `http://www.electric-cloud.com`.

[5] S. I. Feldman. Make—A program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–265, Apr. 1979.

[6] N. Jørgensen. Safeness of Make-based incremental recompilation. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 126–145, London, UK, 2002. Springer-Verlag.

[7] B. W. Kernighan and R. Pike. *The Unix Programming Environment*. Prentice Hall, Inc., 1984.

[8] G. Lagorio. *Type systems for Java separate compilation and selective recompilation*. PhD thesis, DISI, University of Genoa, Mar. 2004.

[9] M. Lindner. Jolt. `http://www.hyperrealm.com/main.php?s=jolt`.

[10] P. Miller. Recursive Make considered harmful. *AUUGN Journal of AUUG, Inc.*, 19(1):14–25, 1998.

[11] Rational ClearCase. `http://www.ibm.com/software/awdtools/clearcase`.

[12] R. M. Stallman, R. McGrath, and P. D. Smith. *GNU Make: A program for directing recompilation, for version 3.81*. Free Software Foundation, June 2004.

[13] Strace, version 4.5.8. `http://www.liacs.nl/~wichert/strace/`.

[14] W. F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.

[15] Veridian Information Solutions Inc. *Portable Batch System Administrator Guide, Release: OpenPBS 2.3*, 2000.

[16] L. Wall, T. Christansen, and J. Orwant. *Programming Perl*. O'Reilly and Associates, third edition, 2000.