



Enhanced eServer iSeries Performance

Memory Affinity



Mark Funk
iSeries Systems Performance
July 2004

For the latest updates and for the latest on iSeries performance information, please refer to the Performance Management Website: <http://www.ibm.com/eserver/iseries/perfmgmt>.

Table of Contents

Memory Affinity Boosts Performance	2
Processor Topology	3
The Node's Role In Performance	4
Memory Affinity	4
Memory Affinity Controls	6
Thread Resource Affinity Level Setting	6
Thread Resource Affinity Group Setting	6
Program Control of Memory Affinity attributes	7
Thread Resources Adjustment - Memory Affinity	
Tuner - QTHDRSCADJ	7
Using the Tuning Knobs	8
Other Considerations	10
Disclaimer - System Performance	12
Trademarks	13

Memory Affinity Boosts Performance

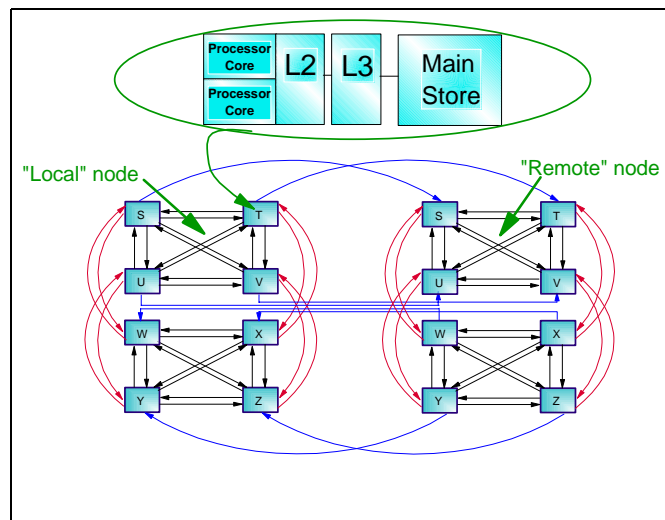
With the dramatic improvements in processor speed and processor optimizations, storage access latencies continue to be a limiting factor in fully utilizing available processing power. Therefore it is important for the Operating System (OS) to provide memory management support to fully optimize processor utilization. IBM i5/OS™ V5R3 takes advantage of a characteristic of the POWER4 and POWER5 Symmetric Multiprocessor (SMP) topology to boost performance. This base OS facility called Memory Affinity controls both how tasks are dispatched to processors and how storage is allocated on their behalf. But further performance may also be available through the use of a provided set of work management controls.

Using these controls where appropriate takes minimal effort, but it comes at a cost; you first need to understand a few basics about cache and multiprocessor topology in POWER4 and POWER5 processors.

Processor Topology

The POWER4 and POWER5 processors are organized within a set of 8-processor MCMs (Multi-Chip Modules) or 2-processor DCMs (Dual-Chip Modules). Within an MCM - see figure below - there are four processor chips, each chip supporting two physical¹ processors. Both processors on a chip share the same Level 2 (L2) cache, but each processor has separate Level 1 (L1) instruction and data caches. A processor executes fastest when the accessed data and instruction stream are in the L1 cache. In the event that the needed data or instruction stream is not in the L1 cache, the processor effectively waits for a number of cycles while a block of storage is loaded into an L1 cache from the L2 cache. The L2 cache is considerably larger than the L1 cache. These systems also support a Level 3 (L3) cache, an equal portion of which can be thought of as being associated with each processor chip and MCM. The L3 cache is considerably larger than each L2 cache, can feed the L2s in the event of an L2 cache miss, and has a considerably longer access latency. The contents of the L2 and L3 caches are called store-in caches since a change to a block of storage can be held in either indefinitely before being finally written back out to main storage. Although by no means precise, think ~10-20 cycles to fill the L1 cache from the processor's L2 cache, ~100 cycles to fill from the processor's L3 cache, and a few hundred to complete an access of main storage. Clearly, a program's performance is largely dictated by the processor's ability to hold data and instruction streams in its cache.

