Performance Tuning
for
Low-Latency Applications

This handbook is a guide to achieving consistent and reliable low-latency operation of applications. It explores how modern operating systems schedule user applications and how to observe and modify application operating parameters; and introduces techniques for making sure that system hardware is working cooperatively with your application.

Table of Contents

Operating System Scheduling

One of the responsibilities of an Operating System (OS) is the task of scheduling many different programs across the various resources available in the current operating environment (i.e. machine).

Each time a program is moved off a processor core (either to make way for another process, or because the OS decides to migrate it elsewhere), a cost is incurred. We refer to this cost as *jitter*, and too much of it will lead to measurable performance degradation, causing undesirable variation in *response time latency* (the time taken for the application to service a request). When building low-latency applications, the aim is to minimise this jitter as much as possible.

Modern CPUs are split into multiple physical *cores*, each of which has dedicated hardware resources. Some CPU architectures also provide parallel threads on the same physical core (sometimes referred to as *hardware threads*, or *hyper-threading*), where the core's hardware resources are shared by concurrently executing threads.

On large server-class systems, there may be multiple *sockets*, each containing a CPU chip. In this case, the machine is said to have a Non-Uniform Memory Architecture (NUMA), meaning that memory access times can be affected by the location of running code. Each CPU chip, or NUMA-node, has an associated block of memory (RAM). If a program is running on socket A, then it can access data from the RAM attached to socket A faster than the RAM attached to socket B.

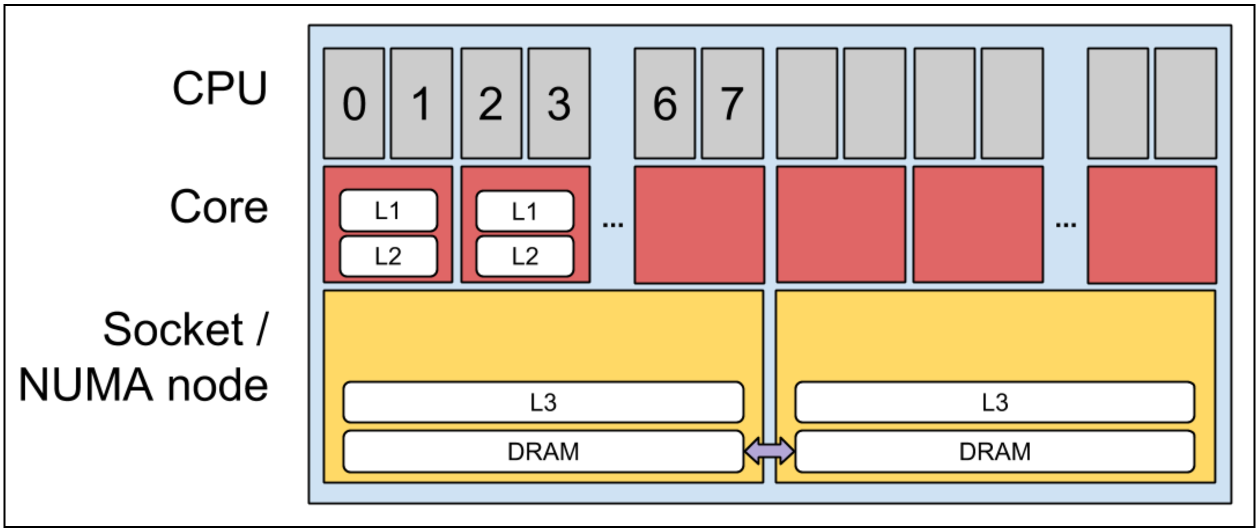

Figure 1: CPU schematic

The dedicated hardware available to each core are its registers and logic units, and its *caches*. The caches are small but fast banks of memory that are utilised by the processor to keep frequently-accessed information close to hand. The caches allow CPUs to process instructions far more quickly than if they were fetching data from main memory (RAM). Caches are small because their fast speed makes them expensive. In a server-class processor chip, the L1 cache (smallest, fastest) might be only 64KB in size; the L2 cache (larger, slightly slower) might be 256KB. The L3 cache, which is shared between all cores in a NUMA node, is slower again than the L2, but might be 64MB.

