Journal on Satisfiability, Boolean Modeling and Computation 7 (2011) 139-144

Controlling a Solver Execution with the *runsolver* Tool

SYSTEM DESCRIPTION

Olivier Roussel olivier.roussel@cril.univ-artois.fr
Université Lille-Nord de France, Artois, F-62307 Lens
CRIL, F-62307 Lens, France
CNRS UMR 8188, F-62307 Lens, France

Abstract

The *runsolver* tool was designed for the 2005 edition of the pseudo-Boolean competition in order to solve the problem of correctly measuring the resources used by solvers, especially solvers with multiple processes. Since then, it has been improved in several directions and adopted by several other competitions or frameworks. This paper presents the inner working of this tool and the technical problems that it addresses.

KEYWORDS: *solver control, resources limitation, competitions*

Submitted April 2011; revised August 2011; published November 2011

1. Introduction

The task of the *runsolver* program is to control the execution of a solver in order to ensure that the solver will not take too much resources (especially time and memory) as well as gathering some data about the running solver (CPU time, exit code,...).

The development of *runsolver* was driven by the organization of the following competitions: pseudo-Boolean competitions (PB05, PB06, PB07, PB09, PB10, PB11), SAT competitions (SAT07, SAT09, SAT11) and CSP/MaxCSP competitions (CPAI06, CPAI08, CSC09). It has been adopted by other competitions or frameworks: ASP competitions, MISC11, EDACC,... *runsolver* is freely available under a GPL license from [1]. This article briefly describes the problem addressed by *runsolver*, its architecture as well as how to use it and decode its data.

2. Controlling the resources

Two major resources to monitor in order to compare solvers are time and memory. There are two distinct notions of time: wall clock time and CPU time. The wall clock time is the real time that elapses between the start and the end of a computing task. The CPU time is the time during which instructions of the program are executed by a processing unit. On a host with a single processing unit, CPU time and wall clock time are equal as long as the system does not interrupt the program. As soon as a time-sharing system is used on a single processing unit, wall clock time will usually be greater than CPU time, because during some time slices the processor will be allocated to another program. On a host with n processing units, if the program is able to use efficiently each of these units and is not interrupted by the system, the CPU time will be equal to n times the wall clock time.

Generally speaking, the CPU time is a good measure of the computing effort, while wall clock time corresponds to the user's perception of the program efficiency. Which notion is the most important depends on the application and *runsolver* allows to monitor both.

There are two notions of memory as well: resident size and virtual size. Resident size is the portion of the program that is stored in main memory. Virtual size is the total size of the program including the parts which are swapped to disk. *runsolver* monitors virtual size which is considered the actual resource consumed. Besides, enforcing a limit on memory is generally used to prevent the solver from swapping, which severely degrades performances and makes the time measures much less representative. This only makes sense if the limit is enforced on virtual size.

3. Genesis of *runsolver*

The main goal of *runsolver* is to obtain a reliable measure of the resources used by a solver, as well as to enforce limits on these. One key point is that *runsolver* should not require any privilege. This is to simplify its adoption and avoid some security risks. Basically, this can be done using time(1)[1.] and ulimit(1) but this approach frequently fails for solvers running multiple processes, which occurs as soon as a shell script is used to start the solver.

Indeed, the time(1) command uses times(2) to display the time statistics of the solver. However, this system call only returns the "resources used by those of its children that *have terminated and have been waited for*". This implies that if, for some reason, the parent process doesn't call wait(2), the resources used by the child will be ignored. This also means that these commands cannot enforce reliable limits for multi-process solvers because the resources used by the child are only reported when it terminates.

The second goal of *runsolver* is to warn the solver that the time limit has been reached and give it a chance to terminate gracefully. This is an essential feature for the pseudo-Boolean competition because, when a solver is not able to find the optimal solution within the time limit, it is still able to print the best solution it obtained.

The first version of *runsolver* in 2005 started by enforcing some resource limits with setrlimit(2), then launched the solver and waited for its completion. Every ten seconds, it fetched some data about the system (average load) as well as information on the solver process such as the current memory consumption and CPU time elapsed so far (obtained from the `/proc/*/stat*` files). On completion of the solver, the solver exit code, as well as a summary of the resources used were printed. When the time limit was reached, a `SIGTERM` signal was sent to the solver. Then the solver had two seconds to output the best result it obtained. After this delay, a `SIGKILL` was sent to terminate the solver.

