PontusVision White Paper

Automatic Optimization of Hadoop Yields throughput gains of 241%

Authors:

Leo Martins

Robin Harker

Abstract

Expecting a new user to optimise a Hadoop platform is similar to expecting somebody to be able to tune a guitar the first time he picks it up: it is not going to happen. Imagine being able to get optimum performance from your Hadoop system, as a musical virtuoso such as Dave Gilmour or Jeff Beck might get from a Gibson Les Paul or Fender Strat' without the years of practise.

Figure 1 - Fanned Fret Bass Guitar By Sheldon Dingwall

This white paper shows how an automated thread manager **improved Hadoop throughput performance ranging from 65% to a staggering 241%**. It also shows that hyperthreading had a significantly negative impact of nearly 10%. The test harness used was YCSB using Hadoop's HBase, Zookeeper and HDFS in a single host, and the technology used for the thread manager, as well as the time series graphs was PVTM.

To illustrate the normal scheduler behaviour versus the automated thread manager, we have produced time series graphs with the core location over time of the busiest software threads. This shows how long they remain on the same processor cores (which is the most efficient for performance), or are moved from core to core by the scheduler. When the scheduler moves threads, it causes a dramatically negative impact on performance (see section "How much delay in a single server?" for more details)

So, why the guitar analogy, you may ask. Similar to the fret on Figure 1[1], Figure 2 shows that a time series graph with efficient thread management looks like the strings on a guitar neck. The y-axis on the left shows the core number of each thread. The parallel lines on the graph denote a high level of thread affinity by pinning over time; the jagged green time series on the top shows the total CPU utilization with the scale on the right y-axis

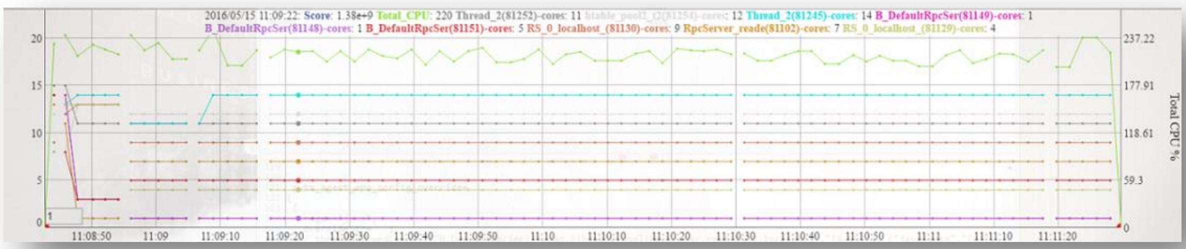

Figure 2 - Guitar strings time series with thread positions over time.

[1] Picture from http://www.dingwallguitars.com/wp-content/gallery/prima-artist-gallery/dingwall-prima-artist-esize.jpg, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=15891935

In contrast, the jagged lines in Figure 3 show the operating system scheduler moving threads from one core to another. This not only pollutes the data caches, but also the translate lookaside buffer (TLB) cache, which is a directory of virtual to physical memory addresses.

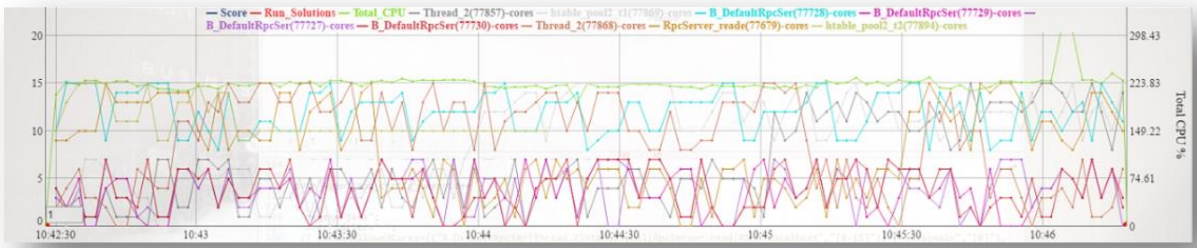

Figure 3 - Threads moved by the OS appear as jagged lines.

