

SPECrate2006: Alternatives Considered, Lessons Learned

John L. Henning

Sun Microsystems

j.henning at computer.org

Abstract. Since 1992, SPEC has used multiple identical benchmarks to measure multi-processor performance. This “Homogeneous Capacity Method” (aka “SPECrate”) has been criticized on the grounds that real workloads are not homogeneous. Nevertheless, SPECrate provides a useful window into how systems perform when stressed by multiple requests for similar resources. This paper reviews SPECrate’s history, and several performance lessons learned using it: (1) a 4:1 performance gain for startup of a benchmark when I/O was reconfigured; (2) a benchmark that improved up to 2:1 when a TLB data structure was re-sized; and (3) a benchmark that improved by 52% after a change to NUMA page allocation. The SPEC CPU workloads usefully exposed several opportunities for performance improvement.

1 Introduction: A Philosophy of Divots

When systems do not perform as expected, performance anomalies are sometimes called “divots”: an unexpected hole where performance sinks. Is a divot something to be ashamed of? Or an opportunity? This tester suggests:

Although it is widely understood that “all software has bugs”, it may not be as widely understood that all systems have performance divots. A repeatable, analyzable workload allows divots to be analyzed. Cherish your divots.

2 Background: About SPECrate

The Original Metric: Speed. The SPEC CPU suites are made up of component benchmarks. The original SPECmark (now referenced as “CPU89”) contained 10 benchmarks, such as gcc, spice, and a lisp interpreter; the most recent suite, CPU2006, contains 29 benchmarks, such as bzip2, GNU Go, GAMESS, POV-Ray, and perl. The SPEC-supplied tool set runs each component benchmark individually, and the time in seconds is reported. For each benchmark, a “SPECratio” is computed by dividing the time on a reference system by the time seen on the system under test. Finally, a bottom line metric (such as SPECint95, SPECfp2000, SPECfp_base2006) is computed as the geometric mean of the benchmark SPECratios.

The bottom line metrics mentioned thus far are called “speed” metrics, and are analogous to speed of travel in the real world in that higher numbers are better, and numbers are comparable. If a sports car takes 1/4 the time to get to Cleveland as a truck, we routinely say that the sports car is 4x as fast; and if a new laptop finishes a well defined task in 1/4 as much time as an old desktop computer, it seems natural to call the laptop 4x as fast as the desktop.

Adding a Throughput Metric. The speed tests run only one copy of each component benchmark at a time, leaving resources idle on multi-processor systems. SPEC addressed this problem in 1992 by adding throughput tests that allow the tester to run multiple copies of identical benchmarks. For example, in a 32-copy SPECint_rate2006 test, the SPEC tool set starts 32 copies of 400.perlbench, waits for all of them to complete; records the time from start of first to finish of last; then starts 32 copies of 401.bzip2, and so forth.

The fact that all copies are running the same workload is the reason that SPECrate was originally known as the “Homogeneous Capacity Method” [1].

The details of the metric calculation have varied somewhat as the suites have evolved, but in all cases a score is calculated for each benchmark which is proportional to the number of copies run divided by the time required to complete the copies. The bottom line metrics (e.g. SPECint_rate95, SPECfp_rate2000) are the geometric means of the benchmark scores.

Interpretation of SPEC CPU throughput metrics is somewhat less intuitive than the speed metrics. For example, if a laptop has a SPECint_rate2006 score of 10, and a server has a SPECint_rate2006 score of 20, it is not immediately obvious if the better result is achieved by running twice as many copies in the same time, or by running the same number of copies in 1/2 the time, or by some other method. The full reports provide the additional level of detail for the motivated reader.

Positioning: A Component Benchmark. Although the throughput metrics exercise more of the system than a single processing unit, while using the compute-intensive portion of real applications [4], it should be understood that SPECrate is not positioned as a whole-system benchmark. It is a design goal to reduce disk I/O, remove network I/O, and eliminate GUIs from the SPEC CPU suites. Use of system services and libraries are also minimized [19].

3 Perceived Weaknesses of SPECrate

Scaling. SPECrate has been criticized because the scaling can sometimes appear to be less than credible. For example, on the metric SPECint_rate_base2000, an Alpha 21264A with 16 chips vs. 32 scales at .975 [6]; an SGI R14000 with 8 chips vs. 128 scales at .976 [7].

Explanations for the excellent scaling include: (1) SPECrate jobs are independent – there are no stalls for cross-job communication. (2) As mentioned

above, they do little IO and use few system services. (3) Although one of the design goals is to exercise the memory hierarchy, it has been shown that the benchmarks usually exercise only a relatively small part of main memory at a time, [3] and therefore caches are usually effective. (4) Even for the benchmarks that exercise main memory the most, there are often access patterns that compilers and hardware can usefully prefetch. (5) Although anyone can publish a rule-compliant SPEC CPU result, in fact, most publications are done by vendors, who are motivated to ensure that systems are properly set up to ensure good scaling. (6) If good scaling is not possible on a particular system, there is typically no particular motivation for the vendor to publish such a result.

