Under the Hood:
POWER7 Logical Partitions

April 4, 2013

IBM Corporation

Table of Contents

Introduction 5
The Virtual Processors 6
  Dispatching of Virtual Processors 8
  The Shared-Processor Pool 10
  Partition Compute Capacity / Entitlement 12
    Utility CoD 15
    Multiple Shared-Processor Pools (MSPP) 16
  The Measurement and Use of Entitled Capacity 16
  Uncapped and Capped Partitions 21
    Capped / Uncapped Summary 23
  Dedicated-Donate 24
  DLPAR and the Desired/Minimum/Maximum Processor Settings 26
  Task Dispatching and the Measure(s) of Consumed Compute Capacity 28
Simultaneous Multi-Threading (SMT) Considerations 31
  iDoctor and CPU Utilization 34
POWER7's Nodal Topology 39
  Affinity Groups 43
  TurboCore in POWER7's Nodal Topology 44
  Processor Licensing and Activation in a Nodal Topology 45
  The Theory and Practice of Controlling Partition Placement 46
  The Easy Button: Dynamic Platform Optimizer 48
  Hypervisor Memory Requirements 52
  Simple (or not so simple) DIMM Placement 52
  NUMA and Dynamic LPAR 53
  The Shared-Processor Pool Trade-offs and the Time Slice 55
Summary 58
Glossary 59
References 63

Disclaimer – POWER7 Logical Partitions

Copyright © 2013 by International Business Machines Corporation.

No part of this document may be reproduced or transmitted in any form without written permission from IBM Corporation.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This information may include technical inaccuracies or typographical errors. IBM may make improvements and/or changes in the product(s) and/or programs(s) at any time without notice. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business.
THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted according to the terms and conditions of the agreements (e.g., IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. IBM is not responsible for the performance or interoperability of any non-IBM products discussed herein.

The performance data contained herein was obtained in a controlled, isolated environment. Actual results that may be obtained in other operating environments may vary significantly. While IBM has reviewed each item for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere.
Statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.
The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Acknowledgements

We would like to thank the many people who made invaluable contributions to this document. Contributions included authoring, insights, ideas, reviews, critiques and reference documents

Our special thanks to key contributors from IBM STG Cross Platform Systems Performance:

Mark Funk – IBM i Performance

Rick Peterson – IBM i Performance

Sergio Reyes – AIX Performance

Our special thanks to key contributors from IBM i Development:

Chris Francois – IBM i Development

Document Responsibilities

The IBM STG Cross Platform Systems Performance organization is responsible for editing and maintaining the Under the Hood – POWER7 Logical Partitions document. Any contributions or suggestions for additions or edits should be forwarded to Sergio Reyes, sergio1@us.ibm.com.

Introduction

Welcome to “Under the Hood: Logical Partitions on POWER7”. In this paper we will show you what is really going on under the abstractions being provided for logical partitions. Reading this, we are assuming that you are familiar with the view of logical partitions provided by the HMC. You already know that you can specify:

- Each partition’s entitled capacity – its “Entitlement” - in terms of whole or fractional processor units (1.0 processor units is approximately one core’s worth of processing capacity),
- The amount of memory that each partition will be allocated,
- Whether the partition is designated as a dedicated-processor or shared-processor partition, and
- The number of Virtual Processors, along with many more configuration settings.

These abstractions are handy in understanding the basics of logical partitioning, but there are also some interesting subtleties that you might also want to influence. This paper will allow you to peek under the hood, to better understand what is really going on, and from there to more intelligently control your multi-partitioned system. This is not a “Virtualization for Dummies” paper. After you read this you will be much more familiar with processor virtualization.

In getting there, we’ll be looking at performance considerations relating to

- Virtual Processors
- Partition Entitlement
- Capped and Uncapped Shared-Processor Partitions
- CPU Utilization and the Measurement of Consumed Compute Capacity
- Simultaneous Multi-Threading as it relates to Processor Virtualization
- Virtualization Effects of Non-Uniform Memory-based Topologies

This document is not intended to be a comprehensive “best practices” document for LPAR performance. Reference the POWER7 Virtualization Best Practices Guide for more details:
POWER7 Virtualization Best Practices Guide

Although much of this performance discussion is applicable to any operating system (OS), be aware that as we discuss the related performance implications of operating system design, the operating system of interest here is primarily IBM i.

The Virtual Processors

The Processor: The hardware entity to which tasks are assigned, executing their programs.

System processors are such a basic feature of your computer system that we tend to forget that each processor, even within a partition, is already virtualized. In a partition with many jobs, processes, and threads, you do not need to know when or which processor is executing your task. All your program needs to do is make a task dispatchable and you can be sure that it will get its opportunity to execute somewhere and soon, even if all processors are busy.

Consider a partition with multiple processor cores. Even if you did know **when** your task was executing, have you really ever wanted to know or control **which core** is being used? The partition provides compute capacity and handles the rest for you. Providing more cores to a partition just means more compute capacity; this provides more opportunity to concurrently execute multiple tasks, and to minimize any task's wait time. Still more compute capacity comes from the fact that most modern processor cores are individually capable of concurrently executing multiple tasks via SMT (Simultaneous Multi-Threading); POWER7 cores can concurrently execute up to four tasks per core. Not only could your task be executing on any core, but it could be executing with three additional tasks on that core.

The point here is that even within a partition, the operating system's Task Dispatcher is virtualizing processors, hiding the details of the processor cores. Although task dispatching is actually quite complex, you need only think of the Task Dispatcher – as in the following figure - as being a hopper of dispatchable tasks, spreading tasks in some fair and performance optimized manner over the available "processors" of its partition.

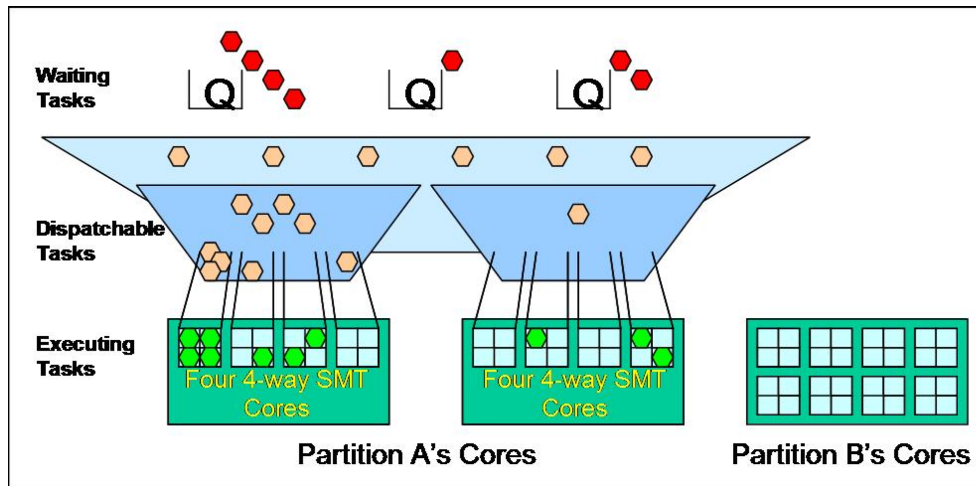

Figure 1 – Task Dispatcher

The innovation that we call "virtualization" is that multiple operating system instances – **Partitions** - can reside on the same Symmetric multiprocessing (SMP) processors and memory previously only used by one. With processor virtualization, each **dedicated-processor** partition uses just a subset of particular cores in the system. And even though they all reside within the same SMP, as far as a dedicated-processor Partition A is concerned, its cores are the whole of this system; Partition A has no visibility outside of that.

Maybe later – perhaps via DLPAR (Dynamic LPAR) - Partition A gets told that it gets to have more physical cores or must instead free up a few; the point is that a partition's fixed view of its resources can change. Partition A might even, on its own, temporarily give up the use of one or more cores for the benefit of one or more other partitions or even energy usage. The partition's Task Dispatcher is flexible enough to handle these changes. Partition A's Task Dispatcher is itself virtualizing even the number of its cores.

Since the reality is that you don't really know upon which core each of your tasks executes, Processor Virtualization allows the partition's processor cores to be further abstracted as "**Virtual Processors**". This abstraction allows us to think of each Virtual Processor as not necessarily tied to any particular core in the system. The partition's Task Dispatcher dispatches tasks instead to Virtual Processors, not cores. The Virtual Processor can be thought of as being assigned to a core shortly thereafter.

In practice, though, **dedicated-processor** partition's Virtual Processors really are tightly tied to particular cores and do have some longer-term persistence to cores. A task assigned to a Virtual Processor really is also being assigned to some particular core; using the same Virtual Processor later typically does mean using the same core as well. Even so, these Virtual Processors can and do move, just not particularly frequently.

A **shared-processor** partition's Virtual Processors, though, might be thought of as having only short-term persistence to a core. Unlike dedicated-processor partitions having persistent association to some specific cores, the shared-processor partition's Virtual Processors are all sharing the processor core resources of something called the "**Shared-Processor Pool**". It is true that even a shared-processor partition's Virtual Processor can remain attached to a core for quite a while, but your general mindset ought to be that there is no long term persistence between a Virtual Processor and any particular core and the processor cache residing there.

There are times when there are many more dispatchable tasks than there are "processors" for them all to execute. When that happens, the partition's tasks take turns executing. The same thing happens with the cores of the Shared-Processor pool; the cores of the Shared-Processor pool get shared by potentially many more active Virtual Processors. Just like tasks waiting their turn for processors, whenever there are more active virtual processors than there are cores in this pool, Virtual Processors must take turns to execute on the pool's cores. Just like tasks switching on and off within a processor, for any shared-processor partition a virtual processor's persistence to a core can be quite temporary. A waiting Virtual Processor may get assigned to the very next available core, no matter its location (or of the core where the Virtual Processor last executed).

Even dedicated-processor cores might be idle; they don't always have tasks dispatched to them. Same thing can be true for Virtual Processors. Any Virtual Processor might be "inactive" because there are no tasks dispatched there. For dedicated-processor partitions, this can – but not always – mean that the associated core is going unused. For shared-processor partitions, this simply means that the empty Virtual Processor is not assigned to any core at this time. Being inactive, it is also not competing with active Virtual Processors for the use of the Shared-Processor pool's cores.

Assigning one or more tasks to a Virtual Processor makes it "active". We would want that Virtual Processor to be attached to a core quickly thereafter. Conversely, when the Virtual Processor's last task ceases its execution and leaves its Virtual Processor (i.e., making it inactive), the Virtual Processor quickly frees up that core. This active period – the time during which the Virtual Processor persists on a core - can be very short, perhaps no longer than between a task's pair of page faults or lock conflicts. Such wait events temporarily remove a task from assignment to a Virtual Processor and, so, a Virtual Processor from executing on a particular core. When a Virtual Processor without tasks is dispatched

there, the Virtual Processor ceases its association with a core. And this is just as you would want it; any waiting active Virtual Processor can now use the freed core.

Each POWER7 Virtual Processor should also be thought of as representing up to four dispatched tasks, because an SMT4 core supports up to four tasks. The Virtual Processor is considered active (for potential use of a core) if there are one through four tasks assigned there; even just one task makes it active. When the last task ceases execution there, the Virtual Processor becomes inactive again.

You can now see again that a Virtual Processor is really just an abstraction; it represents the notion of a processor core – with all of its SMT hardware threads (4 in the following figure) – to the partition itself, or more specifically, to a partition's Task Dispatcher. The Virtual Processor effectively provides the means by which a partition need not know the physical location of the cores on which its tasks are executing, or, for that matter, when they really are executing.

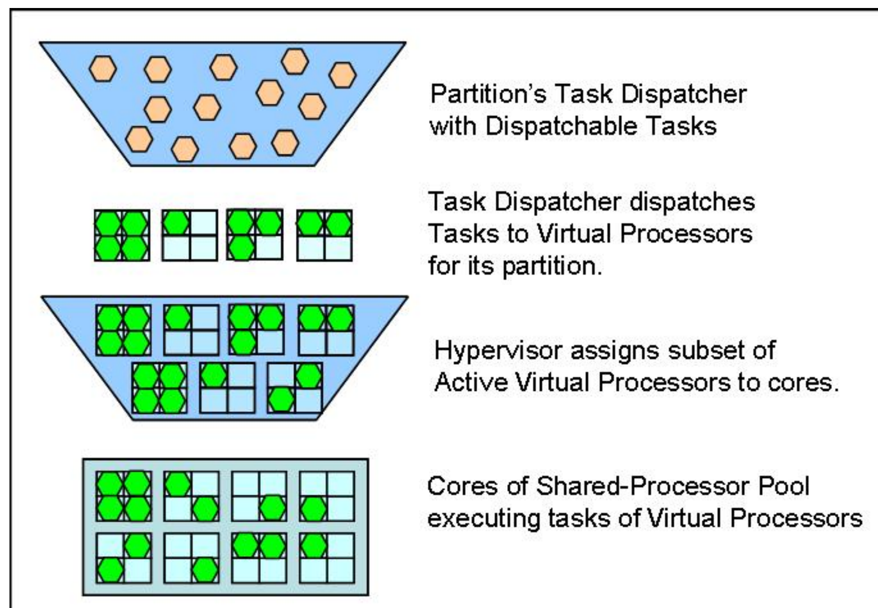

Figure 2 – Virtual Processors

Dispatching of Virtual Processors

The difference in Virtual Processor persistence between dedicated and shared-processor partitions results in some interesting differences in performance behavior as well.

For a POWER7 processor, a dedicated-processor partition is executing at maximum compute capacity only when all of the partition's cores are executing four (SMT4) tasks. When there are SMT hardware threads available – because there are fewer tasks executing – any newly dispatchable task can begin executing immediately. There is no queuing delay. Any additional dispatchable tasks above four tasks per core wait for a while; often this wait is until an executing task stops, freeing up a processor. These tasks are going to perceive a queuing delay as they wait their turn for a processor. The wait period is dependent on both the individual task's priority and the number of dispatchable tasks.

This wait before getting to execute is what you normally think of as a "CPU Queuing" delay. Like waits due to I/O and lock conflicts, you know that CPU Queuing delays are a component of the response time

of your application. Decreasing response time, when high for this reason, might require more compute capacity (e.g., more cores).

Contrasting these observations with shared-processor partitions, getting a task to execute there is occasionally a two step process;

1. First dispatch a task to a virtual processor (which might itself introduce CPU queuing delays), then
2. Attach a Virtual Processor to a physical core to execute the task's instruction stream; this can also introduce a delay when there are too many active Virtual Processors contending for the available cores.

To be more complete, when a task gets dispatched to a POWER7 Virtual Processor, the Virtual Processor will be in one of the following states:

- Already active and attached to a core, executing fewer than four other tasks on an SMT4 core. The new task gets to execute immediately here without delay.
- Already in an active state, but waiting for an available core (i.e., all of the shared-processor pool's cores already have Virtual Processors assigned). The new task dispatched to this Virtual Processor waits because its Virtual Processor has to wait.
- In an inactive state (i.e., no tasks yet assigned there), the one newly dispatched task makes the Virtual Processor active, but
    - The newly active Virtual Processor gets immediately assigned to an available core (so the new task gets to immediately execute),
    - All of the shared-processor pool's cores are busy (so the new task continues to wait to execute).

You already know that tasks can experience queuing delays. Here you also see that for shared-processor partitions there is a related effect which is a function of the over-subscription of active virtual processors for the cores of the shared-processor pool.

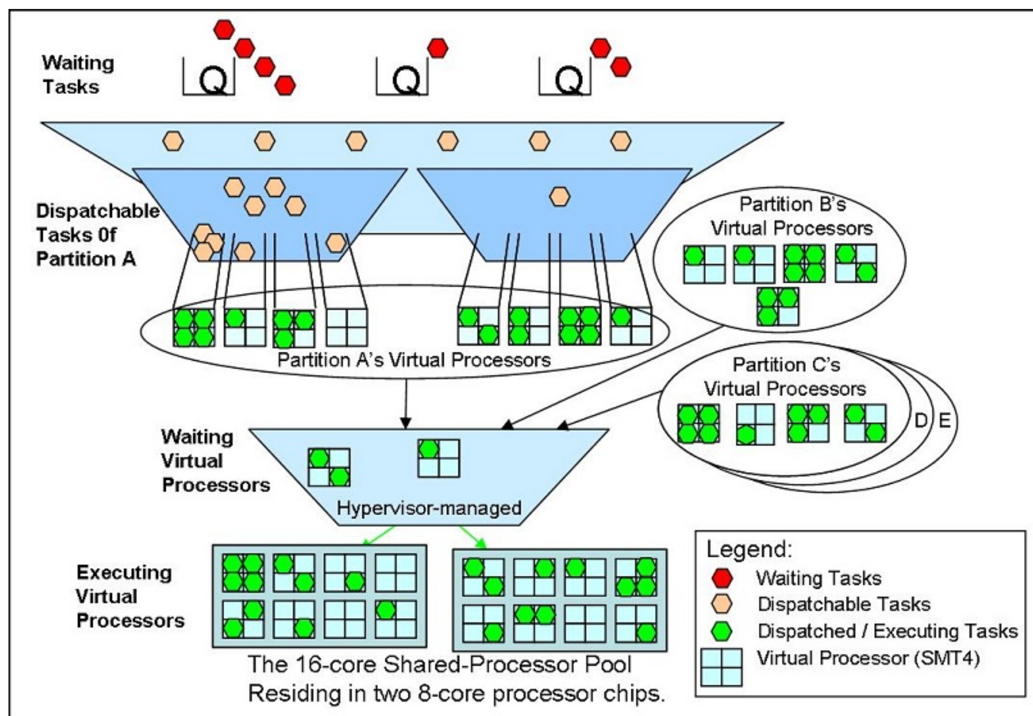

Figure 3 - – Virtual Processors – Dispatching Partitions

The previous figure pulls together what you have seen so far. It outlines the various states of tasks and virtual processors.

- Starting at the top, in red are tasks which are waiting on some resource to be made available or event to occur. Think here in terms of tasks waiting on locks and completion of IO operations as examples. These waiting tasks might also simply be tasks waiting for their next piece of work to arrive (e.g., like you hitting an Enter key). These tasks are not in a dispatchable state and not contending for a processor.
- In orange are tasks which are dispatchable, their previous wait period on a resource or event has ended. These tasks have not yet been assigned to a virtual processor for execution. This is often because the partition's virtual processors are simply totally busy at that moment, with each virtual processor already fully committed supporting its set of tasks.
- In green are tasks which have been dispatched to an SMT hardware thread of a virtual processor. These 4-way subdivided boxes in blue are Virtual Processors; there are four parts because each POWER7 core is capable of concurrently executing four tasks.
- In dedicated-processor partitions, these dispatched tasks are executing once they have been assigned to a virtual processor (each virtual processor here also representing a core). In shared-processor partitions, the associated Virtual Processor
    - might be waiting for the hypervisor to assign it to a core,
    - might simply have no tasks assigned there and so are inactive, or
    - might be already assigned to a core, in which case the task(s) dispatched there are also executing.

Performance Tip: Perhaps the most often noticed performance effect is increased response time. One important component of that is processor queuing delays. One typical way to minimize that is to increase the number of "processors", which here can include POWER7's SMT4 hardware threads. With dedicated-processor partitions, there is always a fixed number of cores available to provide this capacity. With shared-processor partitions it is possible that the Virtual Processor count can provide the same effect. But, for shared-processor partitions, these same Virtual Processors must compete for the compute resources of the shared-processor pool with the Virtual Processors of this and other partitions. This can add its own form of queuing delays, resulting in increasing response time.

As you have now seen, a shared-processor partition's Virtual Processors – and the dispatched task(s) they represent - may need to wait to be attached to a core and, yes, that can take a while. But, because of the logical partition notions of "Entitlement" and of "Time Slicing" – concepts we'll be getting into shortly - each virtual processor is guaranteed that it will soon have its opportunity to be attached to a core, allowing its task(s) to execute, for at least a short while.

The Shared-Processor Pool

Any SMP system has a specified number of physical cores; this is the number of cores (capacity) in your system's hardware. Of these physical cores, potentially fewer are "**licensed**" (activated) for use by any partitions.

Of these activated cores, all dedicated-processor partitions are assigned the number of these activated cores based on their configuration. Once the dedicated-partition cores have been accounted for, the remainder of the SMP's licensed cores are added to the "**shared-processor pool**". The shared-processor

pool will be used to allocate resources for shared-processor partitions when they are activated. For example, in the previous section's figure above, the shared-processor pool is a 16-core subset of a potentially larger system; the pool's cores here happen to reside on just two of the SMP's chips. Any one of this shared-processor pool's cores can be the temporary home for any active shared-processor partition's virtual processor.

(Please do not confuse the terms "activated" – an attribute of a physical core - and "active" – an attribute of a virtual processor. An "active" virtual processor is one upon which a task has been dispatched and has been made available to the hypervisor for assignment to a core.)

Altogether the shared-processor pool's cores represent a maximum amount of compute capacity, just like the compute capacity that is available from the cores owned by any dedicated-processor partition. There is no more compute capacity. And it is this set of cores that are used by all of the virtual processors of all of the shared-processor partitions. The full set of active virtual processors over all these partitions can reasonably be configured to exceed – even far exceed – the number of cores in the shared-processor pool. But all of these are sharing only this maximum compute capacity.

Performance Tip: Since the shared-processor partitions all together normally use only the cores of the shared-processor pool, there is no value in a shared partition having more virtual processors than the number of cores in this pool. As you will see, the virtual Processor count should normally be considerably less than this maximum but large enough to handle peak loads.

You can see a sample of this effect in the following figure. Here the hypervisor is aware of many more active virtual processors than there are cores in the shared-processor pool. In this figure, each color represents the virtual processors of a different shared-processor partition. The green hexagons are tasks, with up to four per virtual processor.

