CHAPTER 10

Improving the Performance of make

make plays a critical role in the development process. It combines the elements of a project to create an application while allowing the developer to avoid the subtle errors caused by accidentally omitting steps of the build. However, if developers avoid using make, because they feel the *makefile* is too slow, all the benefits of make are lost. It is important, therefore, to ensure that the *makefile* be crafted to be as efficient as possible.

Performance issues are always tricky, but become even more so when the perception of users and different paths through the code are considered. Not every target of a *makefile* is worth optimizing. Even radical optimizations might not be worth the effort depending on your environment. For instance, reducing the time of an operation from 90 minutes to 45 minutes may be immaterial since even the faster time is a "go get lunch" operation. On the other hand, reducing a task from 2 minutes to 1 might be received with cheers if developers are twiddling their thumbs during that time.

When writing a *makefile* for efficient execution, it is important to know the costs of various operations and to know what operations are being performed. In the following sections, we will perform some simple benchmarking to quantify these general comments and present techniques to help identify bottlenecks.

A complementary approach to improving performance is to take advantage of parallelism and local network topology. By running more than one command script at a time (even on a uniprocessor), build times can be reduced.

Benchmarking

Here we measure the performance of some basic operations in make. Table 10-1 shows the results of these measurements. We'll explain each test and suggest how they might affect *makefile*s you write.

Table 10-1. Cost of operations

Operation	Executions	Seconds per execution (Windows)	Executions per second (Windows)	Seconds per execution (Linux)	Executions per second (Linux)
make (bash)	1000	0.0436	22	0.0162	61
make (ash)	1000	0.0413	24	0.0151	66
make (sh)	1000	0.0452	22	0.0159	62
assignment	10,000	0.0001	8130	0.0001	10,989
subst (short)	10,000	0.0003	3891	0.0003	3846
subst (long)	10,000	0.0018	547	0.0014	704
sed (bash)	1000	0.0910	10	0.0342	29
sed (ash)	1000	0.0699	14	0.0069	144
sed (sh)	1000	0.0911	10	0.0139	71
shell (bash)	1000	0.0398	25	0.0261	38
shell (ash)	1000	0.0253	39	0.0018	555
shell (sh)	1000	0.0399	25	0.0050	198

The Windows tests were run on a 1.9-GHz Pentium 4 (approximately 3578 BogoMips)* with 512 MB RAM running Windows XP. The Cygwin version of make 3.80 was used, started from an rxvt window. The Linux tests were run on a 450-MHz Pentium 2 (891 BogoMips) with 256 MB of RAM running Linux RedHat 9.

The subshell used by make can have a significant effect on the overall performance of the *makefile*. The bash shell is a complex, fully featured shell, and therefore large. The ash shell is a much smaller, with fewer features but adequate for most tasks. To complicate matters, if bash is invoked from the filename */bin/sh*, it alters its behavior significantly to conform more closely to the standard shell. On most Linux systems the file */bin/sh* is a symbolic link to bash, while in Cygwin */bin/sh* is really *ash*. To account for these differences, some of the tests were run three times, each time using a different shell. The shell used is indicated in parentheses. When "(sh)" appears, it means that bash was linked to the file named */bin/sh*.

The first three tests, labeled make, give an indication of how expensive it is to run make if there is nothing to do. The *makefile* contains:

```
SHELL := /bin/bash
.PHONY: x
x:
        $(MAKE) --no-print-directory --silent --question make-bash.mk; \
        …this command repeated 99 more times…
```

The word "bash" is replaced with the appropriate shell name as required.

* See *http://www.clifton.nl/bogomips.html* for an explanation of BogoMips.

We use the `--no-print-directory` and `--silent` commands to eliminate unnecessary computation that might skew the timing test and to avoid cluttering the timing output values with irrelevant text. The `--question` option tells `make` to simply check the dependencies without executing any commands and return an exit status of zero if the files are up to date. This allows `make` to do as little work as possible. No commands will be executed by this *makefile* and dependencies exist for only one `.PHONY` target. The command script executes `make` 100 times. This *makefile*, called *make-bash.mk*, is executed 10 times by a parent *makefile* with this code:

```
define ten-times
  TESTS += $1
  .PHONY: $1
  $1:
        @echo $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2; \
        time $(MAKE) --no-print-directory --silent $2
endef

.PHONY: all
all:

$(eval $(call ten-times, make-bash, -f make-bash.mk))

all: $(TESTS)
```

The time for these 1,000 executions is then averaged.

