MAINFRAME EFFICIENCY AT HIGH UTILIZATION

Bob Rogers

Overview

In 2007, Gary King wrote a short paper to answer the questions,
“Can Any Single Workload Be Run At 100% CPU Busy?” and,
“Is there a growth in CPU time per transaction at higher utilization?”.

Unfortunately, the answers are “No, they can’t” and “Yes, it grows”.

Since homogeneous workloads cannot be efficiently run at 100% CPU busy,
we can change the question to:
“How can a platform efficiently run multiple diverse workloads?”.

In this presentation, we discuss some of the more recent technologies that
enable the IBM mainframe platform to perform very well at high utilization
with heterogeneous workloads despite these challenging realities.

What it takes to be efficient

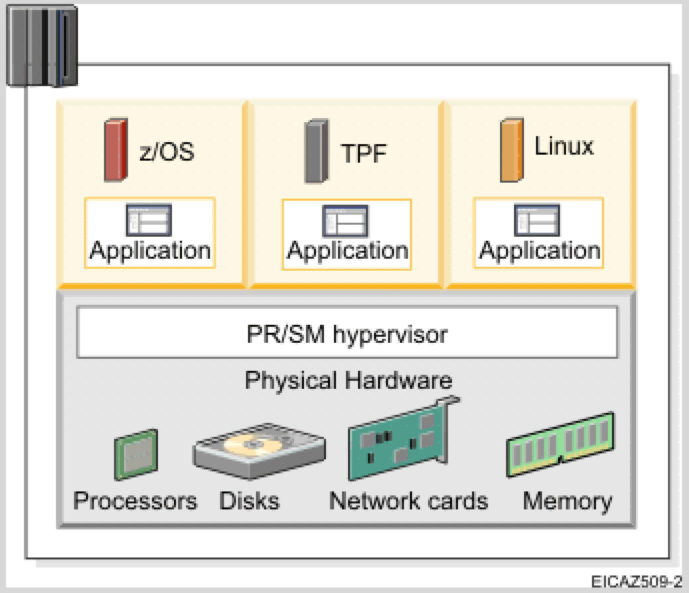
To operate efficiently at high utilization it is necessary to:

1. keep the CPUs running near 100% busy.
   - This can be prevented by insufficient overlap of CPU and I/O. Work is delayed waiting for its data and the CPUs can go idle.
2. devote nearly all the CPU cycles to doing useful work rather than unproductive overhead.
   - For example, inter-processor serialization can lead to “spinning” which accomplishes no real work.
3. ensure that the most important work achieves it performance goals.
   - This may cause less important work to suffer when demand is high, but such is life.

Running Multiple Diverse Workloads

- The primary technology that is used to run multiple workloads on a single processor is virtualization.
- The IBM mainframe supports powerful, multi-tier virtualization with PR/SM in the hardware and z/VM as a software product.

Here is a simple schematic of PR/SM sitting between the operating systems and the hardware.



z/VM also can be added on top of PR/SM to provide a second level of virtualization.

* Multiple workload on z/OS will be discussed later.

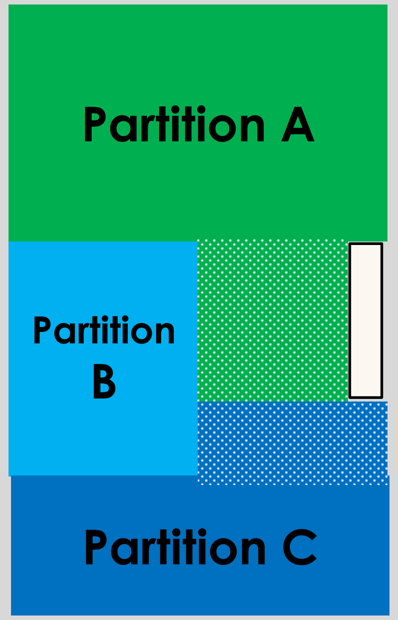
Virtualizing Production Workloads

Installations have been able to run workloads in a virtualized environment since the 1970s, but it wasn’t the dominant way to operate until the introduction of PR/SM in the late 1980s.

