

Opportunities for workload analysis via extended accounting

Alan Burlison and Stephen Hahn

Solaris Kernel Development

alanbur@uk.sun.com, sch@eng.sun.com

ABSTRACT

Extended accounting, introduced in Solaris 8 6/00, provides a number of mechanisms that may offer new insight into the characteristics of large, complex workloads. These mechanisms are reviewed, and applied to an example involving the OS/NET consolidation build. We make performance recommendations and requests for enhancement on both the OS/NET build and the extended accounting facility itself.

INTRODUCTION

The Solaris operating environment provides rich statistics on system activity such as CPU loads, disk service times and network packets sent. As Solaris has been enriched with new features, new statistics describing the behaviour or performance of these features have been published, generally via the kernel statistics facility described in `kstat(3KSTAT)`.¹

An additional set of statistics associated with process-based usage was exported in a non-persistent fashion via `/proc`, and made easily visible via `prstat(1)`. However, the enhancements to these statistics were not reported via the existing System V-derived accounting facility.

System V accounting data is composed of fixed-size structures which use a 16-bit `comp_t` to represent floating-point data. The binary format is unversioned and shared by Solaris and other System V derivatives. While the data itself can be useful for capacity planning applications, its representation of workload is entirely focused on the process. This is no longer appropriate given the established multiprocess application model, and no longer sufficient to present the rich structure of the emerging multithreaded application model.

Accordingly, an extended accounting mechanism with a flexible file format and enhancements to the process model was introduced into Solaris to address these short-

comings. In this paper, we briefly describe the new facility and offer general advice about its application. We then present an example of workload analysis using the new facility: we detail one approach to instrumenting the OS/NET consolidation build. We conclude with some observations about the OS/NET build, as well as some comments regarding future directions for extended accounting.

EXTENDED ACCOUNTING

The extended accounting facility, known informally as `exacct`, consists of a number of components:

- two extensions to the process model: the *task* and the *project*,
- a set of new system calls,
- a new accounting file format along with a library for accessing and creating files of this format, and
- a number of new commands.

In this section, we briefly review some the features of `exacct`, and describe their application to capturing workload resource utilization. As mentioned earlier, this feature set is available in Solaris 8 6/00 and, of course, Solaris 9 and beyond.

General features

Extended accounting introduced two extensions to the process model: the *project* and the *task*. The project abstraction provides a name and ID space with which workloads can be associated. That is, an organization might provide project entries that allow workloads to be associated with a department, a user class, or a specific project. The project database is made available via the name service, such that common definitions can be shared across an administrative domain. Although future extensions to the project are being planned, for the purposes of `exacct` it may be viewed as an accounting ID.

¹The variety of `kstats` available on the system is easily visible using the `kstat(1)` command, introduced in Solaris 8.

A *task* is a new process collective that associates a set of processes with a specific `project`. Thus, the `task` is intended to represent an identifiable workload or workload component. Unlike the existing process collectives, a `task` has no leader, meaning that the `task` can represent a series of invoked processes without having a persistent monitoring process, as with the `session`. A new `task` is created by the `settaskid(2)` system call; `task` membership is inherited on `fork(2)`. When the last process in a `task` exits, the `exacct` framework will write a record describing the total resource usage of the `task`.

Extended accounting, via the `exacct` file format, provides an extensible data file for each of processes and tasks. (Either of these files can be active independently). `Task` accounting is particularly inexpensive as it aggregates the accounting data for potentially many processes. Using `acctadm(1M)`, we can add the microstate accounting information detailed in `proc(4)` (with the exception of the state times themselves) to the process or the `task` records. The accounting data is available in an instantaneous fashion via `getacct(2)` which returns an in-memory copy of the current `task` or process record; we can also force the instantaneous data into the current accounting files via the `wracct(2)` system call.

putacct(2) and complex workloads

The system call, `putacct(2)`, allows a process with `root` privilege to embed its own data in the appropriate kernel accounting file, and to associate the data with its process or `task`. This facility is particularly well-suited to multiprocess workloads, as the `task` can be used for gross aggregation, while records placed with `putacct(2)` can give finer-scale data than provided by the process record. The OS/NET build instrumentation we discuss below utilizes this mechanism.

libexacct(3LIB) for private datastreams

An alternative and more flexible approach is to construct one's own record definitions using the `EXC_LOCAL` catalog tag and `libexacct(3LIB)` routines to write an application specific accounting file. For workloads that cannot utilize a process or `task` as a container to demarcate their resource usage (such as a long-running threaded server) or for workloads that do not associate their workload with individual processes (such as a web server), this avenue may be more appropriate.

A possible enhancement to `sar(1)` to utilize `exacct`

files as an on-disk data format would also provide several benefits, including a common data storage and manipulation framework for both performance and accounting data. A major problem with the existing `sar(1)` data files is that they are both architecture and OS release dependent. This severely restricts their utility for both cross-platform and cross-release data manipulation.

INSTRUMENTING THE OS/NET BUILD PROCESS

As a concrete example of the techniques of the previous section, we present performance analysis of the OS/NET consolidation build. The OS/NET consolidation consists of approximately 6 million lines of predominantly C code spread over more than 15,000 files, which are split into three groups: kernel, libraries, and commands. Additionally, the kernel is built in both debug and production versions and in both 32- and 64-bit modes, packages are built from the results of the compilation, and the kernel (and various libraries and commands) are run through two passes of `lint`.

On an Ultra Enterprise 450 with 4 CPUs, this process can take approximately nine hours. Given the size of the product, build analysis is a reasonably serious example to test the methods outlined above.²

Method

In order to instrument the OS/NET build process we first had to identify points at which instrumentation could usefully be inserted and data gathered. The simplest approach was merely to enable per-process extended accounting using the `acctadm(1M)` command, and analyse the resulting `exacct` data. The data that can be obtained this way includes the process ID, user ID, group ID, CPU usage, elapsed time, command, project ID and task ID for each process run during the build process. However, this simplistic approach has some serious flaws:

- There is no way of knowing what phase of the build process invoked a given instance of a command.
- The behaviour of a command depends on the context in which it was invoked, and on its arguments. The only invocation data presently collected is the base-name of the executable, making it impossible to distinguish the context in which the command was executed.
- There is no way of aggregating commands to obtain an overall picture of the resources consumed in build-

²Current operations that reduce build time consist of dependency elimination, pruning of unbuild or unneeded files and targets, and modification of tools invocation. Generally, this sort of work is not so much focused on performance as keeping the already large source tree from becoming unwieldy.

ing a particular subset of the source tree. For example, we cannot answer how much CPU time the C compiler consumed in building `libc`.