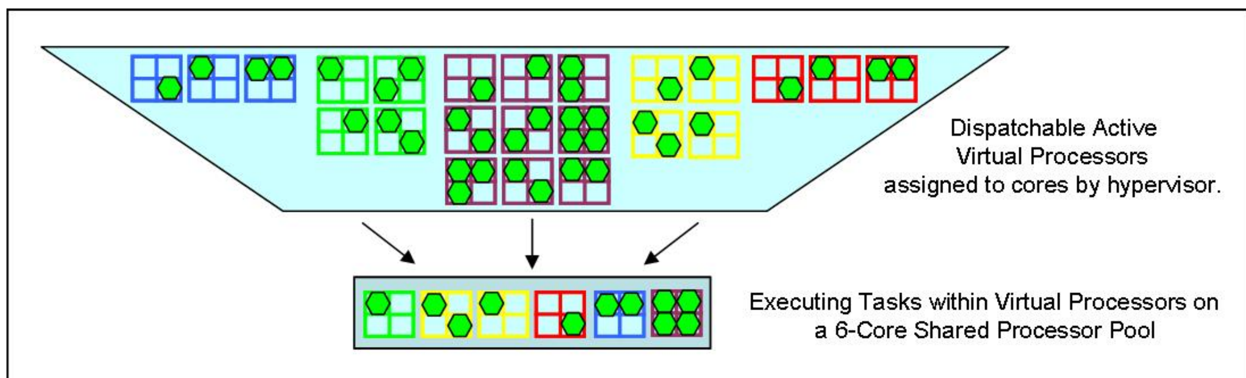

Figure 4 – Shared Processor Pool

As we described earlier, each partition's Task Dispatcher has a "hopper" of dispatchable tasks which get assigned to cores/virtual processors. The hypervisor uses the same concept to manage a hopper of active virtual processors; the hypervisor takes some or all of the active virtual processors in the hopper and assigns them the cores of the shared-processor pool. Just as the Task Dispatcher's hopper may have far more dispatchable tasks than there are available processor threads, the number of active virtual processors in the hypervisor's hopper can also far exceed the number of cores available in the shared-processor pool. Just as the Task Dispatcher needs to provide fairness in allowing all those dispatchable tasks to have their

opportunity to execute, the hypervisor must similarly ensure fair use by allowing the currently active set of virtual processors to each have their opportunity to execute. (The inactive virtual processors are also like waiting tasks in that neither is contending for processor resources - neither type is considered to be "in the hopper" – at that moment.)

Recall also that within a partition, you can provide some control over fair use of the processors by tasks. Within a partition, fairness provided by a partition's Task Dispatcher is partly based on each task's "priority" to execute; for IBM i this is just job priority (e.g., SBMJOB's JOBPTY parameter). Job priority in this context aids the Task Dispatcher in determining which of the dispatchable tasks ought to next be assigned to a processor.

Fairness as calculated by the hypervisor (when determining which virtual processor to next assign to a core) ensures that each active virtual processor gets allocated its share of processor cycles. Hypervisor-provided fairness is at least partly a function of each partition's "**entitlement**" to use the shared-processor pool's entitled capacity. Partitions, though, can also have an attribute of priority, a value which you can set, and which we will describe shortly. We're next, though, going to look at the notion of entitlement.

Restating the performance note earlier, notice in the figure above that one of these partitions (in purple) has 9 virtual processors contending for only 6 cores in the shared-processor pool. Since there is only the compute capacity of six cores available there, only ever a maximum of 6 virtual processors from this partition will be able to concurrently execute. So having more virtual processors than cores in the shared-processor pool is always considered a mistake. In addition, notice that even these 6 virtual processors will occasionally – if not often - need to compete with the active virtual processors of the other partitions for the pool's cores.

Partition Compute Capacity / Entitlement

Entitlement helps to provide fair usage of the compute capacity of the shared-processor pool. There might be a lot of compute capacity in that processor pool, but you may have a lot of partitions – and within each of these, a lot of virtual processors - sharing it. So what is entitlement really and what do you need to know to set it up correctly? Let's start with concepts that you already know.

The compute capacity available within a dedicated-processor partition is proportional to the number of cores available to that partition; more cores means more compute capacity. Since the dedicated partition's virtual processors are tied to particular sets of cores for relatively long periods, compute capacity is defined there in terms of integer numbers of cores. Each dedicated-processor partition's Task Dispatcher ensures fair and efficient use of this compute capacity by appropriate assignment of tasks over its "processors" (a.k.a., its core's SMT hardware threads). You can see such compute capacity limits in the dedicated-processor partitions in the figure below. For example here, dedicated-processor Partitions A and B have the compute capacity of three cores each, Partition C is limited to two cores. Again, compute capacity here is fixed per an integer number of cores. Each partition's Task Dispatcher's "hopper" assigns its task (green hexagons) to the SMT4 cores.

As a side observation, you might also notice that Partitions A-C are not here consuming all of their available compute capacity, even though all cores are being used; here only one or two of each core's SMT threads are being used, meaning that there is still more compute capacity for when more tasks become dispatchable. This observation is important later.

Similarly, the entire compute capacity of the shared-processor pool is only as large as the integer number of cores in this pool; this, in turn, can be no larger than the number of cores in the SMP. It follows that the hypervisor perceives all of the pool's compute capacity as being consumed if there is an active virtual

processor assigned to all of the pool's cores. Picture this shared pool core count as representing the maximum compute capacity available for use by all of the shared-processor partition's virtual processors, no matter the number of active virtual processors.

So, finally, **each partition's entitled capacity can be thought of as a fraction of the total compute capacity of this shared-processor pool**. If you add them all up, their total entitled capacity can be no greater than the compute capacity of this pool. Unlike dedicated-processor partitions compute capacity being defined as an integer number of cores, shared-processor partition entitled capacity can be stated with much finer granularity.

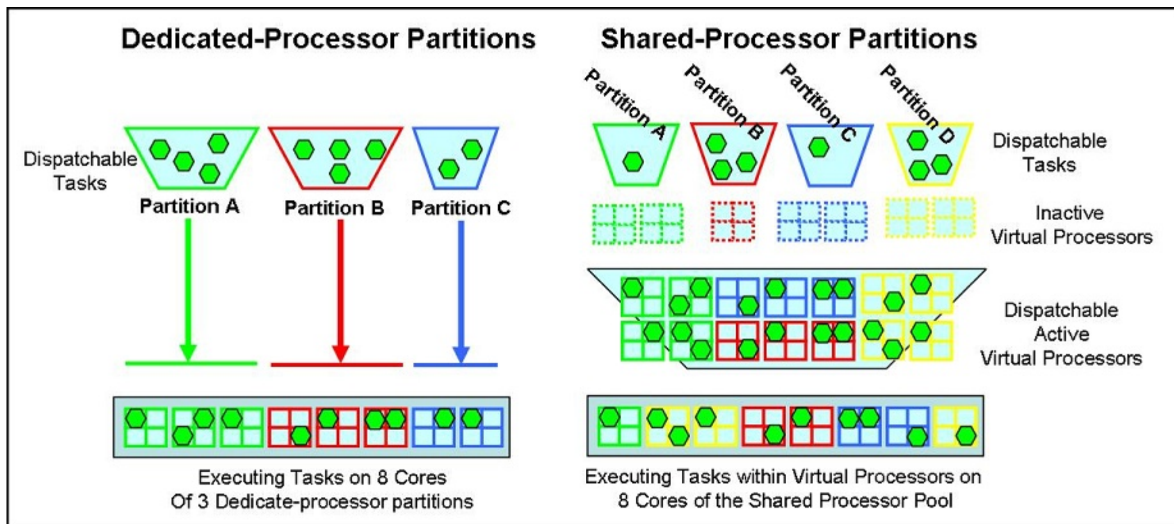

Figure 5 – Partition Comparison

To further explain the reason for entitlement, recall that each partition has an integer number of virtual processors. As in the following figure, it is completely possible – but not necessarily often advised – for any shared-processor partition to have a virtual processor count equal to the number of cores in the shared-processor pool. If such a partition became very active with a large number of dispatchable tasks, it is possible for such a partition to be temporarily using all of the cores in the pool. It can, for that moment, be using all of the compute capacity that is available in the pool. It follows that when this happens, none of the other partition's virtual processors are getting to execute. Of course, when there are other active virtual processors the hypervisor can and does have the virtual processors take turns using the pool's cores.

Without this notion of "Entitlement", the hypervisor would make a best guess attempt at ensuring fairness amongst all of the partition's active virtual processors. Without this advice from a system administrator, the partition(s) with the most active virtual processors could consume the most compute capacity, independent of the number of tasks per virtual processor.

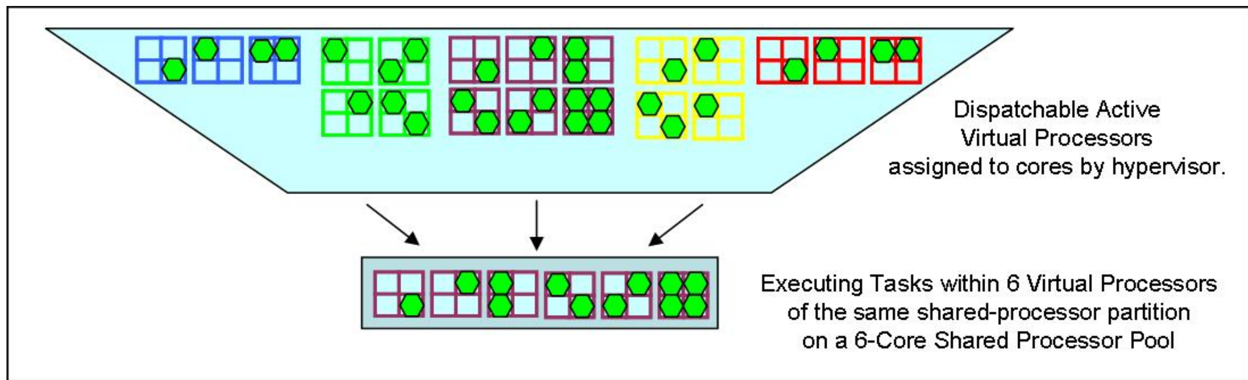

Figure 6 – Dispatching Active Virtual Processors

This is where Entitlement comes in. Entitlement is your means of providing advice to the hypervisor concerning fair use. For each partition you can configure the portion of this entire shared pool's compute capacity that partition ought to be allowed to consume. This portion, its **Entitlement**, stated in terms of fractional core counts (e.g., 2.2 cores of 16 in the pool), is the entitled capacity of a shared-processor partition.

The sum of all the shared-processor partition's Entitlement can not be configured to be more than the compute capacity represented by the cores in the shared-processor pool. These partitions can have many more virtual processors, but the overall compute capacity available is limited by the shared pool's core count. As you will be seeing, for **Capped** partitions, this Entitlement represents the maximum compute capacity – a limit - that a partition's virtual processors are allowed to consume; for **Uncapped** partitions, this value also represents a guaranteed compute capacity if needed, not necessarily a limit.

Performance Tip: Estimating Entitlement and Virtual Processor Values

What do you really need for entitlement values? First, entitlement represents desired capacity; capacity only indirectly represents response time. Further, every workload is different; IBM i publishes capacity in terms of CPW Ratings based on a particular multi-job database OLTP workload which uses the SMT4 threads and all cores of a partition. It does so for a number of different dedicated-processor partition sizes for a number of different systems. Still with these caveats, and more to be discussed, you can estimate the initial capacity of a shared-processor partition.

We assume here that you know your needed capacity, again in terms of CPW rating.

1. What is the capacity of your shared-processor pool in terms of CPW rating? Find a CPW rating for a similar system with a dedicated-processor partition size at or above your shared-processor pool size. Assume – for now - a linear relationship to roughly calculate your shared pool's capacity. Example: Given 12 cores in shared-processor pool, a 16-core CPW rating of 120K…. 90K = (12/16)*120K
2. How much capacity does your partition need (as a CPW rating)? Let's assume 18K.
3. Minimum Entitled capacity required in terms of fractional cores ….
   2.4 cores = 18K * 12 cores / 90K

In order to determine the partition's virtual processor count to use this 2.4 cores of capacity, this partition will need to round this value to the next larger integer. In this case, this partition would require **3** virtual processors. As you will see, more virtual processors could be used by an Uncapped shared-processor partition, but whether 3 or more, these virtual processors would only see more than 2.4 cores of capacity when there remains more capacity in the shared-processor pool.

This is an initial starting point. You'll be adjusting this as you learn more and per your needs.

Utility CoD

Assume that your system has been configured for partition capacity to handle most processing needs. However, there may be peak processing demand periods which require additional processing power beyond the configured capacity. Uncapped partitions provide this function for licensed/activated processor cores.

Suppose that you only want to pay to license these additional processor cores when they are needed. **Utility CoD** provides this support. The extra capacity is in the form of inactive processor units located in the shared-processor pool. When the processor cores are required to handle peak processing, they are put into service. They are available for use by uncapped partitions. This addition processor cores become inactive when the workload returns to its normal level.

For more information on Utility CoD see the following IBM Redpaper:

http://www.redbooks.ibm.com/redpapers/pdfs/redp4416.pdf

Multiple Shared-Processor Pools (MSPP)

The Multiple Shared-Processor Pools (MSPP) function provides the capability for processors resources to be allocated to more than a single specifically configured shared-processor pool. A subset of the shared-processor partitions may be allocated to a given shared-processor pool and the remainder of the shared-processor partitions can be allocated to one or more other shared-processor pool(s). Systems can currently support up to 64 shared-processor pools. Each of these shared-processor pools may be configured individually to support its shared-processor partitions. The total capacity used at any point cannot exceed the pool maximum.

An advantage of MSPP is that shared-processor partitions may be grouped together under a given shared-processor pool and processing capacity can be controlled across all these partitions. The total capacity of that shared-processor pool can be shared among them. If these partitions are running the same software then this function can be useful for software capacity license management. Within this shared processor pool all the virtual servers can be uncapped, allowing them flexibility within the license boundary set by the MSPP.

The Measurement and Use of Entitled Capacity

The previous section outlined the concept of Entitlement. This section outlines how the hypervisor keeps track of each partition's consumption of its entitled capacity and what that means to the performance characteristics of shared-processor partitions.

The hypervisor uses each partition's Entitlement in its job of ensuring fair use of the shared-pool's cores. The real trick comes from how the hypervisor keeps track of each partition's compute capacity consumption, and then how the hypervisor manages it. This has some interesting side effects which we'll be looking at soon.

Whenever a shared-processor partition's virtual processor is assigned to a core, the hypervisor views that virtual processor as consuming some portion of the partition's entitled capacity. A virtual processor that is assigned to a core is also consuming the entire compute capacity of that core; if a core is used by a virtual processor, the core is simply not available for use by any other. So compute capacity consumption is measured merely as **that period of time that a partition's virtual processor is attached to a core.**

[Technical Note: The hypervisor's measure of compute capacity consumed by virtual processors has nothing at all to do with what the virtual processor is actually doing. It is not a function of the number of tasks (up to four on a POWER7 core) that are dispatched to that virtual processor. As a result, CPU utilization – a synonym of compute capacity consumption - as measured by the hypervisor can be very different and typically higher than CPU utilization as measured by the partition. The hypervisor is measuring whether or not a core is being used, period. The partitions themselves are often measuring CPU utilization based on how much compute capacity remains within its SMT-based cores; here a core is perceived as 100% utilized only if all four of POWER7's SMT threads are executing tasks. Both approaches are valid forms of measuring compute capacity consumed; they are just different. We'll comment more on this later.]

The hypervisor tracks each partition's entitled capacity consumption within time slices. When a partition's virtual processors reach their entitled capacity limit within a time slice, it is possible that the hypervisor will have that partition temporarily cease execution. For example, consider the following figure showing the execution of two partitions, both with the entitled capacity of 1 core. The upper

partition has 1 virtual processor, the lower partition has 4. We'll have all of these virtual processors as continuously active. Even continuously active, the 1 VP partition with its 1-core entitled capacity (i.e., the uppermost partition) will never reach its entitled capacity limit and so will continue to execute. The lower partition, though, consumes its entitled capacity four times faster and, as a result, also only has virtual processors executing for ¼ of the time.

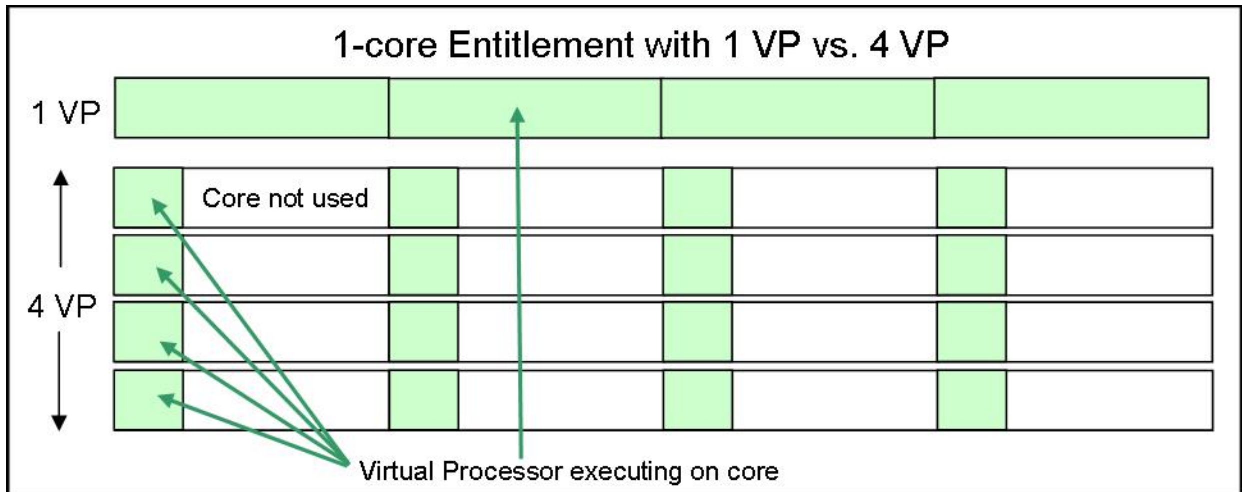

Figure 7 – 1 Virtual Processor vs. 4 Virtual Processors

As you can see, the point at which the hypervisor perceives a partition's entitled capacity as having been consumed is important. It can mean that the partition's virtual processors will temporarily lose their opportunity to execute for a short while. This allows other partitions' active virtual processors their opportunity on a core. Entitlement helps with fair use of processing resources. Entitlement means that every partition is guaranteed at least its "entitled" portion of the shared-processor pool's compute capacity. Remember that the sum of the Entitlement over all of the shared-processor partitions is no larger than the number of cores in the shared-processor pool. There is no guarantee of when or where each virtual processor gets to execute, only together a partition's virtual processors will be ensured that partition's entitled capacity.

The hypervisor measures entitled capacity consumption within time slices of well defined lengths (e.g., default 10 milliseconds. A later section further discusses this value.). If, in one time slice, a partition has exceeded its entitled capacity and so has lost its right to use any core of the shared pool, its virtual processors get another opportunity to execute in the next time slice. Indeed, a virtual processor still executing (i.e., without having exceeded that partition's entitled capacity) at the end of one time slice, can continue executing to at least its entitled capacity limit in the next time slice.

As an example, a partition specified with the entitlement of ½ core and having 1 virtual processor will be guaranteed the right to execute for a total time of at least for 5 milliseconds in each 10 millisecond time slice. This can be for a continuous 5 milliseconds or many short bursts totaling 5 milliseconds. It does not matter to the hypervisor, but it might matter to response time of other active virtual processors waiting for a core.

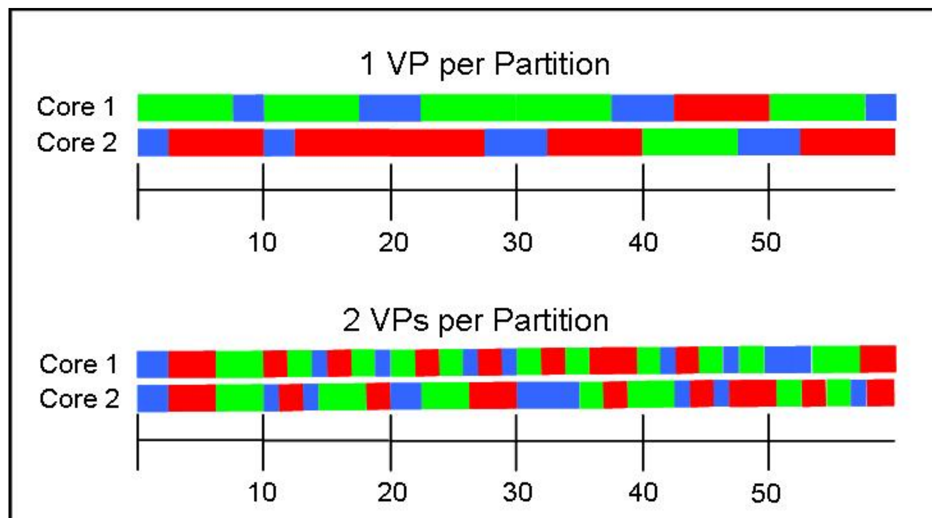

Figure 8 – 1 Virtual Processor per Partition vs. 2 Virtual Processors per Partition

The figure above shows two cases of three partitions (each in a different color), together using a two-core (called Core 1 and Core 2) shared-processor pool. Time is progressing from left to right and divided into 10 millisecond time slices. Each of these shared-processor partitions also has a different value for Entitlement, but – for any given partition – that entitled capacity is the same in both cases shown.

- In the upper case, each of three partitions was assigned one virtual processor. Let's also have these virtual processors continuously active (i.e., with one or more tasks dispatched there) and so intending to use all of the partition's entitled capacity. In most cases, these virtual processors cease their execution when they reach their entitled capacity limit in a time slice. When they don't, they may continue executing with the next time slice again until their entitled capacity limit is reached.
- In the lower of the two cases, with entitled capacity as in the first case, all three partitions now have two virtual processors. Unlike the first case where we have the virtual processors largely continuously active, in this case we'll have each virtual processor as active for only short snippets of time. These virtual processors are deactivating at different times, often because the tasks dispatched there themselves pause their execution (e.g., page fault, lock conflict). But in this figure, even these virtual processors – executing only momentarily – are together still consuming the partition's entitled capacity and may cease execution if that entitled capacity limit is exceeded in a time slice. Note also in this example that all of the entitled capacity of this 2-core shared-processor pool is being consumed.

The point: Whether executing for extended periods of time – as in the first case – or for short bursts of time – as in the second case – if a partition reaches its entitled capacity limit, these partition's virtual processor may temporarily cease their execution. The now available cores then become available for use by waiting virtual processors of partitions which have not yet reached their entitled capacity limit.