In short, a concern with the scaling is that it may appear to be “too good”. Customer applications that depend on interprocess coordination, system services, I/O, or other components not measured by the SPEC CPU benchmarks are unlikely to scale as well as does SPECrate. This is not to say that SPECrate is a dishonest measurement; rather that it is a component benchmark, and that it needs to be understood as such.

Homogeneity and Convoys. A second perceived weakness of SPECrate is its homogeneity. In real life, servers often host a variety of applications. Even in an environment where everyone has a common interest (say, all are molecular chemists, running a common tool set) jobs do not all start at the same instant, and run identical workloads.

It has been hypothesized that SPEC’s implementation of SPECrate may lead to “convoy effects”: for example, 128 memory-intensive programs all hit their most intense memory bandwidth demand at the same time; or 128 programs all try to read their startup data at the same time, thrashing the disk; or 128 programs all try to acquire an OS lock on the filesystem at the same instant.

4 Alternatives Considered by SPEC

In response to concerns about SPECrate, the SPEC CPU Subcommittee has considered various alternatives over the years. Two alternatives are described in this section: “heterogeneous” and “staggered homogeneous”.

4.1 Heterogeneous

During the development of SPEC CPU2006, a prototype was implemented that ran the CPU2000 jobs in a heterogeneous fashion. Tables 1 and 2 show the difference in run order on a system running 4 queues (which would, typically, use 4 processors).

For the homogeneous method, each processor runs the same program and workload.

For the heterogeneous prototype, each processor starts off running a different job than the other processors (provided that the number of processors is less than the number of benchmarks in the suite).

Table 1. Homogeneous run order. All processors run identical jobs.

Queue 0	Queue 1	Queue 2	Queue 3
164.zip	164.zip	164.zip	164.zip
175.vpr	175.vpr	175.vpr	175.vpr
176.gcc	176.gcc	176.gcc	176.gcc
181.mcf	181.mcf	181.mcf	181.mcf
186.crafty	186.crafty	186.crafty	186.crafty
197.parser	197.parser	197.parser	197.parser
252.eon	252.eon	252.eon	252.eon
254.gap	254.gap	254.gap	254.gap
253.perlbmk	253.perlbmk	253.perlbmk	253.perlbmk
255.vortex	255.vortex	255.vortex	255.vortex
256.bzip2	256.bzip2	256.bzip2	256.bzip2
300.twolf	300.twolf	300.twolf	300.twolf

It is important to note that for homogeneous SPECrate, all copies of a benchmark finish, and then the next benchmark is started. Thus one may read Table 1 as implicitly containing 12 phases: between each row there is a pause to wait for all of the row to finish. No such pause occurs with Table 2. In the heterogeneous prototype, each queue runs independently.

Table 2. Heterogeneous run order. Different processors run different jobs.

Queue 0	Queue 1	Queue 2	Queue 3
164.zip	175.vpr	176.gcc	181.mcf
175.vpr	176.gcc	181.mcf	186.crafty
176.gcc	181.mcf	186.crafty	197.parser
181.mcf	186.crafty	197.parser	252.eon
186.crafty	197.parser	252.eon	254.gap
197.parser	252.eon	254.gap	253.perlbmk
252.eon	254.gap	253.perlbmk	255.vortex
254.gap	253.perlbmk	255.vortex	256.bzip2
253.perlbmk	255.vortex	256.bzip2	300.twolf
255.vortex	256.bzip2	300.twolf	164.zip
256.bzip2	300.twolf	164.zip	175.vpr
300.twolf	164.zip	175.vpr	176.gcc

Results. Informal (mostly non-quantitative) reports of results with the heterogeneous prototype fell into two categories. Some reports indicated only minor differences in observed run times, within the usual range for run-to-run variation. Others said that benchmarks with noticeable main memory traffic ran

noticeably faster, presumably because they tended to compete with less intense jobs, rather than with equally intense copies of themselves. Therefore, the likely bottom line with a heterogeneous method would be slightly better scaling than with the homogeneous method. The possibility that scaling would improve may be seen as a negative aspect of the heterogeneous method, if one is concerned that homogeneous scaling already appears to be “too good”.

It seems intuitive that any particular resource stressed by a homogeneous workload would be less stressed by the above heterogeneous method.¹

Difficulties with the heterogeneous method. On the assumption that such resource stresses are useful to study, reducing their levels in a heterogeneous workload is bad, because it makes them less apparent and harder to analyze. A heterogeneous workload also makes it much more difficult to reproduce performance conditions. For example, suppose that 255.vortex runs more slowly than desired. To reproduce its conditions from Table 1, to a first approximation, one can simply run the 4 copies of vortex. To reproduce its conditions from Table 2, it is necessary to run the whole suite. One cannot try to just run selected “rows”, because the rows in Table 2 do not represent separate phases.