In order to obviate these problems we needed some way of providing aggregated statistics. The `exacct` framework conveniently provides this in the form of the previously described task abstraction. Since each process accounting record includes the ID of the task of which the process is a part, individual process records can be correlated with their corresponding task record.

Having ascertained that tasks provided a suitable mechanism for aggregating the individual process data, we then had to decide on a suitable granularity and mechanism for the creation of new tasks during the build process. After some thought, it was apparent that having each invocation of `make` start a new task would be sufficiently fine-grained to allow accurate analysis of the resulting task data, whilst providing a significant reduction in the total number of accounting records being written.

In the case of the OS/NET build process, there is a rough one-to-one correspondence between source code components and `make` invocations. Generally speaking each component lives in a separate directory and is processed by a nested invocation of `make`. In order to allow the creation of a new task whenever `make` was invoked we wrote a simple wrapper that performed the necessary `exacct` calls to create a task before executing the real `make` executable.

Once this instrumentation point had been identified it was possible to remedy some of the other shortcomings identified above, by writing out supplementary information into the task accounting file for each invocation of the `make` wrapper. The `exacct` framework allows the embedding of arbitrary application-defined accounting records into the system accounting stream, via the `putacct(2)` system call. This facility was used to embed the following supplementary information into the system accounting stream:

- *Ancestor task ID*: To be able to identify the relationships between the many `make` invocations during a build it was necessary to record some sort of relationship information in the system accounting stream. We decided to include the ancestor or parent task ID of each task into the supplementary information recorded by the `putacct(2)` call.
- *Current working directory*: As previously stated it is necessary to know the context in which a command

is executed if you are to make any useful observations about it. It was obvious that recording this information per process was impractical, as it would require a wrapper for every executable invoked during the entire build process, which even if it were possible would place an unsupportable performance overhead on the build. Instead the working directory for each invocation of `make` was recorded in the task accounting file. The task ID recorded for each process would allow processes to be tied to the part of the source tree in which they were invoked.

- *make invocation*: The OS/NET build process consists of a series of sequential phases including `clobber`, `debug`, `nondebug` and `lint`. We hoped that by recording the `make` command-line for each invocation it would be possible to identify which build phase each `make` process (and thereby each task) was associated with.

The relevant part of the code used to wrap `make` and embed the additional information into the system accounting stream is given in Figure 1.

Unfortunately it became rapidly apparent that simply wrapping `make` did not allow individual build phases to be identified by inspection of the `make` command-lines. This is because the script that controls the build process (`nightly`) propagates a large part of the state information to the child `make` processes via environment variables.

One solution would have been to embed environment information into the accounting data stream as supplementary information, as was done for working directory and command-line. However we felt that this would drastically increase both the overhead associated with the `make` wrapper, and the size of the task accounting files.

Instead another powerful feature of the `exacct` framework was used—the project. A project is a way of grouping users and/or processes by activity type, for example projects could be set up for development, testing or documentation. Individual users can be given permission to be active in one or more projects, and may switch between projects depending on the type of activity they are currently performing. All tasks and processes that users generate whilst active in a project are tagged with the project ID, and can be aggregated by project as well as by task ID. Normally a user can switch between projects by using the `newtask(1)` command. This changes to the required project, subject to authorization, and forks a new login environment for the user. The reason for the

```

/* Obtain the current task id before creating a new one. */
if ((pjid = getpr          ojid()) == -1) goto EXEC;
if ((ptid = gettaskid()) == -1) goto EXEC;

/* Create a new task. */
if ((tid = settaskid(pjid, TASK_NORMAL)) == -1) goto EXEC;

/* Create the data items to be embedded in the accounting stream. */
ea_set_item(&ptskid, EXT_UINT32 | EXC_LOCAL | MY_PTID, &ptid, 0);
ea_set_item(&cwd, EXT_STRING | EXC_LOCAL | MY_CWD,
            cwdbuf, strlen(cwdbuf));
ea_set_item(&cmd, EXT_STRING | EXC_LOCAL | MY_CMD,
            cmdbuf, strlen(cmdbuf));

/* Create an accounting group record. */
ea_set_group(&grp, EXT_GROUP | EXC_LOCAL | EXD_GROUP_HEADER);
ea_attach_to_group(&grp, &ptskid);
ea_attach_to_group(&grp, &cwd);
ea_attach_to_group(&grp, &cmd);

/* Pack the record and embed in the accounting stream. */
ea_bufalen = ea_pack_object(&grp, ea_buf, sizeof(ea_buf));
putacct(P_TASKID, tid, ea_buf, ea_bufalen, EP_EXACCT_OBJECT);

/* Run the real make executable. */
EXEC:
setuid(getuid());
setgid(getgid());
execvp(REAL_MAKE, argv);
exit(1);

```

Figure 1: Code fragment illustrating the wrapping of `make(1)` and embedding the required additional information in the kernel accounting stream.

`fork(2)` requirement is because `newtask(1)` is a setuid root executable, and in turn this is required because the `settaskid(2)` system call requires root privilege. Root privilege is required because the `settaskid(2)` system call makes the assumption that the caller is trusted, and has validated that the calling user has the necessary permission to switch to the new `project` by examining the contents of the project database.

We decided to split the nightly build into separate projects, each one corresponding to a distinct phase of the build, for example `clobber`, `bringover`, `debug`, `nondebug` etc. However, the nightly script is a large (2000+ line) monolithic `ksh` script, which makes extensive use of environment variables for managing state information. Whilst it would be possible to split the script into several parts and pass in state to each sub-script, it would be difficult to propagate environment changes back into the parent script. In addition such large scale changes would make it difficult to keep the resulting set of scripts in synchronization with any changes made to the nightly script used to build the OS/NET gate.

We therefore decided to produce a customized version of `ksh(1)`, with ‘`newtask`’ as a new shell builtin. Unfortunately the version of `ksh(1)` shipped with Solaris does not have the loadable shared object mechanism of newer versions of `ksh`, so a customized executable was the only practical option.