We want you to picture just how dynamic this environment can be. Whether virtual processors are active and attached to cores for long periods or active for very short snippets of time and so rapidly switching on and off cores, the hypervisor is handling the fair use of a fixed resource, the entitled capacity of the number of cores in the shared-processor pool. Remember the notion of the hypervisor's "hopper" of virtual processors, shown again below? At times, the hopper might …

- Be empty or have fewer active virtual processors than cores in the shared-processor pool. (The next virtual processor becoming active can then be immediately assigned to a core.)
- Have a number of active virtual processors equal to the number of core in the shared-processor pool.
- Have more (or even far more) virtual processors than there are cores in the shared-processor pool.
- Have virtual processors for partitions which have reached their entitled capacity limit, and so which might temporarily not be in contention for the shared-pool's cores.

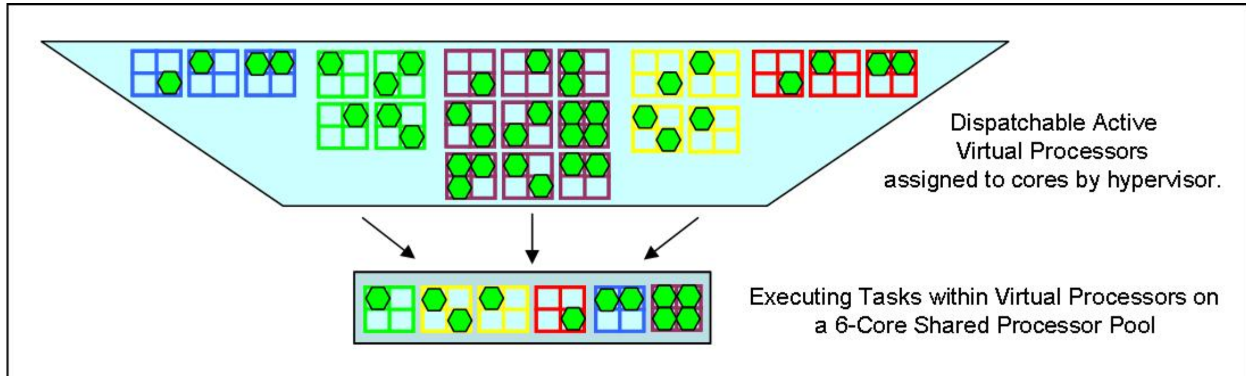

Figure 9 – Dispatchable Active Virtual Processors and Executing Tasks

> **Performance Tip:** Shared-processor partitions can be an efficient use of processing capacity. This dynamic environment can make good use of the shared-processor pool's capacity. But be aware that configuring too many active virtual processors can cause queuing-related response time and throughput (capacity) variability compared to dedicated-processor partitions.

Having, at least occasionally, some excess number of active virtual processors versus the size of the shared-processor pool is not unreasonable. Even if the total static number of virtual processors over all partitions is much larger than the number of cores in the shared-processor pool, when cores are available (because most of these virtual processors happened to be inactive), the oversubscription allows the cores to be used as subsets of partitions become more active. But keep in mind that there is similarity between what happens in the following two quite different cases:

1. An excessive number of active virtual processors of shared-processor partitions competing for cores and
2. An excessive number of tasks contending for the SMT hardware threads of a dedicated-processor partition (i.e., the definition of 100% utilization).

In either case, some tasks are going to wait, and the more tasks – or virtual processors - there are, the longer the wait. Either of these types of waits are a component of response time. But this is where the hypervisor's fairness policy comes in. With each shared-processor partition having some entitled capacity, each partition's virtual processors – and the tasks assigned there - are <u>guaranteed</u> at least that amount of compute capacity. The trick is to define each partition's entitled capacity to ensure good response time while at the same time allowing the resources of the shared-processor pool to be frequently utilized.

[*Technical Note: Since the OS' Task Dispatcher chooses which tasks to dispatch based on a notion of task priority, often the Task Dispatcher has a better idea of which tasks ought to execute than the hypervisor. (The hypervisor is dispatching virtual processors, not tasks.) Configuring your shared-processor partition with too many virtual processors has a way of defeating the intent of task priority. Task priority aids in deciding which dispatchable task is actually dispatched. If, because there are many virtual processors, every task can immediately get dispatched onto a partition's virtual processor(s), dispatch priority provides no value. Instead, tasks of all priorities might find themselves waiting for their virtual processor to be assigned to a core.*]

There is, though, aside from Entitlement, a way to influence the "fairness" relating to the choice of which virtual processor gets assigned next to a core. This additional means is called the Uncapped partition's "**weight**", a value set in the partition's profile. In what follows, let's assume that the remaining active virtual processors are all from partitions already above their entitled capacity limit. (*If in this set, one virtual processor becomes active and it is still below its limit, it gets immediate use of a core.)* Without entitlement being an issue, given equal weighting, the hypervisor would normally tend to randomly select which virtual processor ought to be attached to the next available core. The partition's weighting factor allows you to increase the odds for rapid assignment to a core for some partitions versus others. This is done by assigning each partition – via its profile - a weighting factor between 0 and 255 with 128 being the default. The greater the weight, the greater the odds of this partition's virtual processor(s) - versus others with lesser weight - being assigned to a core. (A partition's weight can be changed "on the fly" by using DLPAR. Otherwise, when the partition's profile is again activated, a change of the weight takes affect.)

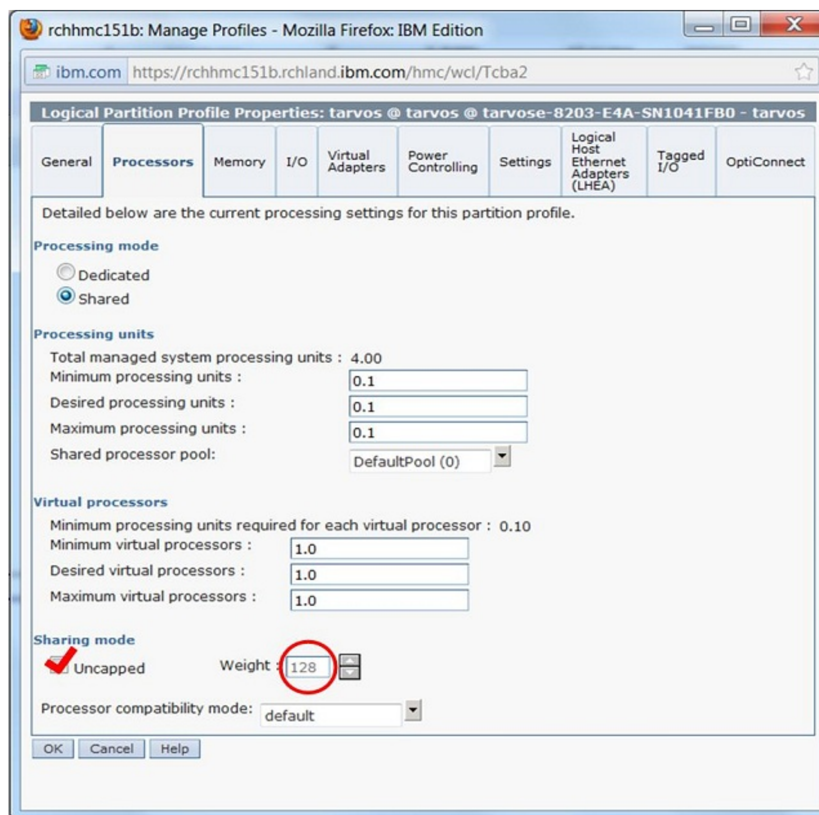

Figure 10 – Weight Value

Note: A partition that is capped and a partition that is uncapped with a weight of 0 are functionally identical. In terms of performance, you get exactly the same result (utilization can only go up to the partition's entitled capacity, not higher), but weighting can be changed from within the partition in workload-managed LPAR groups, whereas capped/uncapped state cannot. If partition capping is an area you would like to change from within a partition using workload management software in the future, an uncapped partition with a weight of 0 should be selected. The HMC is dynamically able to change a partition from capped to uncapped, and to change the weight.

Performance Tip: A partition's Entitled Capacity still remains your primary means of controlling a partition's performance. The partition's weight gets used to slightly help its response time; in a busy shared-processor pool, Partition A's higher weight versus Partition B allows Partition A's virtual processors to tend to be sooner assigned to a next available core.

But, given a set of virtual processors from uncapped partitions all already exceeding their entitled capacity, this also has the side effect of influencing how much extra time Partition A – with its higher weight – is allowed to use a core (versus Partition B with a lower weight). The next section talks to Capped versus Uncapped partitions.

Uncapped and Capped Partitions

In the previous section we spoke of the consumption and measurement of each partition's entitled capacity. We observed that any partition's virtual processors, having consumed their entitled capacity, may be forced to temporarily cease their execution. It also observed that any other partition's active virtual processors, each waiting for an available core, can then begin execution on these now available cores.

Let's suppose that often there was no such contention. Let's assume each partition reaches its entitled capacity limit when there are also no waiting virtual processors. Having reached its entitled capacity, should that partition cease its execution temporarily anyway or should it continue to execute, at least until there is contention? That is the basic difference between **Capped** and **Uncapped** shared-processor partitions.

You can see this in the following figures showing consumption of a 4-core shared-processor pool. A set of 6 partitions (A-F), all with entitled capacity of 2/3 of a core, are consuming some or all of their entitled capacity. We have here arranged for Partitions A and B to twice reach their entitled capacity limit (see periods starting at 10:31 and 10:33). Even so, as seen in the first figure of all Capped partitions, the available compute capacity of the 4-core pool is not ever being fully utilized. In the second figure, Partitions C-F remain Capped and execute as in the first figure, but we have altered Partitions A and B to be Uncapped. When Partitions A and B reached the entitled capacity, rather than ceasing execution, because the shared pool still has compute capacity remaining when needed, Partition A and B's virtual processors continue executing, equally consuming – when needed – the remaining compute capacity of the 4-core shared-processor pool.

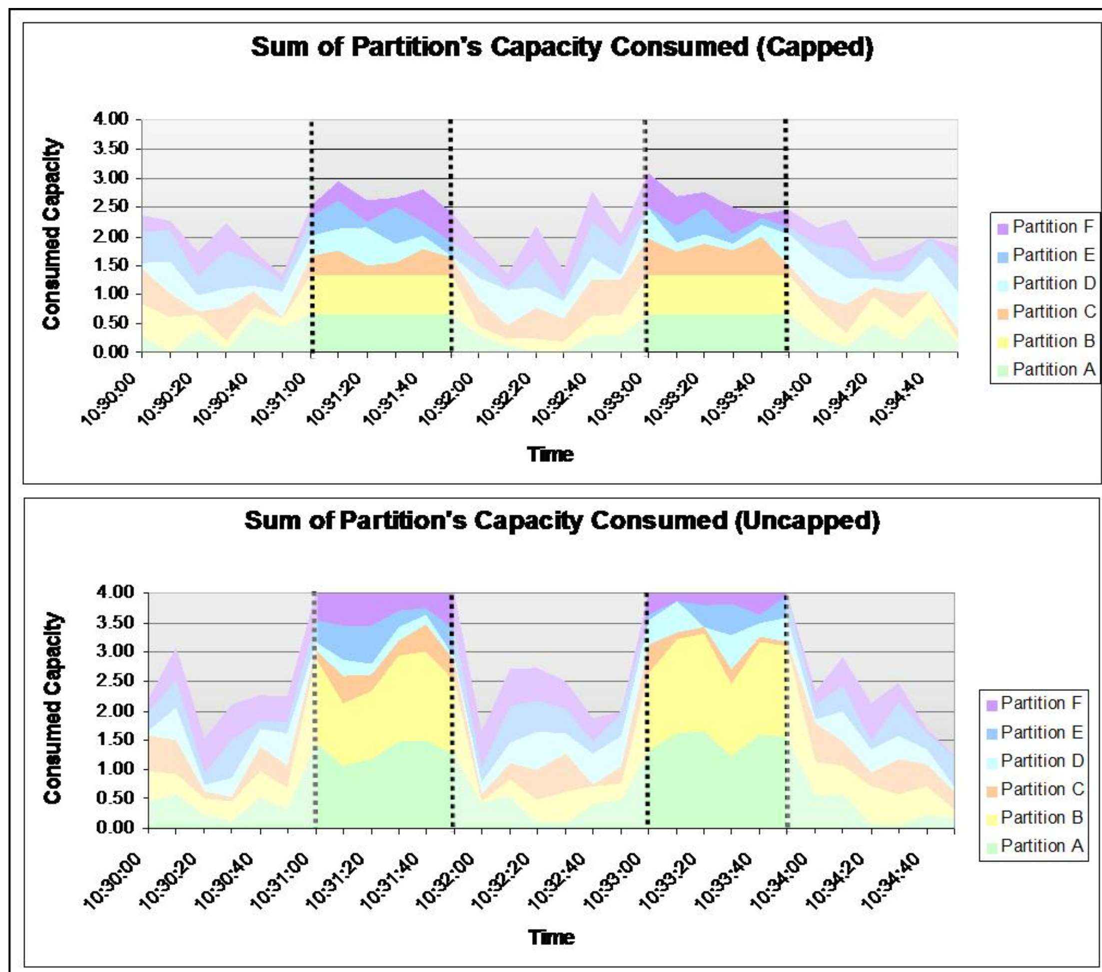

Figure 11 – Sum of Partition's Capacity Consumed

Stating it again, the difference between Uncapped and Capped partitions relates primarily to what happens when a partition's entitled capacity limit is reached. Stated most simply, the difference is that

- A **Capped** partition's virtual processors together cease their execution when the entitled capacity limit is reached. Even if there are unused cores in the shared-processor pool, once the compute capacity consumed reaches the entitled capacity, processing for that partition ceases until the next time slice boundary. **The Capped partition's entitlement provides a limit on consumable compute capacity.** Other shared-processor partitions can be assured that any Capped partition will use no more than this compute capacity.
- **Uncapped** partitions virtual processors can continue execution past their entitled capacity limit**,** but only if there is still available compute capacity in the shared-processor pool. As with Capped partitions, if shared pool compute capacity is not available, all or some of the virtual processors of Uncapped partitions will also cease their execution when the entitled capacity is reached. **The Uncapped partition's entitlement can also be considered a limit, but it is also a guarantee of some minimum consumable compute capacity when needed.** Said differently, if the Uncapped partition needs this compute capacity, it is guaranteed its entitled capacity, even if other partitions have active virtual processors also waiting for cores. But when the shared-processor pool has unused compute capacity, Uncapped partitions are allowed to consume it. (Unlike Capped where the limit is the entitled capacity, the actual maximum compute capacity limit for Uncapped partitions is instead represented by the number of virtual processors when there is no contention.)

So in the second of the preceding figures where Partitions A and B have become Uncapped partitions, the Capped partition's compute capacity limits, which would have been imposed on these partitions starting at times 10:31 and 10:33, are as Uncapped partition not really limits. As Uncapped partitions, Partitions A and B are allowed to consume whatever compute capacity remains in the shared-processor pool. But before doing so, these and the other partitions are allowed to consume compute capacity up to their entitled capacity.

A Financial Note: Before going on to discuss best performance practices, let's first observe that there are some financial aspects to using Capped versus Uncapped partitions. These relate to IBM i OS licensing. These aspects might best be stated as in the following:

- For dedicated-processor partitions, the number of licenses is equal to the desired number of configured cores.
- For Capped shared-processor partitions, the number of licenses equals the total of the desired number of processing units – their overall Entitlement - configured rounded up to the next whole number. This total includes all Capped partitions. Notice that this total can not exceed the number of cores in the shared-processor pool, but it can be less. Such partitions can have more virtual processors than the number of cores defined by this entitled capacity, but the maximum compute capacity they can use is still limited by this pool size as well.
- For Uncapped shared-processor partitions, the number of licenses equals the maximum number of virtual processors configured, but only up to the limit of the number of cores in the shared-processor pool. Each partition can have a lot of virtual processors, but the compute capacity that they together can consume (over multiple or all shared-processor partitions) is no more than what is available in the shared-processor pool.
- As stated earlier MSPP function may be used to control software licensing capacity across multiple shared-processor partitions.

For On/Off Capacity on Demand (CoD), there are no additional licensing charges associated with a temporary processor activation. (Recall that the number of cores in the shared-processor pool is equal to the number of activated cores less the number of dedicated-processor partition cores.)

We had outlined in the previous section how each partition's entitled capacity is measured. To review:

- The shared-processor pool has only the compute capacity defined by the pool's core count.
- Every shared-processor partition has an entitled capacity. The total compute capacity over all shared-processor partitions is less than or equal to the compute capacity available in the shared-processor pool.
- Each virtual processor's compute capacity consumption is represented by the amount of time that the virtual processor is assigned to a core. If multiple of a shared-processor partition's virtual processors are attached to cores at the same time, the rate of compute capacity consumption is proportional of the number of such virtual processors (i.e., three concurrently executing virtual processors are consuming compute capacity at 3X the rate of just one).
- Compute capacity is accounted for within the notion of a time slice. It is effectively reset in the next time slice. Within such time slices, some – or potentially all – partitions measured compute capacity consumption may reach their partition's entitled capacity.

You also know that **Capped** partition's virtual processors cease their execution within a time slice when/if the compute capacity limit is reached, making its cores available at that moment.

Capped / Uncapped Summary

You can now also see the basic philosophy differences behind dedicated-processor and shared-processor partitions.

- **Dedicated-processor partitions** allow having a fixed compute capacity always available. With these, a set amount of compute capacity is reserved such that, if there is a "processor" available when a task become dispatchable, that task is guaranteed near immediate access to that processor. Again, dedicated-processor partitions effectively reserve some amount of a system's compute capacity to have ensured this. And this assurance of immediate execution – with minimal processor queuing effects – can be very important. This, of course, assumes that there are enough cores defined to such partitions to minimize such queuing delays.
- **Shared-processor partitions** maximize the usage of the compute capacity available in a pool of processor cores, a pool which might include the entire system's cores. But, as with any highly used resource, a core might not always be available when needed. And that means there is a potential for additional queuing delays during processing, with each virtual processor needing to wait for a short while for its opportunity to execute. Said differently, shared-processor partitions are trading off the opportunity to increase the usage of your available compute capacity (i.e., cores are used more often) for occasionally longer processing response times. As with adding cores to each dedicated-processor partition, adding more cores in the one shared-processor pool can improve response time for the shared-processor partitions.
    - **Capped Partitions:** As with Dedicate-processor partitions, capped partitions have a fixed compute capacity and no more. Unlike dedicated-processor partitions, this compute capacity can be provided in terms of fractions of a core's compute capacity. Too small a compute capacity for the work required, and too few virtual processors for the number of tasks can result in both task and virtual processor queuing delays to occur.
    - **Uncapped Partitions**: It is possible for such partitions to perceive compute capacity up to the total that is available to the number of virtual processors in the partition, potentially well above the entitled capacity. But, because of shared-pool contention and because of the opportunity for fractional core Entitlement, it is also possible that such partitions will be allowed only the compute capacity implied by the partition's Entitlement. This difference can be considerable and might be perceivable as a difference in response time.

Performance Tip: It is not unreasonable for an Uncapped partition to have more virtual processors than suggested by its entitlement. When cores are available in the shared-processor pool, this allows this partition to consume that capacity, thereby improving the response time perceived by the partition's users. But the capacity implied by the number of virtual processors is not the guaranteed capacity. The Uncapped partition's entitled capacity is its only guarantee. When the shared-processor pool gets busy supporting the virtual processors of other partitions, the lower entitled capacity – and so its potentially perceptibly slower response time – is what will be perceived by the partition's users.

Dedicated-Donate

Thus far we have showed that the maximum compute capacity available to all shared-processor partitions is equal to the sum of all entitlements; this total compute capacity can be available to even a single uncapped partition if that partition has an equal number of virtual processors. True enough, but there is a way that the number of cores in the shared-processor pool can occasionally grow above even this compute capacity limit.

Recall first that the size of the shared-process pool is normally equal to the number of activated cores left over after the dedicated-processor partitions are assigned their cores. Growing the compute capacity of the shared-processor pool means borrowing available compute capacity from one or more of these

dedicated-processor partitions. Such dedicated-processor partitions can be configured – see figure below - to donate their temporarily unused cores to the shared-processor pool. This, in turn, can decrease the contention of active virtual processors for cores, and, with that, decrease virtual processor queuing delays. Compute capacity, otherwise thought of as tightly tied to dedicated-processor partitions, can – when the circumstances are right – be made available to shared-processor partitions.

Note that temporarily unused cores are being donated when available and, more to the point, only donated when a dedicated-processor partition perceives some of its cores as likely to remain unused. Even when average CPU utilization of such dedicated donating partitions is relatively low, all of its cores might nonetheless be used. Having even one task on every dedicated partition's core – recalling that on POWER7 there can be four – is all it takes for every core to be busy. But if one or more cores are tending not to be used for a while, even for short periods of time, such cores can be donated for temporary – and occasionally longer term - use by a shared-processor partition's virtual processor.
[*Technical note: When the donating partition again needs its core, the partition can request its use and get it quickly. But returning ownership to the owning partition does add some additional time over that of simply dispatching a newly dispatchable task to a still available (i.e. un-donated) core. The point is that there can be a typically minor response time impact on the dedicated-donating partition.*]

All it takes to define dedicated-processor partition as one willing to donate its cores is to flag it as in the following example HMC window …. Check under Processor Sharing "**Allow when partition is active**."

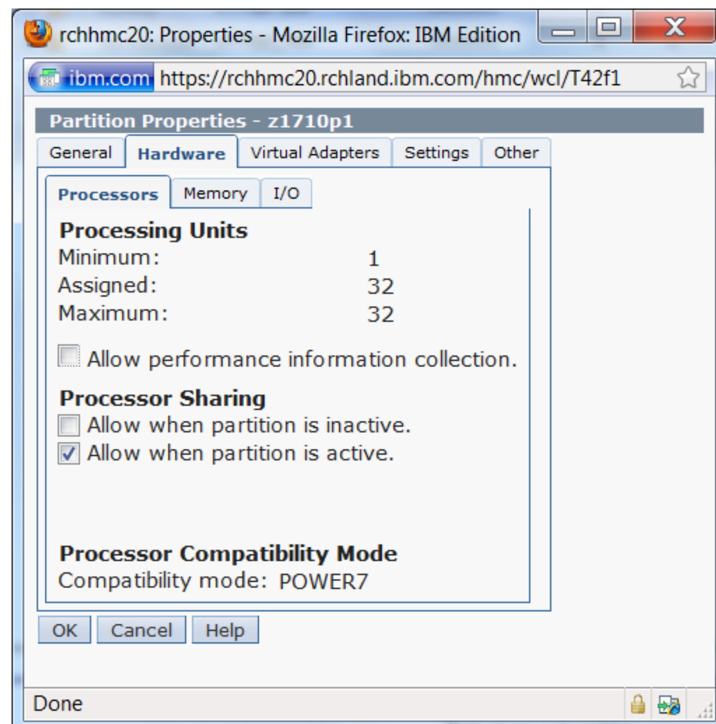

Figure 12 – Partition Sharing

Interesting, but of lesser use, is the other flag "Allow when partition is inactive". When that flag is set, this dedicated-processor partition's cores are added to the shared-processor pool when this partition is shut down. These same cores are often restored when the partition is again activated.

