

CPU Caps Implementation

Andrei Dorofeev

Alexander Kolbasov

Jonathan Chew

December 15, 2006

Abstract

This document describes implementation of the CPU Caps projects. It provides the general implementation overview, discusses the major data structures and interaction with scheduling classes. The CPU Caps design is discussed in [1]. It also outlines the general implementation strategy. The on-line version of this paper is available at [2].

1 Introduction

Solaris provides different mechanisms which can be used to control CPU usage of applications - mainly *Processor Binding*, *Processor Sets*, *Dynamic Resource Pools* and *Fair Share Scheduler*.

There are disadvantages associated with using processor and processor set bindings to control CPU usage. They are too coarse (have CPU granularity), require users to choose specific processors to be used for binding, and can prevent certain Dynamic reconfiguration (*DR*) operations.

The Fair Share Scheduler provides a way to limit CPU usage of zones or projects in situations when workloads compete with each other for the same CPU resources. However, in situations when not all CPU resources are being actively used, CPU shares do not limit CPU usage in any way.

Various customers requested a hard and fine-grained limit on the CPU usage by a zone or a project. The primary motivation for such request is providing consistent user experience independent of the actual machine load.

CPU caps provide a hard CPU usage cap which is enforced even if some CPUs are idle. This is a

stronger performance guarantee than the one, provided by FSS shares, where the end result greatly depends on other workloads that run on the system at the same time.

The rest of this document describes the implementation of CPU caps.

2 Implementation Overview

A CPU cap can be set on any project or any zone. Zone CPU cap limits the CPU usage for all projects running inside the zone. For any project or zone with a cap set to *C* the combined usage of all LWPs running in that project or zone should not exceed *C*% of a single CPU. It is possible to set project caps for projects belonging to capped zones. CPU caps are enforced only for threads running in *TS*, *IA*, *FX*, and *FSS* scheduling classes.

When CPU usage of projects or zones reaches specified caps, LWPs in them do not get scheduled and, instead, are placed on special *wait queues*, associated with the project or zone reaching the cap. These LWPs will start running again only when CPU usage drops below the cap level. Each cap has its own *wait queue*. The time, spent by threads on *wait queues* is reported as “*wait-cpu*” (latency) time by `procfs`¹. Wait times can be seen in the `LAT` column when `prstat(1M)` is invoked with `-m` (report microstate process accounting information) option. CPU time spent by threads on *wait queues* is also accumulated at the `LMS_WAIT_CPU` micro-state accounting state².

¹See `pr_wtime` field of `struct prusage` in `proc(4)` .

²Currently there is a fixed set of micro-state accounting types. Any extension of this set will cause offsets of fields in data structures, embedding micro-state data, to change.

This time, however, is not accounted for, when calculating CPU load averages, unlike the time spent by threads in runnable (sitting on run queues) state. There is no separate accounting for time spent on the wait queue.

When a CPU cap is set for a project or zone, the kernel continuously keeps track of the CPU time used by all user threads within capped zones and projects over a short time interval and calculates their current CPU usage as a percentage. When the accumulated usage reaches the cap, LWPs running in the user-land (when they are not holding any critical kernel locks) are placed on special *wait queues* until their project's or zone's CPU usage drops below the cap.

The system maintains a list of all capped projects and all capped zones. On every clock tick every active thread belonging to a capped project adds total accumulated CPU usage for this thread since it was last checked to its project. The threads on-CPU time is also added to its project when a thread is leaving CPU or is exiting. On every clock tick the current project usage value is decayed by one per cent of its value. Usage from all projects belonging to a capped zone is aggregated to get the zone usage. All accounting uses thread micro-state accounting data.

When the current CPU usage is above the cap, a project or zone is considered over-capped. Every user thread caught running in an over-capped project or zone is marked by setting `TS_PROJWAITQ` flag in thread's `t_schedflag` field and is requested to surrender its CPU. This causes scheduling class specific `CL_TRAP()` callback to be invoked. The callback function places threads marked as `TS_PROJWAIT` on a *wait queue* and calls `switch()`.