In addition the `root-only` restriction on the `settaskid(2)` system call was unnecessarily burdensome for a test and development environment, so we decided to circumvent the `root-only` requirement on the `settaskid(2)` system call. Rather than producing a customized version of the kernel with the `root-only` restriction on `settaskid(2)` removed, we wrote a loadable kernel module that could be installed and uninstalled dynamically with `modload(1M)/modunload(1M)`. On installation via `modload(1M)` this module interposed itself on the appropriate entry in the `sysent` system call table and replaced the `settaskid(2)` system call with a version with the `root-only` restriction removed. On deinstallation of the module via `modunload(1M)` the original system call vector was patched back into the `sysent` table.

The `tasksys` system call is not a loadable system call, so this procedure is somewhat risky. In practice, this has not been a problem, as the system is inactive when the module is loaded or unloaded, but this is definitely a development-only (and unsupported) technique! How-

ever, it is a good indication that the severity of the current privilege model of `settaskid(2)` will probably restrict customer adoption of this feature. The `root`-based build environment of OS/NET is unusual as it stands, and the `root` requirement of `settaskid(2)` all but precludes its use for this type of instrumentation.

Once the customized version of `ksh` and the replacement `settaskid(2)` system call were in place it was then a trivial job to put each phase of the build into a separate project. Appropriately-named projects were added to the `/etc/project` file as follows:

```
onbld:100::root,gk::
onbld.clobber:101::root,gk::
onbld.bringover:102::root,gk::
onbld.realmode:103::root,gk::
onbld.debug:104::root,gk::
onbld.nondebug:105::root,gk::
onbld.lint:106::root,gk::
onbld.gprof:107::root,gk::
onbld.trace:108::root,gk::
onbld.check:109::root,gk::
```

And once that was done the nightly script was modified to place each phase of the build into a separate project. An outline of the type of change made to the nightly script is given in Figure 2.

Use of the `(...) ksh` bracketing construct means that when the closing parenthesis is reached, the script automatically switches back into the enclosing project. In the segment above the script switches first into the `onbld` project, then into the `onbld_nondebug` project, back out into the `onbld` project, enters the `onbld_debug` project and then finally returns to the `onbld` project.

With these modifications in place, the framework for the instrumentation of the nightly build was in place, with three levels of data aggregation: process, task and project. The next problem was to find some way of manipulating the data. A full nightly build produces a process accounting file with more than 845,000 records, and a task accounting file with some 28,000 task records. The file extended accounting sizes are approximately 330MB and 16MB respectively. To merely scan the process accounting file and dump it out as text takes 23 minutes, so producing reports by directly manipulating the accounting files was obviously not feasible. We therefore decided to transfer the `exacct` accounting data into Oracle and then to use SQL and the Oracle report generation facilities to analyze the data.³

³A similar technique is used by HPC and enterprise chargeback systems, such as Instrumental’s PerfAcct product.

```

#!/ws/on81-gateutils-uk/sparc/aksh -p
export SHELL=/ws/on81-gateutils-uk/$MACH/aksh
newtask onbld
...
normal_build()
    # non-DEBUG build begins
    if [ "$F_FLAG" = "n" ]; then
        (
            newtask onbld_nondebug
            ...
        )
        ...
    # DEBUG build begins
    if [ "$D_FLAG" = "y" ]; then
        (
            newtask onbld_debug
            ...
        )
    ...

```

Figure 2: Outline of the changes made to the nightly script to separate the build into projects.

To transfer the `exacct` data into Oracle it was first necessary to decide on the mapping between the `exacct` data files and the corresponding Oracle tables, and the mapping of `exacct` data types onto Oracle equivalents. The table mapping was simple: two tables were created, one for task records and one for process records. Each build was stored in two new tables, where the table names were suffixed with the MMDD date of the build.

Mapping the data fields was a little more problematic. The data captured in the accounting files falls into three main categories:

- *String data.* This includes data such as command name and working directory. These data types were mapped to the Oracle `VARCHAR2` data type.
- *Integer values and counts.* This includes data such as major and minor faults, task ID, system calls made, etc. These were mapped onto a corresponding Oracle `NUMBER` data type that was large enough to represent the precision of the underlying data value. Most `exacct` data of this type is stored in 64 bit integers so the Oracle `NUMBER(20)` type was used.
- *Dates and time intervals.* This type was problematic. Data values such as process start and finish times, CPU usage etc are stored by `exacct` as (seconds, nanoseconds) tuples, where each member of the tuple is a 64-bit integer. In the case of dates the value is expressed

in (seconds, nanoseconds) since the epoch. The obvious first choice would have been to store such values in the Oracle `DATE` type, but this poses problems because the precision of the Oracle `DATE` type is only to the nearest second. The only practical alternative was to store dates and time intervals in a form equivalent to that used in the `exacct` data files, and then to provide conversion routines for mapping between the `exacct` and Oracle equivalents. Accordingly all such values were stored as `NUMBER(20,9)`, which allows the full range and precision of `exacct` values to be represented.

Once the database schema had been established, we provided a mechanism for transferring the data between the `exacct` files and the Oracle database. During the `exacct` development process a utility had been created to allow `exacct` data files to be dumped in a text format. This was modified to support the additional tasks records added to the accounting stream by the `make` wrapper. The output of this utility was then passed through a perl script which performed additional processing to:

- Join the second and nanosecond subfields of time and interval data such as user CPU usage into a single field.
- Form composite task records from the system task records and the additional records output by the `make` wrapper.

- Format the output so that it was suitable for processing by the Oracle database bulk load utility.

The resulting data stream was then passed to the Oracle loader utility and used to populate the appropriate task and project tables in the database. The final stage of the process was to add indexing to the tables and analyse the tables for the Oracle query optimizer.

A Perl module to allow direct access to the `exacct` data files is currently under development and is targeted for release with Solaris 9 [2]. This module will remove the need for the two-step processing required to convert the `exacct` data files into a form suitable for processing by the Oracle bulk loader.

The time and date conversion routines mentioned above were implemented as an Oracle database package containing stored procedures. This makes it possible to use the date manipulation functions directly in the SQL used to query the database, rather than requiring subsequent post-processing. Given these manipulation facilities it is then possible to formulate SQL queries such as those shown in figure 3.

At this point we were in a position to start analyzing the data, and the results of this analysis are presented in the following section.

Results

The table below shows the start times of the various build phases for the build described in figure 7 and for all the other associated graphs in this paper.

