# IBM S/390 storage hierarchy—G5 and G6 performance considerations

by K. M. Jackson
K. N. Langston

**The CMOS-based IBM S/390 Parallel Enterprise Servers™ have always employed the technique of memory caching to bridge the gap between processor speed and main-memory access time. However, that gap has widened with each succeeding system generation, requiring increasingly sophisticated, multiple-level cache structures in order to minimize memory-access latency. The IBM S/390® G5 and G6 include two-level caching, with a binodal second-level cache. This paper reviews the principles of cache design, discusses the performance requirements of S/390 relative to caching, and describes how those requirements are addressed by the binodal L2 cache in the G5 and G6 systems.**

## Introduction

So often is it the case that one could call it a computer design adage—processors ever faster, while main memory grows ever larger but little faster. With each succeeding generation, main-memory access takes longer in terms of processor cycles. For more than 30 years, IBM has incorporated small (relative to main-memory size), high-speed level-1 (L1) memory caching in its flagship computers. L1 caching reduces the number of memory requests requiring main-memory access, which significantly reduces average memory access latency. In 1991, the IBM Enterprise System 9021 introduced a robust, fully shared level-2 (L2) implementation of memory caching between L1 and main memory. This further reduced the number of memory requests requiring main-memory access. The trend of adding memory caching levels is likely to continue unless technology or packaging breakthroughs occur that stem the continually widening disparity between processor speed and main-memory speed (access time). The IBM S/390 G5/G6 Parallel Enterprise Servers* include two-level memory caching. The second level is implemented in a binodal arrangement. The microprocessors are divided into two groups, or nodes, and all of the processors in a node share a second-level cache. The binodal organization [1] exploits the modularity of CMOS technology and the packaging benefit of a multichip module (MCM), while achieving much of the data-sharing benefit of a fully shared cache. The following sections discuss some considerations pertaining to caches in general, as well as the specifics of the G5 and G6 L2 cache.



**0018-8646/99/$5.00 © 1999 IBM**

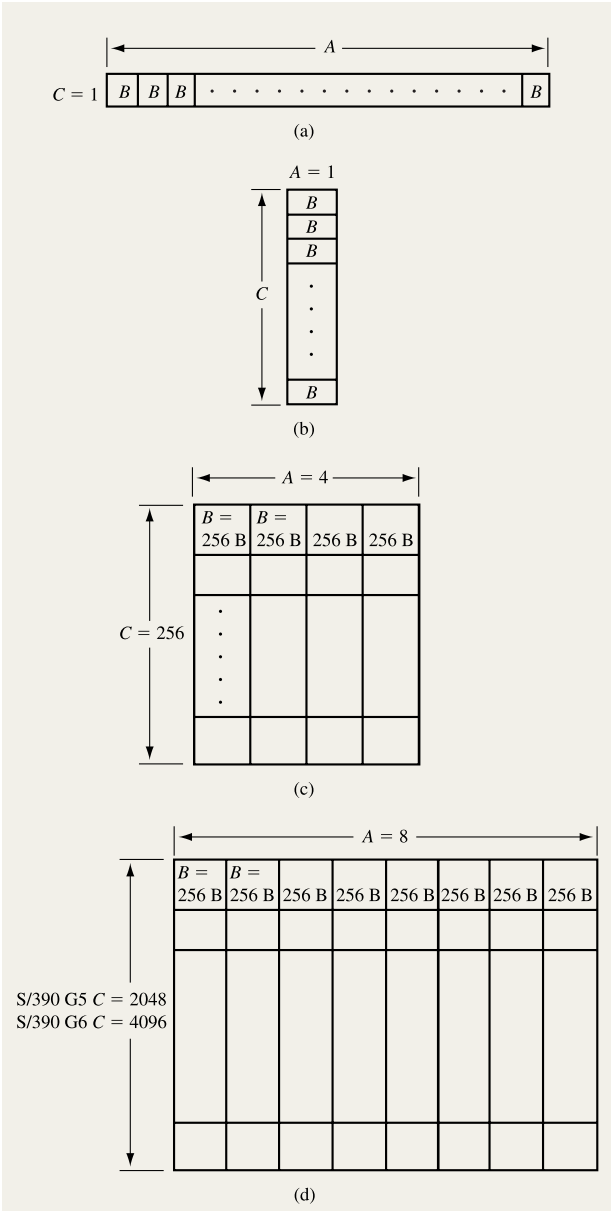


**Figure 1**

Cache form-factor elements: (a) Fully associative cache; (b) direct-mapped cache; (c) S/390 Generation 5 and 6 L1 cache; (d) S/390 Generation 5 and 6 L2 caches.