As you can see from the table, the Cygwin `make` ran at roughly 22 executions per second or 0.044 seconds per run, while the Linux version (even on a drastically slower CPU) performed roughly 61 executions per second or 0.016 seconds per run. To verify these results, the native Windows version of `make` was also tested and did not yield any dramatic speed up. Conclusion: while process creation in Cygwin `make` is slightly slower than a native Windows `make`, both are dramatically slower than Linux. It also suggests that use of recursive `make` on a Windows platform may perform significantly slower than the same build run on Linux.

As you would expect, the shell used in this test had no effect on execution time. Because the command script contained no shell special characters, the shell was not invoked at all. Rather, `make` executed the commands directly. This can be verified by setting the `SHELL` variable to a completely bogus value and noting that the test still runs correctly. The difference in performance between the three shells must be attributed to normal system variance.

The next benchmark measures the speed of variable assignment. This calibrates the most elementary make operation. The *makefile*, called *assign.mk*, contains:

```
# 10000 assignments
z := 10
…repeated 10000 times…
.PHONY: x
x: ;
```

This *makefile* is then run using our `ten-times` function in the parent *makefile*.

The assignment is obviously very fast. Cygwin make will execute 8130 assignments per second while the Linux system can do 10,989. I believe the performance of Windows for most of these operations is actually better than the benchmark indicates because the cost of creating the make process 10 times cannot be reliably factored out of the time. Conclusion: because it is unlikely that the average *makefile* would perform 10,000 assignments, the cost of variable assignment in an average *makefile* is negligible.

The next two benchmarks measure the cost of a `subst` function call. The first uses a short 10-character string with three substitutions:

```
# 10000 subst on a 10 char string
dir := ab/cd/ef/g
x := $(subst /, ,$(dir))
…repeated 10000 times…
.PHONY: x
x: ;
```

This operation takes roughly twice as long as a simple assignment, or 3891 operations per second on Windows. Again, the Linux system appears to outperform the Windows system by a wide margin. (Remember, the Linux system is running at less than one quarter the clock speed of the Windows system.)

The longer substitution operates on a 1000-character string with roughly 100 substitutions:

```
# Ten character file
dir := ab/cd/ef/g
# 1000 character path
p100 := $(dir);$(dir);$(dir);$(dir);$(dir);…
p1000 := $(p100)$(p100)$(p100)$(p100)$(p100)…

# 10000 subst on a 1000 char string
x := $(subst ;, ,$(p1000))
…repeated 10000 times…
.PHONY: x
x: ;
```

The next three benchmarks measure the speed of the same substitution using sed. The benchmark contains:

```
# 100 sed using bash
SHELL := /bin/bash
```

```
.PHONY: sed-bash
sed-bash:
        echo '$(p1000)' | sed 's/;/ /g' > /dev/null
        …repeated 100 times…
```

As usual, this *makefile* is executed using the ten-times function. On Windows, sed execution takes about 50 times longer than the subst function. On our Linux system, sed is only 24 times slower.

When we factor in the cost of the shell, we see that ash on Windows does provide a useful speed-up. With ash, the sed is only 39 times slower than subst! (wink) On Linux, the shell used has a much more profound effect. Using ash, the sed is only five times slower than subst. Here we also notice the curious effect of renaming bash to sh. On Cygwin, there is no difference between a bash named /bin/bash and one named /bin/sh, but on Linux, a bash linked to /bin/sh performs significantly better.

The final benchmark simply invokes the make shell command to evaluate the cost of running a subshell. The *makefile* contains:

```
# 100 $(shell ) using bash
SHELL := /bin/bash
x := $(shell :)
…repeated 100 times…
.PHONY: x
x: ;
```

There are no surprises here. The Windows system is slower than Linux, with ash having an edge over bash. The performance gain of ash is more pronounced—about 50% faster. The Linux system performs best with ash and slowest with bash (when named "bash").