An important design decision is to only put threads on wait queues after being trapped in the user-land (they could be holding some user locks, but no kernel locks) and while returning from the trap back to the user-land when no kernel locks are held either. Putting threads on wait queues in random places while running in the kernel might lead to all kinds of locking problems.

3 Data structures

Previous section described the overall CPU caps implementation. Here we describe the major data structures used in the implementation. The two major data structures are the `cpucap` structure (3.1) and the *wait queue* structure (3.2). We also introduce the `caps_sc` data structure (3.3) used to hold per-thread caps-specific data for its scheduling class.

3.1 cpucap

Most of the per-project or per-zone state related to CPU caps is kept in the following `cpucap` structure:

```
typedef struct cpucap {
    kproject_t  *cap_project;
    zone_t      *cap_zone;
    list_node_t cap_link;
    int64_t     cap_value;
    int64_t     cap_usage;
    disp_lock_t cap_usagelock;
    uint_t      cap_flags;
    waitq_t     cap_waitq;
    uint64_t    cap_below;
    uint64_t    cap_above;
    kstat_t     *cap_kstat;
} cpucap_t;
```

The fields of this structure have the following meaning:

cap_project points to the project for which the cap is set;

cap_zone points to the zone for which the cap is set;

cap_link links caps in a list;

cap_value the actual cap value, expressed in nanoseconds per tick;

cap_usage current CPU usage, associated with the cap, expressed in nanoseconds per tick;

cap_usagelock dispatcher lock protecting the `cap_usage` field;

cap_flags describes the state of the cap. It can have the following bits set:

- CAP_ACTIVE** - this cap is enabled
- CAP_ZONE** - this is the zone cap
- CAP_PROJECT** - this is the project cap
- CAP_REACHED** - the cap usage have reached the cap value

cap_waitq *wait queue*, associated with the cap (see 3.2);

cap_below number of ticks spent below the cap;

cap_above number of ticks spent above the cap;

cap_kstat per-cap kstat pointer.

The `cpucap` structure is associated with the project or zone the first time a cap is set for that project or zone. The `cap_flags` field tells whether this is a project or zone cap. For project caps the `cap_project` field points to the associated project and for zone caps the `cap_zone` field points to the associated zone.

All enabled project caps are linked together in a single list called `capped_projects` and all enabled zone caps are linked together in a single list called `capped_zones`. Both lists are protected with the `caps_lock` mutex. The lists are traversed by the clock thread and are modified when the cap is modified or removed, so there is usually no lock contention for the mutex.

When a zone cap is enabled, all projects belonging to that zone are also automatically enabled, but they have a special `cap_value` of zero. If a project already has its own cap, its cap value is unchanged when its zone gets a cap as well. Projects with a cap value of zero participate in CPU usage accounting for the zone, but are never used to enforce a cap.

Whenever an LWP loses its CPU, its on-CPU time, obtained from micro-state accounting data, is added to its project usage. Also, for any thread found on CPU during `clock()` thread scan, its new CPU usage is added to its project usage.

During each clock tick, the `clock()` calls the `cpucaps_clock_callout()` function which scans all

capped project and decays each cap usage by one per cent of its value. For caps belonging to capped zones, it also aggregates the cap usage into their zone cap. The `cpucaps_clock_callout()` function also sets or clears the `CAP_REACHED` flag if the cap usage is above or below the cap, respectively. If the cap is not reached and there are threads waiting on its *wait queue*, a single thread is removed from the wait queue and is made runnable. Note that only one threads is relased from the wait queue per clock tick. This slows down the potential load increase and prevents system thrashing when threads are constantly put on and off wait queues.

Each cap keeps some statistics that it exports through *kstats*. Currently it exports the time spent below and above the cap. The time is calculated in ticks, but is exported in seconds. Kstats are only present for enabled caps and allow users and administrators to check what caps are enabled, what their value and CPU usage is and how much time LWPs they spend above and below the cap. They also show the maximum CPU usage reached.