## Some cache basics

- *Form factor*

The form factor of a cache is determined by its "ABCs," as shown in **Figure 1**:

- **A**ssociativity – number of cache compartments (or lines) in each congruence class. The lines in each class are associated, in that a portion of their main-memory address bits have the same value for every line.
- **B**lock (or line) size – byte capacity of each cache compartment. This also defines the memory address-byte boundary on which each cache line begins, i.e., the minimum address boundary that a cache directory entry must maintain in order to distinguish unique lines.
- **C**ongruence classes – number of unique classes (or sets), typically $2^n$, i.e., all permutations of $n$ address bits, where $n$ is a design parameter.

$A \times C$ = number of cache compartments; $A \times B \times C$ = cache capacity in bytes. If $C = 1$, the cache is fully associative. A cache having $A = 1$ is direct-mapped or fully constrained. If $A > 1$, each congruence class requires some sort of replacement algorithm to determine which line of the associative set to replace when a new line is loaded into the class as the result of a cache miss. Cache modeling shows least-recently-used (LRU) to be an effective replacement algorithm.

A direct-mapped cache ($A = 1$) is the easiest to implement because there is no choice about which line will be replaced on a cache miss. With rare exceptions, the greater a cache's associativity, the lower its miss rate. For example, **Figure 2** shows the results from some model runs for an S/390* commercial database workload as described in [2]. A fully associative cache ($C = 1$) provides the greatest choice when it is necessary to choose a line for replacement, but is difficult or impossible to implement if the number of cache lines is large. A cache organized with modest associativity can still reduce misses considerably. Cache modeling has shown that a cache organized with an associativity of 4 can have fewer than half the misses of a direct-mapped cache of equal size; i.e., the direct-mapped cache may have to be four times larger to achieve a comparable miss rate. Modeling also shows that shared

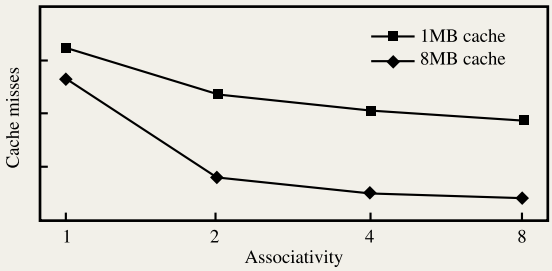


**Figure 2**

Effect of associativity on cache performance for a commercial on-line database workload running on a single S/390 processor.

caches benefit even more from higher associativity than do private caches. The G5 and G6 have taken advantage of this associativity benefit by implementing the L1 and L2 with associativities of 4 and 8, respectively.

The G5 and G6 L1 cache ABCs are 4, 256, 256, respectively (256 KB). The G5 L2 cache (per node) ABCs are 8, 256, 2048 (4 MB); G6 L2 cache (per node) ABCs are 8, 256, 4096 (8 MB).

- *Leading-edge and trailing-edge penalties*

When a requester references a cache but the line is missing, a line-fetch request is generated to obtain the missing data. The leading-edge penalty (latency) is the amount of time the requester waits before receiving the requested data. Resource contention will lengthen the leading-edge penalty. The minimum latency each cache miss must experience consists of the following components:

- Launching the cache-miss request to the facility that will begin processing the miss.
- If there are multiple processors, cross-cache communications (i.e., cache snooping) must take place to maintain the correct state in all caches.
- Fetching data from the appropriate source.
- Returning the data to the requester.