DLPAR and the Desired/Minimum/Maximum Processor Settings

For dedicated-processor partitions, the hypervisor attempts to give each partition exactly the number of physical cores specified as the "desired" number of processing units within the partition's profile. Failing there, the hypervisor chooses a number between the "desired" number and the "minimum" number. Either way, while the partition is active, you can request a change in the number of the currently active cores used by a dedicated-processor partition via the DLPAR (Dynamic Logical PARtition) function and have it quickly take effect. You can also, of course, alter the profile, but such changes take effect at a later time as you will see.

To use DLPAR to increase the core count for a dedicated-processor partition, the active partition being altered must know ahead of time to expect the increase. Providing this pre-knowledge is the purpose of the "maximum" processing unit setting on the partition profile. When the partition is activated, each partition allocates internal resources – virtual processors - up to this maximum, but only the hypervisor-selected number (often the specified "desired" core count) of virtual processors are actually allowed to be assigned to cores. For example, if "maximum" is 4 and "desired" is 2, a dedicated-processor partition will only be assigned two physical cores to use. When the DLPAR operation requests an increase by two to be changed to four active virtual processors, the partition begins using the other two of what it had previously perceived as inactive virtual processors, resulting in all four virtual processors bound to physical cores if these additional cores were available.

[*Technical Note: If DLPAR is used to alter the assigned number, the change is immediately reported to PHYP which also immediately initiates the activation/deactivation in the partition. If the virtual processor count is changed via the profile - altering "desired" and saving the profile - the change takes effect with the next explicit activation with that profile. A DLPAR operation does not alter the partition's profile.*]

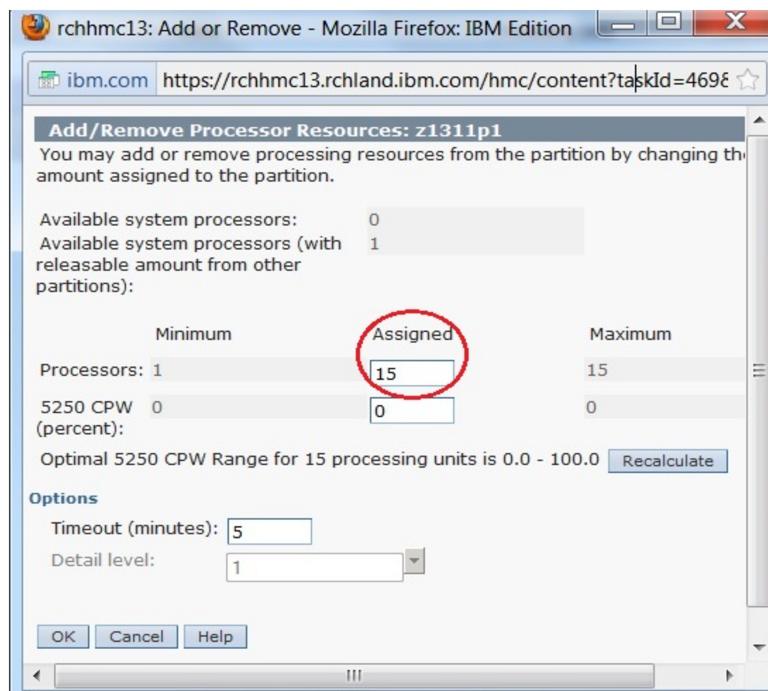

Figure 13 – DLPAR

We had previously commented on how a dedicated-processor partition's virtual processors have some strong level of persistence to particular cores. Here you can see that even for dedicated-processor partitions this persistence can change; the number and location of a partition's cores can be changed fairly easily. When you choose to decrease the number assigned, the number of virtual processors remain, but fewer of them remain active and assigned to dedicated cores. For example, when decreasing from 4 core to 2, what had been four active virtual processors assigned to cores still remains four virtual processors, but only two of them will be used and be assigned to cores.

These changes in the number of dedicated-processor cores can also result in changes to the locations of the cores being used. With other partitions also changing their core counts, the locations of the cores assigned to any one of them can change quite a bit. As you will see shortly, though, the location of which cores are selected for these purposes matters to performance. (This will be discussed in the section "POWER7's Nodal Topology".)

DLPAR provides shared-processor partitions a similar capability, but with some considerable differences. You can see some of them in the following HMC window of a DLPAR operation on a shared-processor partition:

Add/Remove Processor Resources: z1714p1
You may add or remove processing resources from the partition by changing the amount assigned to the partition.
Available system processing units: 28.0
Available system processing units (with releasable amount from other partitions): 28.0

	Minimum	Assigned	Maximum
Processing units:	0.05	2.5	8.0
Virtual processors:	1	4	8
5250 CPW (percent):	0	0	0

Optimal 5250 CPW Range for 4.0 processing units is 0.0 - 100.0 Recalculate
Uncapped Weight: 128
Options
Timeout (minutes): 5
Detail level: 1
OK Cancel Help

Figure 14 – DLPAR with Shared-Processor Partition

- As with dedicated-processor partitions, there are a "maximum" number of virtual processors understood to be potentially used in some future. Each partition's OS needs to know this number to internally represent their potential future use. But this or some lower number (i.e., often "desired" but potentially down to "minimum") are the number that are actually being used. It is this number of virtual processors to which the OS assigns tasks, making them active, and which then can be assigned to the shared-processor pool's cores. For dedicated-processor partitions, the virtual processor count also designates the entitled capacity of the partition, because the partition is entitled to full-time use of the associated physical processor core. For shared-processor partitions, the virtual processor count does not define entitled capacity precisely; it only designates the range in which the entitled capacity may vary.

So, for shared-processor partitions, DLPAR also allows for the partition's entitled capacity – here stated in terms of fractional **Processing units** – to be modified. Here, again, entitled capacity is stated as being within a profile's predefined bounds of a minimum and maximum compute capacity. Here, too, the total entitled capacity over all shared-process partitions can not exceed the compute capacity represented by the cores associated with the shared-processor pool.

*[**Technical Notes**:*
- *Increasing a shared-processor partition's virtual processor count alone might occasionally help with response time, but more often the constraint is entitled capacity. A virtual processor count increase might provide more opportunity for execution parallelism, but it might also only mean that the compute capacity "limit" is reached sooner. The exception is the case of an uncapped partition where there is sufficient unused capacity in the shared-processor pool to satisfy the demands created by additional virtual processors.*
- *Increasing the virtual processor count can increase how quickly the entitled capacity limit is reached, but we'll see later that the partition's Task Dispatcher – rather than assigning tasks over <u>all</u> virtual processors - attempts to assign dispatchable task fewer than this maximum number of virtual processors; the lower processor count chosen is one more in line with the partition's entitled capacity. The Task Dispatcher can later use all available cores when workload increases. This difference in assigning tasks means an earlier increase in the use of the SMT capabilities of a core – and so fewer virtual processors - than would be the case for dedicated-processor partitions.*
- *Recall that the size (and so location) of the shared-processor pool is essentially that set of activated cores which are not part of any dedicated-processor partition. As DLPAR changes are made to increase or decrease the size of any dedicated-processor partitions, the size – and so compute capacity – of the shared-processor pool changes as well. Partition location will be discussed later in the section on **POWER7's Nodal Topology**.]*

Task Dispatching and the Measure(s) of Consumed Compute Capacity

You know that the hypervisor is responsible for ensuring, over all of the shared-processor partitions, that their virtual processors get fair use of the often fewer number of cores of the shared-processor pool. In fact, that has been a theme of a number of the preceding sections. This section stays with that theme and discusses two subtly related concepts associated with the measure and presentation of consumed compute capacity. They relate to:
1. The different ways that the partition's OS and the hypervisor measure CPU utilization, and
2. The notion of maximum compute capacity consumption for Capped and Uncapped virtual processors.

The way that the partitions themselves measure consumed compute capacity (i.e., CPU utilization) is different than the way that the hypervisor measures consumed compute capacity.
- **Hypervisor Compute Capacity Measurement:** The hypervisor's means of measuring consumed compute capacity of a shared-processor partition is determined merely by how much wall clock time any of the partition's virtual processors had been assigned to a core. This is independent of the number of tasks – whether on POWER7 1, 2, 3, or 4 tasks – that are actually attached to any virtual processor. A virtual processor executing on behalf of even just one task – one SMT hardware thread – is still considered a fully utilized core for purposes of tracking a partition's consumption of its entitled capacity. Since no other virtual processor can execute on a core already used by a virtual processor, no matter the number of tasks also assigned there, that core is perceived by the hypervisor as being fully used.
- **Partition Compute Capacity Measurement:** The reason for SMT on POWER processors is to provide more compute capacity than can normally be used by a single task executing on a core. (See "SMT" in glossary for more.) On POWER7 processors with four SMT hardware threads per core,

that means that each unused hardware thread within any core or virtual processor also represents some amount of still available compute capacity. For that reason a partition can only present itself as at 100% utilization – meaning that there is no additional compute capacity - if every SMT hardware thread of every core is used over a period. Said differently, if even one SMT thread within the partition's cores/virtual processors had momentarily gone unused during some period, CPU utilization will be presented as less than 100%, this because the unused thread(s) represents still available compute capacity. Interestingly, because of this effect, on POWER7 if all of a dedicated-processor partition's virtual processors were executing throughout a measurement period on behalf of only one task each, that partition would be presenting itself as roughly 60% CPU utilization; that is a representation of just how much compute capacity remains in a core for up to three more tasks. Note: The OS provides controls over the SMT mode of the partition's virtual processors. The preceding discussion assumed SMT4 mode. When ST mode is used, the partition's processor utilization measurement is essentially the same as the Hypervisor's.

It is the hypervisor's means of measuring compute capacity – not the partition's means – that is used by the hypervisor for enforcing entitlement limits and fair use of the shared-processor pool.

But, as you will be seeing, **this difference results in the OS' Task Dispatcher using different algorithms for dedicated-processor versus a shared-processor partitions.** This difference in task dispatching algorithms is intentional and is a function of the different purposes of these partition types. There is a detectable difference in the performance characteristics of individual tasks executing in one type versus the other. We are going to attempt to explain this effect in the remainder of this section.

As you know, whether Capped or Uncapped, each shared-processor partition has an entitled capacity. For Capped partitions this entitlement represents a compute capacity limit; the partition's virtual processors cease executing at this limit within each time slice. For Uncapped, the entitled capacity is also a guarantee that some minimum compute capacity is available if needed, but virtual processors can also continue executing past this limit.

Even so, Uncapped partitions also have an additional compute capacity limit; an Uncapped partition can perceive no more compute capacity than is available in the concurrent execution of all of its virtual processors. Said differently, a shared-processor partition with a given number of virtual processors continuously executing has available to it no more compute capacity than is available in that number of cores.

With these notions in mind, when a Capped partition reaches its entitled capacity within a time slice, the hypervisor clearly considers that partition to have reached its 100% compute capacity limit. The partition is Capped, it reached its entitled capacity limit, and it ceased executing; that is the definition of consuming 100% of its compute capacity as seen by the hypervisor. But the Capped partition itself sees things differently from the inside. The partition will present itself as having consumed 100% of its compute capacity at this same moment only if every SMT hardware thread of all of its virtual processors were also used whenever its virtual processors were using a core. If even one SMT hardware thread was not used when the partition's virtual processors were executing, the Capped partition would present itself as still having available compute capacity. The point is that **even though there is compute capacity available as seen by the partition, no more throughput will be produced for this Capped partition until the next time slice**.

This same definition – of 100% utilization – can also be applied to Uncapped partitions, but only up to a point. That is, at the point in time when the Uncapped partition's entitled capacity limit is reached, if every hardware thread had been constantly used, the partition's OS would also consider this as being 100% CPU utilization. But since an Uncapped partition's processing is allowed – but not guaranteed - to

continue past this entitled capacity point, the measured CPU utilization of this partition is also allowed to continue, even well above 100% (which is reported at the entitled capacity point).

For example, let's define an Uncapped partition as having an entitled capacity of 1 core and 4 virtual processors. Let's also keep all of the SMT hardware threads of all four virtual processors busy executing tasks (totaling 16 on POWER7) throughout some measurement period. This being the case, the partition is able to consume 4 times as much compute capacity as was implied by that partition's entitled capacity. Since 100% means executing for only the entitled capacity of 1 core (say, executing with only ever one virtual processor), executing continually with four virtual processors (i.e., at maximum compute capacity) also means that this partition's OS is measuring itself as executing at 400% utilization.

As another example, as in the top portion of the following figure, suppose we have the same partition (1-core compute capacity, four virtual processors) but now Capped. Further, instead of having 16 tasks, let's only have 4. Let's place the 4 tasks onto only one of the four virtual processors and keep the tasks executing continuously; only the one of the four virtual processors is active. Because of the one-core entitled capacity for this Capped partition, this fully utilized virtual processor can continue executing. Again, because of the one-core compute capacity of this partition, this Capped partition would be perceived by the hypervisor within each time slice as consuming all of its compute capacity (and no more) and so would also be perceived by the OS as at 100% utilization.

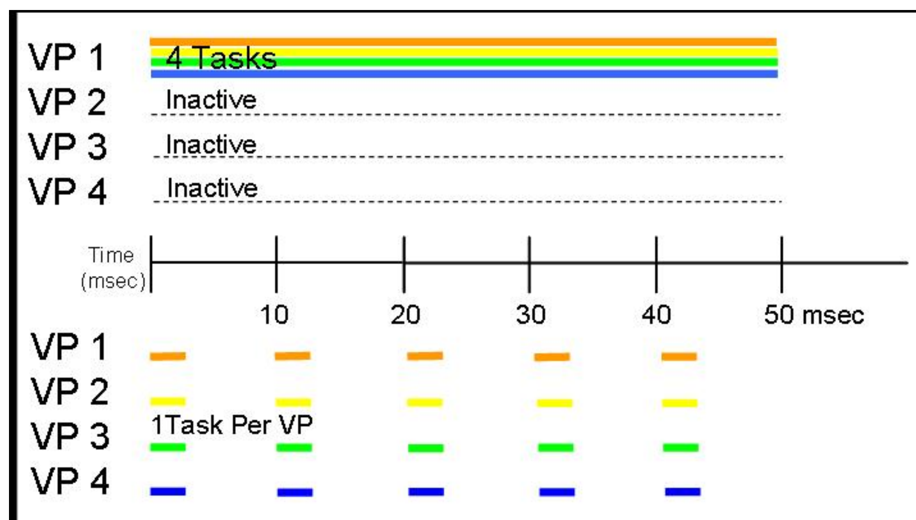

Figure 15 – Virtual Processor Task Dispatching

Next, as in the lower portion of the previous figure, let's take exactly the same Capped partition and the same number of tasks (4), but this time let's spread the tasks in such a way that there is one task per virtual processor. (This is what would be done for dedicated-processor partitions to provide best individual task performance.) Just as with the previous case, we would like to keep all four virtual processors continuously executing on behalf of their one task. However, we can't because of the one-core entitled capacity limit – as measured by the hypervisor – is reached four times faster than in the upper case. There are four virtual processor executing – one each over four cores - and the hypervisor measures compute capacity based on core usage. With the partition's 1-core compute capacity limit being reached four times faster, these virtual processors will get to execute for only ¼ of each time slice.

Four tasks on one virtual processor were here allowed to continually execute. Four tasks with one each on four virtual processors were only allowed to execute ¼ of the time. If all tasks here were individually executing at the same speed, the upper case would seem to produce four times as much throughput. However, as you'll see in the next section, tasks sharing a core individually execute somewhat slower

than when executing alone on a core, but not four times slower. So, for shared-processor partitions, it seems clear enough that we are ahead of the game of producing more throughput if tasks were not spread – spreading is often used with dedicated-processor partitions – and were instead dispatched to fewer than the available number virtual processors. We'll be going over this notion more in a later section after we look more at SMT-based performance effects.

We have now seen SMT effects show up a number of times. So let's take a bit more complete look at SMT as it relates to processor virtualization in the next section.

Simultaneous Multi-Threading (SMT) Considerations

SMT in POWER7 and preceding processor designs is not really capable of being virtualized. In POWER7, each virtual processor gets up to four SMT hardware threads. (For POWER5 and POWER6, the number of hardware threads per core is two.) It is the entire physical core that is virtualized, not the number of hardware threads within it. But as you have already seen, virtualized or not, some of SMT's effects do show through. We'll be looking at more effects shortly.

The unit of processing that a virtual processor perceives is that of an entire core, each core with POWER7's four SMT hardware threads. Even for shared-processor partitions, the hypervisor assigns an active virtual processor with its four hardware threads as a unit to a core. This also means that, whether that virtual processor was supporting one or multiple tasks when active, these tasks are dispatched to a core by the hypervisor as a single unit.

The reason that SMT exists in the first place is that there is far more compute capacity in a core than can typically be used by the instruction stream of a single task; it typically takes multiple independent instruction streams (i.e., multiple tasks) on a core to consume that compute capacity. One reason for this is that when one task's instruction stream does not happen to find the data it needs in the core's cache, that processor must take time to access the data from slower resources, as for example memory DIMMs. These accesses can take a while (typically many 100s of processor cycles, with these accesses done frequently) and during all that time the compute capacity represented by the massive fine-grained instruction parallelism that is a processor core is effectively going unused. So, why not allow one or more other task's independent instruction streams to execute on that core in the mean time? By doing so the core continues to stay busy while another part of the system is concurrently handling the cache miss(es). This, in a nut shell, is the concept of SMT.

It happens that it is not only during such cache misses that the multiple instruction streams are executing. With SMT, all of the tasks dispatched to that core really are concurrently executing their instruction streams through the processors pipes even when none of them happen to be delayed on the likes of a cache miss. You can find a link to more on SMT in the glossary entry on SMT.

The effect of all this is that SMT provides more compute capacity per core – on POWER7 using SMT4, this is roughly 1.6 to 2 times more - to concurrently execute four dispatchable tasks as compared to a similar processor core which happens to only support a single task per core.

SMT, though, is not really like having a separate core for every task. Unlike tasks each executing alone on individual cores, in SMT the instruction streams of multiple tasks **really are** using the common resources of a single core. And, often enough, one such task needs to wait until the core's shared resource is again available. The near linear scaling assumed from having additional cores is not available when adding additional tasks to the same core via SMT. Because the multiple instruction streams are sharing the core's resources (cache, pipes, TLB, store queues, what have you), when these instruction streams conflict on some common resource, one of the threads needs to wait; the wait is often little more than a

processor cycle or two, but the waits are frequent. The high level result of all of this is that as one, then two, then more tasks are assigned to a core, each of the individual tasks do get to execute, but they each also appear to slow down. So, yes, SMT provides more compute capacity, but it does so while slowing the execution speed of the individual tasks that it is supporting. This is all quite normal, being just a characteristic of SMT. And, yes, the tasks may execute slower, but without SMT's additional "processors", the alternative is that they may need to simply wait their turn to execute. This is normal, but also important to virtualization as we will see next.

The OS' Task Dispatcher is aware of these SMT effects. The Task Dispatcher's answer to this effect is to spread dispatchable tasks over the available cores of dedicated-processor partitions. Four dispatchable tasks over four cores often means one task per core; each getting full use of that core. Doing so – as opposed to 2-to-4 tasks per core - allows each task to execute faster, freeing up that "processor" for subsequent tasks sooner. Of course, as the workload increases, this same partition starts to have more tasks per core, up to POWER7's four. But, because of this faster single-task execution, such spreading is the right thing to do for dedicated-processor partitions. This effect is possible partly because this dedicated-processor partition has a fixed compute capacity always available to it.

This spreading, though, is not necessarily appropriate for shared-processor partitions. The reason for the difference stems from the quite reasonable over-subscription of the total number of active virtual processors – well over the shared-processor pool's core count - AND because of the way that the hypervisor measures the consumption of a partition's compute capacity. This is why we are discussing SMT here in the context of a virtualization discussion.

To explain, suppose that a 16-core shared-processor pool is being shared equally by 16 partitions (entitlement = 1 core/partition), and each partition has four virtual processors. Overall, the result is 64 virtual processors equally sharing the shared-processor pool's 16 cores. If all 64 virtual processors are concurrently active, the hypervisor will allow each virtual processor to execute only ¼ (i.e., 16 cores/64 VPs) of each 10 msec time slice. This is true whether each virtual processor was executing on behalf of one task, two, three or four tasks. So, in the extreme, let's have each virtual processor support just one task. For this single-task per virtual processor case, each is able to execute faster alone on its own core rather than with three other tasks (let's say, by 2X). Whatever the benefit of executing alone, each of these virtual processors is nonetheless only executing at this full speed for only ¼ of the time. When executing, the individual tasks were executing faster, but they simply are not executing enough to make up the difference in throughput. This is not a particularly good trade-off. Because of this shared-pool contention, executing tasks alone actually resulted in their producing about half the throughput possible for these same tasks in a 4-core dedicated-processor partition. We can do better.

Shared-processor partitions exist to allow higher consumption of the entire shared-processor pool's compute capacity. But POWER7's SMT-based cores with only one or two tasks per virtual processor leaves a considerable fraction of the pool's compute capacity is not being used; the cores might all be used, but individually each core still has a lot more compute capacity available because of SMT. To allow more use of the pool's compute capacity, and decrease the number of virtual processors contending for it, you'll notice next that there is value in dispatching the same number of tasks on to fewer virtual processors.