Build phase	Time
Build started	11:30:24
Begin clobber	11:30:25
Begin non-debug build	11:52:45
Begin non-debug cpio archives	15:19:10
Begin non-debug packages	15:23:31
Begin debug build	15:28:10
Begin debug cpio archives	16:39:31
Begin debug packages	16:43:58
Begin protolist	16:48:34
Begin gprof build	16:49:40
Begin trace build	17:56:35
Begin lint	19:01:53
Build finished	20:39:23

One immediately useful side-effect of the instrumentation is that it is possible to see what phase of the OS/NET is currently executing, using the `prstat(1M)` utility. Project data is available via the `-J` option, and task data is available via the `-T` option. For an example of

the `-J` flag in operation, see Figure 4. Note that the `onbld_nondebug` project is active.

We have produced a set of standard reports for examining the accounting data once it is loaded into Oracle. The most immediately interesting of these reports is the ‘CPU hogs report’, which totals up the CPU usage of all the commands executed during the build, and shows them in order of decreasing total user+system CPU usage. (See Figure 5.)

These numbers are not entirely unsurprising, with `make` and the various compiler phases being the most frequently executed commands. What is somewhat surprising however is the number of times `sh` is invoked. The shell is executed 430,407 times out of a total of 846,649 processes invoked per build, therefore representing more than 50% of the total process invocations. Bearing in mind this represents nearly one and a half hours of CPU time, we felt that this required further investigation. The first step was to discover what processes invoked `sh`, and with what frequency. The simplest way to do this would have been to query the process accounting data to find the parent process of every `sh` invocation. Unfortunately parent process ID is not currently captured by the `exacct` framework at present, so this was not possible.

As an alternative, the `ptree(1)` utility was used to snapshot the build process as it was running, which we present in Figure 6.

The immediately obvious thing to note is that each invocation of `cc` is being made via an additional `sh` process. Inspection of the makefiles concerned did not show that this was being done directly by the makefiles themselves. The command-line of a typical `cc` invocation was examined more carefully for potential clues. We noticed that the `cc` command-line invariably included arguments such as `-xarch=v9a` and `-Qiselect-regsym=0`. The manpage for `make(1)` states:

To rebuild a target, `make` expands macros, strips off initial TAB characters and either executes the command directly (if it contains no shell metacharacters), or passes each command line to a Bourne shell for execution.

We surmised that `make` was finding the ‘=’ character in the `cc` command-line and was interpreting it as a shell meta-character, and was therefore passing the command off to `sh` to execute. A quick search of the `make` executable identified the embedded string `#|=^ (); &<>*? [] : $ \ ' " \`, which looks suspiciously like something that would be used to match shell meta-characters. Unfortunately the `make` executable used to

```
-- How much CPU did ld use?
SQL> select uxt.hhmmss(sum(cpu_user_sec + cpu_sys_sec), 2) "ld CPU usage"
  2   from process_0227
  3   where command = 'ld';
```

```
ld CPU usage
-----
00:08:17.92
```

```
-- What projects/commands were active at 01:00pm?
SQL> select projname "Project", count(command) "Number",
  2   command "Command"
  3   from process_0227, project
  4   where process_0227.projid >= 100 and process_0227.projid <= 109
  5   and start_sec <=
  6     uxt.utime(to_date('27/02/01 13:00:00', 'DD/MM/YY HH24:MI:SS'))
  7   and finish_sec >=
  8     uxt.utime(to_date('27/02/01 13:00:00', 'DD/MM/YY HH24:MI:SS'))
  9   and process_0227.projid = project.projid
 10   group by projname, command
 11   order by projname, command
```

Project	Number	Command
onbld	1	nightly.task
onbld_nondebug	12	CC
onbld_nondebug	10	ccfe
onbld_nondebug	1	cg
onbld_nondebug	1	iropt
onbld_nondebug	20	make
onbld_nondebug	1	nightly.task
onbld_nondebug	18	sh
onbld_nondebug	1	tee
onbld_nondebug	1	time

Figure 3: Example SQL showing the use of the stored procedure package for date and time manipulation.

```

    PID USERNAME  SIZE  RSS STATE PRI NICE   TIME    CPU PROCESS/NLWP
371654 root         32M   17M cpu2   0   0 0:00:02  2.9% javac/9
152812 root      5848K 5136K cpu1   58   0 0:00:36  0.1% prstat/1
100215 root      3720K 2648K sleep   8   0 0:00:06  0.1% automountd/5
369251 root      2464K 2048K sleep  48   0 0:00:00  0.0% make/1
371653 root      1024K  848K sleep  48   0 0:00:00  0.0% sh/1
100248 root      3192K 2712K sleep  58   0 0:00:00  0.0% nscd/9
152815 alanbur  1816K 1320K sleep  58   0 0:00:00  0.0% ksh/1
100230 root      2072K 1688K sleep  58   0 0:00:00  0.0% cron/1
100207 root      2576K 2144K sleep  58   0 0:00:00  0.0% inetd/1
100253 root      3048K  992K sleep  55   0 0:00:00  0.0% lpsched/1
100214 daemon   2496K 1888K sleep  60   0 0:00:00  0.0% statd/4
100177 root      2120K 1456K sleep  42   0 0:00:00  0.0% ypbind/1
100167 root      2328K 1376K sleep  50   0 0:00:00  0.0% keyserv/4
100298 root      1008K  680K sleep  58   0 0:00:00  0.0% utmpd/1
100164 root      2408K 1544K sleep  58   0 0:01:00  0.0% rpcbind/1
PROJID  NPROC  SIZE  RSS MEMORY   TIME    CPU PROJECT
   105     13   53M   32M   3.3%  0:00:07  2.9% onbld_nondebug
     3     43  128M   95M   9.6%  0:01:10  0.1% default
     0     44  131M   78M   7.9%  0:01:12  0.1% system
   100     1  2016K 1600K   0.2%  0:00:00  0.0% onbld

```

Total: 101 processes, 224 lwps, load averages: 1.86, 4.76, 6.50

Figure 4: Example `prstat(1M)` output during an OS/NET instrumented build.

build OS/NET is the defunct parallel make, and no source could be found in order for us to verify this assumption.

If this supposition is correct, this would account for at least 39,207 of the total 430,407 shell invocations. However, that still leaves 90% of the shell invocations unaccounted for.

We examined the OS/NET makefiles further to see if there were any other potential large users of `sh`. We noticed that the master makefiles `Makefile.master` and `Makefile.master.64` contained a number of constructs similar to these:

```

CH:sh= echo \\043
...
MACH64:sh= if [ "$MACH" = "sparc" ]; \
            then echo sparcv9; \
            elif [ "$MACH" = "i386" ]; \
            then echo ia64; \
            else echo unknown; \
            fi

```

Each of these make macro assignments invokes the shell, passing the rest of the line as input, and assigns the output of the shell to the named macro. The `echo` construct is there merely so a make macro can be set to the `#` character, which is then used for selectively commenting out parts of the remainder of the makefile. The second con-

struct is used to allow conditional assignment to make macros, depending on the value of the appropriate environment variable.