Most of the fields in the `cpucap` structure are protected by the `caps_lock` mutex. The `cap_usage` field is protected by `cap_usage_lock` dispatcher lock.

3.2 Wait Queues

CPU Caps introcude the notion of the *wait queue*. This queue hosts threads sorted in priority order while they can't run because their project or zone CPU usage exceeded its cap limits. *Wait queues* are always associated with a `cpucap` structure (see 3.1). The wait queue has the following definition:

```
typedef struct waitq {
    disp_lock_t wq_lock;
    kthread_t   *wq_first;
    int         wq_count;
    boolean_t   wq_blocked;
} waitq_t;
```

Here is the description of the various fields of the wait queue structure:

wq_lock protects all operations on the wait queue.

When a thread is placed on the wait queue, its

thread lock is replaced by the `wq_lock` of the queue it is enqueued to.

wq_first is the pointer to the first thread on the queue.

wq_count is the counter of threads in the queue. The caps code polls this value every clock tick without holding any locks, trying to make one thread from a wait queue runnable if the load drops below the cap.

wq_blocked A flag, indicating whether any new threads can be placed on the wait queue. When the CPU cap of a project or zone is disabled, its wait queue is blocked. Any attempts by scheduling class code to enqueue threads on the wait queue will fail. This ensures that no threads are placed on any wait queue of a project or zone which does not have a cap set.

The *wait queue* is very similar to the *sleep queue* and uses the same mechanism to provide list of threads sorted by priority order.

A wait queue is a singly linked NULL-terminated list with doubly linked circular sublists. The singly linked list is in descending priority order and FIFO for threads of the same priority. It links through the `t_link` field of the thread structure. The doubly linked sublists link threads of the same priority. They use the `t_priforw` and `t_priback` fields of the thread structure.

3.2.1 Wait Queue Manipulation

There are three interesting operations on a waitq list: inserting a thread into the proper position according to priority; removing a thread given a pointer to it; and walking the list, possibly removing threads along the way. This design allows all three operations to be performed efficiently and easily.

To insert a thread, traverse the list looking for the sublist of the same priority as the thread (or one of a lower priority, meaning there are no other threads in the list of the same priority). This can be done without touching all threads in the list by following the links between the first threads in each sublist.

Given a thread `t` that is the head of a sublist (the first thread of that priority found when following the `t_link` pointers), `t->t_priback->t_link` points to the head of the next sublist. It's important to do this since a waitq may contain lots of threads.

Removing a thread from the list is also efficient. First, the `t_waitq` field contains a pointer to the waitq on which a thread is waiting (or NULL if it's not on a waitq). This is used to determine if the given thread is on the given waitq without searching the list. Assuming it is, if it's not the head of a sublist, just remove it from the sublist and use the `t_priback` pointer to find the thread that points to it with `t_link`. If it is the head of a sublist, search for it by walking the sublist heads, similar to searching for a given priority level when inserting a thread.

To walk the list, simply follow the `t_link` pointers. Removing threads along the way can be done easily if the code maintains a pointer to the `t_link` field that pointed to the thread being removed.

3.2.2 Wait Queue Interface

Each project and zone cap has its own wait queue. The project wait queue is preferred over the zone wait queue so when both the project and zone caps are reached the thread is placed on the project wait queue since project usage is usually more accurate. Threads on the wait queue are considered to be in the `TS_WAIT` state. The wait queue abstraction provides the following interface:

- `waitq_enqueue(waitq_t *, kthread_t *)`

Place the thread on the wait queue. An attempt to enqueue a thread onto a *blocked* queue fails and returns zero. Successful enqueue returns non-zero value.

- `waitq_setrun(kthread_t *t)`

Take thread off its wait queue and make it runnable.

- `waitq_runone(waitq_t *)`

Take the first thread off the wait queue and make it runnable.