The White Paper is sub-divided into the following sections:

YCSB, Yahoo Cloud Serving Benchmark

The **Yahoo! Cloud Serving Benchmark** (YCSB) is an open-source specification and program suite for evaluating retrieval and maintenance capabilities of computer programs. Testers often use it to compare relative performance of NoSQL database management systems.
Yahoo! developed and released the original benchmark in 2010 with the goal of comparing performance of the new generation of cloud data serving systems, such as BigTable, PNUTS, Cassandra, HBase, Azure, CouchDB, SimpleDB, and Voldemort. YCSB is included in the major Hadoop distributions of Cloudera, MapR and Hortonworks, and this makes it ideal as a benchmark to test Hadoop hardware setups and is why it was used to measure the benefits of automated thread pinning with PVTM.

The following URL has further details on YCSB:

https://www.cs.duke.edu/courses/fall13/cps296.4/838-CloudPapers/ycsb.pdf

Hadoop

Hadoop or the Hadoop ecosystem, which includes a growing number of components (e.g. map/reduce, Spark, Yarn, HBase, Impala), is becoming the standard framework for large scale, or “Big Data” analytics. Originally, Google and Yahoo developed Hadoop for internet-scale analysis. Now, most major enterprises as well in government research laboratories, universities, retailers, and hospitals are using Hadoop to analyse large-scale unstructured data.

Such is the groundswell of support and interest in Hadoop that many of the proprietary data analysis vendors, such as Informatica, IBM, Teradata, and Syncsort are offering integration of Hadoop components with their existing analytics packages.

Objectives

The main objective of this white paper was to determine whether YCSB benchmark results using HBase on top of the Hadoop Distributed File System (HDFS) could get increased throughput from an automated thread manager. As we were running the tests, we also measured the impact of the following environmental changes on the YCSB throughput:

1. hyperthreading (on Sandy Bridge),
2. the impact of automatic thread management on a freshly built Broadwell server tuned to save power.

The secondary objective of the white paper was to make the tests easily reproducible. To do so, we performed all the tests in a single server hosting both a stand-alone version of HBase (version 0.98), with embedded Zookeper and HDFS instances, as well as the YCSB client. Lastly, because disk performance may vary from environment to environment, we ran the tests using a tmpfs-mounted drive; this should be easily reproducible on any server, and should display similar performance characteristics to the SUTs used here.

Test Cases Summary

Table 1 shows a summary of the six test cases we ran to produce this white paper:

Table 1 - Summary of Test Cases

Test ID	SUT Intel Micro-architecture	Hyper Threading	Server manually tuned for performance	Thread Manager (PVTM)	Overall Throughput (avg of 5 tests)
T-001	Sandy Bridge (E5-2680)	Enabled	Yes	Disabled	3987.30
T-002	Sandy Bridge (E5-2680)	Disabled	Yes	Disabled	4365.28
T-003	Sandy Bridge (E5-2680)	Disabled	Yes	Enabled (v23)	5305.21
T-004	Sandy Bridge (E5-2680)	Disabled	Yes	Enabled (v24)	**6596.51**
T-005	Broadwell (E5-2650 v4)	Disabled	No (Fresh Install)	Disabled	2090.80
T-006	Broadwell (E5-2650 v4)	Disabled	No (Fresh Install)	Enabled (v24)	**7134.89**

Here is a brief description of Table 1's columns:

- The first column has a test identifier, which is used to track the test cases throughout this document;
- The second column has the microarchitecture of the system under test (SUT); the Sandy Bridge processor we tested (E5 2680) was launched in March 2012, whereas the Broadwell processor (E5 2650 v4) was launched in April 2016;
- The third column shows whether or not hyperthreading was enabled in the platform; Hyperthreading enables cores to run parallel threads of code sharing the same level 1 cache;
- The fourth column shows whether or not the system administrator applied any previous manual tuning to the system;
- The fifth column shows whether the Pontus Vision Thread Manager (PVTM) was enabled, and which version was running (as seen in the results, the new version 4.7.0.24 has dramatic performance improvements).
- Lastly, the sixth column has the average overall throughput over five runs for each test case. Table 2 shows the full set of throughput figures reported by YCSB for each of the tests, which were used to calculate the sixth column in Table 1:

Table 2 - Full set of test results across all 5 runs for each test case

Test Case	Average	Run 1	Run 2	Run 3	Run 4	Run 5
T-001	3987.30	4656.73	3822.06	3654.58	3797.29	4005.85
T-002	4365.28	4165.00	3991.67	4883.58	4174.46	4611.72
T-003	5305.21	5344.91	5206.76	5275.79	5321.16	5377.41
T-004	6596.51	6540.39	6664.53	6617.48	6619.01	6541.12
T-005	2090.80	2352.11	2056.16	1819.38	2031.86	2194.48
T-006	7134.89	7278.97	7099.15	7022.67	7129.11	7144.54