PR/SM allows an installation to define a number of partitions on a physical box and give each a ***guaranteed share*** of the total processor capacity.

It redistributes any guaranteed share not consumed by a partition (called ***white space***) to other partitions which have more work than they can do with their guaranteed share.

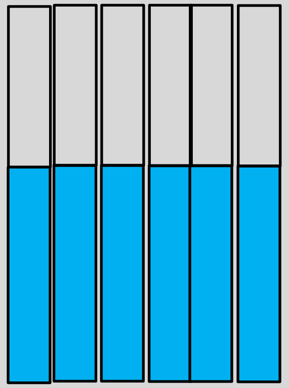
Obviously, this goes a long way toward getting the utilization of the whole box up near 100%.



Partitions A and C are given access to the share that partition B is unable to consume

Hiperdispatch

Horizontal CPU Management



Partition with 3.3 CPU share. 6 horizontals each with 55% share.

To use the extra capacity other partition couldn't use, a partition needs to have extra logical processors beyond what it needed to consume its guaranteed share.

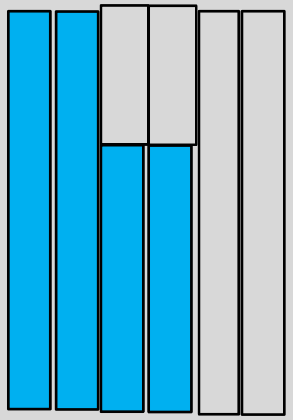
Non-hiperdispatch spreads the guaranteed share and any extra share across all the logical processors and experience PR/SM dispatching and cache damage overheads.

Hiperdispatch assigns vertical high processors that have 100% share and experience no PR/SM dispatching or cache damage.

The vertical medium processors have split the remainder of the share and can consume some additional share.

Vertical low processors are left ***parked***, i.e. running no work, until there is additional share, called white space, left by other partitions. When white space disappears, they are parked again.

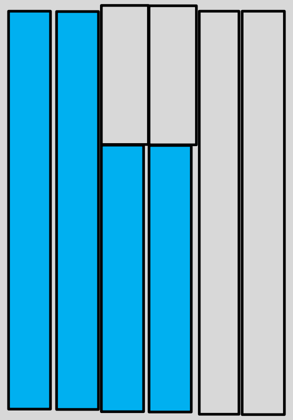
Vertical CPU Management



Partition with 3.3 CPU share. 2 Hi w/ 100%, 2 Med w/ 65% and 2 Lo w/ 0%.

Unparking to consume white space

Vertical CPU Management



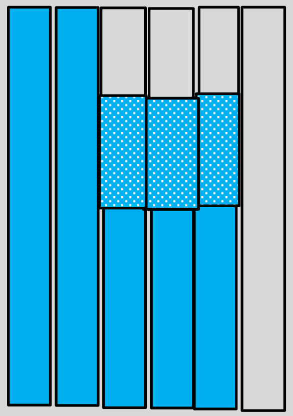
Partition with 3.3 CPU share. 2 Hi w/ 100%, 2 Med w/ 65% and 2 Lo w/ 0%.

Periodically, z/OS asks PR/SM if there is any white space left unconsumed by other partitions.

If it can get more white space than can be consumed by the vertical medium processors, z/OS may ***unpark*** one of the vertical low processors and start dispatching work to it.

Later, there may be insufficient white space available so z/OS will stop dispatching work to the unparked vertical low and park it again.

Vertical CPU Management



Partition with 3.3 CPU share plus 0.8 white space. 2 Hi w/ 100%, 3 med* w/ 70% and 1 Lo w/ 0%.

***** When a vertical low is unparked, it acts like a medium. The guaranteed medium share plus the donated white space is shared across all the mediums.***

The Short Engine Problem

The ***short engine problem*** is not new, but hiperdispatch has created a new, more virulent variation.