- `waitq_block(waitq_t *)`
Block the wait queue, than take all threads off the waitq and make them runnable.
- `waitq_unblock(waitq_t *)`
Unblock the wait queue.
- `waitq_isempty(waitq_t *)`
Return *True* if the wait queue has no threads on it, *False* otherwise. The check is performed without holding any locks.

Threads on wait queues are marked as non-swappable to avoid having situations where a thread is on the wait queue but it can't be made runnable quickly. The same thing happens for threads on the run queues, and wait queues are viewed as another place where threads are not supposed to spend a lot time on. Threads can be swapped out when they become runnable again.

3.3 The caps_sc structure

There is a small amount of accounting data that should be kept by each scheduling class for each thread which is only used by CPU caps code. This data is kept in the `caps_sc` structure which is transparent for all scheduling classes:

```
typedef struct caps_sc {
    clock_t    csc_timestamp;
    hrtime_t   csc_cputime;
} caps_sc_t;
```

The structure has the following fields:

csc_timestamp time stamp taken the last time the structure was updated.

csc_cputime Total time spent on CPU during thread lifetime, obtained as the sum of *user*, *system* and *trap* time, reported by microstate accounting.

The `caps_sc` structure is used to keep track of the CPU usage by using micro-state accounting data.

Whenever LWP is switched off its CPU (by going through `TS_PREEMPT()` or `TS_SLEEP` or `TS_EXIT`) scheduling classes call the `cpucaps_charge_adjust()` function which does the following:

```
void
cpucaps_charge_adjust(kthread_t *t,
    caps_sc_t *csc)
{
    clock_t    timestamp = lbolt;
    uint64_t   usage = mstate_thread_onproc_time(t);
    clock_t    delta = timestamp -
                    csc->csc_timestamp;

    /*
     * Check what time is it now and when
     * was the last time we charged this
     * thread. If the delta is within two
     * seconds, trust the data, otherwise
     * discard it as stale.
     */
    if (delta >= 0 &&
        delta <= two_seconds_tck) {
        int64_t usage_delta = usage -
                            csc->csc_cputime;
        if (usage_delta > 0) {
            kproject_t *kpj = ttoproj(t);
            cap_project_charge(kpj->kpj_cpucap,
                            usage_delta);
        }
    }
    csc->csc_timestamp = timestamp;
    csc->csc_cputime = usage;
}
```

This function is also called once per tick for all running threads.

3.4 Locking

- The `caps_status` flag is protected by `caps_lock`.
- Lists of projects and zone caps are protected by `caps_lock`
- Wait queues are protected by per wait queue disp lock.
- Wait queue count is protected by wait queue lock.

- Wait queue lock can be grabbed while holding `caps_lock`.

4 Interfaces

The CPU Caps facility provides the following interfaces to the rest of the system:

- `cpucaps_project_add()`
Set project cap of the specified project to the specified value. Setting the value to `MAXCAP` is equivalent to removing the cap.
- `cpucaps_project_remove()`
Remove the association between the specified project and its cap.
- `cpucaps_zone_set()`
Set zone cap of the specified zone to the specified value. Setting the value to `MAXCAP` is equivalent to removing the cap.
- `cpucaps_zone_remove()`
Remove the association between the specified zone and its cap.
- `cpucaps_charge_tick()`
Charges specified thread's project for the time it spent on CPU since last checked and return *True* if project or zone should be penalized because its project or zone is exceeding its cap. Also sets `TS_PROJWAITQ` or `TS_ZONEWAITQ` bits in `t_schedflag` in this case. This function is called by the `CL_TICK()` scheduling class callback.
- `cpucaps_charge_adjust()`
Adjusts specified thread's project CPU usage with micro-state accounting data for thread's on-CPU time. See 3.3.
- `cpucaps_enforce()`
Enforces CPU caps for a specified thread. Places LWPs running in `LWP_USER` state on project or zone wait queues, as requested by `TS_PROJWAITQ` or `TS_ZONEWAITQ` bits in `t_schedflag`. Returns

True if the thread was placed on a wait queue or *False* otherwise.