Benchmarking is a never-ending task, however, the measurements we've made can provide some useful insight. Create as many variables as you like if they help clarify the structure of the *makefile* because they are essentially free. Built-in make functions are preferred over running commands even if you are required by the structure of your code to reexecute the make function repeatedly. Avoid recursive make or unnecessary process creation on Windows. While on Linux, use ash if you are creating many processes.

Finally, remember that in most *makefile*s, the time a *makefile* takes to run is due almost entirely to the cost of the programs run, not make or the structure of the *makefile*. Usually, reducing the number of programs run will be most helpful in reducing the execution time of a *makefile*.

Identifying and Handling Bottlenecks

Unnecessary delays in *makefile*s come from several sources: poor structuring of the *makefile*, poor dependency analysis, and poor use of make functions and variables.

These problems can be masked by `make` functions such as `shell` that invoke commands without echoing them, making it difficult to find the source of the delay.

Dependency analysis is a two-edged sword. On the one hand, if complete dependency analysis is performed, the analysis itself may incur significant delays. Without special compiler support, such as supplied by `gcc` or `jikes`, creating a dependency file requires running another program, nearly doubling compilation time.[*] The advantage of complete dependency analysis is that it allows `make` to perform fewer compiles. Unfortunately, developers may not believe this benefit is realized and write *makefile*s with less complete dependency information. This compromise almost always leads to an increase in development problems, leading other developers to overcompensate by compiling more code than would be required with the original, complete dependency information.

To formulate a dependency analysis strategy, begin by understanding the dependencies inherent in the project. Once complete dependency information is understood, you can choose how much to represent in the *makefile* (computed or hardcoded) and what shortcuts can be taken during the build. Although none of this is exactly simple, it is straightforward.

Once you've determined your *makefile* structure and necessary dependencies, implementing an efficient *makefile* is usually a matter of avoiding some simple pitfalls.

Simple Variables Versus Recursive

One of the most common performance-related problems is using recursive variables instead of simple variables. For example, because the following code uses the `=` operator instead of `:=`, it will execute the `date` command every time the `DATE` variable is used:

```
DATE = $(shell date +%F)
```

The `+%F` option instructs `date` to return the date in "yyyy-mm-dd" format, so for most users the repeated execution of `date` would never be noticed. Of course, developers working around midnight might get a surprise!

Because `make` doesn't echo commands executed from the `shell` function, it can be difficult to determine what is actually being run. By resetting the `SHELL` variable to `/bin/sh -x`, you can trick `make` into revealing all the commands it executes.

[*] In practice, compilation time grows linearly with the size of the input text and this time is almost always dominated by disk I/O. Similarly, the time to compute dependencies using the simple `-M` option is linear and bound by disk I/O.

This *makefile* creates its output directory before performing any actions. The name of the output directory is composed of the word "out" and the date:

```
DATE = $(shell date +%F)
OUTPUT_DIR = out-$(DATE)

make-directories := $(shell [ -d $(OUTPUT_DIR) ] || mkdir -p $(OUTPUT_DIR))

all: ;
```

When run with a debugging shell, we can see:

```
$ make SHELL='/bin/sh -x'
+ date +%F
+ date +%F
+ '[' -d out-2004-03-30 ']'
+ mkdir -p out-2004-03-30
make: all is up to date.
```

This clearly shows us that the `date` command was executed twice. If you need to perform this kind of shell trace often, you can make it easier to access with:

```
ifdef DEBUG_SHELL
  SHELL = /bin/sh -x
endif
```

Disabling @

Another way commands are hidden is through the use of the silent command modifier, `@`. It can be useful at times to be able to disable this feature. You can make this easy by defining a variable, `QUIET`, to hold the @ sign and use the variable in commands:

```
ifndef VERBOSE
  QUIET := @
endif
…
target:
        $(QUIET) echo Building target...
```

When it becomes necessary to see commands hidden by the silent modifier, simply define `VERBOSE` on the command line:

```
$ make VERBOSE=1
echo Building target...
Building target...
```

Lazy Initialization

When simple variables are used in conjunction with the `shell` function, `make` evaluates all the `shell` function calls as it reads the *makefile*. If there are many of these, or if they perform expensive computations, `make` can feel sluggish. The responsiveness of `make` can be measured by timing `make` when invoked with a nonexistent target:

```
$ time make no-such-target
make: *** No rule to make target no-such-target.  Stop.
```

```
real    0m0.058s
user    0m0.062s
sys     0m0.015s
```

This code times the overhead that make will add to any command executed, even trivial or erroneous commands.

