University of Bristol

DEPARTMENT OF COMPUTER SCIENCE

An Automatic Make Facility

Ian Holyer   Huseyin Pehlivan

January 2000 | CSTR-00-001

An Automatic *Make* Facility

Ian Holyer, Hüseyin Pehlivan
Department of Computer Science, University of Bristol

January 11, 2000

Abstract

A recompilation program, often called *make*, is a crucial utility employed by many programmers during the development of sophisticated programs. It provides an elegant method for the compilation of programs which are made up of many components, written in any programming language. In this report, we present a new implementation of *make* based on command tracing. Our approach takes advantages of command transactions monitored by a user shell named `brush` [1]. All the information that is required to check for the need to re-compile a particular program component is acquired via these transactions. There is no requirement for a file to specify dependencies and commands, as with the *Makefile* used by *make*. All that is needed is that each program component must be compiled at least once, by hand or from a shell script, to establish the command used.

1 Introduction

Programmers of any programming language normally split up large programs into multiple source files. In Unix, this is often done so that a change to one source file only requires one file to be recompiled and not all the files. In order to help programmers to use this kind of programming facility, Unix provides a utility named *make* that completes the compilation of all source programs in a favourable way. With the *make* program, not having to recompile everything because one file has been changed greatly increases the speed at which programmers can work.

Before being able to use the *make* utility, a programmer must create a particular file, generally called *Makefile*. The file contains the information that is necessary to compile and link the files needed for an executable program. Using the *Makefile*, the *make* program also determines which source files need recompiling. The programmer may have to modify the *Makefile* during the development of a program as a result of incorporating new source files, or making the production of an object file dependent on new files. For large projects, the *Makefile* can become very complicated, dealing with sub-projects, libraries etc., and maintaining it and keeping it correct can become a burden.

In this report, we present a new implementation of the `make` utility, which is called `bmake` throughout the report. The `bmake` utility is implemented in an environment controlled by `brush` [6]. Unlike an ordinary shell, `brush` keeps track of transactions caused by the interaction of user commands with the file system. In `brush` these transactions are basically used for recovery purposes. However, they can also be used to meet the requirements of `bmake`, since they monitor which files are involved in the execution of a command, and what kind of operations (reading, writing, etc.) are made on these files.

In fact, `bmake` has an implementation which is based on command tracing. Each source file is compiled under the monitoring of `bmake` to allow it to discover which files the corresponding object file depends on. By examining the transactions since the last compilation of a source file, `bmake` can determine whether or not it is necessary to recompile the file.

Just as `brush` works in a command independent way, `bmake` works in a compiler-independent way. The `bmake` utility has the ability to monitor any compiler (or other translator) supported by the system and keep track of changes made to the files that it includes in the compilation. To make use of `bmake`, programmers do not have to write or maintain a *Makefile*-like component. Instead, each command involved in the compilation must be executed "by hand" or from a shell script at least once, the first time. If the history list maintained by `brush` includes multiple commands that create the same object file, `bmake` deals with the most recent one, and the transactions belonging to it.

The `bmake` program strives to provide all the facilities that come with the original `make` program. It also partly supports some facilities that are not in common use, such as associating a target with different sets of

[1] `Brush` is a contraction of 'BRistol Undo SHell'.

commands via a double-colon ("::"). Whichever set of commands is more recent is executed. The implementation saves the programmer from the effort of maintaining definitions, by providing the following features:

- there is no need to maintain a description file,
- the functionalities achieved are command independent,
- local variables, such as macros, are not needed,
- there are no restrictions on the generation of commands,
- dependency checking is done automatically,
- synchronization is handled automatically.

The rest of the report is organized as follows. Section 2 describes some improved versions of `make` with new techniques, Section 3 gives a short description of `brush`, showing how it deals with user commands. In Section 4, the method that `bmake` uses to control compilations is introduced. The behaviour of *make* and some issues concerning our method are discussed in Section 5. Section 6 presents `bmake`-related issues that the implementation must resolve. The implementation details are given in Section 7. Finally we summarize the current status of `bmake`, as well as some additional work required, in Section 8.

2 Related Work

The `make` utility was first developed at the Bell Laboratories [4]. It is most naturally used to sort out dependency relations among files [8]. Over the years, many extensions of `make` have been produced to increase its utilization by using very attractive techniques. These extensions equip `make` with more powerful features and remove some arbitrary restrictions. Representative examples are `nmake`, `optimistic make`, `pmake`, `mk`.