So alternatively, the OS' Task Dispatcher - knowing that its partition is part of a shared-processor pool - has the option of having different task dispatching algorithms, one for dedicated-processor, and another for shared-processor partitions. Knowing that there may be too many virtual processors competing for the available cores, rather than automatically spreading dispatchable tasks over the partition's virtual processors, the Task Dispatcher can alternatively first dispatch the same number of tasks onto fewer virtual processors, using the virtual processor's SMT capabilities sooner. When applied to many partitions, the result is fewer virtual processors competing for the cores of the shared-processor pool.

With more tasks per virtual processor compute capacity consumption is actually increased and overall throughput grows.

With this in mind, using the example of the same 16 4-virtual processor partitions mentioned above, let's instead picture each OS' Task Dispatcher as using first just two of its four virtual processors, dispatching its four tasks as two tasks per virtual processor. This is instead of one task each on four virtual processors.

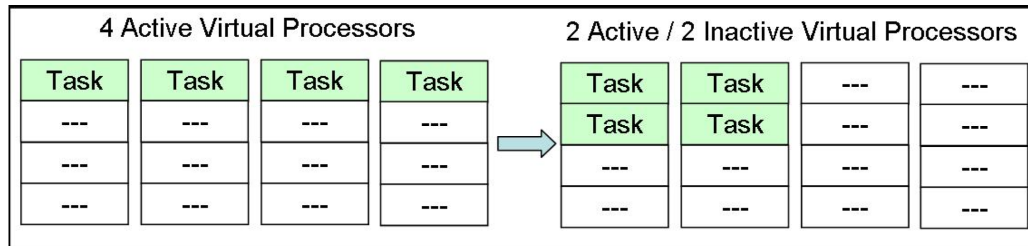

Figure 16 –Virtual Processor Tasks

Now with half (i.e., 32 active virtual processors, rather than 64) of the virtual processors contending for the cores, each virtual processor gets to execute for ½ of each 10 msec time slice (i.e., 32 VPs over 16 cores), rather than ¼ of the time. With two tasks per core, each task is executing only slightly slower than it would alone on a core, but each is now also executing twice as long per time slice. Roughly speaking, with this approach, each task perceives only slightly less than twice as much performance as compared to the previously mentioned (i.e., spread-based) dispatch algorithm. And, given that all partitions are doing this, all partitions are similarly being benefitted by the lower contention for cores. More of the system's compute capacity is being used and tasks effectively execute faster. What a deal!

On IBM i, the decision of when to start doubling up like this is a function of the partition's entitlement. If a partition is entitled to one core's compute capacity or less, consolidating onto fewer virtual processors can happen almost immediately. If the partition's entitlement is – say – two units, tasks may be spread across two virtual processors before moving onto more tasks per virtual processor. Once these two virtual processors are completely used, additional virtual processors will be activated and assigned tasks. So, using Entitlement, you can influence the previously mentioned behavior.

As you can see, you can have a virtual processor count well in excess of the partition's entitled capacity, but the number of virtual processors in excess of the entitlement will tend not to be used until the number of dispatchable tasks grows high enough that they are needed. In the mean time, you'll have each of the tasks perceiving the SMT performance effects outlined earlier. Once again, there is benefit from keeping the partition's entitled capacity and virtual processor count relatively close.

So why did we so verbosely tell you about this difference? It stems largely from the difference in philosophy between dedicated- and shared-processor partitions. Shared-processor partitions exist to maximize the use of the compute capacity of the shared-processor pool (which can be an entire system), potentially at the cost of the performance of individual tasks. Shared-processor partitions are designed to increase the oversubscription of the compute capacity of the shared-processor pool in order to maximize the compute capacity of that processing resource. Although the total entitled capacity over each shared-processor partition can not be more than the compute capacity of the shared-processor pool, the number of active virtual processors concurrently contending for those cores can far exceed the pools core count. This is a way of decreasing that contention of having an excessive number of virtual processors over a fewer number of cores. But there is a trade-off. In a dedicated-processor partition, the same number of tasks would

typically be spread over the partition's cores with the result being fewer tasks per core. For shared-processor partitions, more tasks are assigned earlier to individual virtual processors than would ordinarily be the case for dedicated-processor partitions. As a result, with more tasks per core, it is also true that the tasks might execute individually slower in a shared-processor partition than in a dedicated-processor partition. **This is the natural trade-off which results from the intent to maximize the system's compute capacity with a shared-processor partition.**

Performance Tip: The Task Dispatcher's algorithm for dispatching tasks in dedicated-processor partitions is different than that of shared-processor partitions. For dedicated-processor partitions, task dispatching focuses more on individual task performance. For shared-processor partitions, it is attempting to assist in the maximum capacity consumption of the shared-processor pool and so can trade off individual task performance.

iDoctor and CPU Utilization

We pause here to take a look at how an IBM i tool called **iDoctor** presents CPU utilization.

To help explain, we use a simple workload which ramps up its throughput over the run. Starting with a set of eight threads executing essentially the same instruction stream, the workload adds another three threads every minute. Each thread is executing with roughly a 25% duty cycle; out of every ¼ second, a given thread attempts to randomly executing for about 25% of the time. As a result of this duty cycle, on an 8-core P7+ with SMT4, in this test throughput reaches a maximum at about 150 threads. This continues until the CPU utilization is very nearly at a maximum. Each of the threads track their "transactions" so we can present increasing throughput over time.

You can see the expected effects of SMT4 on both compute capacity and response time in the following graph of this ramp-up workload. At first, as the thread count increases, most threads are executing alone on one of the eight cores. Throughput increases quickly as a result. As two, then three, then four tasks are executing on a core, throughput increases more slowly.

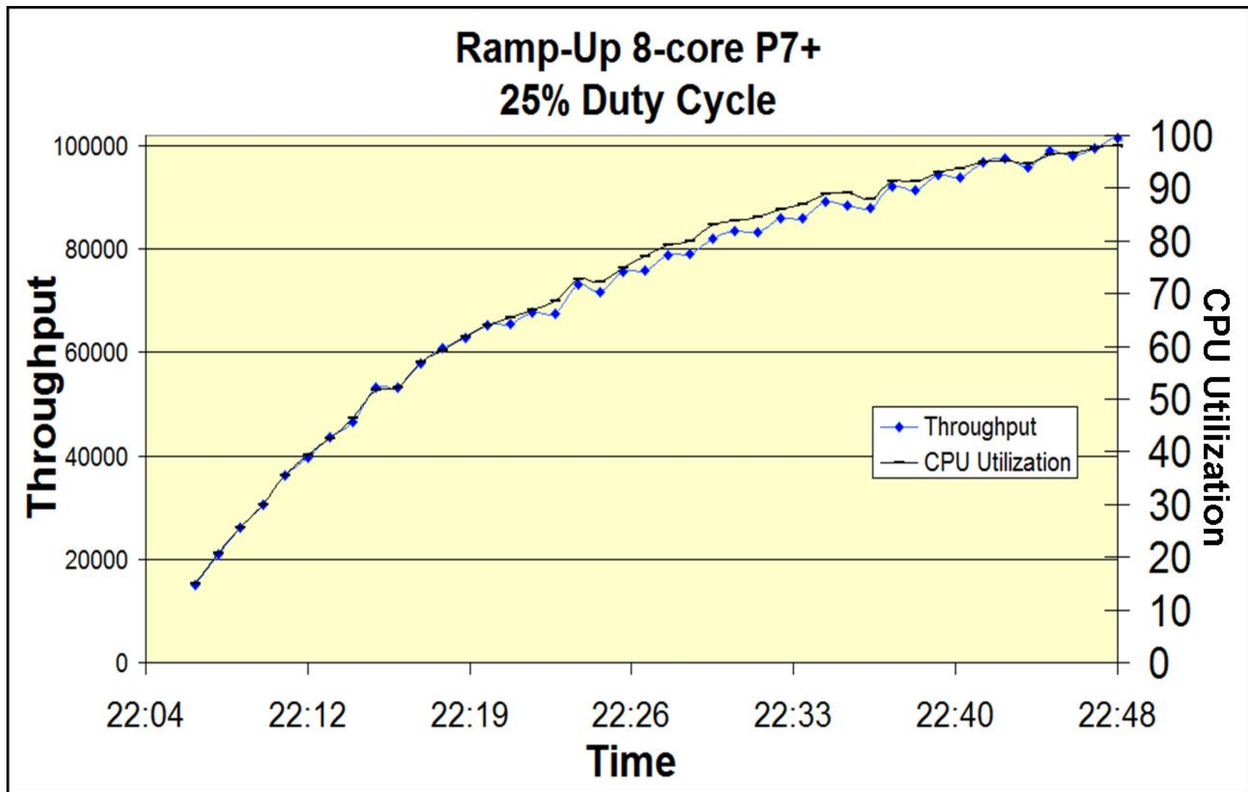

Figure 17 – SMT4 and CPU utilization

Also notice that the workload itself is measuring CPU utilization. As we would want, the measured CPU utilization is tracking often perfectly, at other times very well, with the throughput being measured for this workload. As we described earlier, this form of CPU utilization is attempting to measure the percentage of each core's compute capacity being consumed by this workload. (*This synthetic workload uses MI instruction MATRMD option 0x26 to calculate CPU utilization*.)

We next present iDoctor's view of this same partition during the same period of time.

In the following figure, the blue line near the top represents CPU utilization in the same way as the workload was measuring CPU utilization. As in the above figure, as more threads are added – three more every minute, CPU utilization increases at first quickly and then more slowly, in an attempt to represent the remaining compute capacity of this 8-core partition. (It is important in the following to keep in mind that this is an 8-core SMT4 partition.)

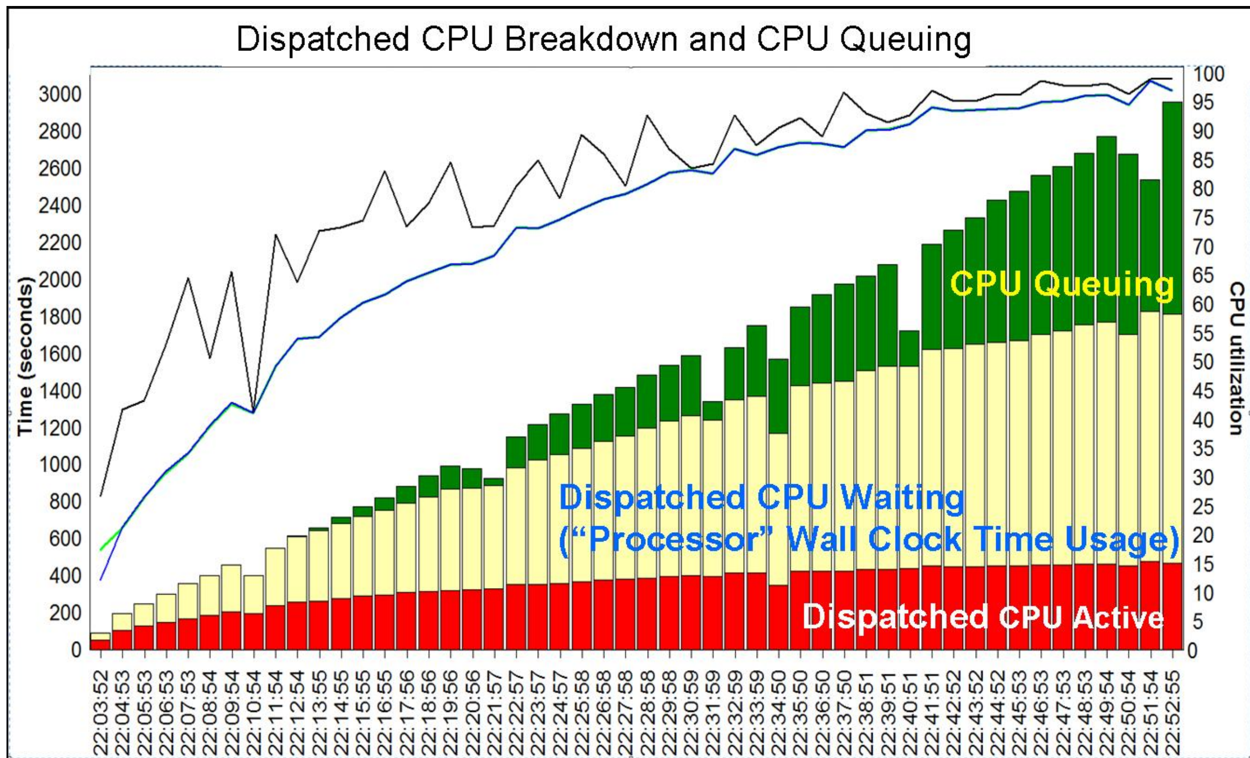

Figure 18 –Dispatched CPU Breakdown and Queuing with Dedicated Processor Partition

So let's next look at the red, yellow, and green regions of this graph.

Starting with green (representing CPU queuing), keep in mind that in this 8-core SMT4 partition there are 32 "processors". By the end of the run, there are about 150 threads active, albeit for roughly 25% of the time. Even – say – half way in to the run, there could be – but typically far fewer – 75 threads contending for the "processors". Whenever there is even one thread waiting its turn to use a processor, the amount of time that it (and others) are waiting is represented here by "CPU Queuing". Notice also that since CPU utilization is not at 100% until the end of the run, there are also periods of time when one or more "processors" are available.

Next, looking very closely, you might notice that the red region – Dispatched CPU Active tracks very closely with the measure of CPU utilization. In this representation, Dispatched CPU active corresponds to the partition's view of capacity used, as described earlier. It is just presented here in terms of time (seconds). Looking at the rightmost red bar, the value in seconds happens to be 466.4 seconds. That bar happens to also represent 60.25 seconds and is occurring when CPU utilization was measured as being 96.8%. Eight cores fully utilized – with four threads each - would have a "Dispatched CPU Active" value of 482 seconds (8 cores * 60.25 seconds); this represents 100% CPU utilization. So the value 466.4 represents 96.8% (466.4 seconds / 482 seconds rounded up). Any one of the red bars would produce the same results being equal to CPU utilization.

So we have red (Dispatch CPU Active) representing CPU utilization in terms of processing seconds. We have green (CPU Queueing) representing how much time some threads are waiting for a "processor" on this system during this synthetic workload. So what is yellow (Dispatched CPU Waiting)? Mathematically, it is the difference between the sum of the task dispatch time, that is, the time that tasks are assigned to a thread of a virtual processor, less the sum of the entitlement charged to the tasks, which is represented by the Dispatch CPU active. Thus, the sum of the Dispatch CPU Active (red) and Dispatch CPU Waiting (yellow) is the sum of the task dispatch time. This sum is depicted as the peaks of the

yellow and labeled “Processor wall clock time usage”, where in this context “Processor” means “Virtual Processor Thread”.

The Dispatch CPU Waiting technically is not a waiting time as it imputes both virtual processor delays and SMT effects. In that sense, it’s a bit of a misnomer, but it really does have meaning and can be useful. For the yellow region, it is instructive to notice that as CPU increases from left to right the red bars are also a decreasing fraction of the value of the corresponding yellow bars. The cause is related to the fact early on the workload there are few enough active threads that, when a core is active, there is also typically only one thread per core. Well later in the ramp-up workload, there are three and often four threads per core, and the processing capacity is being charged to the dispatched tasks accordingly.

We next make a minor change in our environment. We switch to an uncapped shared-processor partition, still with 8 virtual processors, but with 6 cores of entitled capacity. The shared-processor pool is 16 cores, sufficient to allow all eight virtual processors to continually execute. With our partition and exactly the same ramp-up workload as above executing alone in this pool, we produced the following Job Watcher graph.

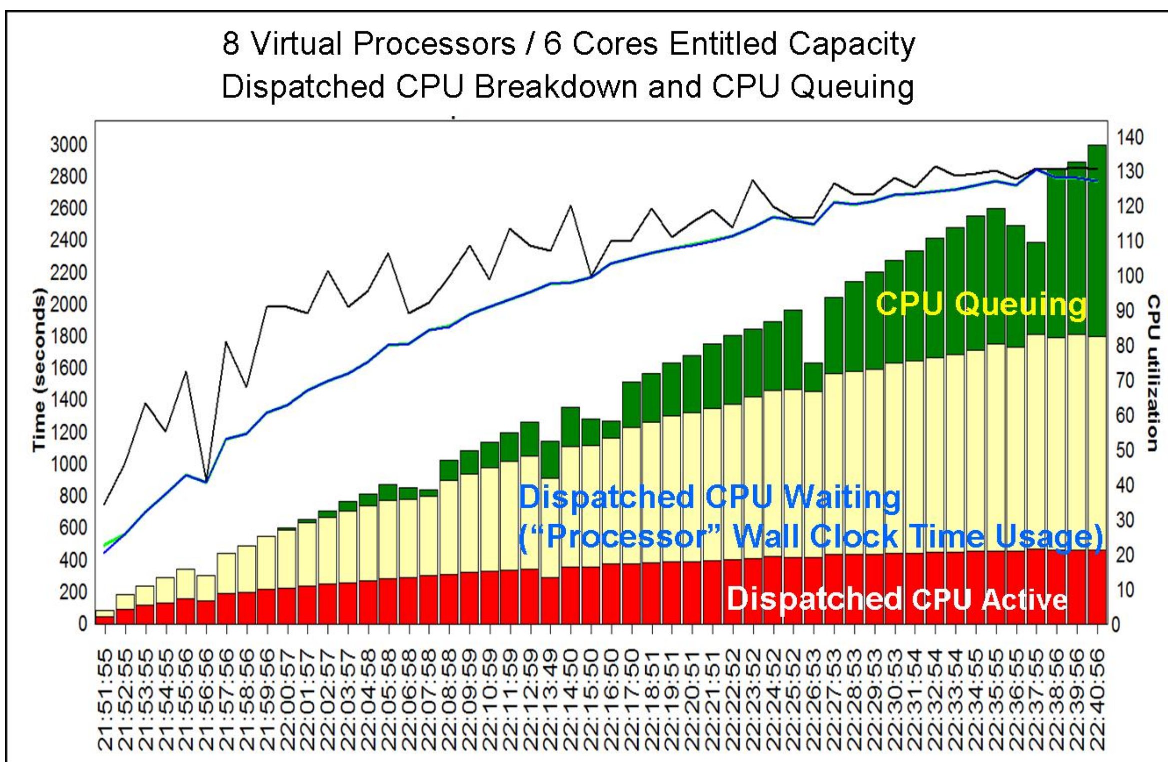

Figure 19 – Dispatched CPU Breakdown and Queuing with Uncapped Shared-Processor Partition

Go back and compare this graph with that produced for the 8-core dedicated-processor partition. With nothing else competing for these cores, the graph is essentially the same, except for one big difference. Check the right hand axis; CPU utilization presented as percent is higher. It is stated relative to the partition’s 6 cores of entitled capacity. But now also notice that the scale of the left hand axis – CPU utilization in seconds – is the same, and so is the shape of the bar graphs. Since each of this shared-processor partition’s eight virtual processors are effectively tied to a core – just like a dedicated-processor partition – the workload is ramping up and using the cores and its compute capacity in essentially the same way.

As a further variation, we take the same ramp-up workload and the same uncapped shared-processor partition (i.e., 8 virtual processors with 6 cores of entitled capacity within a 12-core shared-processor pool) and replicate it in a second partition. Job Watcher's view of one of these partitions is shown as in the following figure.

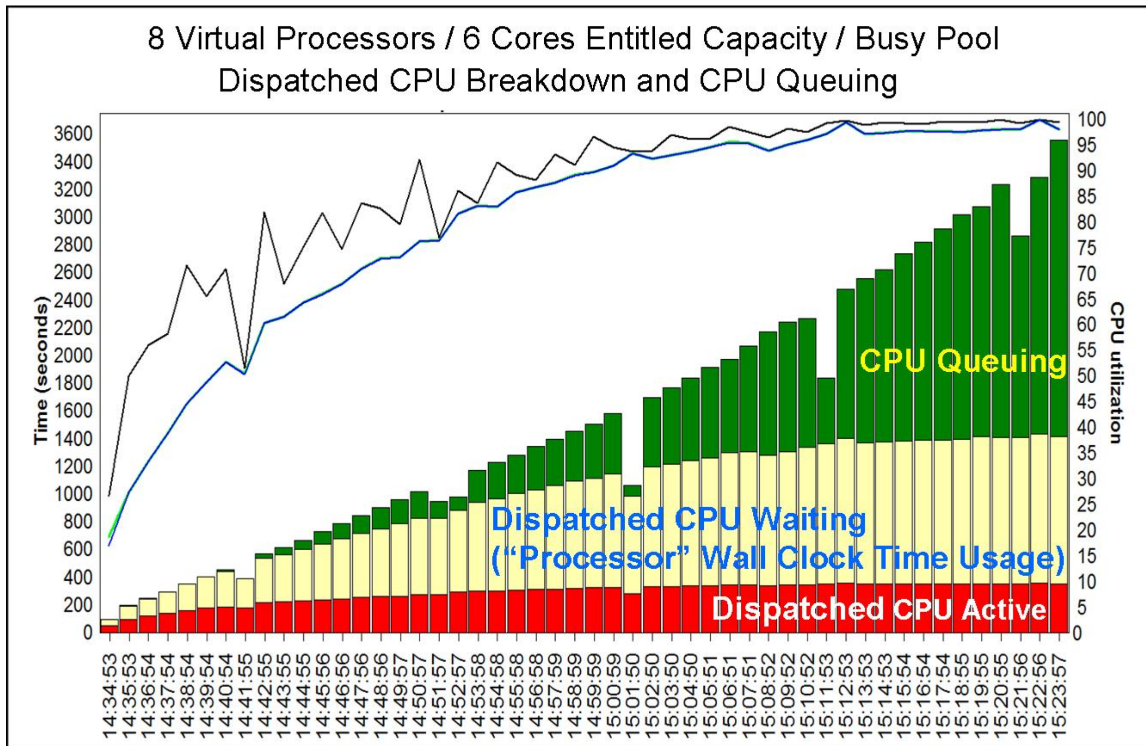

Figure 20 – Dispatched CPU Breakdown and Queuing with 2 Active Uncapped Shared-Processor Partitions

There are only 12 cores in the shared-processor pool. With both partitions executing essentially the same thing, at some point – in fact, for most to the run – these cores are also equally sharing the pool with a maximum of 6 cores per partition; this even though each partition also has 8 virtual processors. Unlike the previous case that was capable of keeping all eight virtual processors busy and attached to a core, here six cores is likely to be the maximum.