- `cpucaps_sc_init()`
Initializes the scheduling-class specific CPU Caps data for a thread.

In addition, the following two macros are provided to quickly check whether the any caps are set or not:

CPUCAPS_ON() *True* if there are any enabled caps;

CPUCAPS_OFF *True* if there are no enabled caps.

5 Implementation details

5.1 Scheduling Class Support

CPU caps are supported by TS/IA, FSS, and FX scheduling classes only. Real-time (RT) scheduling class threads cannot be capped (or more accurately, caps defined for RT threads will have no effect). Each scheduling class, supporting CPU caps should provide the following:

- `CL_TICK()` processing: on every clock tick each participating scheduling class should do the following:
 - Provide per-tick accounting for thread CPU usage by calling `cpucaps_charge_tick()`.
 - Make threads belonging to over-charged projects or zones surrender their CPU.
- `CL_SLEEP()`: Account for thread's CPU usage by calling `cpucaps_charge_adjust()`.
- `CL_PREEMPT()`: Account for thread's CPU usage by calling `cpucaps_charge_adjust()`. For threads belonging to zones/projects exceeding their cap and preempted in user mode should be placed on the project or zone wait queue.
- `CL_EXIT()` Account for thread's CPU usage for time thread spent on CPU before exiting. This is important to improve accuracy in charging project/zones with many short-running threads.

The CPU caps code provides common functions for scheduling classes to perform these tasks.

Once per tick the `CL_TICK()` callback should call the `cpucaps_charge()` function and pass it a pointer to a thread and a pointer to the `caps_sc` structure. The function returns *True* if the thread should be surrender its CPU because of cap violation and also sets `TS_PROJWAITQ` and `TS_ZONEWAITQ` bits in the `t_schedflag` field for the thread if it project and/or zone caps are violated.

The `cpucaps_enforce()` function, called by `CL_PREEMPT()` class methods, is responsible for actually placing threads on wait queues. It returns *True* if the thread passed as an argument was placed on wait queue and *False* otherwise.

5.2 Caps and FSS

Threads running in FSS class whose time quanta have not yet expired but which should be put on wait queues because their project/zone caps is reached need to have their priority recalculated because their priority, in part, depends on the overall CPU usage of the project they belong to. The FSS code is trying to simulate the same behavior as if thread's time quantum have just expired.

5.3 Accounting

All CPU caps accounting is done by adding `LMS_USER`, `LMS_SYSTEM` and `LMS_TRAP` micro-state accounting buckets. Any inaccuracy in the micro-state accounting data will influence the accuracy of the CPU caps mechanism³.

It is extremely important to avoid over-accounting. Since project usage is an aggregate of threads usage and number of running threads may be huge, a small per-thread over-accounting adds up in the project and zone usage. As a result, the usage may become very high and this will cause all running threads to be placed on the wait queue. When the usage decays below the cap value, threads will be released from the

³An interesting discrepancy in CPU accounting data for threads with very high dispatcher activity was discovered during CPU caps development. See bug 6498304 for interesting details.

wait queue and the cycle may repeat, creating load thrashing. Doing accounting updates in small increments whenever thread lives CPU and, in addition to that, once per tick, avoids this problem.

5.4 Accuracy

The accounting accuracy depends on the micro-state data and reflects whatever inaccuracies are present there. Since usage for a project is aggregated over all threads, any accounting error is a sum of per-thread error and may become significant when the number of threads is high.

5.5 Decay

CPU usage is decayed by the `caps_update()` routine which is called once per every clock tick. It walks lists of project caps and decays their usages. If CPU usage drops below cap levels, threads on wait queues are made runnable again, one thread per clock tick. When caps are removed, all threads on wait queue are made runnable immediately.