Because recursive variables reevaluate their righthand side every time they are expanded, there is a tendency to express complex calculations as simple variables. However, this decreases the responsiveness of make for all targets. It seems that there is a need for another kind of variable, one whose righthand side is evaluated only once the first time the variable is evaluated, but not before.

An example illustrating the need for this type of initialization is the find-compilation-dirs function introduced in the section "The Fast Approach: All-in-One Compile" in Chapter 9:

```
# $(call find-compilation-dirs, root-directory)
find-compilation-dirs =                        \
  $(patsubst %/,%,                             \
    $(sort                                     \
      $(dir                                    \
        $(shell $(FIND) $1 -name '*.java'))))

PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR))
```

Ideally, we would like to perform this find operation only once per execution, but only when the PACKAGE_DIRS variable is actually used. This might be called *lazy initialization*. We can build such a variable using eval like this:

```
PACKAGE_DIRS = $(redefine-package-dirs) $(PACKAGE_DIRS)

redefine-package-dirs = \
   $(eval PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR)))
```

The basic approach is to define PACKAGE_DIRS first as a recursive variable. When expanded, the variable evaluates the expensive function, here find-compilation-dirs, and redefines itself as a simple variable. Finally, the (now simple) variable value is returned from the original recursive variable definition.

Let's go over this in detail:

1. When make reads these variables, it simply records their righthand side because the variables are recursive.
2. The first time the PACKAGE_DIRS variable is used, make retrieves the righthand side and expands the first variable, redefine-package-dirs.
3. The value of redefine-package-dirs is a single function call, eval.
4. The body of the eval redefines the recursive variable, PACKAGE_DIRS, as a simple variable whose value is the set of directories returned by find-compilation-dirs. Now PACKAGE_DIRS has been initialized with the directory list.

5. The `redefine-package-dirs` variable is expanded to the empty string (because eval expands to the empty string).
6. Now `make` continues to expand the original righthand side of `PACKAGE_DIRS`. The only thing left to do is expand the variable `PACKAGE_DIRS`. `make` looks up the value of the variable, sees a simple variable, and returns its value.

The only really tricky part of this code is relying on `make` to evaluate the righthand side of a recursive variable from left to right. If, for instance, `make` decided to evaluate `$(PACKAGE_DIRS)` before `$(redefine-package-dirs)`, the code would fail.

The procedure I just described can be refactored into a function, `lazy-init`:

```
# $(call lazy-init,variable-name,value)
define lazy-init
  $1 = $$(redefine-$1) $$($1)
  redefine-$1 = $$(eval $1 := $2)
endef

# PACKAGE_DIRS - a lazy list of directories
$(eval                           \
  $(call lazy-init,PACKAGE_DIRS, \
    $$(call find-compilation-dirs,$(SOURCE_DIRS))))
```

Parallel make

Another way to improve the performance of a build is to take advantage of the parallelism inherent in the problem the *makefile* is solving. Most *makefile*s perform many tasks that are easily carried out in parallel, such as compiling C source to object files or creating libraries out of object files. Furthermore, the very structure of a well-written *makefile* provides all the information necessary to automatically control the concurrent processes.

Example 10-1 shows our `mp3_player` program executed with the jobs option, `--jobs=2` (or `-j 2`). Figure 10-1 shows the same `make` run in a pseudo UML sequence diagram. Using `--jobs=2` tells `make` to update two targets in parallel when that is possible. When `make` updates targets in parallel, it echos commands in the order in which they are executed, interleaving them in the output. This can make reading the output from parallel `make` more difficult. Let's look at this output more carefully.