The trailing-edge penalty is the cost of caching the remainder of the line. The trailing-edge penalty would be minimal if the entire line could be returned and installed in the cache in a single cycle. However, few designs can afford the implied luxuries of line-wide data-return buses and cache-write capability. Some contributors to trailing-edge penalty are the following:

- Line-install cache writes. Typically, several cache-write cycles are required to install the entire line, any of which may preempt and delay other cache users. Implementing a line-fetch buffer to temporarily hold the returning line allows line-install cache writes to be delayed, to be done on later available cache cycles or at less disruptive times. However, disadvantages of a line-fetch buffer include complication of the line-write controls and, if fetching from the line-fetch buffer is allowed, an additional source of data for subsequent fetches. This can complicate the critical timings surrounding the logic that controls which one of multiple data sources to gate to the requester.
- Subsequent fetch-request data in the line being loaded. Access to this line may be denied until the entire line is loaded into cache and the directory entry is marked valid. This delay can be reduced by allowing a partially loaded line to be accessed if the desired data is in the cache or line-fetch buffer. A disadvantage is the need for logic denoting which portion of the partially loaded line is available. Also, the cache-hit logic is further complicated. This can complicate critical timings that often surround the creation of a cache hit and control which data source to gate to the requester.
- Completion of a previous cache-miss data transfer, which delays subsequent cache-miss data transfers. Because cache misses cause line transfers rather than the transmittal of just the amount of data requested, the data transfer persists far beyond the point at which data is returned to the requester. Cache misses occurring in close time proximity can proceed no faster than lines can be loaded into the cache. This delay can be reduced if the requested data from a subsequent miss can preempt the transfer of nonrequested data from earlier cache misses. This complicates cache-miss handling by requiring support for multiple outstanding cache misses and data tagging.

- *Line size*

When a cache miss occurs, the requester's data, along with all remaining data comprised by the line, is loaded into the cache. Cache is effective partly because data that is likely to be referenced in the near future is loaded in addition to what has been specifically requested. Making cache lines longer means that even more additional data is loaded into the cache with each miss, which further increases the likelihood that subsequent references will get a cache hit. From this reasoning, why not divide the cache into just a few long lines, e.g., 8 or 16? One could cite advantages:

- Fewer cache misses.
- Fewer and shorter cache directory entries.
- A surrogate for next-sequential line prefetch. This prefetch algorithm operates under the premise that references to line $n + 1$ will occur soon after $n$ is referenced. If line $n + 1$ is missing from cache but is loaded or being loaded into cache before $n + 1$ is referenced, a cache miss can be avoided or its latency reduced. The downside is a wasted line load and additional trailing-edge interference if $n + 1$ is not referenced.

However, longer cache lines may be a poor choice when it comes to performance; reasons for this include

- Poor payback from loading extra data. For example, doubling the line size would result in a 50% cache-miss reduction if the extra data loaded for each line were referenced just once during its cache residency. Cache modeling shows the actual reduction to be closer to 30%. From a different perspective, loading twice as much data per cache miss is about 60% effective for

many S/390 benchmarks, because the leading-edge penalty is eliminated. Conversely, 40% of the extra data fetched is never touched during its cache life.
- Elongation of trailing-edge penalty. Doubling the line size will degrade performance instead of improving it if the growth in trailing-edge penalty exceeds the reduction of leading-edge penalty.

In general, increasing the line size becomes advantageous when latencies become large.

- *Cache state information in SMP*

In a shared multiprocessor (SMP) system, cache lines are classified to distinguish permitted uses. A frequently used classification is "MESI": modified, exclusive, shared, or invalid [1]. Modified lines contain changed information; exclusive lines can be modified by only one processor; and shared lines can be used, but not modified, by more than one processor simultaneously.

## S/390 reliability requirements and performance implications

- *Write-through and write-back L1 cache*

Although the choice between a write-through and a write-back L1 cache is usually based on reliability and availability considerations, it can have significant performance implications.

In a write-back L1 cache design, all fetch and store references are installed in the L1 cache. Modified lines are sent to the next level in the storage hierarchy only when the L1 line is replaced. S/390 data reliability expectations would necessitate error-correction codes (ECC) in the write-back L1 cache and directory, since the write-back is the only copy of the modified data. There are certain unrecoverable error situations which, without ECC, are not acceptable under S/390 reliability requirements. One such event is a faulty directory entry when the corresponding line is modified.