The `nmake` [5] program embodies major semantic and syntactic enhancements to the standard `make` program. It also provides improved functionality and performance. In `nmake`, an extremely modular directory structure is used. Some directories should be devoted to source files (.c), some to header files (.h), some to libraries (.a), and so on. The update commands are executed by sending the command blocks to the shell *sh*, which runs as a co-process.

`Optimistic make` [2] begins execution of the update commands before the user issues the make request. Outputs of these `optimistic computations` (such as file or screen updates) are concealed until the request is issued. If the inputs read by the optimistic computations have not been changed by the time of the `make` request, the results of the optimistic computations are used. Otherwise, the necessary computations are re-executed. Thus `optimistic make` requires minor modifications to the kernel and to some of the servers.

Parallel make (`pmake`) [1] speeds up the operation of `make` by doing compilations in parallel. For each command block to be executed, `pmake` creates a child process called a *virtual processor*. `pmake` doesn't wait for the virtual processor to finish, but continues processing the list of dependencies in which the target appeared. The parallelism is hidden from the description file writer. It doesn't burden the programmer with all the details of the parallelism.

`Mk` [7], which is an enhanced version of the original `make`, supports program construction in a heterogeneous environment. To exploit the power of multiprocessors, it executes maintenance actions in parallel. The number of jobs run in parallel is user-settable by defining the macro `$NPROC`. `Mk` interacts seamlessly with the Plan 9 command interpreter *rc* and accepts pattern-based dependency specifications.

3 An Overview of `Brush`

`Brush` [6] is a Unix shell with undo support facilities. It creates a secure environment for controlling file versions. `Brush` is equipped with features similar to usual shells, except that it provides `undo` and `redo` commands which are capable of operating selectively on commands and programs. An `undo` command reverses the effects of a program, while a `redo` command reverses the effects of an `undo` command. With these commands, users can recover any desired version of a file without needing the help of the system administrator.

Conventionally, the provision of selective undo involves discovering what changes a command makes to the file system state. In order to keep all command executions under its control, `brush` takes advantage of tracing facilities provided by the **/proc** file system [3]. It attaches itself to each process created to run user commands, and monitors it to trace system calls, particularly file system calls. In this way, selected system calls are stopped on entry to the kernel and necessary items of recovery information (e.g. file backups, command transactions, etc.) are stored. Then the stopped system call is resumed. `Brush` also traces child processes spawned dynamically by programs.

For purposes of recovery *history* and *transaction* lists are maintained through the interaction. `Brush` records in them commands, and file transactions which are performed by commands. File versions that are saved are named by using the corresponding transaction numbers. Initially, for example, suppose the transaction list for two files, `fileA` and `fileB`, created from scratch is as follows:

```
1> Create fileA-1
2> Create fileB-2
```

where each number that is tagged to the filenames shows on which version of the file the transaction is performed. At this stage of the interaction, suppose the following standard Unix command is executed:

```
cp fileA fileB
```

`Brush`, monitoring the execution, intercepts a *read* system call on `fileA` and a *creat* system call on `fileB`, and thus stores new transactions:

```
3> Read fileA-1
4> Delete fileB-2
5> Create fileB-5
```

`Brush` represents the effects of commands on the file system by using three basic transactions: `Read`, `Create`, and `Delete`. With these transactions, it is quite easy to determine which file(s) a particular command needs and which file(s) it produces. `Brush` only deals with files that are owned by the user and keeps the transactions that operate on them. Files that are owned by other users or shared across the system are ignored in terms of recovery requirements.

4 Compiler Tracing

The `Tracer` mechanism that `brush` uses to keep command executions under control presents a new approach to implementing the Unix *make* program. From `brush`'s perspective, a compiler does nothing but read existing files (source files) and create new files (object files or executables). In our approach, transactions that represent all file accesses made by the compiler, thus play an important role in implementing `bmake`.

The approach to `bmake` is based on tracking the compilation of source files. As with ordinary commands, compilers work under the monitoring of `brush` and file transactions that are performed are stored for the requirements of `bmake`. In order to describe information gathered with compiler tracing, we will use the C programming language and the *gcc* compiler.