When a program instructs the CPU to fetch some data, the data is automatically copied into the core's cache so that it will be available if required again in the near future. If the required data is not already in the core's cache, then it must be retrieved from main memory while the CPU waits. Minimising this wait means that the CPU (and the program executing on it) can continue performing useful work.

When the OS moves a new process onto a CPU, it will probably require different data to the program that was running before it, replacing the data that is stored in the cache for that core. Now, when the original program is moved back onto the CPU, the cache will be filled with data that it doesn't need, so it will go back to main memory to retrieve necessary data. This process is called *cache pollution*, and it leads to increased data access times, which contributes to undesirable variations in response time latency.

The diagram below represents the cost of cache pollution, showing the relative access times for retrieving data from L1, L2, L3 and main memory.

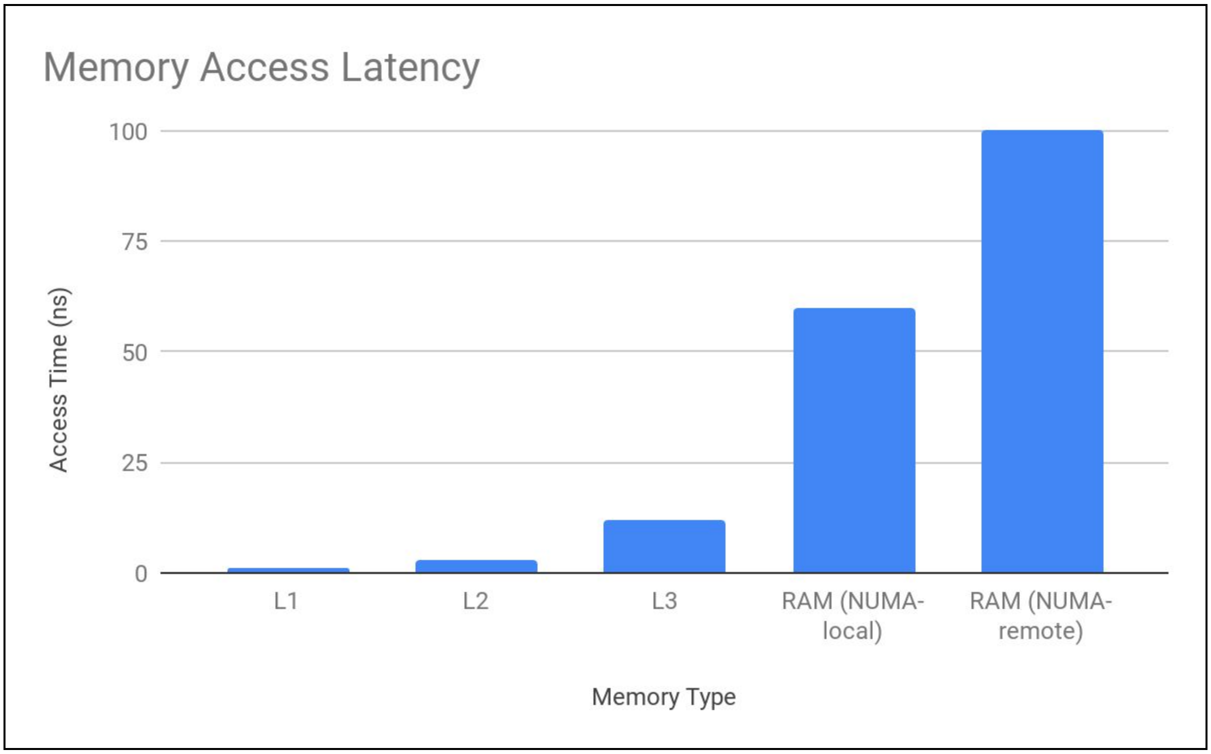

Figure 2: Memory access latency[1]

[1] http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

Running application threads on dedicated cores will ensure that they are not be moved off by the OS, avoiding cache pollution and reducing jitter.

Identifying Important Application Threads

Even very large machines have a relatively small number of available CPUs (e.g. 16 - 128). An application may create hundreds of threads, but in most cases, not all of these threads require continuous access to a CPU. Some housekeeping or monitoring tasks only need to run every few milliseconds. For these threads, it is enough to let the OS do its normal job of scheduling the tasks over the available CPUs.