The contents of all SMP caches is called "coherent". This means that a changed block of storage² residing in one processor's cache is visible to storage accesses by all other processors. For example, when a processor "A" executes a read from the block of storage changed by processor "B", it can access the changed block by accessing the cache storage of processor "B", even if "A" and



¹ POWER5 supports Simultaneous Multithreading for which there are two "logical" processors per physical processor, enabling twice as many concurrently executing tasks as there are physical processors.

² L1 and L2 cache lines hold the contents of 128-byte blocks of main storage aligned on 128-byte boundaries.

“B” are in a different MCM/DCM. An unchanged block of storage can be held in the cache of multiple processors, but upon a request to change that block, the block is effectively removed from all those caches and held only in the cache of the processor making the change. It helps to think of these blocks of storage as data packets being rapidly pulled and copied from processor cache to processor cache.

The Node’s Role In Performance

Recall the cache and multiprocessor topology of this system. Each block (marked S,T,U,V in the earlier figure) represents two processors, cache, and some fraction of system main storage. It is considerably faster for a changed block of storage residing in one processor’s L2 cache to be filled into the cache of a processor on the same module (i.e., MCM) than it is to request and fill the changed block of storage into a processor’s cache in another module. It is also considerably faster to fill a processor’s cache with a block of storage from an L3 cache associated with the same module than it is to pull the block of storage from another module’s L3 cache, although the percentage difference is not as great as with the L2 cache. Because of this difference, we’ll be using the term “**local**” to describe those cache fills satisfied from within a module and the term “**remote**” for cache fills satisfied from another module. So, said differently, the latency of a storage access satisfied from local cache is measurably faster than the same access satisfied from remote cache.

So far we’ve described a topology where there exists a set of processors, each with their own L1 caches, a pair of processors with their own larger on-chip L2, and L3 caches all within a module. Each module, indeed each processor, also has associated with it a number of main storage cards. Even so, every processor on any module can read any portion of main storage in the system. Similarly, a changed storage block in any processor’s cache can be returned to the portion of a module’s main storage which owns that block. As with the cache accesses, a processor’s access of main storage local to a processor is faster than an access of remote main storage. The difference in access times between local and remote main storage accesses is moderate³, but quite measurable. For this reason, we define another term “**node**”. A node is that set of processors, their caches, and main storage cards which are associated with a single module. An access by a processor (i.e., a cache fill by that processor), satisfied by the contents of any cache or main storage on the same node is called an access from a “**local node**”. It then follows that an access from a “**remote node**” is an access satisfied by the cache or main storage associated with a different node.

Memory Affinity

It follows from all of this that if a task/thread executing on a processor is able to have its cache fills satisfied by local cache or local main storage, it will execute faster than if the cache fills were satisfied from a similar but remote cache or main storage. It is from this that the name “Memory Affinity” is derived. V5R3 attempts to maximize the probability of these local accesses. Simply stated, **the OS attempts to allocate storage from main storage local to the processor on which a task is executing and attempts to dispatch a task onto a processor local to where it typically allocates storage**. In order to do this, every task thread is assigned a “Home Node”. Every time that a task again becomes dispatchable, the task dispatcher attempts to assign that task to a processor on its Home Node.

³ You might be familiar with the term NUMA or CC-NUMA; it stands for Cache Coherent Non-Uniform Memory Access. In a sense, multi-module POWER4 and POWER5 systems fall into this category. Key to performance of such systems is the difference in latency between local and remote main storage. POWER4 and POWER5 relative latencies are sufficiently moderate that a system could be used quite efficiently while ignoring this effect. However, the point of this discussion is to suggest that system capacity and application response time can be enhanced by taking into account these latencies.

So not only does a thread executing on its Home Node often find that its storage is in local main storage, but some of its working state is still often somewhere within the cache of the local node. For instance, consider just a thread's stack, its automatic storage. Without Memory Affinity, that thread could execute on one node and later be dispatched to a processor on another node. Its stack, though, can still be represented by a set of changed cache lines on the original node. Having the thread execute on a processor of a different node means that this changed stack state must be transferred across nodal boundaries one block at a time. But if the current processor is on the same node as the previous processor, this transfer would occur more quickly with the result being that the task executes in less time. And, of course, this notion applies to any data and instruction stream actively being accessed by this thread. This point of view shows that limiting this thread to only run on the Home Node seems like an obvious win. And for a large set of situations, this Memory Affinity support really does boost performance.