Consider the following transaction list for two source files, `fileA.c` and `fileB.c` and for a programmer-defined header file, `fileA.h`.

```
1> Create fileA.c-1
2> Create fileA.h-2
3> Create fileB.c-3
```

Suppose that `fileA.c` includes the header file specified with the #include preprocessor directive, and that `fileB.c` does not. If the programmer attempts to produce the corresponding object files by typing in

```
gcc -g -c fileA.c fileB.c
```

the compiler performs the following *open* system calls of interest, consecutively.

```
open("fileA.c", O_RDONLY)
open("fileA.h", O_RDONLY)
open("fileA.o", O_RDWR|O_CREAT, 0666)
open("fileB.c", O_RDONLY)
open("fileB.o", O_RDWR|O_CREAT, 0666)
```

Then the transaction list is extended with additonal transactions:

```
4> Read fileA.c-1
5> Read fileA.h-2
6> Create fileA.o-6
7> Read fileB.c-3
8> Create fileB.o-8
```

Note that shared library header files such as `stdio.h` are not of concern in `brush`, as they do not belong to the current user.

Given the transaction list, it is possible to discover all the files (*dependencies*) which are used to produce an object file. However, for the cases where two or more source files are compiled together, there is no graceful way to find out which dependency a source file uses. In the above example, one can't simply say that `fileA.o` doesn't depend on `fileB.c`. The order in which dependencies are read and the corresponding object files are created can't be used to work this out, since the creation of `fileA.o` may not have been completed before `fileB.c` is opened for reading. As a result, each of the files read during the compilation tends to be a potential dependency for every object file created.

Now, consider what happens if the programmer reedits `fileB.c`. Such a situation creates the following transactions.

```
9> Read fileB.c-3
10> Delete fileB.c-3
11> Create fileB.c-11
```

Of these transactions, `Delete` or `Create` explicitly shows that `fileB.c` has been modified after its first compilation. In this case, both `fileA.c` and `fileB.c` must be recompiled and the recompilation must be repeated after any of the files they depend on are modified.

5 Make Analysis

The way the Unix *make* program normally works is simple. After changing some source files, the programmer starts *make*, which examines the times at which all the source and object files were last modified. If a source file has an earlier modification time than the corresponding object file, no compilation is needed. Otherwise, *make* knows that the source file has been changed since the object file was created, and thus the source file must be recompiled. In this way, *make* goes through all the source files to find out which ones need recompiling, and calls the compiler to recompile them.

A particular file, usually named as `Makefile`, tells *make* which source files to check for the recompilation. In `Makefile` each source file must be referred to with the programmer-defined files (e.g. header files) it depends on, because *make* also needs to compare their modification times to the time of the corresponding object file.

Sophisticated programs can force *make* to employ quite a complicated checking procedure. In general the production of an executable program includes multiple-level checking of dependencies. Given the executable *program* in Figure 1, there are many potential checking points that *make* has to deal with, depending on the programmer's specification. If the programmer includes these points into the *Makefile*, specifying input files for each file that is to be produced, the `make` utility ensures that all the points are checked from *program* downwards and, if necessary, recompiled in reverse order.

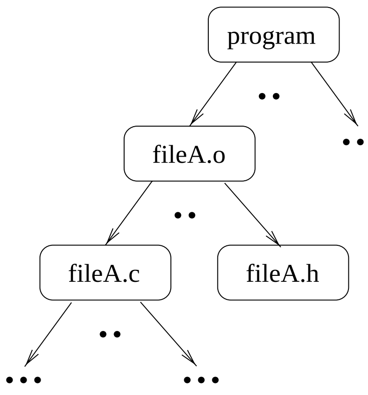

Figure 1: A typical tree of checking dependencies

In our approach, it would also be possible to use modification times to discover changes made to dependencies. However, sometimes there are some shortcomings in using modification times to decide when a target is out of date. For example, consider two source files, `fileA.c` and `fileB.c`, where the modification times of `fileB.c` and its object file `fileB.o` are more recent than that of `fileA.c`. If `fileA.c` is renamed as `fileB.c`, so replacing it, on many operating systems the new `fileB.c` has the old modification time of `fileA.c`. Then the target (`fileB.o`) would not appear to be out of date with its source file (`fileB.c`), and `make` would wrongly fail to recompile it. To avoid situations like this, there is an alternative approach which is adopted by `bmake`. By examining the transaction list, the need for recompiling can be determined easily and precisely.

Once an instance of the compilation command is submitted which contains the compiler name, flags, source files, and so on, the compilation can be executed again at any time by typing in `bmake` with the related object file. However, if one of the following situations occurs the programmer cannot naturally use `bmake`, and the relevant compilation commands need to be reissued:

1. Compiler options need to be changed,
2. A new source file is added into the compilation.
3. Files involved in the compilation are renamed.

On the other hand, for example, with the use of the #include directive, including new header files, or removing existing ones, does not affect the use of `bmake`. In this kind of situation, the changes are reflected in the transactions recorded for a subsequent compilation, and so are detected by `bmake`, which then becomes aware of the change in the dependencies and can take any necessary remedial action.