A short engine is a logical processor that is allotted a very small share. There can be relatively long periods when a short engine is not dispatched by PR/SM. This is a problem if there is important work running on it when it stalls.

Hiperdispatch, if anything, makes the problem worse because a vertical low processor that is unparked because momentarily there is some white space can get stalled for an indefinite duration if the whitespace disappears.

Since z/OS asks PR/SM about whitespace only every few seconds, it will repark an unparked vertical low if the amount of white space available to the vertical low drops below a threshold. This impedes the ability to run the box at 100% utilization.

Warning Track

An architectural feature called ***Warning Track*** greatly mitigates the short engine problem. Like the warning track of a baseball field, it warns that you're about to run into a wall.

Several microseconds before PR/SM is scheduled to take the physical processor away from a logical processor, it signals the operating system with an external interrupt.

If z/OS enables in time to receive the signal, it undispatches the work it is currently running on that logical processor so that it can later be dispatched on some other logical processor. Then it notifies PR/SM that it's safe to undispatch the logical processor

The work is not stranded on logical processor that may not run again for an extended time

Any remaining microseconds left to the timeslice when PR/SM takes the physical processor are later given back to the partition.

Second Metric: Doing Useful Work

As capacity increases there is a phenomenon called the ***large systems effect***.

- The added capacity enables more in-flight units of work
- There are longer queues and larger data structure to represent the work
- These queues and structures are accessed more frequently
- Overhead accessing the structures increases with the square of the capacity increase.
- So, the percentage of overhead increases as the capacity increases
- This can only be addressed with better algorithms.

As more processors are added to the system there is an ***MP Effect***

- With more processors, there is more inter-processor communication
- There is more contention for access to control structures so serialization overhead goes up – in some cases exponentially.

Flattening the MP Effect

- One way to reduce the overhead of an increasing number of processors is to reduce interprocessor communication. An invention called ***reduced preemption*** opened the possibility for z/OS to support 10s of processor engines in a single image. Reduced preemption is so interesting because it uses the source of the problem to construct a solution.
- Another source of MP overhead is serialization. The more processors there are, the more contention there will be for system locks. There are several unique architecture features that help reduce the cost of this contention.
  - Strongly coherent memory – no need for a sync instruction.
  - Signal Processor Sense Running Status – find out if other logical processors are running.
  - The interlocked update facility – atomic update of memory fields.
  - Transactional Execution – serialized update without explicit locks.

Signal Processor Sense Running Status

- In a system with a large number of processors, the processors can spend significant time ***spinning*** for system locks. That is just staying in a tight loop until the lock is freed by the current holder. Multiple processors can be spinning on the same lock.
- In a virtual environment, it is an option to yield the remaining timeslice of a processor rather than continue spinning so that some other logical processor can use the capacity.
- Unfortunately, the trip through PR/SM to yield is quite expensive. Yet, spinning while the holder is not even dispatched by PR/SM is a total waste.
- The Signal Processor Sense Running Status enables the lock requester to know if the current hold is actually running. If it is, the spinning continues. If not, the remainder of the timeslice is yielded. This greatly reduces the waste of spinning unnecessarily.

Third Metric: Doing the Important Work

Decades back, IBM started on the road to managing system capacity based on workload characteristics and business importance. This was manifested in the System Resource Manager (SRM).

SRM was effective, but it required a human performance analyst to set about 60 parameters that would then guide system behaviour. It also was not dynamically adjusted based on changing conditions.

It was not unusual for an installation to be using specifications that were many years old.

These deficiencies were addressed with the Workload Manager (WLM) which was introduced in the mid 1990s and has been repeatedly expanded and enhances.

The Workload Manager

WLM relieves an installation of having to set the individual performance parameters. Instead, the installation can assign business importance and performance goals to the diverse workloads running on a z/OS system.