`Makefile.master` and `Makefile.master.64` are included into nearly every other makefile in the OS/NET build environment, so any such use of the shell is repeated many, many times throughout the build. To investigate the impact of this on the build process, we modified `Makefile.master` and `Makefile.master.64` to remove as many of these constructs as possible, replacing them as far as practicable with make macros that did not require a shell invocation. The build was then rerun and the accounting data again imported into Oracle for comparison with the unmodified version. The results were gratifying:

Makefiles	Invocations (sh / total)	sh CPU usage
Original	430,407 / 846,649	01:06:05
Modified	213,588 / 630,103	00:42:35

However, although the amount of CPU time consumed by the shell had decreased, there was no corresponding decrease in the overall build time. Our assumption was that if the build was significantly non-parallel then decreasing the amount of time consumed by the shell wouldn't necessarily result in a corresponding decrease in overall build time. To verify this we examined the `sar(1)` data captured during the build (see Figure 7).

Command	User+Sys Time	User Time	System Time	Executions	Average U+S Time
acompc	05:40:59.09	04:48:42.90	00:52:16.19	37395	0.547108
cg	04:39:19.95	04:10:49.32	00:28:30.63	37690	0.444679
make	02:38:25.99	01:23:33.22	01:14:52.77	82630	0.115043
sh	01:28:58.52	00:22:53.46	01:06:05.06	430407	0.012403
iropt	01:28:14.96	01:06:35.97	00:21:38.99	37689	0.140491
lint2	01:25:06.24	01:24:23.46	00:00:42.78	1060	4.817208
lint1	00:58:31.52	00:50:28.77	00:08:02.75	4193	0.837472
javac	00:51:51.67	00:46:18.78	00:05:32.89	1164	2.673256
ccfe	00:17:01.11	00:14:47.20	00:02:13.91	708	1.442246
cc	00:16:58.21	00:03:18.90	00:13:39.31	39207	0.025970
mcs	00:11:59.73	00:02:52.03	00:09:07.70	24163	0.029786
ld	00:09:51.97	00:06:17.14	00:03:34.83	5692	0.104000
gzip	00:06:35.82	00:06:07.12	00:00:28.70	85	4.656706
rm	00:06:35.49	00:01:03.04	00:05:32.45	24821	0.015934
install	00:05:24.66	00:00:49.69	00:04:34.97	18675	0.017385
perl	00:04:55.38	00:03:23.95	00:01:31.43	3017	0.097905
pkgmk	00:04:17.61	00:00:42.99	00:03:34.62	655	0.393298
lint	00:04:12.35	00:00:56.63	00:03:15.72	4540	0.055584
expr	00:03:55.55	00:01:00.32	00:02:55.23	10672	0.022072
miniperl	00:03:04.71	00:02:30.62	00:00:34.09	796	0.232048
cpp	00:02:41.85	00:01:05.79	00:01:36.06	5007	0.032325
grep	00:02:14.96	00:00:31.52	00:01:43.44	9124	0.014792
sed	00:02:12.86	00:00:32.29	00:01:40.57	8333	0.015944
xgettext	00:02:09.33	00:01:53.24	00:00:16.09	1150	0.112461
date	00:01:59.93	00:00:28.25	00:01:31.68	9888	0.012129
macrogen	00:01:55.72	00:01:03.77	00:00:51.95	2484	0.046586
as	00:01:37.00	00:00:55.62	00:00:41.38	2297	0.042229
find	00:01:22.36	00:00:06.20	00:01:16.16	137	0.601168
elfdump	00:01:22.27	00:00:19.83	00:01:02.44	3293	0.024983
jar	00:01:17.40	00:00:55.44	00:00:21.96	79	0.979747

Figure 5: Top 30 consumers of CPU during a build.

```

...
864018 make install.targ
  864102 make install.targ
    864104 sh -ce /opt/SUNWspro/SC5.0/bin/cc -xchip=ultra -Wc,-xcode=abs32 -W
      864109 /opt/SUNWspro/SC5.0/bin/cc -xchip=ultra -Wc,-xcode=abs32 -Wc,-Qi
        864179 /opt/SUNWspro/SC5.0/bin/cg -xcode=abs64 -oo obj64/cpu_states.o
  864108 make install.targ
    864110 sh -ce /opt/SUNWspro/SC5.0/bin/cc -xchip=ultra -Wc,-xcode=abs32 -W
      864112 /opt/SUNWspro/SC5.0/bin/cc -xchip=ultra -Wc,-xcode=abs32 -Wc,-Qi
        864178 /opt/SUNWspro/SC5.0/bin/cg -xcode=abs64 -oo obj64/ddi_impl.o -
  864122 make install.targ
    864123 sh -ce /opt/SUNWspro/SC5.0/bin/cc -xchip=ultra -Wc,-xcode=abs32 -W
      864124 /opt/SUNWspro/SC5.0/bin/cc -xchip=ultra -Wc,-xcode=abs32 -Wc,-Qi
        864125 /opt/SUNWspro/SC5.0/bin/acompc -i ../../sun4u/os/ecc.c -o /tmp/
...

```

Figure 6: Partial ptree snapshot of the OS/NET build process.