The baseline test T-001 on the Sandy Bridge server started with hyperthreading enabled, but this yielded poor results, so all other tests had hyperthreading disabled. Unfortunately, we did not have time to run hyperthreading tests in the Broadwell platform to see whether there are any improvements in behaviour with a newer platform. Disabling hyperthreading on the Sandy Bridge platform improved performance by 9.5% when compared to test case T-002's results. Note that we used PVTM to do a 'soft-disable' of the hyperthreads, making them 'junk cores' rather than physically switching them off on the BIOS. On previous tests, we have noticed that this has the same effect as changing the BIOS settings, with the advantage of not requiring a system administrator to log into the server's LOM interface and reboot the server.

Test cases T-003 and T-004 show the impact of an automatic thread manager on system performance. Test case T-003 shows the impact of running version 4.7.0.23 of PVTM to manage threads, which increased performance by 21.5% when compared to T-002, and by 33.5% when compared to T-001. Similarly, T-004 shows that the latest version of PVTM (4.7.0.24) had an even bigger performance boost of 51.11%, or, when compared to T-001, a **65.44% performance boost**. The key difference between PVTM versions .23 and .24 is a new mechanism that PVTM uses to highlight the busiest I/O threads in the system. This enables PVTM to quickly identify and move the most relevant I/O threads as close to each other as possible, resulting in massive performance boosts; the only downside of this new technique (which can actually be switched on or off) is increased CPU utilization; however, the results are well worth the cost.

The most staggering improvements were the 241% between the freshly installed Broadwell system on test cases T-005 and T-006 with and without PVTM 4.7.0.24. This shows how PVTM can have a massive impact on a fresh installation without requiring any experience to tune the system. This 'freshly installed' state of the server is quite common in the industry, where system administrators build servers to reduce operational expenditure (OpEx) by conserving power during idle periods. Though this works well for idle systems, the downside is a dramatic impact in performance when the systems are used. When users have long-running jobs, the extra 241% of time it takes to run the workload often negates the idle power savings, costing more in OpEx. Nevertheless, this configuration reflects the state of several data centres that have general-purpose servers.

System Under Test

The following sections describe the software configuration as well as the hardware details of the SUT for the six test cases

Test Harness Configuration:

To setting up the tmpfs partitions, we added the following entries to the /etc/fstab file:

```
tmpfs /tmp tmpfs defaults,noatime,nosuid,nodev,mpol=local,mode=1777,size=10G 0 0
tmpfs /home/ycsb/hbase/  tmpfs
defaults,noatime,nosuid,nodev,mpol=local,mode=1777,size=10G 0 0
```

And then we executed the following commands (as root):

```
mount -a
chown -R ycsb:ycsb /home/ycsb/hbase
```

The first line ensures that the /tmp directory is mapped into memory, and the second line ensures that the /home/ycsb/hbase directory is mapped into memory. For convenience, the locations of the tmpfs memory areas were set to be local to the threads accessing them.

The YCSB configuration and start-up scripts used for all six tests was identical, here is the bash script used:

```
for ((i=0;i<5;i++));
do
  sudo sysctl -w vm.drop_caches=3;

  /home/ycsb/hbase-0.98.16.1-hadoop1/bin/start-hbase.sh;

  sleep 60;
  /home/ycsb/ycsb-0.5.0/bin/ycsb run hbase098 -P /home/ycsb/ycsb-
0.5.0/workloads/workloadcustomread -jvm-args=-javaagent:/home/ycsb/pontus-
vision/4.7.0/linux/pvtm-agent.jar -cp /home/ycsb/hbase-0.98.16.1-
hadoop1/conf -p table=usertable -p columnfamily=family;

  /home/ycsb/hbase-0.98.16.1-hadoop1/bin/stop-hbase.sh;

done > ~/T-00x.out 2>&1 &
```

Where:
`~/T-00x.out` was replaced with the relevant test ID (e.g. T-001.out).