6 Implementation Issues

Our implementation aims to preserve the original behaviour of the *make* utility, but without requiring the provision of a `Makefile`. Compared to *make*, there is less burden put on programmers, reducing their involvement in the compiling process.

In the case of *make*, programmers use *Makefile* to specify how compilations are to be carried out. Our method expects programmers to start the compilation of each source file by typing the corresponding command line at least once. The most recent command line for a particular source file determines the way that *bmake* carries out later compilations.

`Brush` inherently associates a command line with the transactions that it executes. `Bmake` uses the transactions to determine dependencies. However, there are circumstances in which care needs to be taken. For example, suppose that a source file was last compiled by bmake itself, and that the execution of bmake involved several compilations. If all the transactions associated with the bmake execution are used to determine dependencies, too many dependencies will be found. To get around this problem, each compilation issued by bmake is stored as a separate command line in the brush history, as if it had been typed in by the user, and the appropriate transactions are associated with it.

The fact that compilation commands can be constructed in a wide variety of ways means that special attention has to be given to recompilations. For instance, a programmer can type in a command that produces several object files in one go. Care needs to be taken by bmake to compute the dependencies properly, and to ensure that the command is executed only once, no matter how many of the object files need to be recompiled.

Another issue concerns the order of recompilations. This causes no problem if an object file can always be produced independently of other object files. However, in many programming languages, modules must be compiled in dependency order. Thus bmake must carry out all required compilations in the right order, finally running the command that creates the executable.

The dependence on other files such as header files raises a new problem. If a source file is made dependent on a new header file, say, the new dependency is not among the ones computed by bmake from the last compilation command issued, so there is a danger of producing an object file with inconsistent contents. To discover this situation, the file is firstly compiled using the old dependencies, and then a comparison is made between the old and new dependencies. Then the proper action must be taken according to the new dependencies, perhaps involving compiling the file again.

A programmer who deals with the development of a program often encounters compilation errors. Most compilations generally start after opening all the input files for reading. Even if such a compilation fails to produce an object file, the transaction list contains all the dependencies that belong to the compilation and there is no problem. However, the compiler may fail before opening all the relevant input files, so that the dependencies computed from the compiling command are incomplete. This has no effect on the operation of bmake. The actions described in the previous paragraph also sort this problem out.

7 Implementation

The fact that brush keeps past transactions makes the implementation of bmake fairly easy. Without the programmer needing to specify which compiler tools to use, it can deal with all compilers supported by the system. In other words, since transactions contain only the file system interactions, the implementation is compiler independent.

A compilation process can implicitly make use of many input files (dependencies). At first sight, it appears that dependencies can be determined from command lines. However, a command line does not include all the dependencies, for example it excludes files imported by source files, and in any case it may not be clear from a command line which arguments are actually file names. Thus, in bmake, all the input files required for an output file are taken from the transactions associated with running the compiler.

We assume that each object file created by a compiler needs all the files read by the compiler (regardless of whether they are read before or after the object file is opened for writing in the transaction list). This reveals an important requirement for source files which can be individually compiled. For bmake to discover dependencies that correspond only to a particular object file, programmers must type in a new command line for each such file.

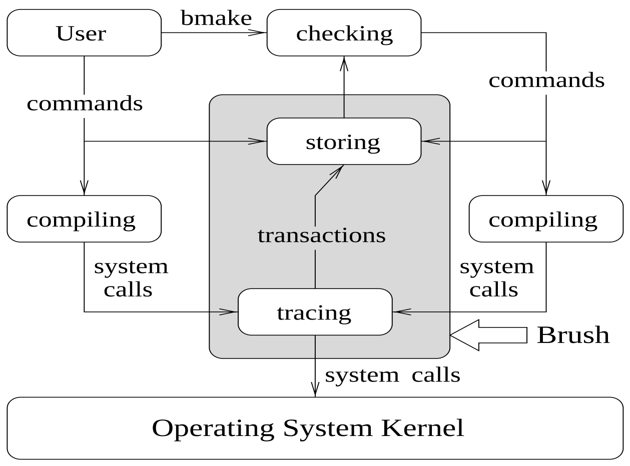

Figure 2: General structure of bmake

There is no difference between a single object file and the executable in terms of discovering dependencies. Bmake always takes the transactions of the last compilation associated with the object file or the executable into consideration so that it can check for modifications to dependencies properly. The desired comparison is made between these transactions and later ones, which includes both file names and version numbers.