All in all, this first version was merely an integration of ulimit(1), time(1) and ps(1) with a few improvements. It had two main weaknesses: it was unreliable for multi-process solvers (cf. the problem with wait(2)) and could not send a SIGTERM when the solver exceeded the memory limit.

The second version of *runsolver* (2005) attempted to fix these problems by intercepting the system calls made by the solver and was inspired by two programs: `strace` [2] which prints the system calls performed by a program and `s4g` [3] which is a generic sandbox for

1. Throughout the paper, the section of manual is indicated in parentheses in order to easily distinguish commands (section 1) from system or library calls (section 2).

programs run on a grid. In order to intercept system calls without any privilege and for any kind of executable, the solver was run in trace mode (see ptrace(2)). In this mode, the solver is suspended by the kernel each time it enters or exits a system call and the *runsolver* process is notified by a `SIGTRAP` signal. The controlling process can examine the system call and intercept its parameters and result. The system calls of interest are the ones related to processes (`clone` and `exit`) as well as memory management (`brk, mmap, munmap, mremap`). The system calls `open, execve` and the ones related to sockets were also intercepted to check if the solver conformed to the evaluation policy. With this technique, *runsolver* easily maintains a list of the processes created by the solver and adds the CPU time of all its child processes to decide if the solver must be stopped by a `SIGTERM`. Tracking the memory usage of the solver was more difficult because there are a number of system calls to allocate memory with subtle interactions. That version only maintained a upper bound of the memory used by the solver and its children and, when this bound exceeded the memory limit, it fetched the actual memory usage of the processes in the `/proc/*/stat*` files.

Stopping a solver when it uses too much memory is actually harder than stopping it when it exceeds the time limit. In fact, there are two limits: a soft limit which sends a `SIGTERM` to the solver and a hard limit (set by setrlimit(2)) which will immediately kill the solver. The hard limit was set as the soft limit plus 50 MB. For these reasons, a solver that allocates too much memory in a single call can hit the hard limit immediately and get killed or see its memory allocation fail.

Intercepting system calls has necessarily a side effect: it slows down the solver. However, the solver is only stopped when it performs system calls and, as it should not happen that often in a solver, only slightly different performances were expected. Unfortunately, the impact was much more important and varied from solver to solver. This second version increased the CPU time by at least 60% and up to 160% for some solvers (see [4] for details). This was clearly unacceptable.

The third version of runsolver (2006) abandoned the idea of intercepting system calls, but improved the identification of the solver sub-processes and increased the monitoring frequency. Every second, *runsolver* scans the list of processes on the host and identifies a tree of processes rooted at the solver. This is enough to identify the processes created by the solver or one of its descendant. Every tenth of a second, *runsolver* updates the CPU time of each of the solver processes by scanning the relevant files in `/proc`. As soon as one of the limits is reached (CPU time, wall clock time or memory), the solver is sent a signal in order to stop it. This current version is detailed in the next section.

4. Current version

As explained previously, the current version maintains a list of processes created by the solver or one of its descendants. This list is indeed a tree and is updated every second. The resource consumption of all the processes in this list is sampled every tenth of a second. Updating the list of processes requires scanning each process in the /proc directory and generating the tree of processes rooted at the solver by using the ppid information. This is more expensive than just updating the resources consumption of each process in the list. This is why the process list is updated at a frequency of 1 Hz, while the resources used

are updated at a frequency of 10 Hz. The CPU time used by the solver is the sum of all CPU times of the processes or threads that it has launched (directly or indirectly). The memory used by the solver is the sum of all virtual sizes of the processes that the solver has launched. The memory of threads is not counted since they share the same memory as their parent process.