The YCSB workload configuration was set as follows:

```
[ycsb@fsl-0032 ~]$ cat /home/ycsb/ycsb-0.5.0/workloads/workloadcustomread
recordcount=1000
operationcount=1000000
workload=com.yahoo.ycsb.workloads.CoreWorkload

readallfields=true

readproportion=0.5
updateproportion=0.5
scanproportion=0
insertproportion=0

requestdistribution=zipfian
```

Hardware Details

The specification of the system under test for test cases T-001 to T-004 was the following:
2 x E5-2680 (20MB Cache, 2.7 GHz) 8-cores per CPU socket (launched in March 2012)
Memory - 64 GB - 1333 MHz (8GB modules)
Operating system, RHEL 7.2, release 7.2.1511 (Core)
N.B. This system under test was kindly provided by Intel Corporation

The specification of the system under test for test cases T-005 and T-006 was the following:
Super micro SYS-6028TP-HTR, dual socket, TWIN Square server, with Broadwell chipset
2 x E5-2650 v4 (30MB Cache, 2.20 GHz) 12-cores per CPU socket (launched in April 2016)
Memory - 128GB - 2133 MHz (16GB modules)
Operating system, RHEL 7.2, release 7.2.1511 (Core)
N.B. This system under test was kindly provided by XMA Ltd, www.xma.co.uk

Test Results Details

The time series graphs on the next page show the thread layouts during the entire runtime of test cases T-001 to T-004. The jagged lines of the two graphs (Figure 4 and Figure 5) reflect the normal Linux scheduler behaviour without a thread manager. This shows threads moving to different cores, which causes several tiny delays over the total test run.

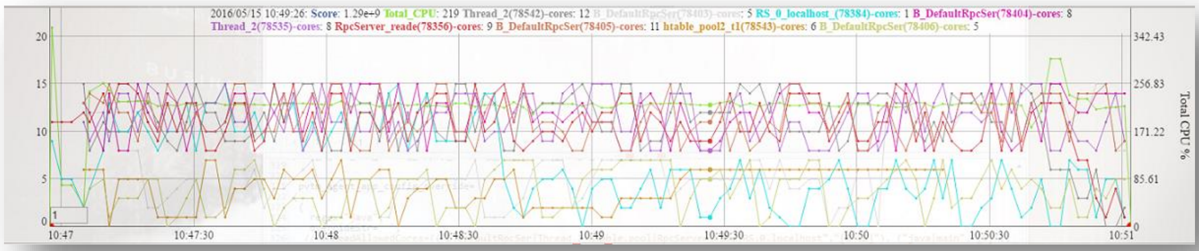

Figure 4 Thread positions over time for T-001 (3987.301 ops/sec)

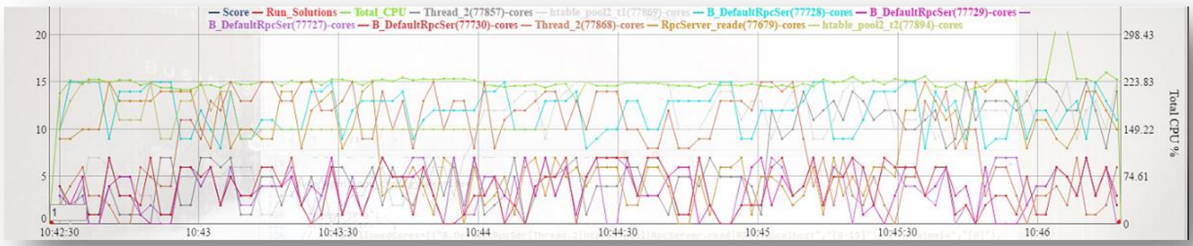

Figure 5 Thread positions over time for T-002 (4365.283 ops/sec - T-001 baseline gain = 9.48%)

In contrast, Figure 6.and Figure 7 show how PVTM creates smooth parallel lines. This "guitar string" pattern maintains much better thread-to-core affinity, which delivers lower latency. The main reason for the performance boost is a reduction in context switching, which causes translate look-aside buffer (TLB) and cache misses. This enables the threads to focus more on the current tasks rather than having to waste time finding previously cached memory addresses through TLB misses, and re-fetch data from slow remote locations through cache misses. The right hand-side axis on these time series also shows the CPU utilization of all the threads in the process. Comparing figures 5 and 6 shows an increase of around 200% CPU utilization between the two tests. At first site, this may seem high; however, in the Sandy Bridge SUT that only represented around 15% of difference in increased power consumption (from 100 to 115 watts) -- well worth the added performance boost.