As an example, assume that a programmer produces an object file named fileA.o by issuing

```
    gcc -c -o fileA.o fileA.c
```

During the compilation, `bmake` ensures that all file transactions the compiler (`gcc`) makes which are associated with the programmer's own address space are stored. After carrying out some editing tasks, if the object file needs checking for recompilation, it is adequate to type in

```
bmake fileA.o
```

In this case, in order to complete the recompilation, `bmake` basically uses the following algorithm. Firstly, the transaction list is searched for the most recent transaction that involves a `Create` on `fileA.o`. Then `bmake` uses the history list to find out the command which created the relevant transaction. Using this command, it discovers the other transactions that were required to produce `fileA.o`. Of these transactions, ones involving files on which a `Read` is made determine the dependencies of `fileA.o`.

Now, `bmake` checks the transactions prior to the time when `fileA.o` is created, and sees whether the dependent files depend on some other files or not, using the above algorithm for each one. If there are more dependencies, the same algorithm must be used recursively for each of them and so on.

After discovering all the dependent files in this way, `bmake` needs to check for any changes made to them. For each dependent file, the checking process covers the transactions that reside between its last creation time before the creation of `fileA.o` and the current time. If a later transaction has affected at least one of the dependent files, specifically one transaction refers to a `Create` or a `Delete` on a dependent file, `fileA.c` should be recompiled.

The editing tasks on `fileA.c` could cause the recompilation to create transactions which are different from ones created by the previous compilation of the same file, and thus to use different dependencies. Therefore, the `bmake` compares these two compilations to be able to discover new dependencies. Each new dependency can require a new recompilation of `fileA.c` after it is checked with the algorithm.

Compilation commands can make the steps involved in this algorithm quite complicated. For example, if the following linking command was initially submitted:

```
gcc -o program fileA.o fileB.o ...
```

the programmer can use `bmake` to update the executable program.

```
bmake program
```

This starts the checking procedure for each object file, in the same way as with `fileA.o`, and ends with a new linking command to create a new `program` file.

8 Conclusion and Future Work

In this report, we have presented the implementation of `bmake` which carries out compilations of source files to build executable programs. It has almost the same behaviour as the Unix *make* program, except that it does not need an auxiliary file (often named *Makefile*). The approach adopted in `bmake` is based on command tracing that is provided by `brush`. All compilations are made under `bmake`'s control to find out the file system transactions, and thus to discover the dependencies of each object file. Not having to specify dependencies for a particular compilation enables programmers to focus entirely on the development of programs.

Our approach has made it necessary to be able to differentiate a compiler from an editor. `Bmake` needs this to relate a file to the right dependencies. For example, suppose that a programmer, currently working on a source file, calls up another file in the editor. In this case, `bmake` must not treat the second file as a dependency, otherwise changing that second file would cause `bmake` to re-run the editor. This can be dealt with either by declaring certain programs to be editors, and thus not to cause dependencies, or by declaring certain types of files to be source files, and thus never to need re-creating.

This information means that the `bmake` program can work cooperatively with `brush` to help avoid saving old versions of files. The use of `bmake` allows the system to determine which files are object files. Thus old versions of them need not be kept by `brush`, because they can be reconstructed whenever necessary. This works even in the presence of undo, because old versions of object files can be reconstucted from old versions of source files.

As far as a compiler independent implementation is concerned, there is a problem in allowing `bmake` to be called without any arguments. It can then be difficult to find the name of the final executable file (or files) to start the checking process. All these problems are the scope of future work.

References

[1] E.H. Baalbergen. Design and implementation of parallel make. *Computing Systems*, 1(2):135–158, Spring 1998.

[2] R. Bubenik and W. Zwaenepoel. Optimistic make. *IEEE Transactions on Computers*, 41(2):207–217, 1992.

[3] R. Faulkner and R. Gomes. The process file system and process model in UNIX system V. In *Pro-*

ceedings of the 1991 Winter USENIX Conference. USENIX Assoc., Berkeley, CA., 1991.

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

[5] G.S. Fowler. The fourth generation make. In *Proceedings of the 1985 Summer USENIX Conference*, pages 159–174, June 1985.

[6] I. Holyer and H. Pehlivan. A recovery mechanism for shells. *To probably appear in Computer*, 1999.

[7] A. Hume. Mk: A successor to make. Technical Report 141, AT&T Bell Laboratories, Murray Hill, NJ, Nov. 1987.

[8] A. Oram and S. Talbott. *Managing Projects with make.* O'Reilly and Associates Inc., 1993.