Obviously, time resources are expected to grow monotonically between each sample. In practice, this is not the case! As an example, if a script shell is used to start the solver, we can have a global CPU time of 100 s at sample i, which drops at only 1 s at sample $i+1$. Once again, this is explained by the fact that the resources of a child are only reported to its parent when wait(2) is called. In our example, the child process has used 99 s CPU time and terminates between the two samples. If the parent has not called wait(2), the CPU time of its child is lost and we can only observe the CPU time of the parent process (1 s). *runsolver* identifies these cases and keeps track of this lost time.

Another essential feature of *runsolver* is its ability to timestamp each line printed by the solver. Even once the solver has exited, this allows to easily identify at what time some events occurred. For example, it can be used to learn how much time it took to parse the instance. This feature was developed specifically for the pseudo-Boolean competition, in order to identify at what times a solver discovers a solution which is better than the previous ones. This does not require a specific instrumentation of the solver: it just prints a line indicating the quality of the new solution, and *runsolver* takes care of printing the time elapsed since the start of the program. *runsolver* waits with select(2) for the solver to print a line. Then the timestamp is obtained by calling gettimeofday(2) which returns wall clock time. Since it would be too expensive to update the CPU time of the solver processes each time a line is printed, the timestamp in CPU time is extrapolated from the two nearest samples. Another point is that the solver must flush its buffer each time it prints a line, otherwise the transmission of the line to *runsolver* may be delayed. If a solver does not flush its output, *runsolver* is able to fool the solver and cause an automatic flush of the stream by using a pseudo-terminal (option `--use-pty`) instead of a plain pipe[2].

Some solvers are able to generate a huge amount of output (several GiB) and it is sometimes necessary to impose a quota to avoid filling the disk with the solver garbage. This can happen if debugging lines or progression bars are used by the solver. To enforce a quota on the solver output, *runsolver* is able to retain only the first x MiB of output as well as the last y MiB. If the solver prints more than $x+y$ MiB, the data in excess is lost, but the amount of dropped data is indicated by a message. This mode is activated by the `--output-limit` option which requires two parameters: the first one is x (the size of the initial part of output to preserve) and the second one is $x+y$ (the total size of output that will be preserved).

If it appears that a solver doesn't have full access to the CPU, *runsolver* is able to display the other processes which are competing with the solver for execution on the CPU. This is triggered when the ratio CPU time/wall clock time drops under a given threshold (0.8 in the current version).

The latest feature added to *runsolver* is the ability to indicate which cores are allocated to a solver (and to each of its processes) as well as to allocate to a solver a given subset

2. The C library automatically flushes its buffer after the end of line when the file is a terminal.

of the host cores (cf. sched_setaffinity(2) and sched_getaffinity(2)). It is interesting to notice that the Linux kernel does not necessarily number the cores of a same chip with consecutive numbers. For example, on the hosts used in the 2011 edition of the SAT and PB competitions, the cores of the first processor are numbered 0, 2, 4, 6 and the ones of the second processor are numbered 1, 3, 5, 7. The *runsolver* process itself attaches to the last available core, which limits interferences between a sequential solver and the *runsolver* program as soon as at least two cores are available.

One key feature of runsolver is that it regularly prints the list of processes and threads that the solver runs, with the information gathered from the system. This is important to identify if some processes terminated unexpectedly or for justifying that a solver exceeded one of the limits. Even if *runsolver* collects data every tenth of a second, if would not make sense to record each available sample[3]. Besides, the most relevant events occur generally at the start and the end of the program. Therefore, *runsolver* uses a buffer of samples which allows it to print a greater number of samples at the start and the end of the solver execution, and only one sample per minute otherwise.

In general, *runsolver* has a very low impact on the solver performances but this cannot be guaranteed. Indeed, since *runsolver* executes concurrently with the solver, they both compete for CPU and memory access. Therefore, there is necessarily a perturbation on the solver, one reason being cache pollution by *runsolver*. Such an effect is obviously highly dependent on the hardware and almost impossible to avoid. To check the impact of *runsolver*, minisat2 was run four times on QG6-gensys-brn007.sat05-2684.reshuffled-07.cnf, on a host with an Intel Core2 quad-core CPU Q9300 at 2.5 GHz. time(1) reported a CPU time of 34.38 s in average, *runsolver* running on 4 cores reported 34.35 s, *runsolver* on 2 cores reported 34.41 s and *runsolver* on a single core reported 34.38 s. In these experiments, *runsolver* itself consumed 0.33 s of CPU time. We can observe that in this case, the perturbation is negligible.