You can also see this in the CPU utilization curve. The value 100% means that all four SMT threads of all entitled – six – cores are being used. [As seen in the previous graph, you can get 100% through other means as well.

In comparing this graph to the previous, notice the difference in scale of the left-hand y-axis. Perhaps even without this observation, you will notice that the contribution due to CPU Queuing is larger in the latter curve. The same amount of work was being requested in both cases, but in the former it was getting handled by 8 virtual processors on 8 cores. Here, with the other partition equally busy, this same work is being requested to be handled by 8 virtual processors on 6 cores. Of course, the compute capacity of six cores – and so the possible throughput – is less than that of eight cores.

POWER7's Nodal Topology

In this section we are going to shift gears. We studied the meaning and control of the compute capacity of your partitioned system. Here we will be looking at how to improve the compute capacity of that same system by decreasing the latency of your program's storage accesses.

The POWER7 system design is based on chips having up to eight processor cores per chip. Each of these chips also contains one or two memory controllers, enabling some amount of memory to be directly attached to each of these processor chips. This unit – a processor chip with its cores, cache, and locally attached memory - represents the basic building block for still larger systems.

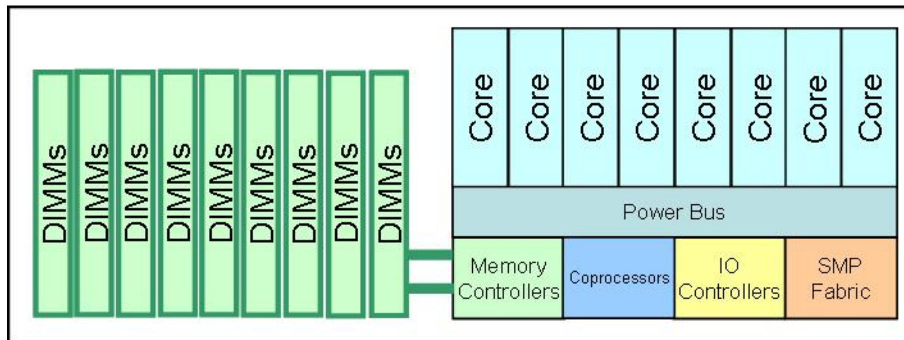

Figure 21 – POWER7 System Design

The following figure shows examples of two different POWER7-based system topologies using these building blocks, units that we also call **Nodes** (and so the reason for this section's title). Green represents the memory DIMMs, blue blocks processor chips; larger blue sub-blocks within these are memory controllers, SMP fabric controllers, and I/O controllers on each processor chip.

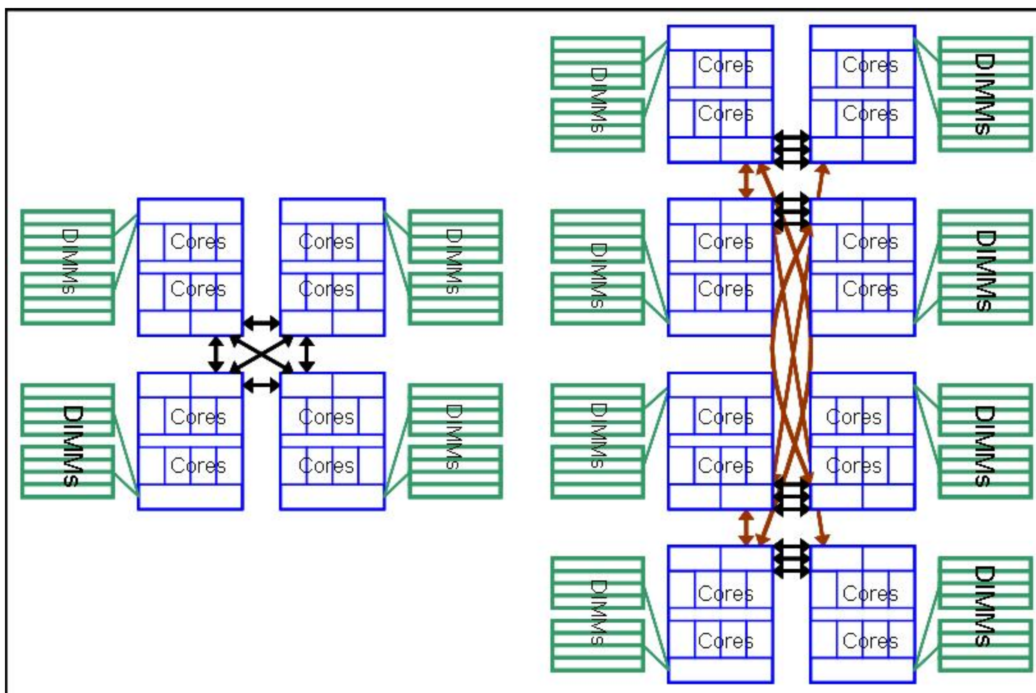

Figure 22 – POWER7 System Topology

As a result of this building block approach (and the memory controller(s) per processor chip), every additional chip also provides still more bandwidth to memory; that much bandwidth is a nice feature which provides the benefit of more rapid access time to memory.

As required by any application's storage model, every byte of memory anywhere can be accessible from any processor core throughout such systems. It does not matter whether the memory is local to the chip of some reference core or attached to another chip. Similarly, a reference core can access the contents of cache associated with any core in this system. Both of these attributes are what define a **cache-coherent Symmetric Multi-Processor (SMP)**, an attribute generally expected by most operating systems and programs executing within it.

In SMPs with topologies similar to POWER7, from the point of view of a reference core, the memory attached to the chip containing this reference core is more rapidly accessed than memory which is attached to another chip in the same SMP. The memory access time is called **memory latency**, and this latency varies depending on the relative location of the data being accessed. Similarly, the content of the cache anywhere on a reference core's chip is accessed much more rapidly than cache residing on some other processor chip.

This characteristic of differing storage access latencies is the primary attribute of a cache coherent Non-Uniform Memory Access-based (ccNUMA) topology. The best performance – and maximum compute capacity, the fewest cycles per instruction - is achieved when a core is close to the data that it is accessing.

Fortunately, the hypervisor and the partition's operating systems know about this difference in storage access latencies and do what they can to increase the probability of local memory access. The performance effect is improved response time and system compute capacity.

At its most simple, the rules to achieve this best performance are to

1. Put the work (a task) close to where its data is most likely to reside and to
2. Put the data into the memory closest to the core that is typically accessing it.
3. Arrange for tasks sharing the same data to execute on cores close to each other.

So why did we bring this up in a discussion on processor LPAR? Remember what parameters you set when describing the resources of each partition. You described

1. the number of cores (or fractional entitled capacity),
2. the amount of memory, and
3. for shared-processor partitions, the number of virtual processors.

There is not one word here – nor do we want there to be - about just where each partition's resources ought to reside in such a NUMA-based SMP. The hypervisor – knowing the physical topology of the system - takes what you describe and attempts to package those partition resources in a manner which, if not ideal, produces acceptable and repeatable performance. At its most simple, the partition's preferred cores are intended to be packaged close to the partition's physical memory.

This works smoothly in theory. The hypervisor takes the attributes you have specified for all partitions and attempts to package the partition's resource cleanly across the nodal hardware resources you saw above. Some sizes of partition memory and – for dedicated-processor partitions – core counts, are more easily ideally packaged together than others. To explain, now having seen the earlier NUMA-topology figures, suppose that you have four dedicated-processor partitions with an equal number of cores and memory. Where would you want each of those partitions to reside? You would want each partition to have their cores (and their cache) on chips close to each other and have their memory attached to such chips. Easy.

But in the real world, partition resources might have first been defined without any knowledge of the hardware's nodal resource organization. If the physical hardware of these systems had just been a set of processor cores and separately a bunch of memory, packaging of partitions resources would be a non-issue. There would be no need to know about nodal performance effects. But the topology is nodal and if best performance - and use of its maximum compute capacity - is desired, partition resource definition should take this nodal topology into account.

Picture yourself, for example, trying to fit – say - luggage into one large container; this luggage might fit exactly right and with all space used. Now try fitting the same pieces into multiple smaller containers, perhaps containers of different sizes, but all together providing the same volume. That's the job being asked of the hypervisor. The hypervisor will succeed because your containers would have fit into the single large container, but the hypervisor might need to cut them up into smaller pieces to correspond to your systems nodal resource boundaries. And, often, that is just fine, if not ideal. Occasionally, though, partition and system compute capacity is better when packaging is "nice". We will be defining "nice" below.

Even with partition resource definitions that were not NUMA-aware, the hypervisor will package all of your partitions. Because both memory and core resources as defined must be packaged, some partitions might find their:

- cores spread across chips because of their memory needs and
- memory spread across multiple chips because of their compute capacity needs.

Indeed, the inefficient packaging of one partition can influence the packaging of others.

So, now that you know that these systems are based on a NUMA topology, you might instead consider the possibility of choosing each partition's resource sizes to fit better, both individually and in groups. Once adjusted, we then still allow the hypervisor to find the efficient ways to package these. We'll be looking at some examples shortly.

As you would now expect, there exists a hypervisor algorithm to place each partition's memory and cores. The hypervisor knows of the desire for preferred placement of each partition's resources and knows of the physical resources of the target system. But before we go into the details, consider a few more items.

Although alterable, the core and memory locations are going to be quite static for dedicated-processor partitions. To minimize memory latency, we want the partition's memory close to its cores and the partition's cores close its memory. So given a static list of partitions and their desired resources,

- If the partition's core count can fit within some chip and the (remaining) memory behind that chip is sufficient to meet the partition's needs, that particular chip is going to be a good location for this partition.
- If the core count needs to span multiple chips, the hypervisor is going to try to arrange for there to be some proportional amount of the partition's memory behind those same chips as well.
- If the partition's core count could fit within a single chip but the required memory can't fit behind that chip, the memory placement will be allowed to reside behind multiple chips, so the partition's cores will attempt to be assigned from within those same chips.

All partitions want to follow these same rules, but after each partition has been assigned its required resources, it is going to consume some specific cores and memory, leaving the remainder for other partitions. It's a bit like fitting together a multi-dimension puzzle (as in the following figure). But in this puzzle, the pieces might be capable of being adjusted subtly to provide a more preferable packaging for all the pieces.

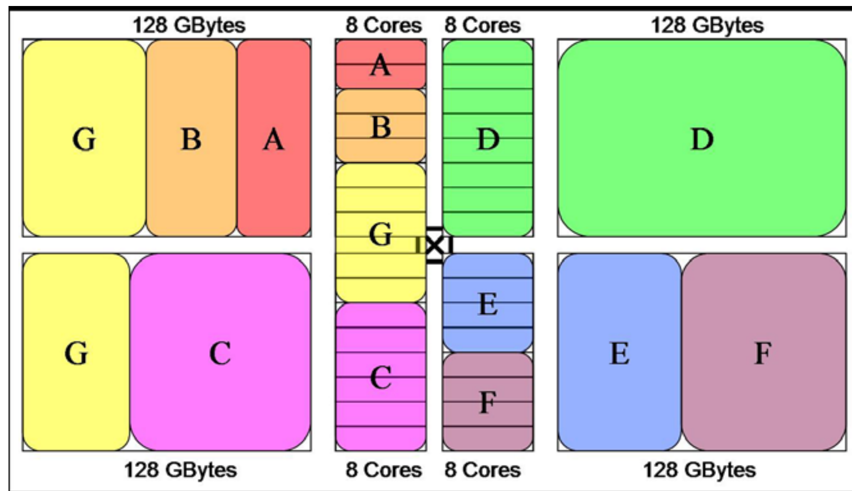

Figure 23– Partition Layout

The operating systems of the resulting dedicated-processor partitions are made aware of this placement. As a result each can further attempt to manage memory affinity for the work being executed. That is, OS kernels of these partitions have a fair amount of NUMA-awareness as well. If a partition happens to find itself with cores and memory residing on multiple chips, the partition's OS can arrange for

- the work to be close to where its data resides and
- the data to be placed where the work normally wants to execute.

To enable this on IBM i, each task is assigned a chip (a Node) upon which are the processor cores where the task would prefer to be dispatched; this preferred chip is called a "**Home Node**". When the task has a need for memory, the location of that allocated memory is best assigned from the task's Home Node. Similarly, if a set of jobs or a set of threads – say in a multi-threaded process – has had its memory assigned from some chip, the entire set of threads may perform best when allocated to cores on the chip attached to the memory.

For shared-processor partitions, the partition's physical memory location is relatively static. The defined size of memory for these partitions tends to be assigned to as few nodes as possible. It tends to anchor those partitions; we still want the partition's work to be executing close to where the partition's data will reside. Unlike dedicated-processor partitions, the shared-processor partition's virtual processors are not tied to specific cores; the partition's virtual processors can potentially execute on any core of the shared-processor pool. (The shared-processor pool consists of any activated core not used by a dedicated-processor partition.) Still, to maximize the desired locality to memory, the hypervisor makes an effort to ensure that a shared-processor partition's virtual processors get assigned even temporarily to shared-pool cores close to the partition's memory.

To enable that, the hypervisor assigns each virtual processor of each partition a "**Preferred Node**" ID based partly on the partition's memory location(s). When the hypervisor assigns virtual processors to cores, the Preferred Node expresses the virtual processor's preference for the node's cores. Additionally, this nodal preference gets used by the partition's Task Dispatcher for dispatching tasks per their own Home Node ID. Recall that, as a similar concept, a dedicated-processor partition's OS - knowing the nodal location of its cores - assigns tasks Home Node IDs, these representing the preferred set of cores on which each task wants to be dispatched. Using this Preferred Node ID, the Task Dispatcher knows also a nodal location of the virtual processors it will be using. An IBM i's task with its Home Node ID is used to select a virtual processor with a matching Preferred Node ID which is then used by the hypervisor to select a core to which to assign the virtual processor. There is no absolute guarantee that a shared-processor partition's virtual processor will be assigned a core of its preferred node, but with few enough

active virtual processors contending for those cores, the hypervisor can often succeed. When successful, it allows for the preferred local access to the partition's memory.

Affinity Groups

The notion of Affinity Groups provides you some additional control over partition placement for advanced users.

Starting with a system on which no partitions had been activated, as each partition is first activated, the hypervisor places that partition's cores and memory resources. The next partition to be activated is placed on remaining core and memory resources in a manner perceived as best for that partition, and so on. The hypervisor will use its algorithms to decide the placement of each of these partitions, and often this approach is fine. Choosing the right order – along with the right resources for each partition – is a way of controlling the placement of these resources, partition by partition.

You can, though, gain some additional control over just where each partition is packaged by grouping together the resources of sets of partitions. Unlike more traditional controls provided by – say – the HMC, this tool is provided starting with the PowerVM Firmware 730 via an HMC CLI (Command Line Interface) command. This command is **CHSYSCFG** and here its purpose is to tie a set of partitions together into one or more groups for purposes of partition placement. The groups so defined effectively identify those partitions which will be activated together at frame reboot time.

For example, suppose your system consists of 16-core drawers. Suppose further that you have a set of partitions A, B, and C that together could use all of the core and memory resources of a single drawer and you want it that way. So you define these partitions as a single group, thereby also guaranteeing that other partitions and other partition's affinity groups will be packaged in other drawers. You don't necessarily know where in the single drawer Partitions A, B, and C are going to be packaged, but you do know that they'll be packaged together. Given this constraint on A, B, and C, the hypervisor will then attempt to package these in what the hypervisor perceives as best placement within that single drawer.

The command's format (actually provided as a single line) – executed for each partition in a group - is as follows:

```
chsyscfg -r prof
         -m <system_name>
         -i "name=<profile_name>,
             lpar_name=<partition_name>,
             affinity_group_id=<group_id>"
```

The "group_id" is any number between 1 and 255. (A group_id=none removes a partition from the group.) During frame reboot, as the hypervisor places partitions, it starts with the highest numbered Group ID (e.g., 255) and works its way down. The "–r prof" is merely saying that the resource involved is partition profiles; this command is executed with the same group_id for each partition A, B, and C.

You can find more on this command at

http://www.redbooks.ibm.com/redbooks/pdfs/sg248000.pdf

or

http://pic.dhe.ibm.com/infocenter/powersys/v3r1m5/index.jsp?topic=/iphcx_p5/chsyscfg.htm

or to learn about HMC commands in general at

http://publib.boulder.ibm.com/infocenter/powersys/v3r1m5/index.jsp?topic=/p7edm/p7edm_kickoff.htm

Affinity Groups provide an advanced function for experienced users or those who plan to use them with IBM guidance. See the POWER7 Virtualization Best Practices Guide for more details:

POWER7 Virtualization Best Practices Guide

TurboCore in POWER7's Nodal Topology

Much of this paper has been written assuming there are always eight physical cores on POWER7's chips. This might simplify packaging of some partitions. But the hypervisor's partition packaging must also take into account that there are POWER7 systems with fewer physical cores per chip. Some systems also have three, four, or six cores per chip. Still others have 8 physical cores – and their caches - per chip but can be changed to allocate just 4 of them. This last 4-core chip is associated with "TurboCore" mode which provides a higher frequency and more cache per core.

To understand TurboCore, first understand that these chips need to be able to be cooled. Eight cores on a chip can produce a lot of heat. The amount of heat they produce increases faster than the increase in frequency; a linear increase in frequency results in a faster increase in heat produced. As a result, there is a maximum frequency allowed for 8-core chips based on the system's capability to cool them. Four active cores per chip, though, produce less heat than eight. Since a system with 8-core chips at their nominal frequency is capable of being cooled, and 4-core chips at that same frequency produce less heat, the 4-core TurboCore mode's cores can be run at a higher frequency at still be cooled.

Further, TurboCore is a system mode based on a system otherwise physically capable of having all eight cores and each core's caches active. TurboCore mode has only four cores executing instructions, but the cache of all eight cores remain active. Because the contents of an active core's L3 caches can be written into the now unused core's L3 caches – a notion called Lateral Cast Out - these chips effectively have twice as much L3 cache for use by the active cores. (For comparison, actual 4-core chips only have their own L3 cache.)

Higher frequency and more cache can, of course, mean better performance.

In this context of NUMA, though, there are some additional items worth observing…..

- Notice that an 8-core partition, for example, could be packaged completely within an 8-core chip. With TurboCore's four physical cores per chip, that means such a partition resides on at least two chips. So we have a bit of a trade-off here. For this 8-core partition, on one hand we have TurboCore's higher frequency, larger cache state, and the extra memory controllers of multiple chips (providing more bandwidth), but this gets traded off for some increased latency to a remote chip's memory and cache. If this partition could instead have been packaged on one chip, we would instead have strictly local access to memory and cache. Additionally, because of lateral cast-out, at lower utilization with only a few of the eight cores busy executing on the chip (with the remainder temporarily idle), we also have the perception of a larger on-chip cache as well without TurboCore.
- The proportion of memory size per core effectively doubles going from 8 cores per chip to TurboCore's 4. With twice as much memory per core attached to each chip, the assignment of partitions within an 8-core-based system will be different than for a 4-core-based system. So the hypervisor needs to know this ahead of the point in time where it begins packaging partitions. From a partition resource placement point of view, a TurboCore-based system can look quite

different from an 8-core chip-based system, so ensure that the system had been powered up in TurboCore mode prior to activating any of the partitions. It is important to also take your intent to use TurboCore into account as you define each partition's memory and core resources.

You can see a related effect for still larger partitions. For example, a 16-core partition residing on two chips in a single 16-core drawer becomes with TurboCore a 16-core partition residing on the four chips of two drawers.

This is not necessarily critical for all environments, but in the context of what we have been discussing here, it should be taken into account. Depending on usage, some of the benefits of TurboCore's higher frequency can become offset by an increased probability of access to remote memory and cache. For more on these trade-offs, refer to the TurboCore link in the Glossary.

Processor Licensing and Activation in a Nodal Topology

So far we have been assuming that all of the system's physical cores and memory will be used. It happens, though, that both cores and memory need to be activated and then licensed for use before being used. A system can have fewer licensed cores than activated cores and fewer activated cores than physical cores. (Some systems have all cores activated at the time of purchase.

The order of core activation has evolved across hypervisor releases. Early releases of the hypervisor activated the specified number of cores in what was called "core order"; how ever the cores were numbered, that was the order in which the cores were activated. You can see this core ordering in the following figure of a two-drawer / 2-chip per drawer system. Only the (**18 in red**) core-order activated cores became the cores subsequently usable as the licensed partition's cores. The remaining cores (12 in black) were simply not used.

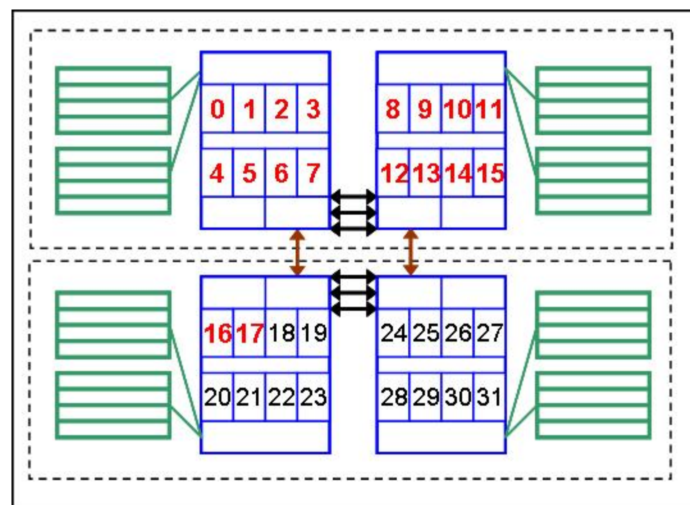

Figure 24 – Partition Resource Packaging

This approach did not always help in producing the most preferred packaging of partition resources. Core order also meant that the cores licensed for use were packed forward. This, in turn, meant that the memory to be used would also be packed forward, this done to have partition memory close to the partition's cores. This rather restricted the hypervisor's options for packaging, occasionally producing situations where there were partition cores having no local memory and partition memory where there were no local cores.

More recent hypervisor versions have altered the algorithm for determining which cores become activated. This algorithm first determines the preferred location of each dedicated-processor partition's cores and all partition type's memory. Once the dedicated-processor cores and preferred locations of the shared-processor partition's virtual processors have been identified, it is these that are first activated and licensed for use by the operating systems. As you have seen before, this newer algorithm is attempting to increase the probability that a storage access request can be satisfied by cache or memory close to the core making the request.