But it's never quite that simple. Suppose that a task becomes dispatchable and it is the next task to be given a processor. If a processor of that task's Home Node is available at that moment, then it is clear that this processor should be assigned to that task. But suppose instead that the only available processor is one which is not on that task's Home Node. Should the task immediately begin executing there? If not, how long should that task wait for a Home Node processor to become available? Or what if another task with a different Home Node shortly becomes dispatchable and it would have consumed that processor? Or, to make matters worse, what if all of the dispatchable tasks share the same Home Node and they are all waiting for any processor from a given node? Each of these tasks could execute more quickly if limited to their Home Node, but the throughput of all of the tasks taken together might suffer. Indeed, the key word here is "might" since the added cache traffic and additional access latency of dispatching the threads off their Home Node can degrade the total throughput.

Similarly, recall that the OS tries to allocate main storage from a task's Home Node. If main storage is available - say having had its contents previously written out to disk and not likely to be used again - then it seems clear that this available local main storage ought to be allocated for use by this task. But what if such local storage is not available and is, in fact, being actively accessed? If there is storage easily available on a remote node, is it better to displace that or to use an active local page, potentially increasing the paging rate?

What we are talking about here is a tradeoff between balanced use of processors and main storage across the modules versus the advantages of faster storage access times resulting from local access. A single thread taken alone is always better off being limited to a single node. A set of 8 threads heavily sharing data are always better off running on a single node (of 8 processors) rather than spreading these threads over multiple nodes if that is all that the system is typically doing. Indeed, if the system's usage is really a set of multithreaded jobs, each utilizing the processor(s) equally, arranging for the multiple threads of each job to share the same Home Node would seem to produce the best throughput. But when there are more dispatchable threads than available local processors and over-utilization of a single node seems likely to continue, balancing the threads over multiple nodes would seem the correct solution⁴. There are many such situations where Memory Affinity will provide improved performance and also other situations where performance may be impacted. It is for this reason that V5R3 offers a set of controls to tune its behavior.

⁴ For all of the advantages of strictly local access, at relatively high utilization we typically recommend relatively balanced use of the nodes, since Local vs. Remote latencies are sufficiently close. Relative latencies in future systems may alter this recommendation.

Memory Affinity Controls

V5R3 provides the capability to configure how Memory Affinity is managed. Basically this is accomplished in two ways. One option is to specify the degree to which the system tries to maintain the affinity between threads and the system resources of their Home Node. Another option allows specifying groups of threads sharing the same Home Node, be they the threads of multithreaded jobs or the threads of multiple jobs. These options are controlled OS-wide by the settings of the system value QTHDRSCAFN or on a job-by-job basis by the RSCAFNGRP (Resource Affinity Group) parameter in subsystem routing entries. This section will address the concepts of setting these values.

Thread Resource Affinity Level Setting

Recall that every thread has a Home Node, first assigned when the thread is created. By default, the system value, QTHDRSCAFN, has the affinity level set to *NORMAL. The *NORMAL affinity tells the task dispatcher to first attempt to dispatch a thread to its Home Node. If no processor is available on its Home Node, the availability of processors on remote nodes is checked. If there is a processor available on a remote node, the thread is dispatched immediately to the remote node. Similarly, if multiple tasks are waiting for an available processor, when one becomes available, preference is given to the task where the available processor would be on the task's local Home Node. Aside from that, any task with *NORMAL affinity can be chosen to execute on potentially any remote node.

Affinity of *NORMAL also has meaning to main storage allocation but it is less precise. When a thread needs main storage allocated, if obviously available or sufficiently aged pages exist in the main storage of the Home Node, the required storage will be allocated locally. If not, remote storage will be similarly checked for available pages, and if successful, will be allocated from there. If remote storage is also not available, pages from local main storage will be found through more aggressive means. Again, this is the default behavior and its use will tend to guarantee relatively even usage of processor and main storage resources. It also produces considerably better performance than treating all processors and main storage as equal.