Example 10-1. Output of make when --jobs = 2

```
 $ make -f ../ch07-separate-binaries/makefile --jobs=2

 1  bison -y  --defines ../ch07-separate-binaries/lib/db/playlist.y

 2  flex  -t ../ch07-separate-binaries/lib/db/scanner.l > lib/db/scanner.c

 3  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
    ../ch07-separate-binaries/app/player/play_mp3.c | \
```

Example 10-1. Output of make when --jobs = 2 (continued)

```
    sed 's,\(play_mp3\.o\) *:,app/player/\1 app/player/play_mp3.d: ,' > app/player/play_
mp3.d.tmp

  4  mv -f y.tab.c lib/db/playlist.c

  5  mv -f y.tab.h lib/db/playlist.h

  6  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
     ../ch07-separate-binaries/lib/codec/codec.c | \
    sed 's,\(codec\.o\) *:,lib/codec/\1 lib/codec/codec.d: ,' > lib/codec/codec.d.tmp

  7  mv -f app/player/play_mp3.d.tmp app/player/play_mp3.d

  8  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
     lib/db/playlist.c | \
    sed 's,\(playlist\.o\) *:,lib/db/\1 lib/db/playlist.d: ,' > lib/db/playlist.d.tmp

  9  mv -f lib/codec/codec.d.tmp lib/codec/codec.d

 10  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
     ../ch07-separate-binaries/lib/ui/ui.c | \
    sed 's,\(ui\.o\) *:,lib/ui/\1 lib/ui/ui.d: ,' > lib/ui/ui.d.tmp

 11  mv -f lib/db/playlist.d.tmp lib/db/playlist.d

 12  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -M
     lib/db/scanner.c | \
    sed 's,\(scanner\.o\) *:,lib/db/\1 lib/db/scanner.d: ,' > lib/db/scanner.d.tmp

 13  mv -f lib/ui/ui.d.tmp lib/ui/ui.d

 14  mv -f lib/db/scanner.d.tmp lib/db/scanner.d

 15  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -c
     -o app/player/play_mp3.o ../ch07-separate-binaries/app/player/play_mp3.c

 16  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -c
     -o lib/codec/codec.o ../ch07-separate-binaries/lib/codec/codec.c

 17  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -c
     -o lib/db/playlist.o lib/db/playlist.c

 18  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -c
     -o lib/db/scanner.o lib/db/scanner.c
    ../ch07-separate-binaries/lib/db/scanner.l: In function yylex:
    ../ch07-separate-binaries/lib/db/scanner.l:9: warning: return makes integer from
pointer without a cast

 19  gcc  -I lib -I ../ch07-separate-binaries/lib -I ../ch07-separate-binaries/include  -c
     -o lib/ui/ui.o ../ch07-separate-binaries/lib/ui/ui.c

 20  ar rv lib/codec/libcodec.a lib/codec/codec.o
```

Example 10-1. Output of make when --jobs = 2 (continued)

```
    ar: creating lib/codec/libcodec.a
    a - lib/codec/codec.o

 21  ar rv lib/db/libdb.a lib/db/playlist.o lib/db/scanner.o
    ar: creating lib/db/libdb.a
    a - lib/db/playlist.o
    a - lib/db/scanner.o

 22  ar rv lib/ui/libui.a lib/ui/ui.o
    ar: creating lib/ui/libui.a
    a - lib/ui/ui.o

 23  gcc   app/player/play_mp3.o lib/codec/libcodec.a lib/db/libdb.a lib/ui/libui.a   -o
     app/player/play_mp3
```