Other threads, however, will benefit from being run on a CPU that is always available. These threads will typically be *busy*, in that they never voluntarily yield the CPU back to the OS by performing a locking or sleeping operation. These might be network threads, responsible for reading incoming messages from a network interface; or business logic threads, responsible for processing application messages as they arrive.

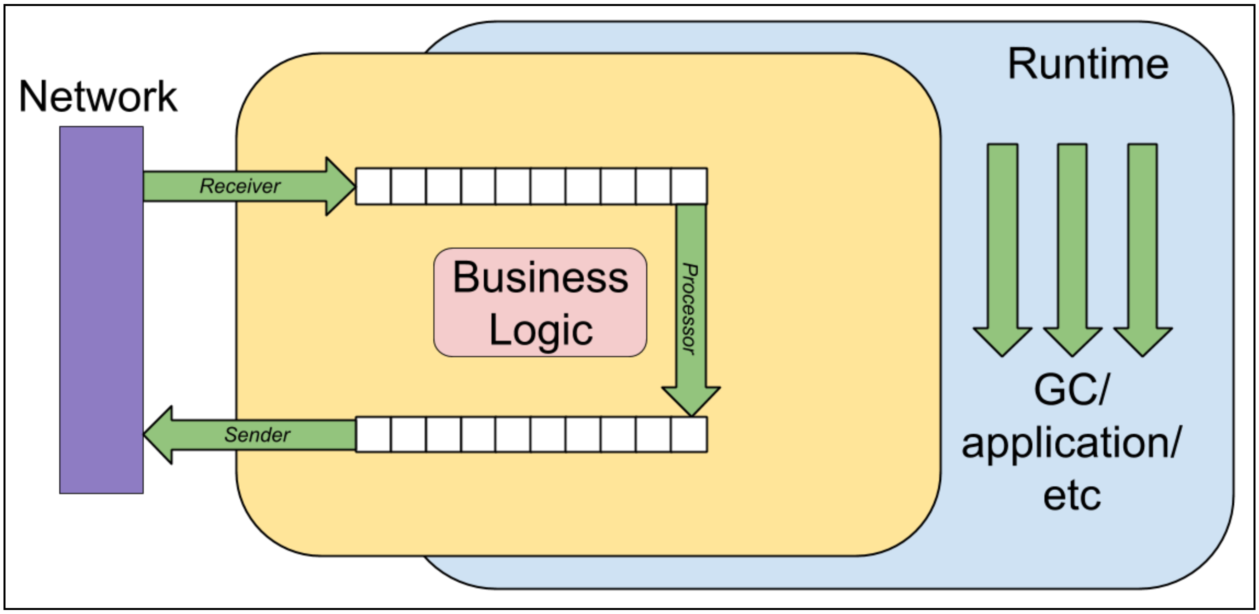

Figure 3: Example threading model

In the above system, there are 3 threads affecting the service time latency. One is responsible for reading from the network (the *receiver*); another for performing logic processing on received messages (the *processor*); and another for sending responses back to the network (the *sender*). Some form of queue is used to move data between each thread. There may be many other threads running as part of the overall process, but their scheduling will not affect latency.

Identifying the important threads is highly application-dependent. To determine important threads, a profiler can reveal where most time is spent when executing the application.

Assigning Threads to CPU cores

Once the important threads have been identified, assigning them to a dedicated CPU means that they will be less likely to suffer from cache pollution. This can happen either as a result of the thread being moved between cores by the OS, or because another thread is contending for the space in the same bank of cache.

The OS treats all CPUs in the machine equally, and while some algorithms can prevent too much CPU migration, even a mostly idle machine will end up scheduling a given thread on many different CPUs over its lifetime.

The process of assigning a thread to a CPU or set of CPUs is referred to as setting its *CPU affinity*. If a thread's CPU affinity is set to 1, then it will only be executed on CPU 1.

On Linux machines, the available CPUs on the system can be viewed by opening a terminal, and typing the `lscpu` command.

To set a thread's affinity, there are two simple options. On Linux systems, the `taskset` command can be used to query and set the affinity of threads running in the system.

