SPEC CPU2006 Benchmark Tools

Cloyce D. Spradling
Systems Group, Sun Microsystems
Release Manager, SPEC CPU Subcommittee
Contact: cloyce.spradling@sun.com

Introduction

The benchmarks that make up the SPEC CPU2006 benchmark suite are set-up, run, timed, and scored by the CPU tools harness. The tools have evolved over time from a collection of edit-it-yourself makefiles, scripts, and an Excel spreadsheet to the current Perl-based suite. The basic purpose of the tools is to make life easier for the benchmarker; they make it easier to tweak compilation settings, easier to keep track of those settings, and most importantly, they make it easier to follow the run and reporting rules.

This paper will explain how the tools work to provide services for users who want to generate benchmark scores and also for those who want to conduct research using the SPEC CPU benchmarks. The course of a normal, score-generating run is followed from setup to report generation, expanding on some sections such as timing that have never before been documented. A couple of new features designed to make life easier for users of the CPU2006 benchmark suite are discussed. Features aimed at researchers are also explored, such as how modified benchmark sources may be tested, ways to work around the tools when they get in your way, and how the tools can help with profiling workloads.

Throughout this paper, you will see references to `$SPEC`. This is just a shorthand way to refer to the top of your CPU2006 installation. If you are a Unix user, you can use this directly as it is the name of an environment variable set by `shrc` or `cshrc`. The Windows equivalent is `%SPEC%`, and is set by `shrc.bat`. The overall directory structure is the same on all platforms.

A Normal Run

The SPEC CPU benchmarks exist primarily to measure the performance of computer systems in a consistent fashion; as such, we consider a "normal" run to be one where the tools are used to build, run, and report on a benchmark or set of benchmarks that have not been modified. Only such a run may produce a publishable score. In this case, all that is necessary is to provide some compilation options (if the benchmarks are not pre-built), run the benchmarks and generate a result. This section describes how a normal run works; a later section discusses some features that can be used when more than this basic level of functionality is desired.

Building the Benchmarks

The CPU2006 benchmarks are supplied as source code, and must be compiled before they can be used to generate a result. Compilation settings including the name of the compiler, compiler flags, and build method are all specified by a configuration file.[1]

In the general case, the setup of a build directory is straightforward. The tools create a working directory for the build if one does not already exist, and copy the source files from the benchmark's `src/` directory. After the copy, the MD5 checksum of each file is checked after it has been copied; if there is a mismatch, the build is aborted. This is done primarily to catch what would be very difficult-to-reproduce errors caused by bad RAM or disks, and also guards against the use of modified sources.

After the files are copied, the tools write the settings found in the configuration file to `Makefile.spec` in the working directory. This file only contains settings for variables; all of the actual compilation rules used are in the file `$SPEC/benchspec/Makefile.defaults`. These rules are the same for all benchmarks.

The tools then invoke `specmake`, which is a slightly modified version of GNU make, to build the benchmark binary. If the compilation completes successfully, the resulting executable is copied into the benchmark's `exe/` directory for future use. The options used to build the benchmark, the MD5 sum of those options, and the MD5 sum of the executable itself are all stored back into the configuration file. This allows the tools to automatically determine when a benchmark needs to be rebuilt based on whether its options have changed. The MD5 checking feature was added to CPU2000; prior to that, it was not possible to know with certainty that a particular set of binaries were generated with a particular set of options.

Running the Benchmarks

When a run of any type is requested, the tools check to see if a binary already exists and if so, that the compilation settings used for it have not been changed. Unless explicitly disallowed by the use of the `--nobuild` flag or the `nobuild` configuration file setting, the benchmark will be rebuilt if either of the above conditions are not true.

The tools will check to see if a suitable working directory for the run already exists under the benchmark's `run/` directory, and create one if it does not already exist. The executable is copied from the benchmark's `exe/` directory, and the input files are copied from the appropriate workload directories under the benchmark's `data/` directory. As with the source files, and for the same reasons, the MD5 sums of all executable and all the input files are verified after copying.

Each benchmark has an `invoke()` method specified in its `object.pm` file which provides information about how to run the benchmark. The information is written to `speccmds.cmd` in the working directory and the tools call `specinvoke` to actually run and time the benchmark. Various utilities mentioned in this article, such as `specinvoke`, are described in utility.html [6].

Timing the Benchmarks

One measured iteration of a benchmark for which an elapsed time is reported may actually consist of several invocations of the benchmark binary with different inputs and outputs. For example, the binary for 450.soplex is run twice during a run using the reference workload; once with `pds-50.mps` as the input file, and once with `ref.mps`.