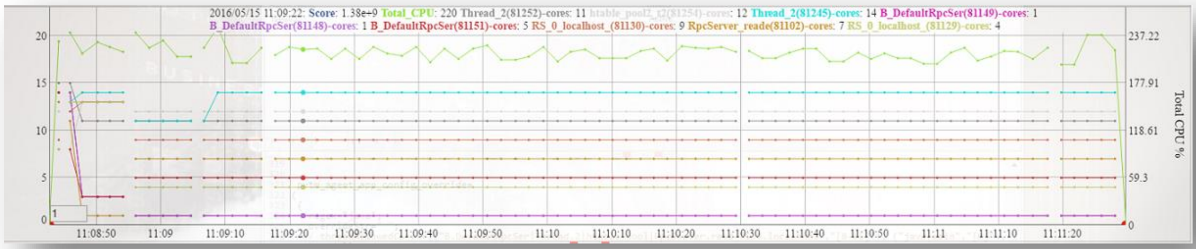

Figure 6 Thread Positions over time for T-003 (5305.206 ops/sec - T-001 baseline gain = 33.05%

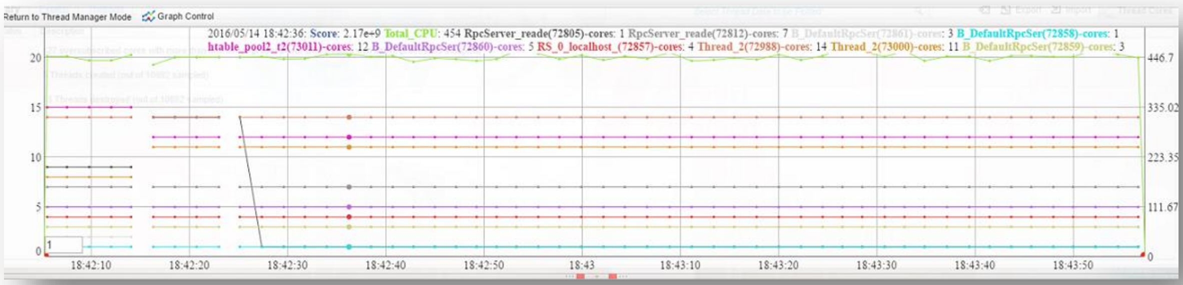

Figure 7 Thread Positions over time for T-004 (6596.506 ops/sec - T-001 gain baseline gain = 65.44%

As the next section will show, the "guitar string" patterns of thread management amount to many improvements of tiny amounts of time. The tiny delays that PVTM saves with thread management only last hundreds of nanoseconds to microseconds; at first, these seem negligible, however, as seen in the test results, their cumulative impact even over a few minutes of tests is enormous. The next section shows more details on how much delay there is in a single server.

How much delay in a single server?

Most people think that the delays inside of a single server are negligible; however, the cumulative effect of these delays is very large. The speed at which threads communicate with each other can be over one hundred (100) times slower depending on which cores they are running in a single server. The relative distance between the CPU cores and their memory access affects how quickly data can move between threads.

Table 3 shows rough relative speeds of sending data between threads measured on a four CPU server. This table reflects the access speed of memory in different cores, as well as the amount of time that it takes to re-fetch data from main memory if not cached. Access speed is relatively fast when the CPU accesses data stored in level 1, level 2 and level 3 caches. The access speed is slower when the CPU accesses local memory, slower when the CPU access remote memory from a neighbouring CPU, and even slower when the memory is not in a local, but a remote neighbour without a direct cross bridge.

Table 3 - Distance Ratio Table

Level 1 Cache	Level 2 Cache	Level 3 Cache	Local Memory	Remote Memory	No-Cross Bridge Memory
1	2	6	10	83	113

It is important to notice that these speeds can vary dramatically depending on the thread behaviour. As such, applications will seldom always run at the fastest speed or at the lowest speed. As the Intel Haswell stats below show, the time to access memory can vary from 4 cycles to access L1 to over 45 cycles plus around 57ns to access main memory. If a thread never moved, and always accessed memory that was within the TLB and in L1 cache, it would use around 4-5 cycles to access the data. In contrast, a thread constantly being moved across cores (in the same CPU) would be missing L1/L2 cached data, having to go to L3 or main memory and having its translate lookaside buffer (TLB) trashed would use 36 cycles + 9 cycles + 57ns to access the data. In reality, threads will never behave as perfectly as the 4-5 cycles 100% of the time; however, when threads are not pinned, they tend to be constantly in the misbehaved state. By managing the location of threads, PVTM is trying to increase the chances of the good 4-5 cycle behaviour.