4.2 Staggered Homogeneous

Another alternative prototyped by SPEC delays the start of each job by a small amount (a “stagger”), while running the same job on all processors. The intent of the staggered homogeneous method is to avoid the hypothesized convoy effects mentioned above. The prototype still exists, latent and unsupported, in SPEC CPU2006. The excerpts below are taken from an unmodified copy of the suite:

```
$ specinvoke -h
    -S msec    sleep between spawning copies (in milliseconds)

$ runspec --stag
Option stag is ambiguous (stagger, staggeredhomogenousrate)

$ runspec --config oct14a --size test \
    --copies 2 --staggeredhomogen --stagger 6000 473.astar
```

The `specinvoke` [11] utility provides a help message that tells us that staggers are expressed in milliseconds. The first `runspec` command tricks the switch parser into reminding us how to spell its undocumented switches, and the second `runspec` command runs 2 copies of the test workload for the benchmark 473.astar, with a delay of 6 seconds between each copy.

As a reminder, the staggered homogeneous prototype is **unsupported**. If the reader plays with it, you are reminded that anything you learn from it cannot

¹ With the notable exception of hardware and OS support for the instruction stream. For SPECrate, each copy has its own data, but all use the same program binary, allowing the OS the opportunity to load only one copy into physical memory. In a heterogeneous context, obviously, multiple program binaries are active.

be represented as an official SPEC metric. If you do decide to use it, you will probably find it easiest to discern what it did by looking in the run directory:

```
$ cd $SPEC/benchspec/CPU2006/473.astar/run/run*000
$ cat speccmds.out
timer ticks over every 1000 ns
running commands in speccmds.cmd 1 times
runs started at 1225226364, 29870000, Tue Oct 28 16:39:24 2008
run 1 started at 1225226364, 29876000, Tue Oct 28 16:39:24 2008
child started: 0, 1225226364, 29883000, pid=3147,
  './run_base_test_oct14a.0000/astar_base.oct14a lake.cfg'
child started: 1, 1225226370, 30218000, pid=3148,
  './run_base_test_oct14a.0000/astar_base.oct14a lake.cfg'
child finished: 0, 1225226376, 980432000, sec=12, nsec=950549000,
  pid=3147, rc=0
child finished: 1, 1225226383, 556000, sec=12, nsec=970338000,
  pid=3148, rc=0
run 1 finished at: 1225226383, 562000, Tue Oct 28 16:39:43 2008
run 1 elapsed time: 18, 970686000, 18.970686000
runs finished at 1225226383, 597000, Tue Oct 28 16:39:43 2008
runs elapsed time: 18, 970727000, 18.970727000
```

Notice above that the two copies were started 6 seconds apart (1225226364 and 1225226370 seconds after Jan. 1, 1970), each took just under 13 seconds, and the total elapsed time was just under 19 seconds. The bottom line includes the time for the stagger, as it is measured from start-of-first copy to finish-of-last. One might want to consider other ways of calculating a bottom line. (Reminder: any use of the prototype may *not* be represented as an official SPEC metric.)

Results. As SPEC experimented with the prototype, the hypothesized convoy effect was not observed. That is, the expectation had been that the normal SPECrate causes unrealistic resource overloads when, for example, 128 copies all try simultaneously to acquire a lock on a filesystem; and that a small stagger (on the order of 10s of milliseconds) would avoid the overloading and actually cause faster overall execution time. Instead, small staggers were observed to make no particular difference to overall time (indistinguishable from noise).

Difficulties with the staggered homogeneous method. Should the metric include the stagger time? If so, unless the staggers are very small, too much idle time may be included. Alternatively, one might try to exclude the staggers by, for example, calculating time from start-of-last to finish-of-first; a disadvantage of this approach is that it could cause performance to be overstated if one copy has more hardware resources than others (e.g. a 16-chip, 64-core system with 4 copies on 15 of the chips, but only 1 copy on the last). Perhaps the most attractive alternative would be to attempt to achieve a steady state of repeated execution, with all processors busy, running staggered workloads; one would compute a metric that sampled execution time for complete jobs during the steady state. The primary disadvantage of this approach is that the suite is sometimes already

criticized as taking too long; running repeated workloads to ramp up to a steady state was not viewed as attractive.

SPEC's Decision. After discussion, neither of the prototyped alternatives was adopted for CPU2006, and SPECrate remains essentially unchanged since 1992.