Thus the elapsed time of one iteration is the sum of the elapsed times for each run of the benchmark binary. That is, the timer starts when the binary begins execution and ends when it completes. This time includes time needed to load and begin execution of the binary, benchmark I/O time for reading inputs and writing results, and calculation time.

The elapsed time reported for a benchmark iteration will never exactly match the wall-clock time that the iteration takes, because the tools also validate the benchmark's output, and the time required for the validation is **not** included in the reported time.

Generating Reports

Once all of the runs are finished, the tools generate reports. In all cases, a "raw" result file is generated. This is really the only required output format; with a copy of the raw file, the tools can generate any of the supported report formats.

The raw file also contains copies of the configuration file used to generate the result and lists of all options used to build the benchmarks. The stored configuration file can be retrieved using the `extract_config` tool. [6]

If a raw file is not available, but an HTML, PostScript, or PDF format result is, the raw file can be extracted from that using the `extract_raw` tool. [6] Thus, in most situations the raw file and all of the ancillary information that it contains can be retrieved.

There are also a couple of "output formats" which do not actually generate a report of their own. The first is "subcheck", which runs as a formatter but is really more of a verification script. It checks the informational fields in a result and reports on the elements that must be changed before a result will be accepted by SPEC for review and publication. It checks for proper units on fields that must have units, mandatory fields that are blank, and other criteria required for a submission to SPEC. A "PASS" from subcheck is no guarantee that a result will pass review, but it is a guarantee that the SPEC result handler will not automatically reject your submission because of the contents of the informational fields of your result.

The other formatter that generates no report of its own is the "mailto" formatter. As the name suggests, it can be used to mail the results of a run to a specified list of addresses. It requires some setup, most of which will go into the configuration file. The primary use is when running a long test: at the end, you will be notified by email that the test is complete. Because the options can not be specified on the command line, it really is not much use when re-formatting results that have been already run. For more information on how to set up report mailing, see the configuration file documentation.

Going Beyond the Numbers

Not all CPU2006 users are interested solely in the scores that a particular system can achieve given a set of benchmark binaries and some time. Some may be interested in the characteristics of a particular benchmark, while others may be concerned with getting better optimization or flag sets. Still others may need to do runs on a wide range of systems using the same binaries. The tools have features to address all of these needs.

Configuration Management

One of the toughest parts of preparing a result for publication in any venue is getting the informational fields correctly filled in. While reviewing the results that are published on the SPEC web site, the review committee has seen many instances of incorrect hardware descriptions. Such errors can be difficult to spot because in many cases it is necessary to be extremely familiar with the nuances of a particular product line to spot these errors.

With CPU2006, it is possible to avoid these errors altogether. The problem of being able to determine the hardware configuration on any platform under any operating system is not one that we attempted to solve. However, no user runs on all operating systems on all platforms. The problem of automated hardware description is tractable when the scope of the problem is narrowed to one operating system.

As with several other difficult problems with which the CPU tools attempt to deal, we enlist the help of the user. The tools do not attempt to do any determination of the system configuration themselves, but they do provide a facility for running a user-provided program which can query the operating system and hardware and dynamically insert that information into the configuration file being used.

For example, the following script could be used on some Linux systems to fill in information about the number of CPU cores enabled:

```
#!/bin/sh

CORES=`grep proc /proc/cpuinfo | wc -l`
echo hw_ncores = $CORES
```

If this script was called `$SPEC/linux_sysinfo`, you could use it by inserting the following into the header section of your configuration file:

```
sysinfo_program = $[top]/linux_sysinfo
```

It is likely that *some* human intervention will be required in order to fill in details and polish names, but the use of this facility should be able to eliminate the grossest errors.

For more information, see the section on `sysinfo_program` in the configuration file documentation, and the sample sysinfo program.[2]

Documenting Compilation Options

Another error-prone part of result preparation is the documentation of the compilation options used. Checking compilation options is also one of the most tedious parts of

reviewing a result for publication. It is not enough to just verify that only the options listed were used, but it is also necessary to ensure that those options are documented. Prior to CPU2006, this tedium was foisted on the result reviewer – the one with the least opportunity to effect change when necessary! The problem was compounded by the "flag counting" rules in CPU95 and CPU2000 which restricted baseline optimization to four flags. The rule itself was not the problem; it was the counting. Some flags are easy. "-O" is clearly one flag. But what about "-qxinline=27 :bob:88:fastturnaround"? Is it one flag or four? The exercise (as usual) is left to the reader.