Here are some stats for Intel's Haswell processor:

- L1 Data Cache Latency = 4 cycles for simple access via pointer

- L1 Data Cache Latency = 5 cycles for access with complex address calculation (size_t n, *p; n = p[n]).
- L2 Cache Latency = 12 cycles
- L3 Cache Latency = 36 cycles
- RAM Latency = 36 cycles + ~ 57 ns
- L1 Data TLB - miss penalty of 1-8 Cycles , with 4x1Gb/ 32x2Mb / 64x4Kb entries
- L2 Data TLB - miss penalty of 9 -22 cycles with 1024 2Mb / 1024 x 4Kb entries

Note that one cycle is equal to the inverse of the core's clock speed, so if a core is running at 2.2GHz, one cycle is 1 / 2.2GHz ~= 4.54 ns

The next section explains at a higher level how PVTM uses these various time constraints to figure out the best location of the threads in a server.

Pontus Vision Thread Manager (PVTM)

PVTM speeds up software applications by better placing software threads onto hardware cores to maximise performance. A secondary advantage, (which is arguably as important), is that it provides a great insight of application behaviour. PVTM is able to show at any point in time where threads were running, how much CPU utilization they had, and which threads were communicating with each other. For Java applications, PVTM also exposes the Java thread names to the operating system, providing even more clarity of what is happening within the JVM.

A large scale Hadoop analysis may take hours or days to run and, due to its resilient design, requires large numbers of physical servers; 100-1,000 servers is not uncommon, so the costs of using Hadoop are substantial and ways of reducing the cost are of interest to all. PVTM is an accelerator, so allows you to either, run more analyses on your Hadoop infrastructure, or reduce the total number of servers and so make capital expenditure, or CapEx savings (e.g. reduction in costs by purchasing fewer servers), as well as operational expenditure, or OpEx savings (e.g. less electricity required to run the servers).

What is the logic behind PVTM?

Modern operating systems (OSs) on modern non-uniform memory access (NUMA) servers are not very good at managing threads for performance-sensitive apps. OSs typically balance the load across various cores rather than focus on application performance. When dealing with performance-sensitive applications, balancing the load across various cores causes application latency to increase significantly.

As an example, in Figure 8 below, there are two Four-CPU servers running the same workload. This hypothetical example shows a pipeline with seven steps. The OS on the left server distributes the red threads across all CPUs in a round robin fashion; by doing this, the distances for data movement are much greater. Therefore, the time to move data between the threads is several times greater than in the server on the right. For many applications, constraining the threads to fewer CPUs can significantly increase performance.

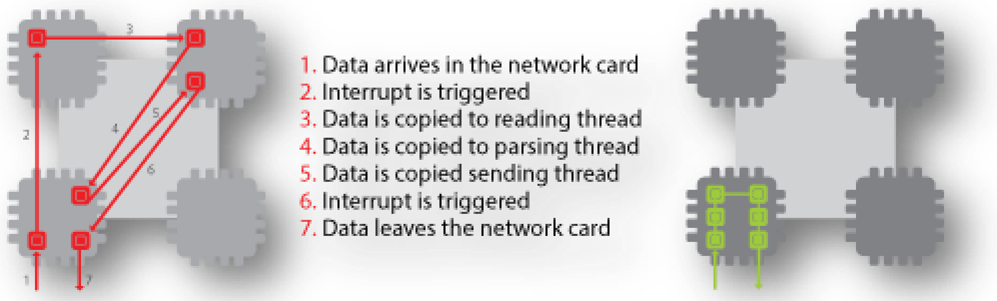
1. Data arrives in the network card
2. Interrupt is triggered
3. Data is copied to reading thread
4. Data is copied to parsing thread
5. Data is copied sending thread
6. Interrupt is triggered
7. Data leaves the network card

Figure 8 - Scheduler behaviour with and without PVTM

PVTM improves the probability that software threads will use less time to communicate with each other. PVTM places threads that have high levels of communication as close to each other as possible whilst at the same time avoiding context switches. To achieve this, PVTM captures the following information about the system:

- Firstly, PVTM analyses the target hardware layout figuring out which cores are closest to each other.
- Secondly, it analyses the communication patterns between threads including which threads use sockets, shared memory and locks to talk to each other.
- Thirdly, PVTM captures the CPU utilization and current location of each thread, as well as optionally the position of I/O devices, such as disks and network cards.