The other option for affinity level is *HIGH affinity. In this case, the task dispatcher attempts to assure that a thread with *HIGH affinity is only run on its Home Node, even if a processor on another node is available. In effect, *HIGH suggests that better performance is expected if this task waits for an available processor on its Home Node - on the theory that a local processor will become available soon - rather than dispatching it immediately on a remote node. In a balanced system, an available remote processor will soon be allocated to a task in its local Home Node, making this delay moot. The *HIGH affinity level is also used by main storage allocation to suggest that more aggressive means be used to allocate storage from the Home Node.

The affinity level attribute can also be set for jobs on a job level, and will take precedence over the system value. This is accomplished by specifying the affinity level value in a routing entry. An affinity level of *SYSVAL on a job says to use the system value set at either the *NORMAL (again default) or *HIGH values.

Thread Resource Affinity Group Setting

It may be the case that the threads of a multithreaded job or even the threads of multiple jobs are accessing some common resource. Perhaps it is a database, or, in the case of the multithreaded jobs,

perhaps it is a common communications area in static storage. If these threads would benefit from having shared resources always be local to a node, then they could make use of support for grouping threads.

The concept of grouping threads to share the same node is the same whether those threads are from a multithreaded job or multiple jobs, but the means of achieving it is somewhat different. In the case of multithreaded jobs, it is possible to indicate either on a job basis (via a routing entry or prestart job entry) or system-wide (by using the QTHDRSCAFN system value) to control whether the multiple threads of a job should share the same Home Node, or whether each should be assigned an arbitrary Home Node. In the case of the common Home Node, the first thread of a job is assigned a Home Node and all subsequent threads share the same Home Node.

To enable the group jobs to share a Home Node, you have to create a resource affinity group via a routing entry or prestarted job entry by specifying RSCAFNGRP = *YES. Then all threads of all jobs in the group are assigned the same Home Node. It is when the first thread of all of these jobs is created that the system assigns the actual Home Node value.

The concept of thread grouping (*GROUP or *NOGROUP) distinct from affinity level. A multithreaded job, or the threads of multiple jobs, can have either *NORMAL or *HIGH affinity.

Program Control of Memory Affinity attributes

The system value and job attributes described above allow a level of control without direct knowledge of the number of nodes in the system. Where you want even tighter control, V5R3 offers limited API programming support to control what physical node a thread can be dispatched to. This requires the knowledge of Machine Interface (MI) programming. Basically, the Materialize Process Attribute (MATPRATR) MI instruction can be used to retrieve the node ID that a particular thread is executing on. Then the SPAWN programming API has been enhanced such that you can specify the node ID that a thread should be dispatched to. For more specifics you should refer to the V5R3 MI specification.

Thread Resources Adjustment - Memory Affinity Tuner - QTHDRSCADJ

Realizing that jobs come and go over time, it is possible that the initial settings of each thread's Home Node might result in a mis-balanced use of processor and main storage resources. For this reason, the system watches for balance and reassigns the Home Node of some threads to return resource utilization to balance. Jobs and threads which have been logically grouped together by specifying *GROUP on the QTHDRSCAFN system value, or via routing entries, will be kept together as a group if they are moved. The determination of a mis-balanced state and the actual movement is not done rashly, instead some time passes before a system is deemed to be mis-balanced. Even so, a system value is provided (QTHDRSCADJ) which specifies whether or not the system should make adjustments to the affinity of threads currently running in the system. By default the QTHDRSCADJ system value is set to automatic adjustment. Although this action is the default behavior, if you wanted to "freeze" the location of all threads (i.e. disable this tuner), you set the QTHDRSCADJ system value to not dynamically adjust the threads.

Using the Tuning Knobs

So how could these new tuning knobs be used? In the examples below, various scenarios will be described along with what the associated affinity controls should be set to. It is important to note that when using the system value QTHDRSCAFN, or routing entry support, for specifying Memory Affinity settings, this can all be done without changing any applications. Therefore, in order to best understand the advantages of Memory Affinity settings, you can try various combinations to assess what provides optimum performance.