5 Applying SPECrate2006

SPECrate provides a useful window into how systems perform when stressed by multiple requests for similar resources such as program startup, data initialization, translation lookaside buffer (TLB) requests, and memory allocation. It is understood that in real life, an OS is unlikely to get 128 simultaneous identical requests, so one must be careful not to over optimize to this, or to any other, benchmark. Nevertheless, the homogeneity may be its virtue: in real life, systems do have to deal with intense requests, traffic jams do occur, and SPECrate presents a compute-intensive workload that is repeatable and analyzable.

In this section, three case studies are briefly summarized from applying SPECrate2006 to Solaris systems.

5.1 A 4:1 Performance Gain for Startup of a SPEC CPU2006 Benchmark when I/O Was Properly Configured

Although the intent of SPEC CPU benchmarks are to be compute intensive, some I/O inevitably remains. When multiple copies are run for SPECrate, I/O is magnified. With each suite, it seems that one or two benchmarks stick out as being especially in need of I/O tuning. For CPU95, a benchmark of concern was 126.gcc: each copy compiles 56 input files and writes 112 output files with a total of 8 MB of output data. For CPU2000, the benchmark 200.sixtrack writes 42 files, with a total of 5.3 MB, per copy. For both CPU95 and CPU2000, testers learned that on large systems, it is useful to have striped disks, preferably with journaling file systems that do not stall waiting for writes.

Problem. For CPU2006, a benchmark of concern in large SPECrate runs is 450.soplex, a Simplex Linear Program (LP) Solver. The program is invoked twice, and I/O becomes a problem in startup of part 2, when each benchmark copy needs to read its copy of the 267 MB input file ref.mps.

Methods. In order to focus on the second part of the benchmark, the utility `convert_to_development` [10] was applied to allow modifications to the ref workload while still using the SPEC tools. The first workload was deleted, leaving only ref.mps in the directory 450.soplex/data/ref/input. Then, 128 run directories were populated on a large server using `runspec --action setup`. The actual runs were done using `specinvoke -r` [11]. In order to avoid unwanted file caching effects (which would not be effective in a full reportable run), memory was cleared between tests by running large copies of STREAM [17] and reading a series of unrelated files. CPU and IO activity were observed using `iostat 30`.

Metrics. As each run began, CPU utilization was low, and disk activity high, as 128 copies of ref.mps were read. Eventually, the io kps fell to zero and the tested processors achieved 100% utilization. Two metrics are reported: (1) Startup time in minutes, determined by counting the 2-per-minute iostat records prior to 100% utilization; (2) kps from the busy period (converted to MB/sec).

Baseline. When a single 10K RPM disk was used, startup required about 24 minutes, reading at about 24 MB/sec.

Software RAID. When Solaris Volume Manager was used with the default 16 KB block size (known as an “interlace size” in the terminology of SVM) on an A5200 Fiber Channel disk array with 6x 10K RPM disks, startup fell to about 20 minutes, reading about 30 MB/sec. With a block size of 256 KB, startup improved to about 8 minutes and 72 MB/sec. For this read-intensive workload, RAID-0 was not particularly faster than RAID-5. Increasing the number of disks in the stripe set had little additional effect on performance, as the maximum observed bandwidth for this somewhat older disk system was about 78 MB/sec.

Hardware RAID. A newer hardware RAID Array, the Sun StorageTek 2540 with 6x 15K RPM disks, did not show sensitivity to block size (called “segment size” for this device) over the tested range of 16 KB through 512 KB. This insensitivity may be viewed as a plus, since it may be hard to know in advance what block size to choose. The bandwidth was about 97 MB/sec, roughly matching the limit of the 1 Gb Host Bus Adapter (HBA) used in this test. Once again, read performance was insensitive to use of RAID-0 vs. RAID-5. Further improvement might be possible with a higher bandwidth HBA.

Divot summary. With hardware RAID, a performance divot of idle CPUs waiting on I/O was reduced from 24 minutes to 6 minutes, which is a 4:1 improvement over the original single-disk configuration.

Lessons for tuning other systems. Even in an allegedly CPU intensive environment, IO lurks. Hardware RAID may offload overhead from the server.

5.2 An Improvement of up to 2:1 for a CPU2006 Benchmark When a TLB Data Structure Was Re-sized

UltraSPARC T2. The UltraSPARC T2 (aka “Niagara2”) and UltraSPARC T2 Plus processors [12] are multi-threaded processors with eight SPARC processor cores. Each core runs 8 hardware threads and has 2 integer units, one floating point unit, an 8 KB L1 data cache, and a 16 KB L1 instruction cache. All cores share a single 4 MB L2 cache. Each core does virtual address translation using a 64-entry instruction Translation Lookaside Buffer (TLB) and a 128-entry data TLB [5]. When TLB misses occur, software-managed direct-mapped Translation Storage Buffers (TSBs) are consulted by a Hardware Table Walker. TSBs are allocated per-process, for up to 4 page sizes (8 KB, 64 KB, 4 MB, 256 MB). By default, each TSB holds 512 entries, but the hardware allows much larger TSBs to be allocated if the operating system so chooses [13].