In a write-through L1 cache design, store references are not installed in the L1 cache. In addition, modified data is transferred immediately to the next level of cache. The next level of cache must have enough bandwidth to handle all of the individual store requests. Parity protection on the L1 cache is sufficient, because the modified data has been forwarded to a more data-secure level. On an L1 cache or directory parity error, the corrective action is to invalidate the line in the L1. The data will be fetched into L1 only if referenced again. If necessary, a processor may be restarted without incident once any queued stores are completed. Of course, to restart the processor without incident requires coordination and close cooperation between the hardware and operating system.

A performance consideration in deciding whether to implement a write-through or a write-back L1 cache is the impact of stores on the processor-to-L2 data bus. In a write-through scheme, each store occupies the processor-to-L2 data bus for a single cycle. In a bidirectional bus implementation, this has little effect on fetch requests on the bus. At most, a store can delay the transfer of fetch data by one cycle. If a priority mechanism is implemented, the fetch data transfer will always receive higher priority, removing any impact caused by store data transfers. In a write-back scheme, the entire line is stored. When the line size of the L1 cache is large and the bus width is several times smaller than the line size, these stores can interfere with fetch data returns. This translates into longer demand-fetch latency and degraded performance.

In a system with a write-through L1 cache, all cache-to-cache (intervention) transfers are between L2 caches. Except for stores which are waiting for bus priority at the L1, all modifications are reflected in the L2 cache.

Since modifications are accumulated in a write-back L1 cache, the L1 cache must provide cache-to-cache data for modified lines. This additional interference can impede both the source and destination processor performance. The impact on the processor-to-L2 bus depends on the bus width and the line size.

On a system with 16-byte-wide processor-to-L2 data buses, a line size of 256 bytes, and a write-back L1 cache, an L1 write-back operation takes 16 data transfers. This could cause significant system performance degradation. Hence, G6 implemented a write-back-managed write-through L1 cache.

- *Write-back-managed write-through L1 cache*

A variation on the write-through scheme is a write-back-managed write-through L1. The G5 and G6 L1 uses this scheme. As with write-back, all fetch and store references are installed in the L1 cache, but modified data is transferred immediately to the next level of cache as with write-through. Although it appears at first that such a scheme incorporates the disadvantage of each of these disciplines, this scheme does have advantages. Write-through permits error detection rather than detection and correction at the L1. The error detection is implemented as byte parity on G5 and G6. A byte is the smallest store granularity allowed by the architecture. With byte parity, no merge of modified and unmodified data is required. Error recovery is accomplished by invalidating the L1 line.

G5 and G6 implement single-error-correct, double-error-detect ECC protection in the L2 cache [3]. This complicates store operations that modify fewer bytes than the ECC word size. These store operations require merging of unmodified and modified data in the ECC word. Ordinarily, the unmodified bytes must be read and

ECC-checked, and corrected if errors are found. The modified bytes are merged, and the ECC code is generated and stored with the new merged data. This requires a cache-read and a cache-write operation.

By having a write-back-managed write-through L1, the unmodified bytes are available from L1 to form a full ECC word. At the same time the modifications are made to the L1 line, the L1 supplies the unmodified bytes. ECC is generated on the merged data before it is sent to L2. The length of this data is a full ECC word; i.e., only ECC generation and write operation are required. With a write-through L1, the L2 with ECC performs a read operation, ECC check, data merge, ECC check, and write operation. The write-back-managed write-through L1 removes the need for the read on behalf of the ECC operation, the associated ECC check operation, and the data merge at L2. This reduces L2 resource utilization.

## L2 cache topology

- *Private vs. shared L2 cache*

L2 caches in an SMP can be implemented either as private to each processor or as shared among multiple processors. In a private L2 cache system, there is a one-to-one correspondence between a processor and an L2. A fully shared L2 cache has all processors sharing one common L2 cache. S/390 commercial database workloads typically perform best with a fully shared L2 cache.

The performance of a cache is determined by the cache hit rate and the time required to return data from that cache to the processor (latency). The IBM S/390 commercial database workloads show consistent results with respect to cache hit rates: Fully shared caches are best. Operating system and database management code and structures are often shared by multiple processors. In a shared L2 cache, a line exists only once, regardless of the number of processors sharing that line. In a private L2 cache implementation, a shared line must be installed in the L2 cache of each requesting processor. For the same total L2 size, the private cache implementation is less efficient.