As a result of the tedium and the difficulty of counting, automated flag auditing was one of the most requested features for CPU2006. As with hardware configuration determination, flag auditing is a difficult problem. Once again, the tools enlist the help of the user. By providing a flag description file[3] with regular expressions to match flags and documentation of those flags, the user enables the tools to provide a complete and accurate report of the flags used for each benchmark, as well as a separate report with the documentation for those flags. In result formats that support linking (currently HTML and PDF), the flags listed in the result page link to the documentation for the flag.

Creating a flag description file from scratch is a non-trivial exercise. Fortunately, it will probably not ever be necessary, as there are a couple of functional examples [4] provided with CPU2006 itself. Additionally, the flag description files used for the results published at SPEC are also publicly available, [5] and together these cover the most common compilers in use today. Each result page contains the URL for the flag description file used for that result.

Sharing the Installation

Both CPU95 and CPU2000 did all benchmark builds and runs in directories under the top-level installation directory. On many systems, this poses a problem when attempting to share one installation among several users. In that situation, it is necessary for everyone to have write access to all parts of the installation. That is not always desirable or even possible.

CPU2006 has a couple of features to address this situation.

For groups where sharing write access is not a problem, for example when everyone runs the benchmarks as "root" (which is not a recommended practice), there is just a need to keep one user's runs and results separate from another. Each user can, in his configuration file, specify an `expid` (short for "experiment ID"). This will cause run directories and result and log files to be deposited in a subdirectory named for the value of `expid` underneath the normal directory where those files would normally appear. For example, result files are normally written to `$SPEC/result/`. With an `expid` setting of "goof-off", results would be written into `$SPEC/result/goof-off/`.

A better way to share a benchmark tree does not require giving everyone write access to the whole tree. Instead, the `output_root` feature can be used to put all writable directories into a completely separate hierarchy. Because the configuration file must be updated when a build is completed, and because the output root can only be specified in a configuration file, the configuration files still live underneath the main install tree, and the "config" directory still needs to be writable by all users. On Unix systems, the potential for mischief can still be mitigated by setting the sticky bit on `$SPEC/config/`, and permissions on individual configuration files can be set to prevent reading by other users if necessary.

Users using the output root feature may be interested in the `ogo` shell alias [6]. Like `go`, it provides a quick way to jump to various places in the benchmark tree. Unlike `go`, it uses the `GO` environment variable to point to the base of an output root. The `ogo` alias knows which parts of a tree are always found under `$SPEC` and which go in an output root, and can navigate accordingly.

Modifying the Sources

Those interested in optimization techniques often want to try source code changes. Unfortunately, the same tools features that ensure that no files have been corrupted can make this difficult.

However, by using the mechanism already in place for using approved source changes, ("src.alt" in SPEC parlance) it is easy to work around this particular feature.

The process is simple, but heretofore undocumented. Under the benchmark's `src/` directory, create a subdirectory named `src.alt/` and another under that with a short name for your modification. For this example, we wish to test some changes where we have re-rolled some hand-unrolled loops in 999.specrand. For something like that, a short name like "rerolled" would be appropriate. The first step would be to create the directory hierarchy `src/src.alt/rerolled/` under the 999.specrand benchmark directory.

Into this directory go the modified source files. Additionally, it is necessary for a `README` file to be present, though it does not need to actually contain anything.

Running the `makesrcalt` utility will cause the differences of the modified files and the existing files to be taken and a control file written. For this example, the invocation would be

```
$ makesrcalt 999.specrand rerolled
```

This command will need to be re-run whenever the sources are modified.

To use the newly generated src.alt, it is necessary to put a line in the configuration file for the benchmark in question. For no good reason, the configuration file tag is "srcalt", and the value to use is the short name that you gave to the directory above. (CPU2006 V1.1 will support the more intuitive configuration file tag "src.alt".) The configuration file addition for our example src.alt would be

```
999.specrand:
srcalt = rerolled
```

Subsequent builds of 999.specrand will use the modified sources.

Note that in general, results generated from binaries built with modified sources may not be published. The only exception is for SPEC-approved alternate sources which address portability concerns in a performance-neutral way. For more information about the rules relating to source changes in CPU2006, please see the Run and Reporting Rules [7].