Table 3. 436.cactusADM SPECratios (higher is better)

	base	peak
run #1	86.04	86.56
run #2	86.09	86.98
run #3	85.82	63.52

Problem. During testing of CPU2006 on UltraSPARC T2 and UltraSPARC T2 Plus processors, unexplained variability was sometimes seen for the benchmark 436.cactusADM. For example, a single reportable run of the floating point suite from December 2007 with 6 runs of the benchmark (3x base tuning and 3x peak tuning) showed inconsistent performance, as detailed in Table 3. Notice that although the median performance for peak was 86.56, the slowest run was off by more than 1/4.

Analysis: Variation by copy. Recall from the metrics discussion at the beginning of this paper that reported benchmark scores depend on the time from start of first copy to completion of last. Therefore, a primary goal for the tester is to attempt to achieve consistency across all tested copies – in this case, 127 copies on a 2-chip system. Table 4 summarizes the copy-by-copy times in the second and third peak runs.

Table 4. Normalized per-copy times (lower is better) for 436.cactusADM

Metric	Peak	Peak
	Run 2	Run 3
Including all 127 copies:		
Median	1.0000	.9947
Arithmetic Mean	.9916	.9990
Std. Deviation	.0232	.0844
Max	1.0104	1.3837
If the worst 6 copies are dropped:		
Median	.9987	.9933
Arithmetic Mean	.9907	.9813
Std. Deviation	.0235	.0284
Max	1.0072	1.0086

In Table 4, times are normalized to the median time from Peak Run 2. Notice the consistency in Peak Run 2, with the worst of the 127 copies needing only 1.04% more time than the median time. By contrast, the slowest copy in Peak Run 3 needed 38.37% more time than the median of Peak Run 2. The problems

in Peak Run 3 are not widespread; in fact, only 6 of the 127 copies were slow. If these 6 copies were eliminated, as shown in the second half of the table, the two runs would match each other. Unfortunately for the tester the metrics do not allow post-processing to eliminate the slow copies.

Considerable time was spent trying to trace the source of the occasional poor copy time for 436.cactusADM, which sometimes was up to 2x worse than the expected time. Analysis of experiment logs did not indicate any particular pattern to the degraded performance. Sometimes, a handful of copies would be slow; often, none would be slow. The slow performance did not appear to be tied to system state, nor to particular virtual processors, as it would move around from one CPU to another. Attempts to instrument the tests were often met by a failure to reproduce the slow performance.

Smoking gun. Eventually, a bad run was caught with `trapstat -T` [14]:

cpu	dtsb-miss	%tim
7	4138331	59.8
11	4117256	60.2
14	4135205	59.9
21	4114273	60.4
23	4139823	59.5

In the `trapstat` output, it can be seen that various copies (on virtual processors 7, 11, 14, 21, 23) are estimated to be spending about 60% of their time processing TSB misses. Once this was found, the solution to the variability of 436.cactusADM was straightforward. As mentioned above, the hardware allows TSBs to be expanded, and Solaris supports the hardware feature with a pair of tunables: `enable_tsb_rss_sizing` and `tsb_rss_factor` [16]. The former is on by default; the latter provides a measure of how full TSBs have to be before they become candidates for resizing. As can be seen in SPEC CPU submissions from early 2008, this Solaris tuning parameter has been used, and 436.cactusADM performance has been steady. For example, in a large SPECrate submission with 630 copies, the three runs differed from each other by less than 1% [8]. If per-copy results are analyzed (as in Table 4), the worst time across all 1890 copies differs from the median by only 1.52%.

Divot summary. SPECrate was useful for uncovering a hard-to-predict, hard-to-reproduce performance divot of up to 2:1. It was resolved by encouraging the operating system to be more willing to expand the size of the data TSBs.

Lessons for tuning other systems. The default TSB sizing is adequate for most applications, especially if large pages are employed. If it is suspected that large applications (e.g. more than 1 GB, with 4 MB pages) may be running more slowly than desired, `trapstat -T` can be used to check for TSB activity, and if it is found, `tsb_rss_factor` can be decreased.

5.3 A Gain of 52% for a CPU2006 Benchmark After a Change to the Operating System Policy for NUMA Page Allocation

Problem. When testing large SPECrate runs, variability was sometimes observed, and, as in the previous section, effort was spent to try to trace it. Unlike the previous case, there appeared to be a pattern, as shown in Figure 1.