Private L2 caches nevertheless have some benefits. Since there is a one-to-one correspondence between each L2 cache and the associated processor, the L2 cache can be placed close to the processor. This reduces the latency from the local L2 cache. The bandwidth between the processor and the L2 cache may be larger, since the connections are limited to one processor and L2. However, the bandwidth constraints may be at the L2-to-main-memory interface instead, since each L2 cache requires addressability to all of memory in an SMP system.

In a fully shared implementation, there is no one-to-one correspondence between the processor and the L2 cache. Therefore, the system can be optimized for cost and performance by changing the total number of L2 cache chips. The system chip count for a large SMP with a fully shared L2 cache implementation may be less than that for a private L2 cache implementation.

- *L2 cache latency tradeoffs*

Although a fully shared L2 cache has a better hit rate than a private L2 cache implementation, the L2 cache hit latency may be worse in a fully shared L2 system. The placement of the L2 cache chips may not be optimized for all processors. For very large SMP systems, it is impossible to place all processors adjacent to the fully shared L2 cache. This contributes to longer latency. Additional cycles may also be needed to determine which request will be serviced when multiple requests are waiting.

Private L2 caches can be implemented on the processor chip, since there is a one-to-one correspondence between the processor and the L2. Fully shared L2 caches are restricted to separate-chip implementations. This further distinguishes the latencies between fully shared and private L2 cache implementations.

In a fully shared L2 cache system with a write-through L1 cache, there is no notion of a cache-to-cache transfer; i.e., L2 and memory are the only sources for processor fetch requests that miss the L1 cache. Further, cache snooping is handled at the L2 cache directory. Some cache snoops must be broadcast to the L1 cache(s). Because state changes are handled at the L2 cache, the average L1 miss-fetch latency is minimized in a fully shared L2 cache system.

Private L2 cache systems require some form of cache-to-cache transfer mechanism between L2 caches. The volume of these transfers depends on the intervention scheme that has been implemented. Private L2 caches must be interconnected to allow the transfer of snoop signals and data between caches. If the interconnection is through a common system bus, the latency for cache-to-cache data transfers can approach or exceed main-memory latency. An alternative is to provide a cross-point switch. Data is transferred from the source cache through the cross-point switch to the destination cache. Cross-point switches are typically implemented on separate chip(s), which adds additional system cost. Cache-to-cache latency depends on the structure, the technology implemented, and the state of the line in the source L2.

In addition to cache-to-cache data transfers, cache snoops can have a detrimental effect on system performance. Every request for an exclusive line that was found to be shared or invalid in the L2 cache may result in a snoop to all processors. For a large SMP with high store rates, this degrades performance even if the line misses all other L1 caches.

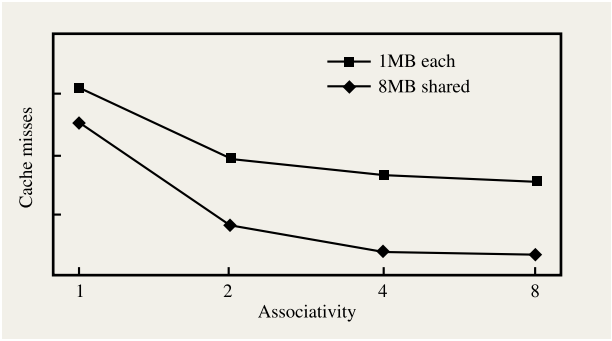


**Figure 3**

Effect of associativity on cache performance for a commercial online database workload running on an eight-processor SMP.

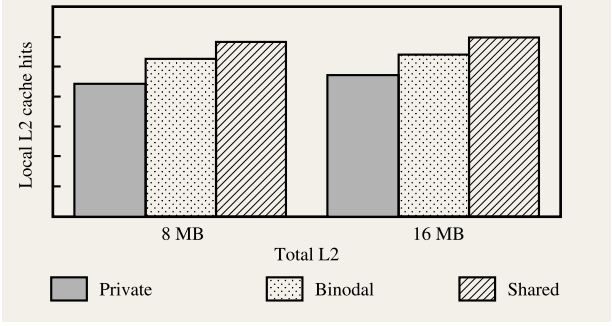


**Figure 4**

Comparison of cache-hit rates for shared, binodal, and private L2 caches for a commercial database workload running on an eight-processor SMP.