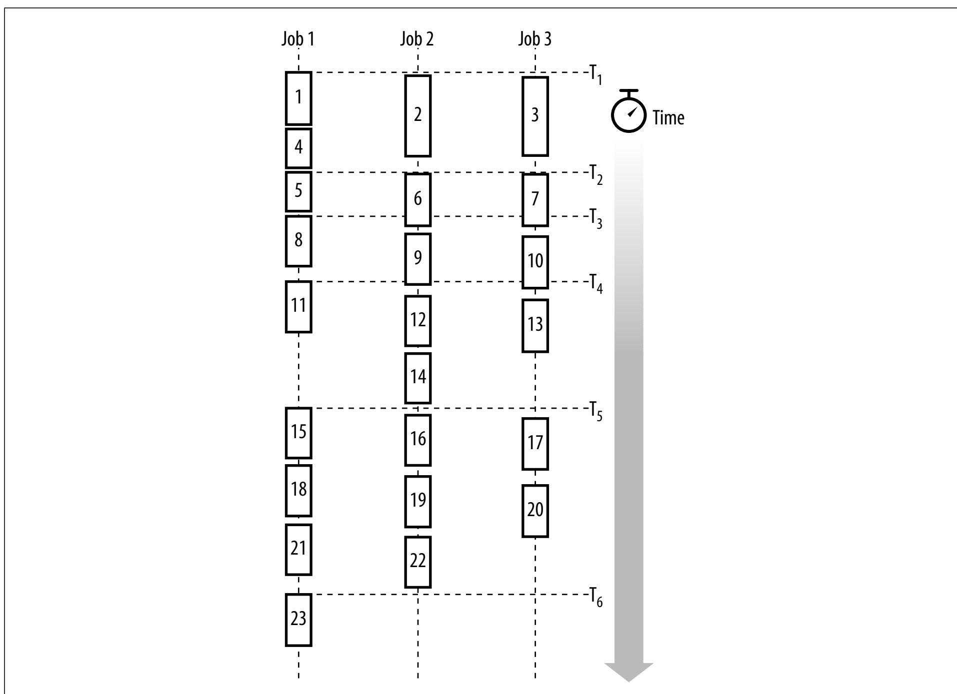

Figure 10-1. Diagram of make when --jobs = 2

First, make must build the generated source and dependency files. The two generated source files are the output of yacc and lex. This accounts for commands 1 and 2. The third command generates the dependency file for *play_mp3.c* and is clearly begun before the dependency files for either *playlist.c* or *scanner.c* are completed (by commands 4, 5, 8, 9, 12, and 14). Therefore, this make is running three jobs in parallel, even though the command-line option requests two jobs.

The `mv` commands, 4 and 5, complete the *playlist.c* source code generation started with command 1. Command 6 begins another dependency file. Each command script is always executed by a single `make`, but each target and prerequisite forms a separate job. Therefore, command 7, which is the second command of the dependency generation script, is being executed by the same `make` process as command 3. While command 6 is probably being executed by a `make` spawned immediately following the completion of the `make` that executed commands 1-4-5 (processing the `yacc` grammar), but before the generation of the dependency file in command 8.

The dependency generation continues in this fashion until command 14. All dependency files must be complete before `make` can move on to the next phase of processing, re-reading the *makefile*. This forms a natural synchronization point that `make` automatically obeys.

Once the *makefile* is reread with the dependency information, `make` can continue the build process in parallel again. This time `make` chooses to compile all the object files before building each of the archive libraries. This order is nondeterministic. That is, if the *makefile* is run again, it may be that the *libcodec.a* library might be built before the *playlist.c* is compiled, since that library doesn't require any objects other than *codec.o*. Thus, the example represents one possible execution order amongst many.

Finally, the program is linked. For this *makefile*, the link phase is also a natural synchronization point and will always occur last. If, however, the goal was not a single program but many programs or libraries, the last command executed might also vary.

Running multiple jobs on a multiprocessor obviously makes sense, but running more than one job on a uniprocessor can also be very useful. This is because of the latency of disk I/O and the large amount of cache on most systems. For instance, if a process, such as `gcc`, is idle waiting for disk I/O it may be that data for another task such as `mv`, `yacc`, or `ar` is currently in memory. In this case, it would be good to allow the task with available data to proceed. In general, running `make` with two jobs on a uniprocessor is almost always faster than running one job, and it is not uncommon for three or even four tasks to be faster than two.

The `--jobs` option can be used without a number. If so, `make` will spawn as many jobs as there are targets to be updated. This is usually a bad idea, because a large number of jobs will usually swamp a processor and can run much slower than even a single job.

Another way to manage multiple jobs is to use the system load average as a guide. The load average is the number of runnable processes averaged over some period of time, typically 1 minute, 5 minutes, and 15 minutes. The load average is expressed as a floating point number. The `--load-average` (or `-l`) option gives `make` a threshold above which new jobs cannot be spawned. For example, the command:

```
$ make --load-average=3.5
```

tells make to spawn new jobs only when the load average is less than or equal to 3.5. If the load average is greater, make waits until the average drops below this number, or until all the other jobs finish.