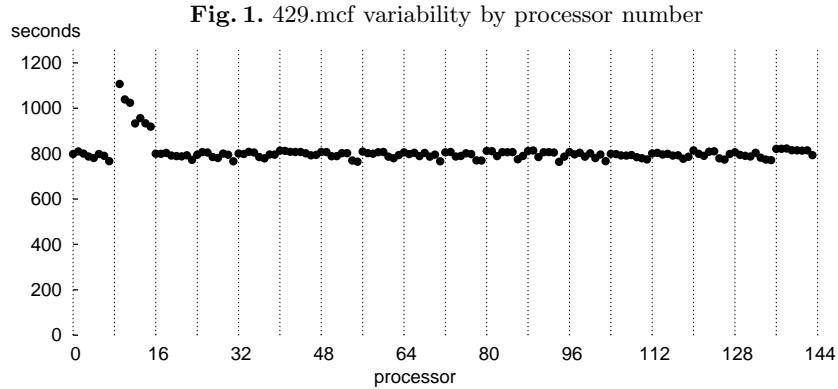
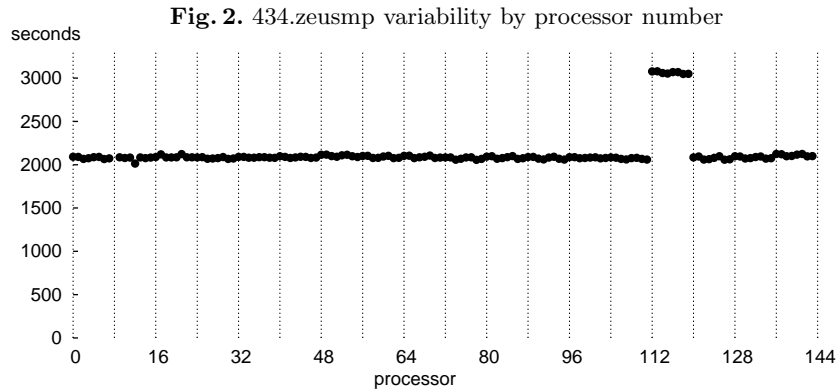


Figure 1 is from a large 72-chip, 144 processor server, running 143 copies of the benchmarks. The server has 18 system boards, each with 8 virtual CPUs. In the graph, the vertical grid delimits system boards. Notice that most copies of 429.mcf completed in about 800 seconds, except for those on the second system board. Attempts to trace the problem showed that generally a single system board would be slow, but it was, at first, hard to predict which board. In Figure 2, taken from a **different** large server, notice that it is the 4th from the last that is slower.



Graphical analysis. Edward R. Tufte suggests that graphs should be used only if one has large amounts of data needing analysis, and they should contain only pixels that are essential to the analysis, avoiding “chartjunk” [18]. The situation at hand has over 14,000 benchmark observations in each 143-copy reportable SPECfp_rate2006 run, and many more from tuning runs. To ease graphical analysis, a perl procedure was written that extracted data from log files, drove `gnuplot` with what was viewed as a minimal amount of chartjunk (as in the above graphs), and joined them into a webpage.

NUMA Hypothesis. Because the graphs showed that problems would tend to occur on a single system board, and because it is known that local system board memory access has shorter latency than remote memory, NUMA (Non Uniform Memory Access) differences were suspected. Solaris supports NUMA using a concept of Memory Placement Optimization (MPO) [2], which attempts to place process resources into “latency groups”. A latency group is a set of resources which are within some latency of each other. Systems can have multiple latency groups, and multiple levels of groups.

Tools. NUMA activity can be seen on Solaris 10 systems with the opensolaris.org “NUMA Observability Tools” [15]. Two useful tools are the extended `pmap` and `lgrpinfo`. The first is easily installed from the tools binary distribution:

```
$ gunzip -c ptools-bin-0.1.7.tar.gz | tar xf -
$ cd ptools-bin-0.1.7/
$ ./pmap -Ls $$ | head -10
Address    Bytes Pgsz Mode   Lgrp Mapped File
00010000   640K 64K r-x--   2 /usr/bin/bash
000C0000    64K 64K rwx--   1 /usr/bin/bash
000E0000   128K 64K rwx--   1 [ heap ]
FF0F4000    8K 8K rwxS-   1 [ anon ]
```

In the `pmap` example above, note that `-L` tells us the locality group for each memory segment, and `-s` displays the page size. (In the interest of space, various output is truncated in both the examples in this section.)