To query affinity:

```
$ taskset -cp 3301
pid 3301's current affinity list: 0-63
```

And set:

```
$ taskset -cp 2 3301
pid 3301's current affinity list: 0-63
pid 3301's new affinity list: 2
```

`taskset` is a fairly crude tool, and it is preferable for the application to set affinity on its threads during start up either by using the `sched_set_affinity` system call on Linux, or the `SetThreadAffinityMask` function on Windows.

Monitoring CPU Time

Once application threads have had their CPU affinity set, the OS will continue to treat all CPUs equally and schedule work on all of them. By measuring this effect, we can gain insight into whether application threads are contending for CPU resources.

The Linux OS uses a *scheduler*, which is responsible for assigning CPU resource to processes that are ready to run. The `perf` tool can be used to instrument and observe the scheduler’s behaviour.

In this example, the thread identified by 3103 has its affinity set to CPU 2:

```
$ taskset -cp 2 3103
```

The `perf` tool is used to record scheduler events on CPU 2:

```
$ sudo perf record -e "sched:sched_switch" -C 2
```

The results of the recording show that the application thread was periodically *descheduled* by the OS scheduler in favour of another process (Timer):

```
java  3103 [002]  2307.125403: sched:sched_switch: \
    prev_comm=java prev_pid=3103 \
    prev_prio=120 prev_state=R ==> \
    next_comm=Timer next_pid=2035 \
    next_prio=120
```

During the time taken for the OS scheduler to execute it on another (or the same) CPU, the application thread will not be running, and this is a common source of jitter in latency-sensitive applications. Even if the application thread is run immediately on the same CPU, the other process may have caused cache pollution, leading to further delays while data is re-fetched from slower caches or main memory.

So, once application threads have had their CPU affinity set it is important to ensure that the OS is not running other threads on those CPUs.

CPU Isolation

To stop the OS from scheduling other threads, Linux has a boot parameter *isolcpus*. To use this parameter, the bootloader config needs to inform the kernel that certain CPUs are restricted to use by user programs.

Adding the following to the boot command-line will cause CPUs 2 - 7 to be isolated from the OS scheduler:

```
isolcpus=2-7
```

Other alternatives to using the `isolcpus` parameter are `cgroups` or the `cset` program.

Once CPU isolation has been completed, the OS will not schedule any other tasks to contend with the important application threads, and thread affinity can be set with confidence. The monitoring techniques described above can be used to confirm that application threads are no longer being descheduled from their CPU.

Combining CPU isolation with thread affinity gives an application thread dedicated CPU resource, removing the possibility of being descheduled and suffering from cache pollution.

The example below uses the application from the previous chapter to demonstrate one way of laying out the processes.