5.6 Observability

The CPU caps implementation provides two facilities for the observability. One is the per cap kstat which shows aggregated project and zone information. Another is the extension of the DTrace `sched` provider for wait queues:

cpucaps-sleep Probe that fires immediately before the current thread is placed on a wait queue. The `lwpsinfo_t` of the waiting thread is pointed to by `args[0]`. The `psinfo_t` of the process containing the waiting thread is pointed to by `args[1]`.

cpucaps-wakeup Probe that fires immediately after a thread is removed from a wait queue. The `lwpsinfo_t` of the waiting thread is pointed to by `args[0]`. The `psinfo_t` of the process containing the waiting thread is pointed to by `args[1]`.

Internally, the probe contains a single pointer to a thread as an argument and this pointer can be used to find the cap structure, controlling the thread.

5.7 Thread States

A new `TS_WAIT` thread state is introduced for threads placed on the wait queue. This state can only be entered from the `TS_ONPROC` state. Threads in the `TS_WAIT` state can only transition to the `TS_RUN` and (less frequently) `TS_ONPROC` states. The state can be checked while holding thread lock. From the userland, `/proc`'s `lwpinfo` structure will report `pr_sname` set to "W" for threads sitting on the wait queues.

5.8 Avoiding CPU preferences

The `clock()` function should walk the list of CPUs starting from different places. If `clock()` always walk the list of CPUs starting from the same CPU with CPU caps present in a steady case when the usage is waving around the cap level, threads running on CPUs that are closer to the head of the list are more likely to be placed on wait queues. This problem is avoided by making `clock()` walk the list of CPUs in more random order.

6 Issues

- Dedicated micro-state for wait queues.

One of the design issues is around whether a new micro-state should be added or not. Right now, `LMS_WAIT_CPU` micro-state is used to keep track of both on-waitq and on-runq CPU times. Adding a new micro-state in an update release may be impossible due to compatibility reasons.

Given the difficulty of extending the micro-state accounting facility and other observability hooks provided by the project we feel that is not very important, at least at the moment, to provide a separate accounting for time spent on wait queues.

- Clock rate

Increasing `clock()` rate might improve accuracy, but it requires careful analysis as it might impact performance on large SMP systems where clock has to do many more things; it might also have negative impact on power consumption on laptops. As one data point, Linux have gone from 100 to 1000 clock rate, and then fell back to 500 times/second.

- Interrupts and pinned threads

Threads which get pinned by interrupt threads don't change their micro-states. Clock tick processing won't happen for pinned threads, but it might look like they've used more CPU time than they actually did just by looking at their micro-state counters. This is a generic micro-state accounting problem though.

- Scalability

Traversing a global list of projects and zones may present a scalability problem when the number of zones or projects in the system is high. We may need more careful algorithms to provide scalable solution in this case.

7 Related Bugs

6464123 `nice(2)` mechanism may starve threads of CPUs

6464127 Time obtained from microstate accounting may go backwards

6464161 Dead `KSLICE` code should be removed

6466380 Project resource set callbacks are needlessly called on every `fork()`

6468003 `prectl` should support the notion of default and infinity

6468451 Errors from setting resource controls should propagate to the caller

6194864 simultaneous `setproject()`'s on the same project can fail to set `rctl`

6498304 too much CPU time winding up in `LMS_WAIT_CPU`

References

- [1] CPU caps web page on OpenSolaris.org
<http://www.opensolaris.org/os/project/rm/rctls/cpu-caps/>
- [2] Implementation description
http://www.opensolaris.org/os/project/rm/rctls/cpu-caps/caps_implementation/.
- [3] PSARC/2006/496 Improved Zones/RM Integration
<http://sac.sfbay.sun.com/PSARC/2006/496>
<http://www.opensolaris.org/os/community/arc/caselog/2006/496>
- [4] PSARC 2006/598 Swap resource control; locked memory RM improvements.
<http://sac.sfbay.sun.com/PSARC/2006/598>
<http://www.opensolaris.org/os/community/arc/caselog/2006/598>