5. Using *runsolver*

Using *runsolver* is extremely easy since it does not require any privilege. The syntax is essentially the same as time(1): `runsolver [options] solver [args...]`. *runsolver* accepts many options beyond those already mentioned. `-o` allows to redirect the solver output to a file while `-i` allows to redirect the solver input from a file. The data gathered about the processes can be redirected to a file with the `-w` option. The delay between the SIGTERM and the SIGKILL signals can be specified with `-d`. Limits on CPU time, wall clock time, size of virtual memory and even stack size can be specified (options `-C, -W, -M, -S` respectively).

As an example, one can type `runsolver --timestamp -w watcher.out -o solver.out minisat file.cnf`[4]. The file solver.out will contain the following lines (by lack of space, irrelevant parts of the output have been omitted):

```
0.00/0.00  This is MiniSat 2.0 beta
[...]
0.00/0.07   ========================[ Search Statistics ]==========================
0.00/0.07   |Conflicts|      ORIGINAL         |      LEARNT          | Progress |
0.00/0.07   |         |Vars  Clauses Literals |Limit  Clauses Lit/Cl |          |
```

3. For a multi-thread program and a runtime of 5000 s, the resulting file would have a size of 190 MiB!
4. The actual file used in this experiment is QG6-gensys-brn007.sat05-2684.reshuffled-07.cnf.

```
0.00/0.07   ============================================================================
0.00/0.07   |       0 |1234     4590    36164 | 1530        0   -nan |  0.000 % |
[...]
15.39/15.44 |  443167 |1232     3780    29076 | 9357     7397     19 |  2.980 % |
24.78/24.89 |  664850 |1230     3754    28976 |10293     6320     15 |  3.115 % |
34.37/34.41 ============================================================================
[...]
34.37/34.41 UNSATISFIABLE
```

The first figure on each line is the estimated CPU time and the second figure is the wall clock time. This information tells us when the solver performed its restarts. One of the samples printed by *runsolver* is the following:

```
[startup+0.700115 s]
/proc/loadavg: 0.27 0.16 0.06 2/272 28125
/proc/meminfo: memFree=1687372/3926240 swapFree=0/0
[pid=28125] ppid=28123 vsize=16632 CPUtime=0.69 cores=0-3
/proc/28125/stat : 28125 (minisat) R 28123 28125 27882 34816 28123 4202496 954 [...]
/proc/28125/statm: 4158 797 241 20 0 752 0
Current children cumulated CPU time (s) 0.69
Current children cumulated vsize (KiB) 16632
```

It indicates the time of the sample, the host load average, the current memory and swap used by the system, the list of processes of the solver and the cumulated CPU time and virtual memory size. For each process, the resources used are displayed, including the cores which are allocated to the process. The stat and statm file are also displayed, to allow post-mortem investigation if needed.

6. Conclusion

runsolver is a useful tool for controlling a solver execution and gathering different relevant information during its execution. The experience accumulated during the different competitions has driven its development. A number of essential features such as timestamping have been incorporated.

This kind of tool would be perfect if it did not use any resource itself, which is obviously impossible. In computer science too, measurement perturbs the experiment. We believe that the current version of *runsolver* is a good compromise but there is still hope that it can be improved.

The future development of *runsolver* will probably be to incorporate the features of the sandbox(8) command which allows to enforce a SElinux policy that sets up a sandbox, preventing the solver from performing undesired actions on the system.

References

[1] The *runsolver* homepage. http://www.cril.univ-artois.fr/~roussel/runsolver/.

[2] W. Akkerman et al. The strace homepage. http://sourceforge.net/projects/strace/.

[3] T. Morlier. S4G: a Sandbox for Grids. http://s4g.gforge.inria.fr/.

[4] V. Manquinho and O. Roussel. The First Evaluation of Pseudo-Boolean Solvers (PB'05), Journal on Satisfiability, Boolean Modeling and Computation **2**:103-143, 2006.