When writing a *makefile* for parallel execution, attention to proper prerequisites is even more important. As mentioned previously, when --jobs is 1, a list of prerequisites will usually be evaluated from left to right. When --jobs is greater than 1, these prerequisites may be evaluated in parallel. Therefore, any dependency relationship that was implicitly handled by the default left to right evaluation order must be made explicit when run in parallel.

Another hazard of parallel make is the problem of shared intermediate files. For example, if a directory contains both *foo.y* and *bar.y*, running yacc twice in parallel could result in one of them getting the other's instance of *y.tab.c* or *y.tab.h* or both moved into its own *.c* or *.h* file. You face a similar hazard with any procedure that stores temporary information in a scratch file that has a fixed name.

Another common idiom that hinders parallel execution is invoking a recursive make from a shell for loop:

```
dir:
        for d in $(SUBDIRS);       \
        do                         \
          $(MAKE) --directory=$$d; \
        done
```

As mentioned in the section "Recursive make" in Chapter 6, make cannot execute these recursive invocations in parallel. To achieve parallel execution, declare the directories .PHONY and make them targets:

```
.PHONY: $(SUBDIRS)
$(SUBDIRS):
        $(MAKE) --directory=$@
```

Distributed make

GNU make supports a little known (and only slightly tested) build option for managing builds that uses multiple systems over a network. The feature relies upon the Customs library distributed with Pmake. Pmake is an alternate version of make written in about 1989 by Adam de Boor (and maintained ever since by Andreas Stolcke) for the Sprite operating system. The Customs library helps to distribute a make execution across many machines in parallel. As of version 3.77, GNU make has included support for the Customs library for distributing make.

To enable Customs library support, you must rebuild make from sources. The instructions for this process are in the *README.customs* file in the make distribution. First, you must download and build the pmake distribution (the URL is in the README), then build make with the --with-customs option.

The heart of the Customs library is the customs daemon that runs on each host participating in the distributed make network. These hosts must all share a common view of the filesystem, such as NFS provides. One instance of the customs daemon is designated the master. The master monitors hosts in the participating hosts list and allocates jobs to each member. When make is run with the --jobs flag greater than 1, make contacts the master and together they spawn jobs on available hosts in the network.

The Customs library supports a wide range of features. Hosts can be grouped by architecture and rated for performance. Arbitrary attributes can be assigned to hosts and jobs can be allocated to hosts based on combinations of attributes and boolean operators. Additionally, host status such as idle time, free disk space, free swap space, and current load average can also be accounted for when processing jobs.

If your project is implemented in C, C++, or Objective-C you should also consider distcc (*http://distcc.samba.org*) for distributing compiles across several hosts. distcc was written by Martin Pool and others to speedup Samba builds. It is a robust and complete solution for projects written in C, C++, or Objective-C. The tool is used by simply replacing the C compiler with the distcc program:

```
$ make --jobs=8 CC=distcc
```

For each compilation, distcc uses the local compiler to preprocess the output, then ships the expanded source to an available remote machine for compilation. Finally, the remote host returns the resulting object file to the master. This approach removes the necessity for having a shared filesystem, greatly simplifying installation and configuration.

The set of worker or *volunteer* hosts can be specified in several ways. The simplest is to list the volunteer hosts in an environment variable before starting distcc:

```
$ export DISTCC_HOSTS='localhost wasatch oops'
```

distcc is very configurable with options for handling host lists, integrating with the native compiler, managing compression, search paths, and handling failure and recovery.

ccache is another tool for improving compilation performance, written by Samba project leader Andrew Tridgell. The idea is simple, cache the results of previous compiles. Before performing a compile, check if the cache already contains the resulting object files. This does not require multiple hosts, or even a network. The author reports a 5 to 10 times speed up in common compilations. The easiest way to use ccache is to prefix your compiler command with ccache:

```
$ make CC='ccache gcc'
```

ccache can be used together with distcc for even greater performance improvements. In addition, both tools are available in the Cygwin tool set.