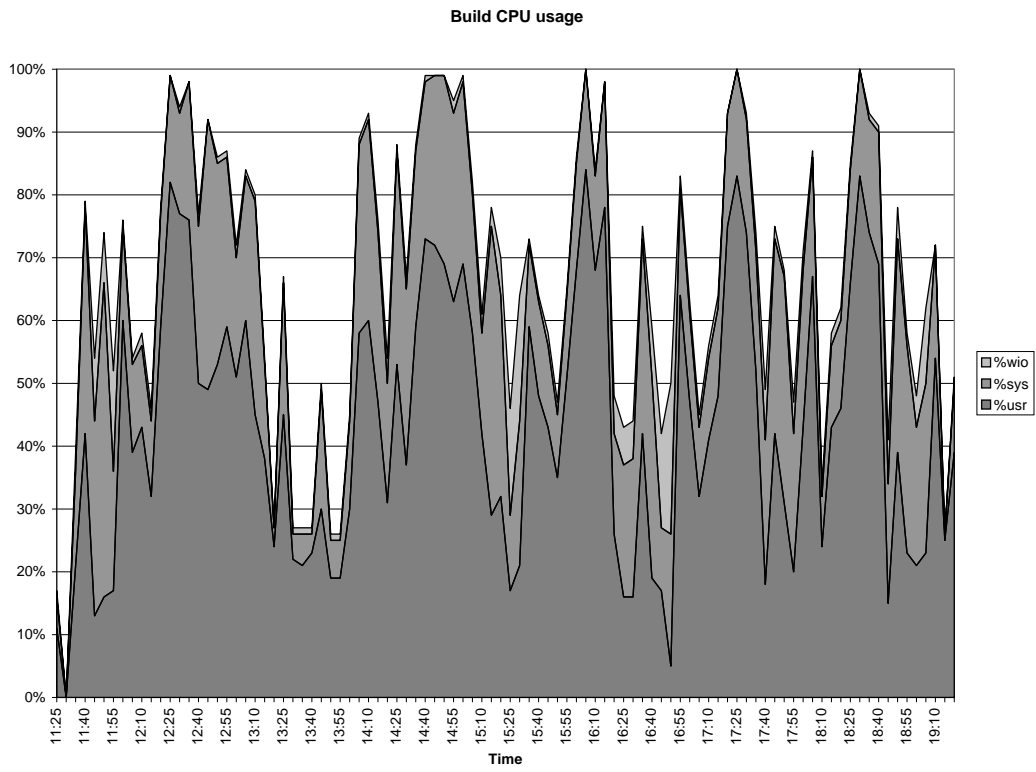


Figure 7: sar(1) CPU data.

This shows that the build process is a very uneven load, with CPU utilisation often being below 30%, for example the period between 13:25 and 13:40. At other times the system is obviously severely overloaded, with very high system time comprising over 30% of the total CPU usage.

To rule out IO bandwidth as a possible performance problem, the `sar(1)` disk I/O data was examined (see Figure 8). During the build no single disk was ever more than 20% busy for any extended period, and the SLVM statistics were usually below 30% busy, and only occasionally peaked to 80% busy. We therefore decided that disk I/O was unlikely to be the constraining factor.

We decided to examine the number of processes active at different points during the build. To do this we queried the process table in Oracle, and calculated the cumulative number of processes that had started and finished every five minutes through the build. For each 5 minute period, the difference between the cumulative number of processes started and finished was graphed; this is given in Figure 9. This gave a measure of the number of active processes in any 5 minute period.

This graph correlates well with both the CPU usage and disk IO graphs, again showing the 'dead spot' in build activity between 13:25 and 13:40. To narrow down the likely cause of this, we examined the tasks that were active during this period using `whattasks`, a SQL script that provides a hierarchical task display analogous to `ptree(1)`. (See Figure 10.)

This shows that the only OS/NET component being built for most of this time period is the `usr/src/lib/ami/jarami` subdirectory, and that this directory takes just under 20 minutes to build. This could be for a number of reasons: other components could have a make dependency on this one, or this could just be the last component to be built during this phase of the overall build process. Whatever the reason, the ability to 'drill down' into the task and process accounting data in this fashion makes it possible to rapidly identify such problems.

As another example of how the data can be manipulated easily once it is inside an RDBMS, a simple SQL report was written to show the number of subprocesses invoked by each `make` invocation, sorted by build phase and working directory, for all `make` invocations that took more than 2 minutes to execute. The SQL used for the report is given in Figure 11, and the resulting report is given in Figure 12.

Again, note that `src/lib/ami/jarami` takes the longest time to build during the optimised build, indicating that this may well be an area that merits further inves-

tigation.

We also noted the extremely irregular load imposed on the system by the OS/NET build. Although the degree of make concurrency was set to 12 jobs, it seemed apparent that the load control mechanism provided by `make` wasn't managing the load imposed on the system very effectively. A SQL query was written to count the number of active tasks (and therefore `make` invocations) active every time a new `make` process started. The SQL for this query is given in figure 13, and a graph of the resulting output is given in figure 14.

From this graph it is evident that the `make` job limit mechanism provides very poor control of the load imposed by the OS/NET build - although the number of concurrent `make` processes is supposed to be limited to 12, the figure is often two or three times the requested number.

Comments

The combined overhead of turning on both `task` and `process` accounting, the `make` wrapper and the interposed `tasksys(2)` system call adds approximately 10 minutes to a 9 hour build, which we felt was a more than acceptable overhead.

We feel that the monolithic nature of the `nightly` script and `makefile` structure currently discourages wrapper/launcher experimentation. The existing statistics provided by the `nightly` script are very coarse-grained and provide little help in identifying problem areas within the build.

We also feel that significant improvements to the OS/NET build process could be made if the `make` utility provided more powerful facilities for controlling the overall workload, for example by accurately enforcing the requested concurrency, and for allowing different degrees of concurrency at different points in the build process.

CONCLUSIONS

In characterizing our attempts to analyse the OS/NET build process with the `exacct` facility, we have identified a number of areas for improvement in the facility itself:

- The security model of `settaskid(2)` is inconvenient for wrapper-based approaches to workload analysis. Even if we had decided to enhance `make(1)` rather than `ksh(1)`, we would still be confronted by a `root` requirement if we wished to invoke `settaskid(2)` directly.
- Given a relaxed security model, the 'task launcher' command, `newtask(1)` should be promoted to the