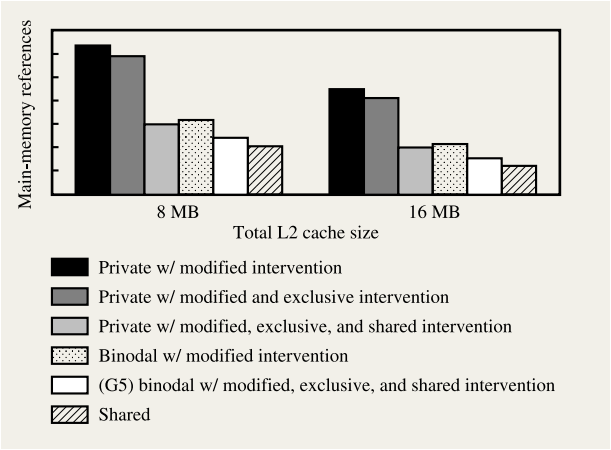


**Figure 5**

Effects of L2 cache sharing on intervention schemes for a commercial database workload running on an eight-processor SMP.

A significant benefit of a private L2 structure is the cost and scalability of the design. The multiprocessor-to-single-processor performance ratio (MP ratio) is a measure of SMP efficiency. In a private L2 cache system, single-processor systems have one L2 cache; *n*-processor systems have *n* L2 caches. In a fully shared L2 system, the L2 cache size is usually optimized for the maximum number of processors. Either special controls are needed to support different-size caches for different numbers of processors, or the same size L2 cache is used for single-processor through many-processor systems. Either case adds to the cost of low-end systems.

Another consideration with regard to a fully shared L2 cache is the number of associativity sets. For a single processor, four sets are usually sufficient. In a large SMP system, more sets may be necessary to avoid thrashing within a congruence class. This can add to the L2 cache hit latency and to the design complexity. **Figure 3** shows the effect of varying the associativity on an eight-processor SMP system running an S/390 commercial database workload.

- *G5 and G6 binodal L2 cache*

A fully shared L2 cache performs best for S/390 commercial database workloads. However, because of pin limitations, a fully shared L2 cache design was not possible on the G5. A reasonable compromise was the binodal cache design [1], in which the microprocessors are divided into two groups, or nodes. Each node has a maximum of six microprocessors on G5; seven microprocessors on G6. All processors on a node share an L2 cache. However, the two L2 caches are private to one another. This binodal L2 cache design provides many of the advantages of both the private and the shared-cache designs.

Since each L2 is connected to only half of the processors, the bus bandwidth between the L2 and the processors can be larger. Additionally, the processors and the L2 cache can be close to one another, reducing L2 cache-hit latency. The L2 cache-hit rate in the G5 and G6 binodal implementation is better than that of a private L2 cache implementation of similar size, but worse than that of a fully shared L2 cache. **Figure 4** shows the L2 cache-hit rates for fully shared, binodal, and private L2 caches. The model results are for an eight-processor SMP system running an S/390 commercial database workload [2].

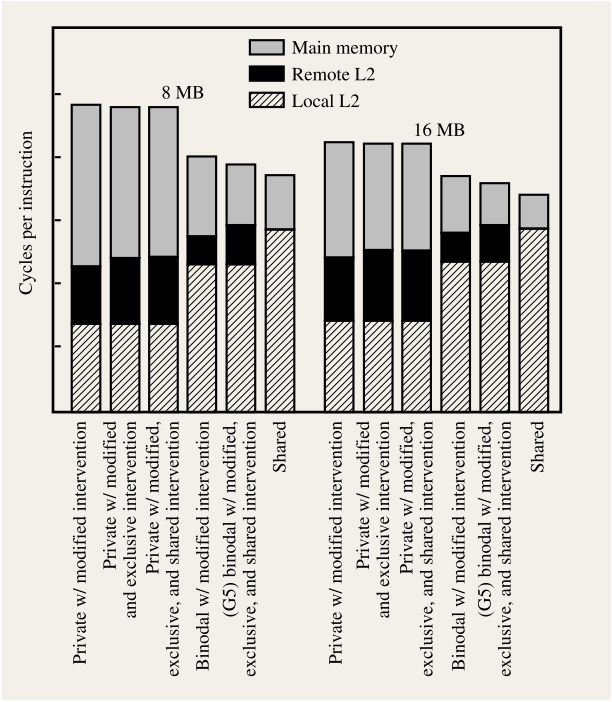


**Figure 6**

Comparison by data source of CPI demand for fully shared, binodal, and private L2 caches.