To install `lgrpinfo` requires a couple of extra steps, because a customization is needed for the local version of `perl`:

```
$ gunzip -c Solaris-Lgrp-0.1.4.tar.gz | tar xf -
$ cd Solaris-Lgrp-0.1.4/
$ perl Makefile.PL
Writing Makefile for Solaris::Lgrp
$ make
$ make test
All tests successful.
$ su
Password:
# make install
# exit
```

```

$ bin/lgrpinfo
lgroup 0 (root):
  Children: 1 2
  CPUs: 0-127
  Memory: installed 130848 Mb, allocated 3924 Mb, free 126924 Mb
  Lgroup resources: 1 2 (CPU); 1 2 (memory)
lgroup 1 (leaf):
  CPUs: 0-63
  Memory: installed 65312 Mb, allocated 1675 Mb, free 63637 Mb
  Lgroup resources: 1 (CPU); 1 (memory)
lgroup 2 (leaf):
  CPUs: 64-127
  Memory: installed 65536 Mb, allocated 2249 Mb, free 63287 Mb
  Lgroup resources: 2 (CPU); 2 (memory)
$

```

In the `lgrpinfo` example, the output describes a system with 128 virtual processors and 128 GB memory, divided into two latency groups. (For the sake of brevity, this example is from a simpler system than the one in the graphs.)

Diagnosis. Use of `pmap` showed that the benchmarks running in the slower locality group were receiving memory of the requested page size (4 MB) but not the desired location. It was also noted that the slow locality group was the one where the SPEC tool suite itself (`runspec`) was started. Observations with `lgrpinfo` showed that during the benchmark setup phase, when `runspec` writes 143 run directories for each of the benchmarks in the suite, physical memory was used up in `runspec`'s locality group, apparently for file system caches.

Workarounds attempted. It was hypothesized that the setup phase may have fragmented memory on `runspec`'s system board; and that the operating system might not be able (or, might not be willing) to coalesce fragmented 8 KB pages into 4 MB pages. Asking for smaller page sizes (such as 64 KB or 512 KB) sometimes appeared to succeed, but this compromise was not considered desirable since the benchmarks are large enough that 4 MB pages are known to be helpful. The size of file system caches was reduced using system tuning parameters such as `bufhwm` and `segmap_percent`, and memory cleanup was encouraged with reasonably active settings for `autoup` and `tune_t_fsflushr` [16]. To improve predictability, `runspec` was initiated in a known location, namely the system board that is also used by Solaris itself, and the amount of physical memory on that board was doubled.

These workarounds were usually helpful, and memory availability usually improved, but the workarounds were viewed as less than completely satisfactory on the grounds that in real life, customers may not have the degree of control that the benchmark tester has.

Colloquially, the problem can be simply summarized as: "Dear Operating System: If I ask for local bigpages, and you don't have them handy, please don't

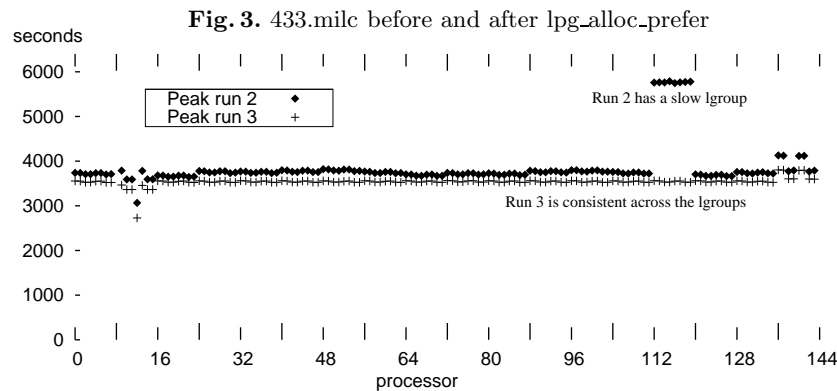
give me remote bigpages instead. Please try harder to create local bigpages.” Given this simple summary, a simple suggestion arises: why not just change the default policy to always try harder?

There are several reasons to hesitate to change the default policy: (1) Coalescing pages may be expensive, as it requires relocating pages for running processes. (2) For processes that run quickly, it may be better to allocate memory quickly rather than spending extra effort. (3) It is unknown how frequently the problem may occur in real life: how often do long-running programs ask for large memory regions with large pages, which are then used intensely enough to amortize any extra cost required to coalesce pages? Given insufficient data to answer questions such as these, the operating system policies must be approached with care.

Changes to Solaris. Over the course of the investigations of these issues, the Solaris development group responded by implementing two changes. First, the algorithm for coalescing pages was made more efficient. Second, a tunable parameter was introduced to allow users to increase the priority of local page allocation: `lpg_alloc_prefer` [16]. If you have single threaded, long running, large memory applications, then consider setting `lpg_alloc_prefer=1`. This causes Solaris to spend more CPU time defragmenting memory to allocate local large pages, versus allocating readily available remote large pages. The long term savings from accessing local rather than remote memory may offset the higher allocation cost.

This tunable parameter is used in the 256 virtual processor Sun SPARC Enterprise T5440 SPECfp_rate2006 result [9]. When the graphical analysis tools are applied to this result, NUMA effects are not seen.

Divot summary. An early version of `lpg_alloc_prefer` was applied to a system in the middle of a SPECrate run. The effect was to remove a NUMA performance divot that would sometimes slow down a single system board. The largest effect was on the benchmark 433.milc, as shown in Figure 3.