Note, though, that once the desired number of cores are activated, it is these particular cores that remain activated as partition resources change over time. More can, of course, be activated (if available).
More on activating of system resources can be found at:
http://www-03.ibm.com/systems/power/hardware/cod/

The Theory and Practice of Controlling Partition Placement

You have seen that partition placement in these NUMA-based systems is important for having performance-optimized partitions. So, now, just how does one go about controlling the positioning of partition resources?

We start by defining the term "**Activation**". Activation of a partition is the act of allowing the partition to know that you really do intend to use the resources defined within a partition's profile. You can have created many partition profiles, but only some of them might get activated. It is only when a profile gets activated that physical system resources are assigned. In doing so, the system tracks the relationship between the activated profiles and the assigned resources. This is true even

- after DLPAR (which does not actually alter the profile),
- when the partition is suspended – perhaps temporarily.

This partition resource information is maintained in a non-volatile storage registry within the system. Again, independent of all of the possible profiles, the system maintains a registry of the physical resources used by all of the currently activated partitions.

Let's then start with a clean slate in that registry (i.e. when a system is first delivered to a customer). The first step for most customers should be an initialize operation which deletes the IBM supplied all resource partition. Deleting this partition clears out the registry of defined partitions. As each partition is defined by the customer and activated, the hypervisor places the partition's required resources in a way which the hypervisor perceives as being best for that partition. The next partition to be activated is similarly placed, but using the core and memory resources that remain. As each is activated, its detailed resource information is preserved in this registry. This is true for both dedicated-processor and shared-processor partitions.

Recall that this initial activation of partitions and the placement of cores together have the effect of also defining the locations of some or all of the "activated" cores. After the dedicated-processor partitions cores are placed, the remaining number of cores that can be activated are assigned to the shared-processor pool. Some systems come with all cores already "activated"; others are activated by utilizing CUoD.

Performance Tip: This process of activating and placing partitions proceeds for as many partitions as you specify. The order of partition activation - starting with an empty registry - has a lot to do with the way that the partitions are placed. So choose an order of activation of partitions and groups which provides each next partition its best placement from the remaining resources, realizing that this partition is removing resources available for subsequent partitions. Think of it a bit like a puzzle, perhaps adjusting the puzzle piece sizes to allow them all to fit better during this initial process.

An observation to keep in mind here is that, once partitions are so activated, the system now has a record of all of these in its registry. Later, starting from this initial/registered state you may be making subsequent changes. Next we look at a few of these changes:

A System "Reboot" … Now with the registered knowledge of all of these activated partitions, let's assume that you need to reboot the entire system. "Reboot" here means having power cycled the CEC – powering off and then powering on the CEC – and, when that is complete, the partitions can be reactivated; the registry persists. If the partition's profile had specified auto-activation, the mere process of power cycling will result in their reactivation. But what this step is really all about is that the hypervisor will (re)define the location of all of the partitions that it knows about via the resource size description in the non-volatile registry (not the state as currently defined in the profiles). Because the registry describes multiple partitions' resource requirements, the hypervisor can find best placement for these partitions together, rather than one after the other. That is, before the partitions are actually activated during (and perhaps after) this reboot, the hypervisor has recalculated the location of their resources. Placement will be in the following order:

1. Dedicated-processor partition, typically largest to smallest.
2. Shared-processor partitions, memory and fractional entitlement for each virtual processor.

With such reboot, given the registry-based knowledge of all of the partitions, the partitions will reside in what the hypervisor perceives as a preferred placement for all of them.

If, during this reboot process, a profile of a reactivated partition is subsequently found to have been changed since the partition was previously activated (i.e., the partition's profile and registry descriptions are different), and recalling that the partition's resource locations have already been assigned per the registry,

- If a partition's profile defines fewer cores or entitled capacity than the number assigned per the registry, the hypervisor will choose to use a subset of the already allocated core of this partition. The ones chosen will be those perceived as the best packaging for this partition. The remainder will be freed up for other use, perhaps as part of the shared-processor pool.
- If the partition's profile specifies more cores, the hypervisor will attempt to first find an available (previously activated) core on a chip close to the partition's memory and place the new core(s) there. Failing there, it chooses some core(s), ideally on a chip in the same package, but it could be any activated core in the CEC.
- If the partition's profile specifies more memory, the hypervisor will attempt to allocate memory from behind a chip where the partition has a core at the moment. Failing there, it could be any available memory in the CEC.
- If both memory and cores are added, the same rules as above are attempted, but, if this is unsuccessful, the hypervisor will attempt to allocate the additional core(s) and memory together from the same chip(s). The partitions perform best when the cores have local access to memory.

The registry will be updated per the changes.

[Technical Note: Note that the powering on or off of a partition will not cause resources to be reassigned unless the profile is also altered.]

A New Profile Created and Activated … Suppose you have a system with partition resources already up and running and their state is known in the registry. You are about to add a new partition, memory for certain, but also cores for dedicated-processor partitions and virtual processors with their fractional entitled capacity for shared-processor partitions. Clearly, at the time that the profile is saved, the hypervisor knows nothing about this partition; it's just one or potentially many partition descriptions. Subsequent activation of this partition, though, requires the hypervisor to fit the new partition's resource requirements into available memory and core resources.

As with any newly activated partition, the hypervisor also wants to place this partition in a manner ideal for this partition. The only resource locations available are those which are not currently being used (or for shared-processor partition's virtual processors, "preferred") by other previously registered partitions. Notice we said "previously registered" partitions here. A previously registered Partition A may or may not be currently active at the time that the new Partition B is activated. If Partition A did happen to be suspended at this moment, the hypervisor - on the assumption that Partition A might be subsequently reactivated - attempts to place Partition B in still available memory and cores. Failing – at least partially – there, Partition B will consume the assigned resources of the now inactive Partition A and the registry will be updated as such.

Dynamic LPAR … Suppose you have a partition up and running, and having previously created a profile which allows its memory, core and/or entitled capacity resources to be increased or decreased. Let's have this partition be nicely packaged as is. Using the HMC's DLPAR controls, we are now going to alter the size of the partition's resources. As we described in previous sections, requesting a decrease of resources is often the cessation of the use of previously allocated resources, leaving the chosen resources as unused right where they are. A request for an increase means finding a location for the resource, ideally close by the core and memory resources already owned by the partition. But already used resources of this and any other partition will remain where they are.

Here, as with the resource changes described previously, the results might produce a partition which remains nicely packaged, but then again it might not.

Notice also that the resources freed by this partition might become used by a next DLPAR of this or some other partition. If this next DLPAR were by the same partition and was to request an increase in some resource and the partition had been previously nicely packaged, this is the best possible outcome; the partition may well reuse the resources that it had previously freed. Appropriate ordering of DLPAR operations with this in mind can help here. But notice that another partition's DLPAR will succeed if the needed resources exists anywhere in the CEC.

Partition Deletion … If a partition is deleted, the memory and core resources become available for use by some other partition. When a new partition is subsequently activated, because this deleted partition's resource were explicitly freed, the hypervisor now has the option of using this now freed resource along with any still unused memory and cores.

The Easy Button: Dynamic Platform Optimizer

We have shown the value of influencing the location of your partition's resources. We have also shown that partition resources sizes may change over time, so their location can as well. After a number of such changes, any partition's allocated resources can become rather fragmented. In fact, for some types of changes, the overall <u>opportunity</u> for better partition placement improves, but the system is not yet able to take advantage of it. You could re-IPL your entire system - with the result being better overall partition placement - but that is not something you would often want to do. A better solution is an "Easy Button";

this is a way to tell the system “I know that the partition resources have become rather fragmented and I want this system cleaned up; and allow the partitions to keep running while this is working.” That is what the **Dynamic Platform Optimizer (DPO)** was designed for.

DPO support, first available in Firmware level 760, reorganizes all the partition resource locations in a manner more efficient for both the individual partitions and the compute capacity of the system as a whole.

For example, consider the following two figures of a 4-chip, 32-core system with identical partition specifications in both, each partition’s core and memory represented by a single color; the first partition’s resources are rather fragmented, the second with its resources having become more nicely packaged:

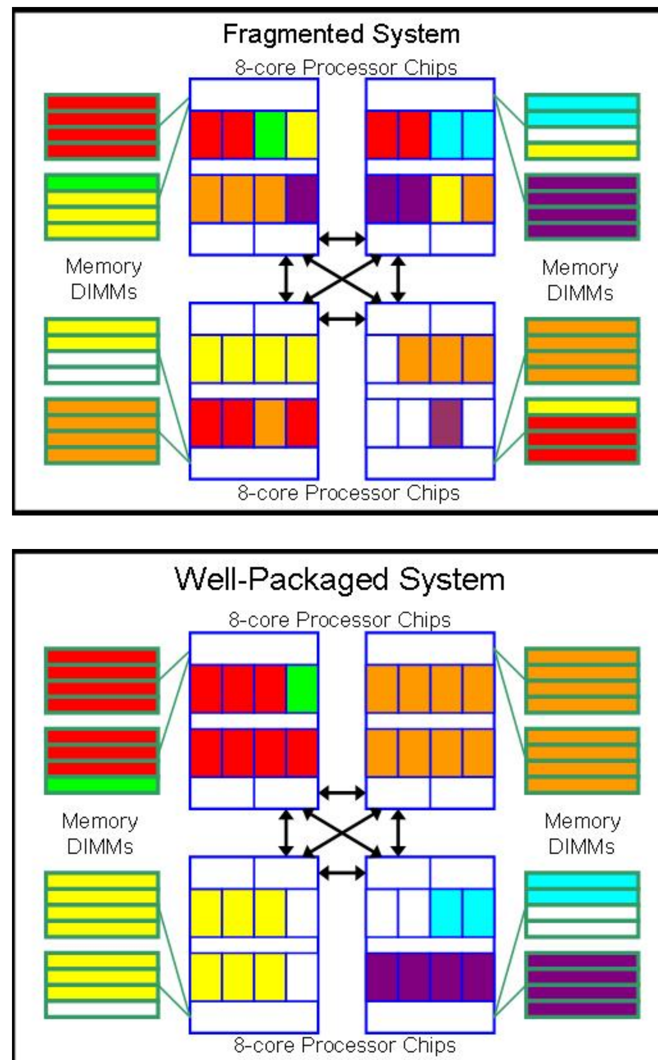

Figure 25 – Partition Fragmentation

In the fragmented system we see colored partitions with

- Partition cores and memory and cores residing on (or behind) more chips than necessary,
- Partition cores residing on chips where none of the partition’s memory resides,
- Partition memory residing behind chips where none of the partition’s cores reside.

The well-packaged system shows exactly the same partitions with exactly the same resources after the system has been optimized. The size of the partitions resources did not change here, only the resource locations. Indeed, one partition – in green - was found to not require a change in location.

During this example's optimization process, all memory available to each partition continues to be available; only its resource locations change. Part of the reason that this is possible is that this system had some amount of memory (i.e., in units of "Logical Memory Blocks", LMBs) not assigned to any partition. Much like a disk defragmentation, the contents of each LMB is copied – page by page - to an available location where the pages are subsequently made accessible again. The speed of this conversion is partly a function of the amount of memory available – memory not assigned to any partition - on each node prior to the operation. More available memory means fewer copy steps needed to achieve the desired packaging. For that matter, the ease by which an "ideal" partition layout can be accomplished is also partly a function of the amount of unallocated memory on each node, both before and after the operation. If need be, the hypervisor will temporarily use unlicensed memory in support of this operation.

The following are a few observations concerning its use:

- This operation can be initiated via the HMC's command language interface (i.e., an HMC CLI operation OPTMEM). Details on these commands will be provided later.
- There is a means to abort the operation prior to its completion.
- Subsets of partitions can be included in and excluded from this operation.
- Two "optimization scores" are available (also available via an HMC CLI):
  1. A value describing the relative fragmentation of the system's resources.
  2. A value relative to the first which describes the packaging state which could occur after an optimization operation.
- DPO on firmware level 760 requires a no-charge license code; subsequent firmware releases no longer require a license code. Systems that ship with firmware level 760 or later will have the VET code installed during manufacturing. Users installing this firmware will need to acquire the code from IBM.

Upon completion of the operation, for each partition repackaged, the partition's OS is informed of the changes. With this notification, each IBM i partition begins an autonomic process of re-optimizing the location of the work and the data that it is accessing to be consistent with the new nodal topology seen by this partition. You might have noticed that in the second figure above, every partition's core and memory resources was able to have been localized to individual chips; when this is the case, each IBM i partition changes, essentially allowing it to do nothing to manage the nodal topology of its partition since it is no longer multi-nodal. (Although preferable for smaller partitions, this might not be typical for an actual customer environment.)

Performance Tip: To take full advantage of the DPO feature, we recommend IBM i partitions be running at or post release 7.1 MF56058.

Since the contents of individual physical pages are being moved from one location to another, the time required to complete such operations can be rather long. This is a function of

- the number of LMBs that need to be moved,
- the number of times those LMBs are moved, and
- the compute capacity available to do it.

The hypervisor is executing this operation using processors cycles otherwise not used by the partitions. As a result, it is prudent to attempt such operations when processor resources are most available.

Such partition reorganization uses the size of the partition resources as they exist on the system when the operation is initiated, not as they might be preferred as stated in a currently saved profile. (This is also true when executing a full system reboot. You might recall that DLPAR operations similarly use the partition's current resource sizes, not the contents of the active profile.) If the intent is to have the post-optimized partitions also represent the state of the partition profiles, either first

- Use DLPAR's resource specification to reflect the currently saved state of the profile, or
- Deactivate and then reactivate a selected partition profile to pick up the changes reflected in the profile. (A simple restart/reboot of a partition is not enough to see a changed profile for that partition.)

Notice that, it is not necessarily prudent to attempt to "defragment" a single partition unless there are enough processor core and memory resources available on some node or node group to contain that partition.

The following represents a quick synopsis of the HMC CLI commands involved:

lsmemopt –m <system_name> -o currscore

reports the current affinity score for the entire server. The score is a number in the range of 0-100 with 0 being poor affinity and 100 being perfect affinity.

lsmemopt –m <system_name> -o calcscore

reports the potential score that could be achieved by optimizing the system with DPO.

optmem –m <system_name> -o start –t affinity

starts the optimization for all partitions on the entire server.

lsmemopt –m <system_name>

displays the status of the optimization as it progresses.

optmem –m <system_name> -o stop

ends an optimization before it has completed all the movement of processor and memory. This can result in affinity being well less optimized for some partitions that were not completed.

lsmemopt –m <system_name>

displays the status of the most recently requested optimization. If one is currently in progress, it displays an estimate of percentage completed.

The HMC command line interface provides help text for these commands. There you can also find options to explicitly request which partitions should take part in such optimization ("requested partitions"), as well as how to protect partitions from being included ("protected partitions").

Here are a few performance observations concerning the actual use of this function:

- Pages within LMBs are being copied from one location in physical memory to another. While this process is occurring, the very same page(s) might be actively being accessed. At the very least, the page(s) can't be allowed to change during this process. This is like a page having been purged from memory and then having to be paged back in again (but considerably faster); the page is temporarily not accessible by any application. This slight delay in having access to these pages might be frequently perceived, but each is unavailable for a very short period of time.
- As with many hypervisor-driven functions, the hypervisor uses otherwise unused processor cycles. Any core (or SMT hardware thread of a core) not being used by a partition is eligible for use by the hypervisor for this purpose. When such unused compute capacity exists, the compute capacity being

actively used by the partitions is not really impacted. So it is largely only when this function needs to be executed AND most cores are being used by partitions that a partition's performance might be observed as degraded (and/or that the DPO operation takes too long).

- Given a core (or more) is in use for this function and a partition then needs a processor, as with many hypervisor functions, there might be a slight delay before the hypervisor can return control to a partition. At the very least, a page being copied – having been made temporarily non-accessible – needs be again made available before the hypervisor frees up that processor.

Hypervisor Memory Requirements

It is natural to picture all of a system's memory as being consumed the partitions themselves. Normally we would like to be able to add up all of the memory requirements of all the active partitions and have that be close to the system's memory size. That mental model, though, ignores an important consumer of system memory, the hypervisor itself. Unfortunately, there is no easy rule of thumb for definitively knowing what the hypervisor's memory needs might be.

In fact, there is an optional high reliability capability on some systems wherein all of the hypervisor's data objects are automatically replicated in different portions of physical memory when written. This provides higher availability in the event of memory failures. This, of course, adds to the hypervisor's memory requirements.

So, for now, keep in mind that the hypervisor requires some portion of the memory behind each chip and add an adjustment of about **10%** of each chip's memory as potentially being consumed by the hypervisor. For more information see the IBM system planning tool:
http://www.ibm.com/systems/support/tools/systemplanningtool/
to estimate the amount of memory that will be reserved by the hypervisor.

[Technical Note: It is the hypervisor's memory which contains the structure(s) used for virtual-to-real address translation by each partition. It is called the Hardware Page Table (HPT), and having one per partition is what keeps the partition's use of physical memory isolated from each other. It happens that the size of this table tends to be proportional to the size of the maximum memory specified for a partition.]

Simple (or not so simple) DIMM Placement

You might have noticed that we have been assuming throughout this document that there are memory DIMMs installed behind each of the processor chips. We have also implied that there is equal memory behind each chip. Although much preferred for performance, neither is strictly required from a functional point of view.

You also know now that for performance reasons we want the hypervisor to place each of the partition's cores where its memory resides and to place the memory where the cores reside. Similarly, each partition's Task Dispatcher does not really want to place tasks onto cores where the partition has no memory. This is an important point. If a processor chip does not happen to have memory, the partitions then also prefer not to use that set of their cores that happen to reside there; the cores can get used (and do at higher utilization), but the partition then incurs an additional cost to access memory. As such, we recommend ensuring that all processor chips have memory in their DIMM slots if there is any current or future need for partitions to use the cores of those chips.

It happens that, by not having first considered the nodal aspects of these systems, it is actually quite easy to accidentally have processor chips lacking memory. For example, if you thought in terms wherein these systems are really just N number of cores somehow connected to M amount of memory, you would also believe that memory in one DIMM slot is as good as any other. (You know now that that is not the case.) So with such a mental model, a set of memory DIMMs get ordered; their total size was chosen to slightly exceed the memory needs of all your partitions. In doing so, a DIMM density was also chosen. The size and the density together effectively defined the number of DIMMs to be installed. With DIMM slot plugging rules that call for all DIMM slots behind chips to be filled before going onto a next, once we have exhausted the purchased DIMMs, we find that we have simply left some chips with all DIMMs slots empty. We have chips, with cores potentially assigned to partitions, which have no local memory.

The system can be quite functional just like this; to be just functional, DIMM-less chips are acceptable. But you also know that performance is better when you avoid having chips with active cores lacking locally attached DIMMs. So if your partition's core and memory resources are intended to span all of the chips of your system, you should also attempt to ensure that the system also has enough DIMMs installed to allow every chip with active cores to have locally attached memory.

Perhaps, for quite reasonable reasons, the amount of memory had been purchased and installed with memory pricing a primary consideration. Notice, though, that even memory is licensed on some systems, being paid for as needed. And, as with activated/licensed cores, the hypervisor can first define where it wants each partition's memory to reside and then activate just that memory for use.
The remaining unused – but nonetheless physically available memory – also remains unlicensed.

As a result, by having more physical memory than needs to be licensed:
1. the hypervisor has more flexibility concerning where to place each partition's memory,
2. the memory placement allows more flexibility concerning where to place the licensed cores, and
3. only the licensed memory is "paid for",
4. Until more is needed, this allows each partition's additional - previously unlicensed - memory to be potentially allocated closer to the partition's current memory.

Performance Tip: To improve the ease of partition resource allocation – this including initial placement, DLPAR, and DPO – and so to improve system and partition performance, having true balance of memory behind all processor chips and potentially having more physical memory than is really needed is recommended.

NUMA and Dynamic LPAR

As we had shown earlier, DLPAR (Dynamic LPAR) is about altering – without a partition IPL – one or more of the following resources:
- The number of cores in a dedicated-processor partition
- The number of virtual processors in a shared-processor partition
- The amount of memory for any partition, and
- A number of other partition parameters (like partition weight, for example).

For some of these, all you needed to have done is describe their maximum and minimum values and then later request the change. For example, if you wanted to have allowed for an increase in the number of cores or virtual processors, the partition's OS would have had to have been configured as part of its IPL to set up its internal structures to allow for more processors to later be activated.

You now know that, before such a DLPAR-enabled change, the partition had some number of cores and some amount of memory residing somewhere in your NUMA-based system. Ideally, the core and memory locations were also nicely packaged there. And the partitions themselves knew, for each node, how much of the partition's memory and cores resides there.

Starting from this relatively static state, let's consider a DLPAR change affecting the number of cores. Let's also start by making this DLPAR change challenging by assuming that every physical core of the system is already licensed and being accounted for within the set of dedicated-processor partitions. In order for Partition A to increase its core count, Partition B and/or others had to have first deactivated some of theirs. It is these cores that are to become activated within Partition A. So, physically, just where are those cores? And, just as important, where are they relative to the physical memory of the source and target partitions? These additional cores can certainly be made to be functional within Partition A, but from a performance point of view their location will also matter.

Suppose, next, that the number of licensed cores is fewer than the number of physical cores. The cores that are at first licensed are also cores being used by the partitions. The cores that are not licensed are essentially just unassigned cores.

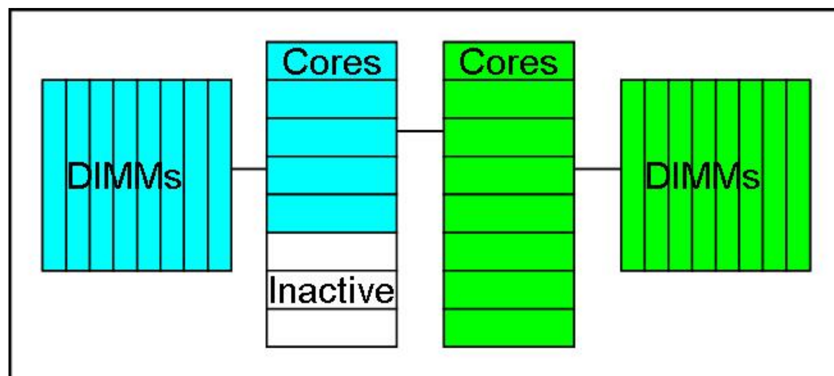

Figure 26 – Partition Layout with Inactive Cores

Let's now have Partition B drop three cores and have Partition A increase its core count by three as in the following figures (Partition A is in blue, Partition B in green). Prior to the 760 firmware, Partition A would have picked up the cores freed up by Partition B.