Figure 1: Usage of the -n option for specinvoke

```
$ specinvoke -n
# Use another -n on the command line to see chdir commands
# Starting run for copy #0
../run_base_ref_none.0000/specrand_base.none 1255432124 234923 > rand.234923.out 2>> rand.234923.err
```

Modifying the Workloads

There are no features to aid in modifying the workloads. If testing modified workloads is your goal, you will need to work around the safety features in the tools. You can still use the tools to prepare your work area and run your experiments. You could also use them to test your outputs, but chances are that if the inputs are changing, the supplied outputs will not match anyway. You also will not be able to generate a report.

The first step is to get a run directory set up. You can do this by using the `runspec "setup"` action. It will make a directory (if necessary), copy your benchmark executable into the directory, copy the input data into the directory, and create the `specinvoke` control files for running the benchmark and also for validating the output. In other words, it does everything it would for a normal run, stopping just before actually executing the benchmark. Here is an example command to set up a run directory for the reference workload of 400.perlbench:

```
$ runspec --action setup --input ref \
  --config macosx.cfg 400.perlbench
```

The above command will build a benchmark executable if necessary, set up the run directory, and exit. It is then up to you to navigate there. The `go` shell alias comes in handy in this case:

```
$ go 400 run
```

This will take you to where the run directories for 400.perlbench are stored. From there you will see the directory that `runspec` prepared for you.

Inside the run directory, you will find several files of interest. If you want to use `specinvoke` to run and time your experiments, pay special attention to `speccmds.cmd`. This file contains specifications of each invocation for one benchmark iteration, as well as specifications for input and output redirection. By modifying this file, you can change which invocations happen, where the outputs go, and which executable is used.

Of course, it is not necessary to use `specinvoke`; you may simply wish to use it to dump out the commands that would normally be issued and then run those by hand. In that case, run "`specinvoke -n`" in the run directory. See Figure 1 for an example. If you are using a Bourne-compatible shell, executing the commands from Figure 1 will have exactly the same effect as running the benchmark with `specinvoke`.

There is more information on avoiding the use of the tools, including different scenarios and many more examples in the CPU2006 documentation.[8]

Watching the Runs

Running code or hardware profilers in conjunction with the benchmarks is a common activity. The CPU2006 tools have all of the monitoring capabilities that the CPU2000 tools did, in largely unchanged form. Unfortunately, the documentation for these features was not completed in time for the release of CPU2006 V1.0. If you used the monitor hooks in CPU2000, your knowledge should still be good. Documentation for these features is planned for a future update of the suite.

In the meantime, here is a short primer on the dedicated monitoring facilities in CPU2006. There are several hooks which provide opportunities to run monitoring applications at various points before, during, and after a run. All of these hooks are set in the configuration file, and may either be set globally (in the header section) or on a per-benchmark basis.

The monitoring hooks called `monitor_pre_bench` and `monitor_post_bench` allow programs to be executed before and after `specinvoke` is called to run the benchmark. This could be used to harvest files written by an instrumented binary or to start and stop a system-level profiler. As an example, Figure 2 has the settings which were used to collect branch and basic block data presented in Darryl Gove's paper on workload correspondence.[9] Before the run (`monitor_pre_bench`), the Binary Improvement Tool (`bit`) [10]) is used to instrument the benchmark executable. After the run (`monitor_post_bench`), `bit` is used again to dump statistics from the run into files in the `$SPEC/analysis/` directory.

The values for `$commandexe` and `$size` are provided by the tools at run time; there are many others available. Exactly which are available vary depending on what is being done – for a list of the variables available for substitution, execute your `runspec` command with the verbosity set to 35

Figure 2: Usage of the monitor hooks

```
monitor_pre_bench = bit instrument ${commandexe}; cp ${commandexe} ${commandexe}.orig; \
                    cp ${commandexe}.instr ${commandexe}

monitor_post_bench = bit analyze -o $[top]/analysis/branches.${benchmark}.${size}.csv \
                                 -a branch ${commandexe}; \
                     bit analyze -o $[top]/analysis/blocks.${benchmark}.${size}.csv \
                                 -a bbc ${commandexe}; cp ${commandexe}.orig ${commandexe}
```

Figure 3: Two ways of using the monitor hooks with `strace`

```
monitor_specrun_wrapper = strace -ff -o $benchmark.calls     $command; \
                          mkdir -p $[top]/calls.$lognum; mv $benchmark.calls* $[top]/calls.$lognum

monitor_wrapper         = strace -f -o $benchmark.calls.\$\$ $command; \
                          mkdir -p $[top]/calls.$lognum; mv $benchmark.calls.\$\$ $[top]/calls.$lognum
```

or greater. This can be accomplished by specifying "-v 35" as arguments on the `runspec` command line.