WLM dynamically adjusts internal performance parameters to ensure that the most important workloads continue to achieve the stated goals and distributes the remaining capacity to other workloads based on their relative importance and goals.

While ensuring that the most important workloads achieve their goals, WLM does not waste capacity by allowing those workloads to overachieve their goals while starving less important workloads. This is important for work that has no specified goal (called ***discretionary***).

Internal performance parameters are adjusted in real time based on statistical observations so that resource utilization is constantly optimized based on installation policy.

WLM doesn’t overdo it

While WLM ensures that in times of shortage, capacity is given to the most important workloads, it must not completely starve any workload.

Even low importance work can hold important resources. For example, a batch job with a discretionary goal can hold an important DB2 latch.

If that job completely stalls while holding an important latch, high importance transactional work may need that latch and back up behind the stalled job that holds it. Depending upon the importance of the lock or latch involved, the negative impact can be dramatic.

WLM prevents this problem with an algorithm called ***Trickle***. An installation can specify a percentage of the capacity that z/OS can distribute “out of priority order” to lower importance work to prevent damaging long-term stalls.

* Promotion of low priority work that holds an ENQ resource needed by higher priority work has been around for a long time. Trickle is need because, unlike ENQs, the operating is unaware of serialization done with private locks/latches. The main example is DB2 latches.

Mainframe Millicode

The name ***millicode*** is given to the high-level microcode used to implement or augment many of the IBM mainframe architectural facilities. It provides a platform for many of the innovations discussed above.

Unlike traditional microcode, millicode has no communication latency between the main CPU and a microcode engine because millicode runs on the same CPU as application code.

In addition to implementing the published z/Architecture, the IBM mainframe CPU chip provides a ***millicode mode*** which implements all the z/Architecure hardware instructions plus additional instructions available only in millicode mode.

There is also another set of registers so that there is no need to save and restore the application program or operating system registers when entering and exiting a millicode routine.

An Example of Millicode Efficiency

The Signal Processor (SIGP) instruction is used to perform a number of functions in a multiprocessor environment. It is implemented in millicode.

An example that illustrates the efficacy of this millicode layer is the SIGP Sense Running Status (SIGP SRS) facility mentioned earlier.

z/OS wants to know if some other logical processor in its partition is currently running or if it is not currently dispatched by PR/SM.

Asking PR/SM would be very expensive because it would cause the guest to undispatched and then redispatched – both are heavy duty operations – in addition to looking at the control structures to find the answer.

Instead, an invocation of the SIGP SRS millicode routine can supply the answer in the time of few dozen instructions.

With SIGP SRS, wasteful spinning for locks can be reduced without introduction additional other overheads.

Conclusion

Without mentioning:

- Chip frequencies of greater than 5 gigahertz
- Independent I/O channels than operate in parallel and free the CPU to do more work.
- High Performance FICON (zHPF) to greatly speed up Disk I/Os
- Parallel Sysplex – nearly unlimited capacity with nearly 100% availability
- Intelligent Resource Manager

…. we have shown why you can efficiently run z/OS workloads on the IBM Mainframe at very high utilization without fear of response time delays for critical transactional and batch work.

The answer is constant innovation, enhancement and optimization on every layer of the mainframe platform.

Some additional reading

Gary King’s paper
https://www.ibm.com/support/pages/system/files/inline-files/gking_high_util_v1.pdf

An article on Reduced Preemption
https://community.ibm.com/community/user/ibmz-and-linuxone/blogs/destination-z1/2019/12/23/the-jiu-jitsu-of-reduced-preemption

An article on Warning Track
https://community.ibm.com/community/user/ibmz-and-linuxone/blogs/destination-z1/2019/12/23/warning-track

Nice write-up on WLM in Wikipedia
https://en.wikipedia.org/wiki/Workload_Manager

See Wikipedia article on millicode (the link to my IBM Systems Magazine article is dead)
https://en.wikipedia.org/wiki/Millicode