Suppose that your system supports a set of HTTP. servers. The core of each HTTP. server tends to be a multithreaded job. The initial number of threads per server is as you specify but the actual number is dependent upon the workload seen by this server. To make things simple, suppose that the number of HTTP. servers is four and the hardware has four nodes. Suppose also that they tend to equally utilize CPU and main storage resources. This is clearly an opportunity of having each job's multiple threads all share the same Home Node, *GROUP, as well as for *HIGH affinity. Almost by definition, we have a balanced system.

Now suppose that these HT. threads pass their database accesses off to another set of jobs. Given the nodality of the HT. servers, how should these database jobs use memory affinity? One option is just to do nothing and allow the default (*NORMAL) affinity with no job grouping to spread these database jobs across all of the nodes to ensure balance. Another is to associate pools of these database jobs with each HT. server, realizing that there is communications going on between them. If this is preferred, you can group these jobs into the same logical node as the HT. server. However, given that these database jobs might all be accessing the same database tables and associated indexes, it might be preferable to group these jobs independent of the HT. server. If balance can be maintained and is desirable, best performance might result by having all of the database jobs share the same node with the result being that the database itself resides in the storage of that node.

We need to point out again that Home Node value of a group of threads is decided at the time that the first of these threads are initially created on the system. This Home Node decision is based upon the relative nodal workload. For instance, if one node is clearly being more heavily utilized than another, then it is much more likely that a new thread will not be assigned to the heavily utilized node.

Also in the realm of client-server computing, consider a set of server jobs - perhaps multithreaded - which are using *HIGH affinity and have balanced utilization over the system's available nodes. Suppose that this environment has a single control job/thread which manages - say - socket connections. Balance is maintained as each connection is made by having the control thread pass the connection to a server thread of a particular node, choosing that node in a round-robin fashion.

Next consider a transaction processing system accessing a set of database tables and associated indexes. Suppose that there is a relationship between some number of threads and a subset of the tables such that these threads tend to more heavily access the tables. An example might be the set of all Order Entry users accessing a common set of customers. The database tables and indexes need to reside in main storage somewhere, they might as well reside in the storage of the node where they are most often accessed. From the thread's point of view, if the database tables exist in one node, because their pages were already brought into storage by another thread, the threads will execute most efficiently when they execute on the processors of the same node as the location of the database tables. In this case you would use *GROUP and *HIGH affinity settings. Conversely, database tables accessed by all threads of the

system do not in themselves suggest any advantages to memory affinity, so you would take the defaults, *NORMAL and *NOGROUP for these types of jobs.

As a last example, consider Java. Java requires a JVM, a multithreaded environment within a single job. If your Java application must be associated with a single JVM (and that this is the only use of the system), then the multiple threads of the JVM may be better served by not grouping (*NOGROUP) these threads on a single node. Instead, as these threads are created, each is given a Home Node, theoretically being evenly spread over the available nodes. If the threads are evenly spread and utilized, setting these threads to have *HIGH affinity assures that at least a portion of their active storage will be found in a local cache. But the common heap storage used to hold the JVM's Java objects will essentially be shared by all of the JVM's threads and so have no real locality. Therefore, thread grouping should not be used (*NOGROUP). However, if multiple applications are spread over multiple JVMs, it may then be feasible to group the threads of a each JVM to a single node, essentially allowing all of its storage and threads to use a single node. Again, this is all a tradeoff between balanced use of the systems resources versus better performance of individual threads from assuring local access.

While the above examples illustrate a variety of workloads that may benefit from the combination of i5/OS and POWER5 Memory Affinity functionality, thorough benchmarking of production workloads is recommended.

Other Considerations

There are also other architectural features which effect the benefit of Memory Affinity that need to be understood, but a complete discussion is beyond the context of this whitepaper. Other whitepapers are being developed to address these effects in more detail.

LPAR Effects

The notion of memory affinity described up to now is associated with a single operating system. It is only when a single partition uses the processors of multiple nodes that memory affinity becomes an issue. However, many larger systems are partitioned via LPAR. Even in a multi-node system, if a partition is given processors only from a single node, memory affinity is not an issue for that partition. For that reason, LPAR support attempts to allocate processors to partitions from a single node. The hypervisor on POWER5 systems also attempts to allocate main storage to these partitions only from the same nodes. (Again, for this reason, since you don't have any direct control over particular nodes, main storage cards should be equally assigned amongst the processors.) As a result some care in specifying the amount of main storage associated with each partition should be taken to assure that strictly local main storage is allocated if possible.