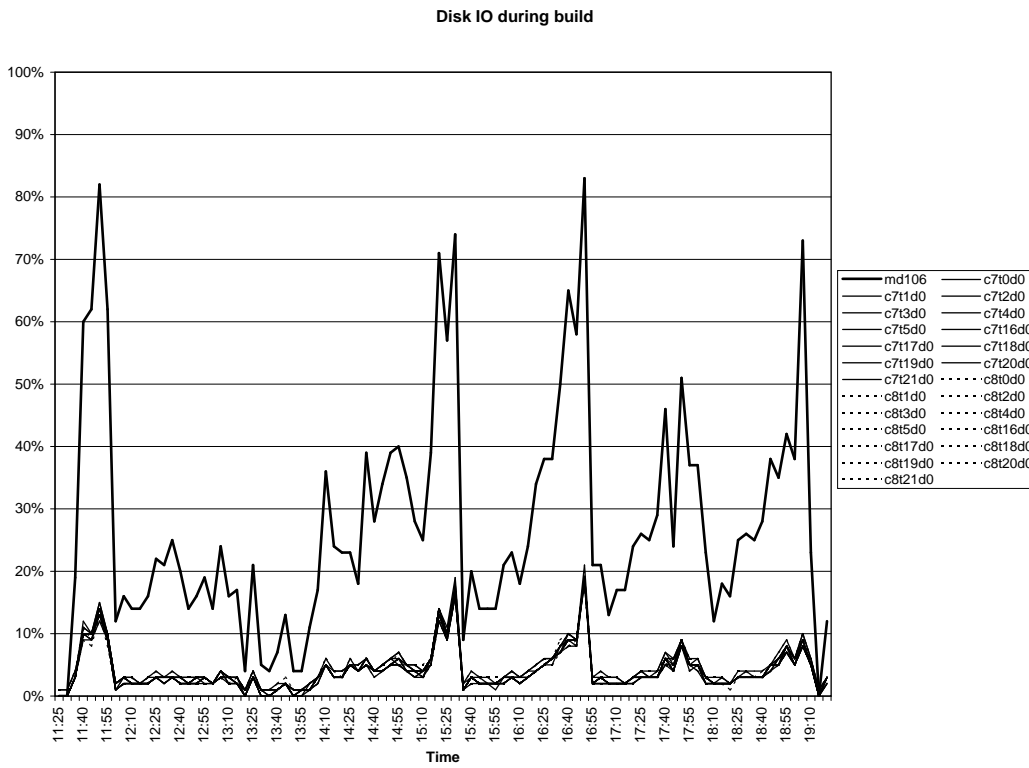


Figure 8: sar(1) disk I/O data.

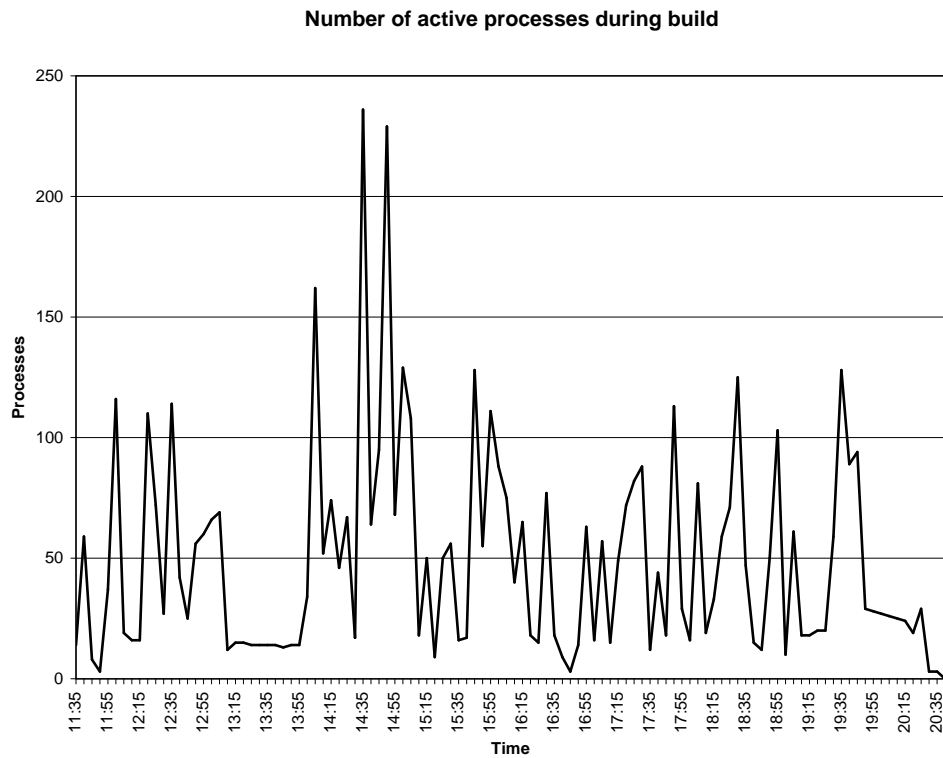


Figure 9: Active processes.

```
basalt> whattasks 0227 27/02/2001 13:25:00 27/02/2001 13:40:00

Tasks active between 27/02/2001 13:25:00 and 27/02/2001 13:40:00
-----
tid:59771 ptid:59770 src (11:52:45.30 -> 15:18:58.33)
  tid:61113 ptid:59771 src/lib (12:36:31.26 -> 14:19:00.16)
    tid:61716 ptid:61113 src/lib/ami (13:23:04.77 -> 13:43:46.92)
      tid:61725 ptid:61716 src/lib/ami/jarami (13:23:05.35 -> 13:41:48.05)
```

Figure 10: Example of the whattasks script being used to identify tasks active during a given time interval.

```

select
  pr.projname "project",
  uxt.hhmmss(t1.cpu_user_sec + t1.cpu_sys_sec) "cpu_tot_sec",
  (select
    count(*)
  from
    process_&&1 p1
  where
    p1.taskid = t1.taskid) "processes",
  substr(t1.cwd, 21) "cwd"
from
  task_&&1 t1,
  project pr
where
  t1.cpu_user_sec + t1.cpu_sys_sec > 120
  and pr.projid = t1.projid
order by
  1, 2 desc, 3 desc;

```

Figure 11: SQL to generate per-invocation statistics for make.

status of a shell built-in, in our standard shells to simplify wrapping. (We could also consider providing a wrapping shell which with certain options would settaskid() itself into a specific project.)

- The ancestor task ID, which will not be made available as a programmatic API, needs to be made available in the task accounting file so that workload components can be placed in sequence.
- The parent process ID is currently not captured in the process accounting file. Availability of this data would have made it possible to identify the make-sh issue directly from the accounting data.

To view *exacct* as merely a better way of doing charge-back accounting is to miss a major opportunity for collecting and analysing performance statistics with a degree of precision and control that until now has not been possible on Solaris. The ability to capture and aggregate data at a range of granularities, along with the ability to embed additional data specific to an application provides an important new tool in the armoury of the systems administrator, capacity planner and performance specialist.

The use of a RDBMS for processing the captured *exacct* data is a key enabler. Without the speed and ease

of access to the data brought to bear by a RDBMS, much of the analysis presented in this paper would not have been possible. This ease of access makes possible ad-hoc examination of the data, which is key when attempting to identify application performance issues.

We look forward to feedback and examples of *exacct* being used to assess and analyse other large and complex workloads.