Because the tools report the time from start-of-first to finish-of-last, the bottom line improved by 52%:

```
Success 433.milc peak ref ratio=226.74, runtime=5789.629458
Success 433.milc peak ref ratio=344.64, runtime=3809.035023
```

SPECrate was useful as a generator of a repeatable, intense workload on NUMA systems, allowing careful study of the divot.

Lessons for tuning other systems. Systems that tend to run large single-threaded programs may benefit from setting `lpg_alloc_prefer`.

6 Summary

Although it is widely understood that “all software has bugs”, it may not be as widely understood that all systems have performance divots. A repeatable, analyzable workload allows divots to be analyzed. Cherish your divots.

(Repetition is a form of emphasis.)

References

1. Carlton, A.: CINT92 and CFP 92 Homogeneous Capacity Method Offers Fair Measure of Processing Capacity.
<http://www.spec.org/cpu92/specrate.txt>
2. Chew, J.: Memory Placement Optimization (MPO),
http://opensolaris.org/os/community/performance/mpo_overview.pdf
3. Gove, D.: CPU2006 Working Set Size. ACM SIGARCH Computer Architecture News, 35(1), 90–96, March 2007.
<http://www.spec.org/cpu2006/publications/>
4. Henning, J.L.: SPEC CPU Suite Growth: An Historical Perspective. ACM SIGARCH Computer Architecture News, 35(1), 65–68, March 2007.
<http://www.spec.org/cpu2006/publications/>
5. McGhan, H.: Niagara 2 Opens the Floodgates. Microprocessor Report November 6, 2006.
http://www.sun.com/processors/niagara/M45_MPFNiagara2_reprint.pdf
6. SPEC CPU2000 published results,
<http://www.spec.org/osg/cpu2000/results/res2000q2/cpu2000-20000511-00104.html> and
[res2000q2/cpu2000-20000511-00105.html](http://www.spec.org/osg/cpu2000/results/res2000q2/cpu2000-20000511-00105.html)
7. SPEC CPU2000 published results,
<http://www.spec.org/osg/cpu2000/results/res2002q2/cpu2000-20020422-01329.html> and
[res2002q1/cpu2000-20020211-01256.html](http://www.spec.org/osg/cpu2000/results/res2002q1/cpu2000-20020211-01256.html)
8. SPEC CPU2006 published results,
<http://www.spec.org/cpu2006/results/res2008q2/cpu2006-20080408-04064.html>

9. SPEC CPU2006 published results,
<http://www.spec.org/cpu2006/results/res2008q4/cpu2006-20080929-05409.html>
10. SPEC CPU2006 Documentation,
http://www.spec.org/cpu2006/docs/utility.html#convert_to_development
11. SPEC CPU2006 Documentation,
<http://www.spec.org/cpu2006/docs/utility.html#specinvoke>
12. Sun Microsystems, UltraSPARC T2 Processor
<http://www.sun.com/processors/UltraSPARC-T2/datasheet.pdf>
13. Sun Microsystems, UltraSPARCT2 Supplement to the UltraSPARC Architecture 2007, section 12.2.
<http://opensparc-t2.sunsource.net/specs/file/UST2-UASuppl-current-draft-P-EXT.pdf>
14. Sun Microsystems, Solaris 10 Reference Manual Collection,
<http://docs.sun.com/app/docs/doc/816-5166/trapstat-1m?a=view>
15. Sun Microsystems, NUMA Observability,
<http://www.opensolaris.org/os/community/performance/numa/observability/>
16. Sun Microsystems, Solaris Tunable Parameters Reference Manual,
<http://docs.sun.com/app/docs/doc/817-0404>
17. STREAM: Sustainable Memory Bandwidth in High Performance Computers
<http://www.cs.virginia.edu/stream/>
18. Tufte, E.R.: The Visual Display of Quantitative Information, Graphics Press, Chesire, Connecticut (1983), pages 107-121.
19. Weicker, R.P., Henning, J.L.: Subroutine Profiling Results for the CPU2006 Benchmarks. ACM SIGARCH Computer Architecture News, 35(1), 102-111, March 2007.
<http://www.spec.org/cpu2006/publications/>

Acknowledgments. Thank you to the SPEC CPU subcommittee for permission to summarize the investigation of SPECrate alternatives, and especially to Cloyce Spradling for implementation of the alternatives. Numerous colleagues within Sun have assisted with the technical investigations summarized in this paper, including Miriam Blatt, Jonathan Chew, Michael Corcoran, Darryl Gove, Aleksandr Guzovskiy, Alexander Kolbasov, Eric Saxe, Steve Sistare, Geetha Valabhaneni, and Brian Whitney. Karsten Guthridge was the first to catch 436.cactusADM in trapstat.