PVTM then sends all this information to a simulator that applies a performance score to the system. The score penalizes threads that have strong communication links to each other, but that are located in cores that have long relative distances to each other. The score also penalizes context switches by avoiding moving active threads to the same core as other active threads as much as possible. The simulator is then capable of running millions of what-if analysis to determine a thread execution layout that improves the score as much as possible.

How does PVTM work?

As seen in Figure 9 below, PONTUS VISION Thread Manager (PVTM) has 3 main components:

1) PVTM Agent – (pvtm-agent) – A lightweight single threaded agent that collects information about the hardware, and discovers the data communication patterns between threads. PVTM Agent is written in C, and usually uses < 1% CPU to capture its data. To aid PVTM Agent discover the thread pinning strategies, the following components may also be used:
   a) libpvtm-agent-preload.so – On Linux, a library that can help the PVTM Agent discover communication patterns between threads. This can be injected in existing applications without recompiling them by using the LD_PRELOAD environment variable.
   b) pvtm-agent.jar – an optional java agent file that exposes the names of Java threads to the operating system. To use this, you need to change your JVM command line to add the `-javaagent:<path to the pvtm-agent.jar file>`.
   c) pvtm-agent-preload-windows.dll – On Windows, a library that helps PVTM Agent discover communication patterns between threads. PVTM Agent automatically connects to the applications that need to be monitored using this DLL along with remote debugging techniques.
2) PVTM Simulator / Thread Manager – (run-threadmgr.sh) – a Java 7 standalone simulation engine that receives TCP/IP connections from PVTM Agent, and can take the hardware information, as well as the data communication patterns between the threads to produce an optimal layout of software threads on the hardware cores.
3) PVTM GUI Server (run-gui.sh) – a self-contained server that hosts a browser-based graphical user interface. The GUI enables users to visualize the layout of the threads, and produce scripts for static thread pinning configurations. Users can use this in environments where PVTM Agent is unable to run.

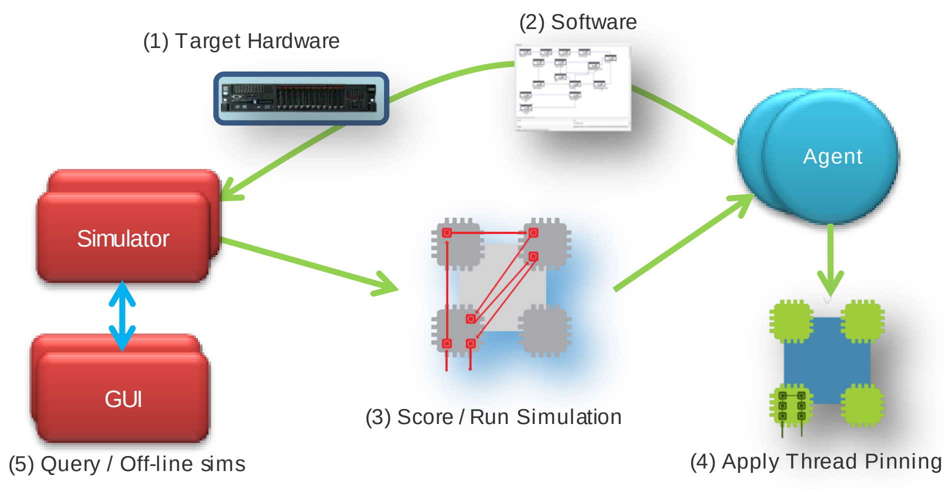

Figure 9 - PVTM Architecture

Conclusion

As seen by the test cases above, Pontus Vision Thread Manager (PVTM) makes a significant performance improvement to the YSCB benchmark. Tuning the threads like guitar strings improved throughput by 241% in a system tuned for power saving without any special sysadmin skills required, and 65% on a system pre-tuned for performance.

To follow on from the guitar analogy, PVTM delivers plenty of power, with a low-strung action that delivers a smooth predictable tone you can rely on. PVTM even allows the novice axeman to become a John Mclaughlin or Eddie Van Halen and play 241% faster.