ACKNOWLEDGMENTS

We have benefited from conversations on this topic with Andrei Dorofeev, Tim Marsland and Mike Shapiro. Blake Jones first conjectured that the inflexibility of the settaskid(2) security model could complicate its application.

REFERENCES

- [1] Stephen Hahn and Tim Marsland, *PSARC/1999/119: Tasks, Projects, and Extended Accounting*.
- [2] Alan Burlison, *PSARC 2000/341: Perl interface to libexacct*.

Processes per directory and build phase
 =====

(Ignoring directories taking < 2 minute to build)

Project Name	User+Sys Time	Procs	Working Directory
onbld_debug	00:06:59.43	1490	src/uts/sun4u/genunix
	00:06:38.61	1490	src/uts/sun4u/genunix
	00:06:29.49	1485	src/uts/sparc/genunix
	00:02:14.03	787	src/uts/sun4u/starfire/unix
	00:02:11.38	769	src/uts/sun4u/starcat/unix
	00:02:10.72	769	src/uts/sun4u/serengeti/unix
	00:02:10.15	771	src/uts/sun4u/unix
	00:02:07.24	787	src/uts/sun4u/starfire/unix
	00:02:03.65	771	src/uts/sun4u/unix
	onbld_nondebug	00:17:50.61	3040
00:11:30.68		11548	src/uts/adb/sparcv9
00:10:54.72		11273	src/uts/adb/sparc
00:09:18.56		818	src/lib/libslp/javali
00:08:20.11		2620	src/cmd/man/src/util/nsgmls.src/lib
00:06:46.47		2920	src/lib/libnsl/sparc
00:06:34.79		6230	src/lib/libc/sparc
00:06:32.88		6003	src/lib/libc/sparcv9
00:06:32.61		6234	src/lib/libc/sparc
00:06:25.68		6238	src/lib/libc/sparc
00:06:24.52		6012	src/lib/libc/sparcv9
00:06:12.37		1488	src/uts/sun4u/genunix
00:05:52.99		1488	src/uts/sun4u/genunix
00:05:47.93		1483	src/uts/sparc/genunix
00:04:28.32		2650	src/lib/gss_mechs/mech_krb5/sparcv9/gl
00:04:28.02		2650	src/lib/gss_mechs/mech_krb5/sparcv9/do
00:04:24.27		2647	src/lib/gss_mechs/mech_krb5/sparc/do
00:04:22.99		2647	src/lib/gss_mechs/mech_krb5/sparc/gl
00:03:55.79		900	src/lib/lvm/libmeta/sparc
00:03:36.15		1472	src/lib/libnsl/sparcv9
00:03:31.78		3340	src/lib/libcurses/sparc
00:03:30.39		3327	src/lib/libcurses/sparcv9
00:03:29.00		3324	src/lib/libcurses/sparc
00:02:48.38		1265	src/lib/libresolv2/sparc
00:02:40.90		2698	src/lib/ami/libami/sparcv9
00:02:38.14		2696	src/lib/ami/libami/sparc
00:02:23.00		3187	src/lib/libbc/sparc
00:02:22.66		3155	src/lib/libbc/sparc
00:02:19.99		240	src/cmd/perl/distrib
00:02:17.04		180	src/lib/fn/context/onc_ns/nisplus/sparcv9
00:02:14.40		364	src/cmd/cmd-inet/usr.sadm/dhccpmgr/com/sun/dhccpmgr/client
00:02:13.55		648	src/cmd/apache/src
00:02:05.73		180	src/lib/fn/context/onc_ns/files/sparcv9

Figure 12: Report showing per-invocation statistics for make.

```

execute uxt.defaults(prec => 2, dfmt => 'HH24:MI:SS');

select
  uxt.odatestr(start_sec) || ',' ||
  (select
    count(*)
  from
    task_&&1 t2
  where
    t2.start_sec <= t1.start_sec
    and t2.finish_sec >= t1.start_sec)
from
  task_&&1 t1;

```

Figure 13: SQL to count the number of make processes that are active every time a new make process is started.

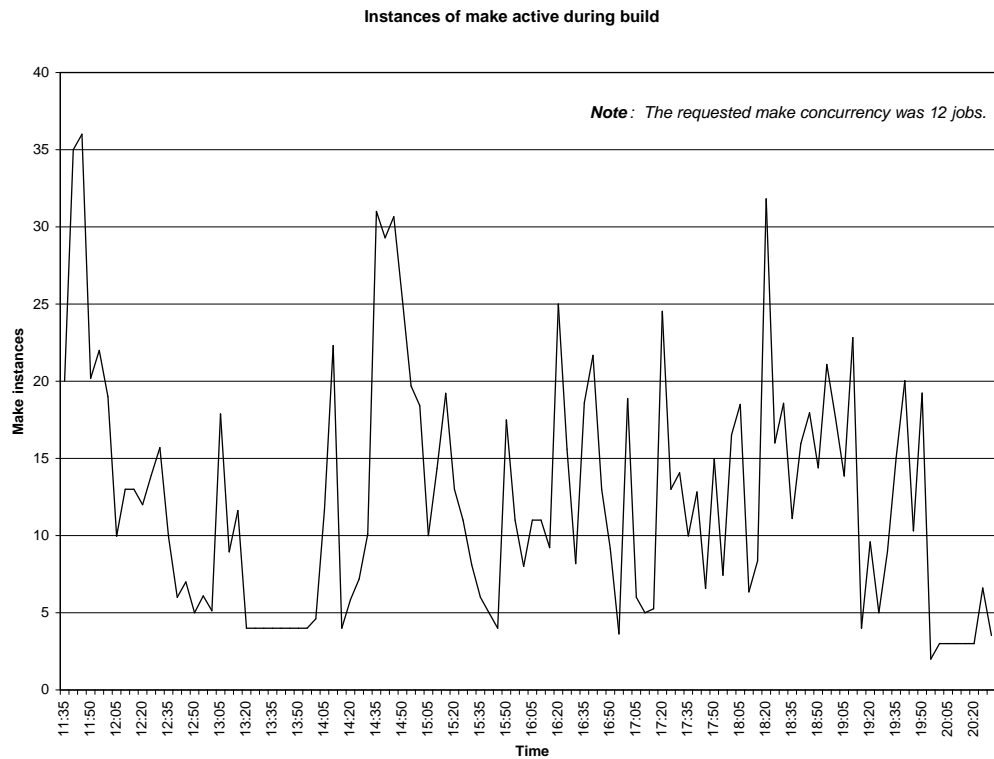


Figure 14: Graph of make concurrency throughout the build.