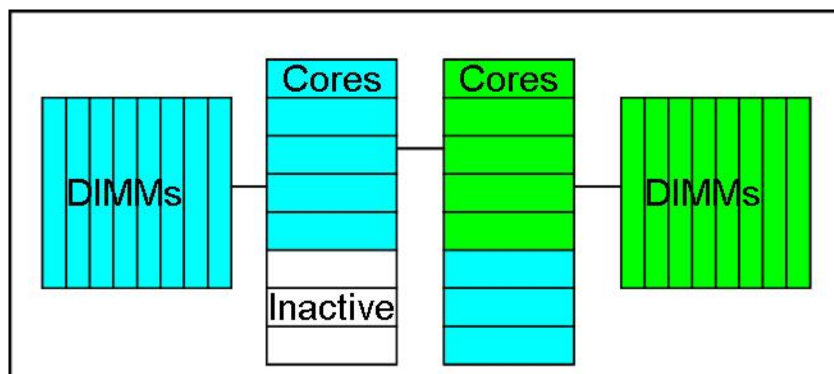

Figure 27 – Partition Layout with Inactive Cores after Processor DLPAR

Starting with 760 firmware, the location of the licensed cores changes to allow the transition to be as shown in the first and third of the following figures, allowing Partition A's new cores to be closer to its

existing core's cache and memory. The existence of the inactive cores allowed Partition A to remain packaged in a single chip.

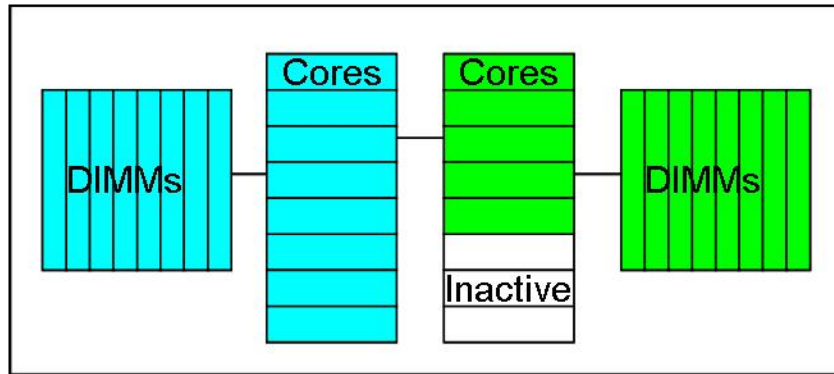

Figure 28 – Partition Layout with Inactive Cores with Transition

The Shared-Processor Pool Trade-offs and the Time Slice

We have previously described how the hypervisor tracks the compute capacity consumed by the virtual processors of shared-processor partitions within time slices defaulting to 10 milliseconds in length. This default time slice value can be changed. This section is to provide some guidance as to just what you are influencing when adjusting this time slice value.

We have noted that these systems have a complex cache and storage topology.   This design is intended to allow the core(s) to access the data and instruction streams quickly. Indeed, each core has three levels of cache of various size and speeds, the smallest – L1 data and instruction caches - also being the fastest. Cache works to speed task execution time and increase system compute capacity by maintaining the most recently accessed data in the core's caches. Of course, when the needed data is not in the core's cache, processing takes longer while a block of data is accessed from elsewhere – local or remote memory for example, in the system.

Each task – and so each virtual processor – tends to access it own data. For example, the task's program stack, although used by a lot of different routines, is really just repeatedly using the same storage. There is a state that tends to get brought into the cache – each as a distinct cache fill – and then reused.

It is worth noting that, for all of our discussion concerning NUMA effects, the notion of cache has a way of making the relative location – whether local or remote - of each partition's and each task's memory irrelevant. If even remote data can persist in a core's cache, it does not tend to matter as much that it took longer to get there.

A dedicated-processor partition's task – and all of its state - can execute on a core indefinitely as long as other tasks don't require the use of that "processor" to provide the perception of progress for all. A Capped shared-processor partition's virtual processor (and so the tasks there) can continue to execute indefinitely if its partition's compute capacity limit is never reached; no matter the number of virtual processors defined, if a capped partition with 1-core entitlement is only using one virtual processor, that virtual processor can execute non-stop. Similarly, an Uncapped partition's virtual processors need not cease execution if there is continually available cores in the shared-processor pool. These are all good things from the point of view of a cache and the associated tasks' performance.   Your task really does want to stay on a processor for as long as possible just so that its data working set can remain in the core's cache.

Tasks may want to stay where they are placed -- executing as quickly as their cached state allows them to. But you have seen that contention by the many virtual processors for a fewer number of cores in the shared-processor pool forces virtual processors to cease executing where they are and permits another virtual processor to take its place. The new virtual processor's state, though, is not necessarily in that core's cache. That state may be in memory or in the cache of another core of the shared-processor pool where the virtual processor had last executed. As that virtual processor (re)starts executing where it is, its working state is pulled into the new core's cache (and aging out the state of the preceding virtual processor). This takes time. This is a completely normal characteristic of cache(s) and cache management.

This is "normal" but from a performance point of view, it is not an optimal characteristic. Upon (re)starting execution, each new task, each new virtual processor incurs this cost of loading its state into the core's cache from wherever that state currently resides. These are just a lot of individual cache fills, each one taking time. And, although there is no well defined moment when it's over, you can think of each of these periods as taking a while, with the effect being that the work is not executing as quickly as it would have if such priming were not required. Again, the task executes fastest with the data already in the core's cache; when a task incurs all the cache fills to restore that state, processing slows down and compute capacity is reduced.

If the rate at which this virtual-processor switching occurs (and so reloading of cache) is minimal, the effect on performance is also minimal. But as this rate of switching grows, it also has a way of reducing the total system's available compute capacity. Just to provide a mental model of the possible effect, picture the next switch out as occurring shortly after the priming period has ended; this results in a lot of cache fills and a lot less work getting done.

This effect of restoring cache state is known by IBM's Workload Estimator Tool (WLE). In some ideal with no such switching (a dedicated-processor partition with a certain number of cores, for example), some number of cores is presented to you as having some set amount of compute capacity. Shared-processor partitions are known to have some lesser compute capacity due to this effect. This reduction can become a quite measurable fraction of the total compute capacity of the system. WLE presents this difference to you. But it is important to understand that this is an estimate based on measured experimentation; it can be better or worse.

Performance Tips: The trick to decreasing this shared-processor capacity reduction is to decrease the probability of such virtual processor switching. One way is to decrease the contention of virtual processors for cores in the shared-processor pool. There are a number of ways to do this:

1. Limit the number of virtual processors contending for the cores of the shared-processor pool. If all active, they are all going to get their time on a core based on their partition's entitled capacity. You can do this by decreasing each partition's virtual processors to be closer to the entitled capacity. As you know, doing so trades off the opportunity for uncapped partitions to grow significantly to use unused capacity in the shared-processor pool (when available). With knowledge of when contention is likely, you can also use DLPAR to adjust the number of active virtual processors.
2. Use of "dedicated-donate" wherein the dedicated partitions allow their cores to temporarily join the shared pool when not being used by their owning partition. Doing so has the effect of increasing the capacity of the shared-processor pool, thereby decreasing the rate of virtual processor switching due to exceeding entitlement.
3. Alter the default length of time representing the hypervisor's time slice (10 msec). Recent versions of the hypervisor allow for changing the time slice value to 50 milliseconds on some system models. This can be set through ASM (Advanced System Management) and needs to be used with care. Using a longer value could result in virtual processors executing for an extended period of time, but once a partition had reached its entitlement limit, it might not be until well into the next – now longer - time slice that that virtual processor's tasks get to execute again. Although tasks execute faster when executing, this less timely use of a processor may show through as longer I/O latencies, for example.

It is also worth noting that time slicing is not the only reason a virtual processor moves from core to core. All that is needed is for a virtual processor to be temporarily deactivated and then soon thereafter reactivated. A virtual processor becomes inactive merely because all of the tasks assigned to a virtual processor have themselves entered a wait state. For example, if the last or only one of the tasks still attached to a virtual processor happens to incur a page fault and requires a DASD I/O read, that task enters a wait state pending the completion of the needed page read. This, in turn, deactivates the virtual processor. But just as with the completion of the DASD page read making a task in a dedicated-processor partition dispatchable and so attached to a core, such a newly dispatched task needs to be associated with a virtual processor of a shared-processor partition to once again begin actual execution. This might, in turn, require the re-activation of that virtual processor. Since there is no guarantee that the core used previously by that virtual processor is the same as the core newly used, the task's cache state might need to be moved between cores. Additionally, as mentioned above, if a longer time-slice value than the default is used, this re-activation might be delayed while other virtual processors continue their execution.

Shared-processor partitions exist largely to maximize the use of the compute capacity available in the cores of the shared-processor pool. The pool's compute capacity, though, is not really a constant; it can be improved with prudent use of the cache in the cores of this pool.

Summary

In this paper we have shown you the primary POWER7 logical partition abstractions and ways to control your virtualized partitions. You should now be able to better understand these concepts and apply them to your system. Using these concepts you can build and maintain a virtualized configuration with the best possible performance.

Glossary

Access Latency: This refers to the length of time needed for a Load or a Store instruction to access its data. In as much as most storage operations outside of the processor core proper access the remainder of the storage architecture using cache-line sized blocks (128 bytes), this also refers to the length of time to complete a cache fill of a referenced block, no matter where it resides in storage.

Activated: Each SMP system has a set of physical and functional cores. Of these, all or some of these cores are purchased for subsequent use by partitions. It is these cores that are activated.

Affinity Group: This is a means of identifying partitions whose memory and processor resources should be assigned together to a location in the SMP. The reason for such grouping stems from the NUMA topology of these SMPs. Partitions of the same group tend to be assigned to the same book or drawer or even socket, based on the needed resources.

Capped: This is an attribute of a shared-processor partition relating to the use of its entitled capacity. Once – within each time slice – a partition's virtual processors have consumed their entitled capacity, the hypervisor cease their execution.

Core: The hardware entity which does the actual execution of instructions in a program. POWER7 cores can support up to four tasks, each concurrently executing their own program.

Dedicated-Donate: This is an unofficial term for the capability whereby a dedicated-processor partition allows its idle cores to be temporarily available in the shared-processor pool.

DIMM: Dual In-line Memory Module. Think of this as a unit of physical memory that can be purchased and plugged into a slot for connection to the processor chips. Some DIMMs have additional buffering for performance.

Dispatchable: One of many states in which a task can exist. Dispatchable means that a task is no longer "Waiting" for some software resource, that it can instead begin executing on a processor. If a "processor" is available when a task becomes dispatchable, that task can begin execution immediately. If not, it waits enqueued for a next processor to become available.

DLPAR: Dynamic Logical Partition. This refers to a set of operations for requesting and then immediately executing changes to the resources of a partition. Rather than changing a partition's profile explicitly and then activating a partition per that profile, DLPAR allows an already active partition to alter some resources within predefined constraints and have the changes quickly take effect.

DPO: Dynamic Partition Optimizer. A hypervisor component which reorganizes the locations of one or more partition's processor and memory resources to provide better performance for the affected partitions and the SMP system as a whole.

Entitled Capacity: Entitled Capacity is a partition's guaranteed compute capacity (for uncapped partitions) or maximum compute capacity (for capped partitions). See **Entitlement**.

Entitlement: Entitlement is the amount of compute capacity (physical cores allocation) to which the partition is guaranteed. Entitlement is used in the fair use allocation of the compute capacity of the shared-processor pool's cores. Entitlement is a partition's maximum (for capped partitions) or guaranteed (for uncapped partitions) compute capacity. The Entitlement of a partition does not change automatically when the compute capacity of a shared-processor partition changes (e.g. DLPAR).

Job Priority: An attribute of an IBM i Job (and tasks/threads within it) which is used to control job scheduling. The IBM i task dispatcher uses job priority as the main criteria to determine the next task to be dispatched to a newly available SMT Processor Thread when the OS scheduler is over-committed (when there are a set of tasks ready to execute).

Home Node: An IBM i concept in which tasks are assigned a Home Node as an attribute. This provides each task a preference for that set of core within a "Node", on POWER7 typically a processor chip. Virtual Processors of shared-processor partitions are similarly provided a Home Node, allowing the OS to match up each task's Home Node ID with that of the virtual processor. This, in turn, allows the task some assurance that the core on which it is executing is close to it the memory where its data often resides.

Hypervisor: Processor virtualization implies the support of multiple OS instances – partitions – using the same SMP. Efficient and fair use of the SMP's resources, as well as isolation of each of the partitions, requires a highly trusted level of code for managing the partition's use of the SMP's resources.

Lateral-Cast Out: A POWER7 capability associated with hardware cache management. As the data blocks are aged out of a core's L3 cache, that same data can be written into the L3 cache of a more idle core on the same processor chip. This increases the length of time that a data block can remain in cache. Without Lateral-Cast Out, the data block would have either been merely removed from the cache or – if changed – written back to main storage. Along with its slightly high core frequency, TurboCore relies on Lateral-Cast Out to effectively increase the amount of cache per POWER7 core.

Licensed: For dedicated-processor partitions, that number cores paid for use by that partition. Licensing of partitions defined as shared-processor partitions is more complex and is described elsewhere in this document. The total number of licensed cores can be fewer than the number of activated cores which can, in turn, be fewer than the number of physical cores in the SMP. Subsets of the available physical memory can similarly be licensed for use.

MSPP: Multiple Shared-Partition Pools. This function provides the capability for processors resources to be allocated to more than a single specifically configured shared-processor pool. A subset of the shared-processor partitions may be allocated to a given shared-processor pool and the remainder of the shared-processor partitions can be allocated to one or more other shared-processor pool(s). MSPP technology provides a mechanism to define pools of processors supporting different licensed software. Within this shared processor pool all the virtual servers can be uncapped, allowing them flexibility within the license boundary set by the MSPP.

NUMA: Non-Uniform Memory Access. Also **ccNUMA.** Cache-Coherent NUMA.
As the name implies, from the point of view of some reference core, the latency to access the contents of some memory is faster than to other memory. This effect falls out naturally from systems with multiple processor chips – each having multiple cores – and each chip having some amount of system memory directly attached to each chip. It is possible for a partition with resources residing completely on a single chip of a multi-chip system to not perceive this effect.
The POWER7 processors support ccNUMA. Any core of the NUMA SMP can access the contents of all memory and any core's cache; the hardware-supported cache coherency allows most software to execute as though the cache did not exist.

OS: Operating System. This is that set of software which abstracts the SMP's hardware and IO for higher levels of software. With LPAR support, this abstraction is further enabled by the hypervisor. A "partition" is also thought of as an OS instance.

Partition Weighting: A relative partition attribute – set via the HMC – which influences the hypervisor's decision of which of a set of virtual processors ought to be assigned to a next available core. Under some circumstances, uncapped partitions with greater weight increase their probability of being dispatched to a core versus virtual processors of partitions with lower weight.

Processor: A now largely generic term representing the entity used to execute a task's instruction stream. When referring to the hardware, the currently preferred term is "Core". See also "Virtual Processor".

Processor Thread: This is an abstraction representing that resource to which a task is assigned to by an OS' Task Dispatcher within an SMT-capable core. Prior to SMT, this would simply have been called a Processor. See SMT below.

Processing Units: A compute capacity metric that is used in defining the entitled capacity of a partition. For shared-processor partitions is stated in terms of the compute capacity of a core. It can take on values with precision stated in terms of hundredths of the compute capacity of one or multiple cores.

Reserve Capacity: Unallocated activated cores – those that are not assigned to dedicated partitions nor included in the total shared processor entitlement allocation. This reserve capacity may be used for uncapped processing needs.

Shared-Processor Partition: A partition whose virtual processors contend for use of the cores of the shared-processor pool. As a result, each of their virtual processors can be assigned temporarily to any one of these cores. This flexibility also allows for these partitions to be limited to compute capacity which is arbitrary fractions of compute capacity otherwise available in some integer number of cores.

Shared-Processor Pool: The set of an SMP's active cores which are not associated with any dedicated-processor partition. Any of these cores can be used to execute the thread(s) of any Shared-Processor Partition's active virtual processors.

SMP: Symmetric Multi-Processor. A set of processor cores, each typically supporting the same instruction set architecture, with access to a common memory represented by a single real address space. All cores, using this single real address space, have access to the contents of all

of the memory. The cores typically have caches holding the contents of this memory and which are managed by SMP hardware in a cache-coherent fashion; this cache-coherency allows most software to execute as though the cache was not present.

SMT: Simultaneous Multi-Threading. The hardware capability in which the instruction streams of multiple tasks can be executed concurrently on the same core. More on SMT can be found at: http://www-03.ibm.com/systems/resources/pwrsysperf_SMT4OnP7.pdf

Task: An operating system concept describing a piece of work. Think of this most simply as describing the location of the instruction stream to be executed next, along with processor register state to be used by that instruction stream. It is this instruction stream address and this register state which gets loaded onto an SMT hardware thread by the OS' Task Dispatcher upon "Task Switch In" and which is saved in memory as the task is switched out.

Task Dispatcher: The operating system component which assigns tasks to virtual processors. On POWER7-based systems, the virtual processor supports up to four tasks.

TurboCore: A capability in which fewer than the available physical cores of a POWER7 chip are enabled, thereby decreasing the chips power consumption. As a result, the active core's frequency can be increased. Further, the cache of the unused cores remains available, allowing it to be used by the active cores. Both can improve the performance of such individual cores. More on TurboCore can be found at:
http://www-03.ibm.com/systems/resources/systems_i_pwrsysperf_turbocore.pdf

Time Slice: A period of time within which the hypervisor tracks the usage of each shared-processor partition's usage of its entitled capacity. It defaults to 10 milliseconds. In the event that a partition's entitled capacity is exceeded within a time slice, the partition's virtual processor(s) might cease execution until the next time slice boundary.

Uncapped: This is an attribute of a shared-processor partition relating to the use of its entitled capacity. Once – within each time slice – that an uncapped Partition A's virtual processors have reached their entitled capacity, AND if there are active virtual processors of other partitions which have not reached their compute capacity limit, the hypervisor ceases the execution of one or more of Partition A's virtual processors. Partition A's virtual processors can continue executing even if its entitled capacity has been exceeded if there are then unused cores in the shared-processor pool.

Utility Capacity on Demand (CoD): A licensing solution to allow assignment of inactive processors to the shared-processor pool on a temporary, as-needed basis. The extra capacity is activated and paid for only when the peak workload requires extra processing power. This extra capacity is available for use by uncapped partitions.

Virtual Processor: An abstracted processor core. The view that an OS' Task Dispatcher has of a processor core in a multi-partition system. In a single-partition system, often one without a hypervisor, the OS' Task Dispatcher can be thought of as assigning tasks to physical cores rather than to Virtual Processors.

References

Utility CoD Redpaper – "Utility Capacity on Demand: What Utility CoD Is and How to Use It":
http://www.redbooks.ibm.com/redpapers/pdfs/redp4416.pdf

Virtualization Redbook – "Logical Partitions on System i5 – A Guide to Planning and Configuring LPAR with HMC on System i":

http://www.redbooks.ibm.com/redbooks/pdfs/sg248000.pdf

Power Systems: Capacity on Demand:
http://pic.dhe.ibm.com/infocenter/powersys/v3r1m5/topic/p7ha2/p7ha2.pdf

IBM System Planning Tool for POWER processor-based systems:
http://www.ibm.com/systems/support/tools/systemplanningtool/

POWER7 Virtualization Best Practices Guide:
POWER7 Virtualization Best Practices Guide

The Power Architecture and Power.org wordmarks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

UNIX is a registered trademark of The Open Group in the United States, other countries or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. In the United States and/or other countries.

TPC-C and TPC-H are trademarks of the Transaction Performance Processing Council (TPPC).

SPECint, SPECfp, SPECjbb, SPECweb, SPECjAppServer, SPEC OMP, SPECviewperf, SPECapc, SPEChpc, SPECjvm, SPECmail, SPECimap and SPECsfs are trademarks of the Standard Performance Evaluation Corporation (SPEC).

InfiniBand, InfiniBand Trade Association and the InfiniBand design marks are trademarks and/or service marks of the InfiniBand Trade Association.

© IBM Corporation 2013
IBM Corporation
Systems and Technology Group
Route 100
Somers, New York 10589

Produced in the United States of America
February 2013
All Rights Reserved

This document was developed for products and/or services offered in the United States. IBM may not offer the products, features, or services discussed in this document in other countries.

The information may be subject to change without notice. Consult your local IBM business contact for information on the products, features and services available in your area.

All statements regarding IBM future directions and intent are subject to change or withdrawal without notice and represent goals and objectives only.

IBM, the IBM logo, ibm.com, AIX, Power Systems, POWER5, POWER5+, POWER6, POWER6+, POWER7, TurboCore and Active Memory are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml

Other company, product, and service names may be trademarks or service marks of others.

IBM hardware products are manufactured from new parts, or new and used parts. In some cases, the hardware product may not be new and may have been previously installed. Regardless, our warranty terms apply.

Photographs show engineering and design models. Changes may be incorporated in production models.

Copying or downloading the images contained in this document is expressly prohibited without the written consent of IBM.

This equipment is subject to FCC rules. It will comply with the appropriate FCC rules before final delivery to the buyer.

Information concerning non-IBM products was obtained from the suppliers of these products or other public sources. Questions on the capabilities of the non-IBM products should be addressed with those suppliers.

All performance information was determined in a controlled environment. Actual results may vary. Performance information is provided “AS IS” and no warranties or guarantees are expressed or implied by IBM. Buyers should consult other sources of information, including system benchmarks, to evaluate the performance of a system they are considering buying.

When referring to storage capacity, 1 TB equals total GB divided by 1000; accessible capacity may be less.

The IBM home page on the Internet can be found at: http://www.ibm.com.

A full list of U.S. trademarks owned by IBM may be found at: http://www.ibm.com/legal/copytrade.shtml.

The IBM Power Systems home page on the Internet can be found at: http://www.ibm.com/systems/power/