As a separate issue, the pool of LPAR's "shared" processors can span multiple nodes. Even when this occurs, if a partition is known to use "shared" processors, even the default form of memory affinity outlined earlier is unsupported.

Also note, due to capacity upgrade on demand , a partition consisting strictly of processors from a single node can later be upgraded from the pool of available processors. If partitions are configured to minimize the number of nodes, available processors might be found on one or more separate nodes. So, as an example, a 4-processor single node partition can become a 6-processor multi-node system when that partition is upgraded from the pool of available processors.

Subsystem Storage Pools

V5R3 has no notion of a nodal pool (i.e. a pool whose storage is bound only to a single node). The storage associated with a pool is allocated in a balanced manner from each node's portion of main storage. If each node and each processor has an equal amount of main storage, as is strongly suggested by IBM, then each pool gets an equal amount of main storage on each node. Now compare this to memory affinity's use of storage, especially when the system or set of jobs use *HIGH affinity. Memory affinity attempts to allocate storage out of a task's Home Node's storage. And in terms of pools, this means that memory affinity will attempt to allocate memory out of the portion of a pool residing on a task's local node. So, as an example, if a single pool is used by a single subsystems and all the jobs of this subsystem are also grouped to have a single Home Node, all of these jobs will first attempt to allocate storage from the portion of the pool on a single node.

Simultaneous Multithreading (SMT)

From one point of view, SMT doubles the number of dispatchable locations by implementing two virtual processors for each physical processor. SMT by no means doubles the capacity of the system, but it does double the number of locations to which a task can be dispatched, allowing a task to begin execution earlier rather than having to wait to an available processor. It therefore can also be thought of as doubling the number of “local” processors on a node. This would seem to significantly increase the probability of a newly dispatched task of finding an available local processor. But SMT doubles the number of “logical” processors, not physical processors. And a task running alone on a physical processor runs faster than when it shares a physical processor with another task. So the natural question is, “Is it better to be dispatched to an available logical processor on a local node or to a completely available physical processor on a remote node?” V5R3 design gives precedence to running a task on its Home Node. For a more complete discussion of SMT, please refer to the SMT Whitepaper at the following website:<http://www.ibm.com/servers/eserver/series/perfmgmt/pdf/SMT.pdf>.

Disclaimer - System Performance

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Information is provided "AS IS" without warranty of any kind. Mention or reference to non-IBM products is for informational purposes only and does not constitute an endorsement of such products by IBM.

The material included in this presentation regarding third parties is based on information obtained from such parties. No effort has been made to independently verify the accuracy of the information. This presentation does not constitute an expressed or implied recommendation or endorsement by IBM of any third party product or service.

The information presented regarding unannounced IBM products is a good faith effort to discuss iSeries plans and directions. IBM does not guarantee that these products will be announced.

Trademarks

© Copyright International Business Machines Corporation 2004. All rights reserved.

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States or other countries or both:

ADSTAR	Client Series	JustMail	PSF
Advanced Function Printing	DataGuide	Net.Commerce	SanFrancisco
AFP	DB2 Universal Database	Net.Data	SmoothStart
AIX	e-business logo	Netfinity	SystemView
AnyNet	IBM	NetView	WebSphere
Application Development	IBM Logo	OfficeVision	
APPN	i5/OS	OS/2	
AS/400	Information Warehouse	Operating System/400	
AS/400e	Integrated Language Environment	OS/400	
AT	Intelligent Printer Data Stream	PowerPC	
BrioQuery	IPDS	PowerPC AS	
BRMS	iSeries	Print Service Facility	

Photographs shown are of engineering prototypes. Changes may be incorporated in production models.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information in this presentation concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources. Sources for non-IBM list prices and performance numbers are taken from publicly available information. Including vendor announcements, vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future direction and intent are subject to change or withdraw without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of the specific Statement of Direction.