By using `monitor_specrun_wrapper`, it's also possible to directly monitor `specinvoke`, and by extension the entire benchmark iteration, no matter how many separate executions that involves. For example, to generate a system call trace for `specinvoke` and all its children on a Linux system, you could use the first example in Figure 3. In that example, the crucial point is `$command`; it expands to the full command to be run, including arguments. As you might guess, if `$command` is omitted or replaced, something other than the desired command will be traced, and the benchmark run will not validate.

The execution time for the commands specified by `monitor_pre_bench`, `monitor_post_bench`, and `monitor_specrun_wrapper` are not included in the benchmark's reported time.

It is also possible to instrument each invocation of a benchmark binary using `monitor_wrapper`. This allows profiling or tracing of the individual workload components without measuring the overhead of `specinvoke`. One potential disadvantage is that the execution time of these commands *is* counted as part of the benchmark's reported run time. The second example in Figure 3 saves each output file directly into the profile directory. This results in the same profile files being stored in `$SPEC/calls.$lognum`, with the exception that there is no output file for `specinvoke`.

One very important point to note about `monitor_wrapper` is that by default any output that the monitoring software writes to stdout will be mixed with the benchmark's output. A setting such as the following will cause many of the benchmarks to fail validation:

```
monitor_wrapper = date; $command
```

The benchmark's expected output certainly does not include the current time of the run. This is a contrived example of a real problem that can occur even if the monitoring program outputs only benign static status information. This is because normally the output redirections are set up by `specinvoke` before executing the benchmark. A similar problem can occur with input if the monitoring application consumes input that the benchmark expects to find on stdin.

The way around this problem is a configuration file switch called `command_add_redirect`. Normally input and output files are opened by `specinvoke` and attached directly to the new process' file descriptors. Setting `command_add_redirect` in the header section of the configuration file causes that step to be skipped and instead modifies the benchmark command to include shell redirection operators. So, in Bourne shell syntax, by default the above example translates to something like

```
(date; $command) < in > out 2>> err
```

With `command_add_redirect` set, this becomes

```
date; $command < in > out 2>> err
```

The output from the date command is still stored, but in a file in the run directory that is not subject to validation.

The monitoring facilities in CPU2006 are not limited to the simple examples presented here. Scripts and programs of any level of complexity may be used to examine command arguments and executables and change their monitoring actions accordingly. In this way only a specific part of a benchmark's workload may be monitored while not wasting time or other resources tracing uninteresting workloads.

Conclusion

This is just a short overview of the operation of and new features in the SPEC CPU2006 toolset. With the exception of the timing description, the src.alt creation process, and the monitoring facilities, everything is documented in much more detail in the CPU2006 documentation set, which is publicly available at http://www.spec.org/cpu2006/Docs/ .

Acknowledgements

Thanks go to the members of SPEC's CPU subcommittee for suggesting many of the new features detailed here. Special thanks go to Miriam Blatt, Darryl Gove, and John Henning of Sun Microsystems, Rick Jones of Hewlett-Packard, and Mat Colgrove of The Portland Group for their critical feedback.

References

[1] "SPEC CPU2006 Config Files", http://www.spec.org/cpu2006/Docs/config.html
[2] "Sample sysinfo program", http://www.spec.org/cpu2006/Docs/sample-sysinfo-program.pl
[3] "CPU2006 Flag Description Format", http://www.spec.org/cpu2006/Docs/flag-description.html
[4] Sample flags files are in the "Docs/flags" subdirectory of the CPU2006 distribution
[5] See http://www.spec.org/cpu2006/flags/
[6] "SPEC CPU2006 Utility Programs", http://www.spec.org/cpu2006/Docs/utility.html
[7] "SPEC CPU2006 Run and Reporting Rules", www.spec.org/cpu2006/Docs/runrules.html#rule_1.2.1
[8] "Runspec Avoidance", www.spec.org/cpu2006/Docs/runspec-avoidance.html
[9] Darryl Gove, Lawrence Spracklen, "Evaluating the correspondence between training and reference workloads in SPEC CPU2006", Computer Architecture News, Vol. 35, no. 1, March 2007
[10] "Cool Tools - Binary Improvement Tool (BIT)," http://cooltools.sunsource.net/bit/.