The binodal L2 cache system has buses between the two nodes for snoops and cache-to-cache line transfers. A cross-point switch mechanism is implemented within the system controller (SC) [1] to minimize cache-to-cache latency. By maintaining a high bandwidth between the nodes, the system can support modified, exclusive, and shared intervention. This reduces main-memory fetches. **Figure 5** shows the relative number of main-memory fetches for shared, binodal, and private L2 implementations. To minimize the impact of snoop requests on each processor, the G5 and G6 L2 maintains information on L1 lines. While this puts additional burden on the L2, it greatly reduces the impact on the processors.

**Figure 6** shows the cycles per instruction (CPI) [(cycles per event) × (events per instruction)] for fully shared L2 cache, binodal L2 cache, and private L2 cache systems with different intervention schemes. The binodal L2 cache design with modified, exclusive, and shared intervention is the next best approach to fully shared. The L2 cache-hit portion of the CPI is best for private L2 because of the reduced latency. However, the cache-to-cache and main-memory portions exceed those for both binodal and fully shared L2. Only the private L2 cache with modified, exclusive, and shared intervention gives results close to those for the binodal L2 cache.

## Conclusions

Cache modeling shows that increased associativity increases cache effectiveness, and both G5 and G6 exploit this.

In comparison with G4, which had 128-byte lines, longer lines were beneficial in this design. The reduction in misses from the longer line size outweighs the additional trailing-edge penalty.

Fully shared caches are best for most OS/390* environments, but are difficult to implement with the technology available today. In an SMP environment with large numbers of processors, the binodal approach was the only acceptable alternative to a fully shared L2 cache. It also has the benefit of allowing additional processors to be added on each node to take advantage of incremental technology density improvements. Smaller improvements in technology density can be exploited. The binodal approach is as close as one can get with current technology to a fully shared design and will continue to meet S/390 performance needs for the foreseeable future.

*Trademark or registered trademark of International Business Machines Corporation.

## References

1. P. R. Turgeon, P. Mak, M. A. Blake, M. F. Fee, C. B. Ford III, P. J. Meaney, R. Seigler, and W. W. Shen, "The S/390 G5/G6 Binodal Cache," *IBM J. Res. Develop.* **43,** No. 5/6, 661–670 (1999, this issue).
2. IBM Corporation, *IBM Large Systems Performance Reference*, Order No. SC28-1187-05; available through IBM branch offices.
3. P. Mak, M. A. Blake, C. C. Jones, G. E. Strait, and P. R. Turgeon, "Shared-Cache Clusters in a System with a Fully Shared Memory," *IBM J. Res. Develop.* **41,** No. 4/5, 429–448 (1997).

*Received February 15, 1999; accepted for publication July 19, 1999*

**Kathryn M. Jackson** *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (kmjackso@us.ibm.com).*
Ms. Jackson is an Advisory Software Engineer in S/390 Custom Hardware Design, responsible for SMP storage hierarchy design and performance analysis for S/390. She received a B.S. degree in computer science and mathematics from Hofstra University in 1982 and joined IBM as a programmer in Poughkeepsie. Previous assignments included design work in Enterprise Systems Central Architecture, for which she holds one U.S. patent. She has received two IBM Outstanding Technical Achievement Awards for her work on the G4 and G5 systems.

**Keith N. Langston** *2545 E. 1100 N, Roanoke, Indiana 46783.*
Mr. Langston, retired in 1995 from IBM after 30 years of service, is an independent system consultant. At the time of his retirement he was a Senior Engineer who, for the previous 20 years, did performance analysis on various S/390 processors and, for the last 15 years, SMP storage hierarchy design and performance analysis. He received an A.A.S. degree in electrical engineering from Purdue University in 1965 and began his career with IBM in Kingston, New York. After working in Kingston and Boeblingen, Germany, he joined the development laboratory in Poughkeepsie, where he was involved in performance analysis for the remainder of his IBM career. He achieved the First-Level Invention Plateau and received a Division Award, an IBM Outstanding Innovation Award, and twelve informal awards.