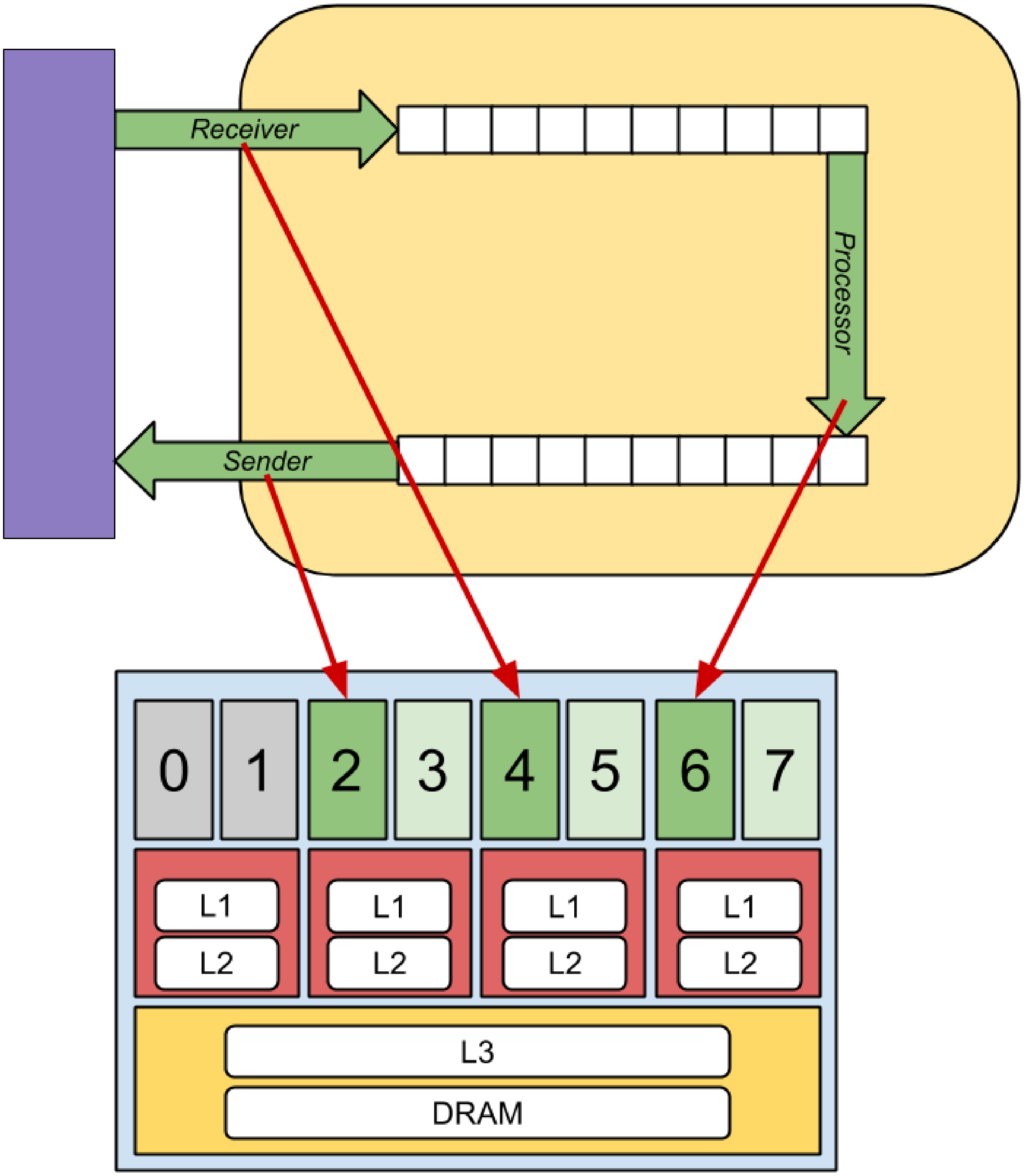

Figure 4: CPU isolation and thread affinity

In this example, CPUs 2-7 are isolated from the OS, and the application's 3 important threads have their affinity set to CPUs 2, 4 and 6 respectively. CPUs 3, 5 and 7 have no workload assigned to them, meaning that nothing will pollute the L1 and L2 caches on those physical cores.

Interrupt Masking

Once these processes have been implemented, it is important to check that the system is behaving as expected. Even when CPUs are isolated from the OS, it is still possible for application threads to be interrupted, introducing unwanted jitter.

This is usually down to *kernel threads*: processes that perform work on behalf of the OS. Kernel threads have a higher priority than user application threads, so they will always interrupt the application when scheduled.

The OS will schedule a kernel thread onto a CPU for a number of reasons. One of the most common is for servicing *interrupts*.

When a hardware device (e.g. network card) wishes to notify the OS of a change in state (e.g. a data packet has arrived), it sends an interrupt signal to a specific CPU. The OS will halt any currently running process, and invoke an *interrupt handler* which will then add an item to a queue to be processed by a *softirq thread*. This process will halt any application threads that are supposed to execute on the interrupted CPU.

Moving interrupts from the isolated CPUs will remove this source of jitter. This can be achieved in the Linux OS by setting the affinity of system interrupts (IRQs). Let’s look at the same example:

To force all hardware devices to only send interrupts to CPU 0 & 1, set the affinity of each IRQ using the interface in the `/proc` file-system:

```
$ for i in $(ls /proc/irq | grep -v default); \
    do echo "0-1" > /proc/irq/$i/smp_affinity_list;
done
```

This final technique should remove almost all sources of process interruption from the CPUs that the application is running on.

Because operating systems are constantly changing, there may well be other kernel functions that are scheduled onto isolated CPUs. Using the `perf` tool to periodically monitor scheduler events on isolated CPUs will ensure that no kernel or other application threads are causing interrupts and jitter.

Further Reading

If this handbook was useful, please consider downloading our other guides:

- Performance Tuning for Cloud Servers
- Hardware Tuning for Low Latency Applications

http://aitusoftware.com/#resources