# On-Chip Cache Coherence and Real-Time Systems

**Paul E. McKenney**
IBM Linux Technology Center
15400 SW Koll Parkway, Beaverton, OR 97006 USA
paulmck@linux.vnet.ibm.com

**Abstract**

In the Communications of the ACM article entitled "Why On-Chip Cache Coherence Is Here to Stay", Martin et al. argue that on-chip cache coherence can scale for the foreseeable future, with only modest impact on performance, even for very large systems. Unfortunately, a number of the authors' arguments apply only to real-fast systems. Although some might argue that real-time response is important only relatively small systems, recent bug reports regarding 200-microsecond latencies on systems with 4096 CPUs provide a strong counter-argument. This paper therefore looks at how Martin et al.'s arguments fare in real-time environments.

## 1 Introduction

A number of researchers have argued that future computer systems would move away from hardware cache coherence due to overhead and scalability concerns [1, 2, 3]. However, Martin et al. recently argued that cache coherence would remain in hardware for the foreseeable future [4], addressing five concerns:

1. System-interconnect traffic
2. In-cache storage overhead
3. Maintaining inclusion
4. Memory latency
5. Energy efficiency

Unfortunately, Martin et al. analyzed only the real-fast case, for the most part ignoring real-time response. The following sections summarize their arguments surrounding each of the above concerns and examine the corresponding real-time implications. But first we review relevant topics in system architecture.

## 2 System Architecture

Figure 1 shows a view of system architecture for a fanciful system having eight CPUs. Each CPU has a private cache, and there is a shared cache for each pair of CPUs. Each cache contains *cache lines*, maintaining multiple copies (from zero upwards) of each fixed-sized block of memory. This fixed size is a power of two, assumed for the purposes of this paper to be 64 bytes.

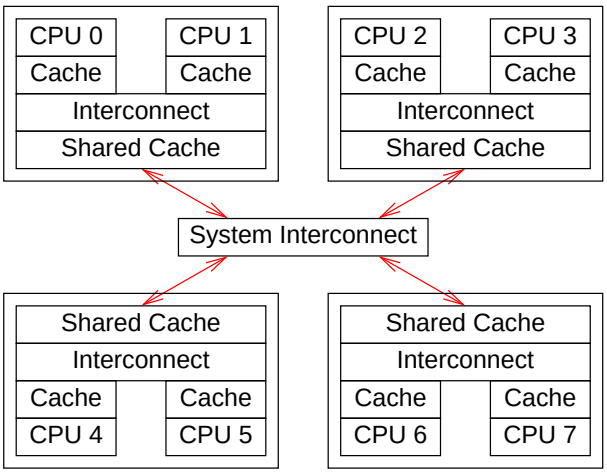


**FIGURE 1:** *System Architecture Schematic*

If the system is *cache coherent*, then all cached copies of a given block of memory will have the same value. In contrast, *cache-incoherent* systems might have conflicting values for the same block of memory in different caches. A key question is whether systems will ever grow so large that cache coherence becomes infeasible. This question is important because cache-incoherent systems have proven to be more difficult to program than have the cache-coherent systems currently in widespread use.

As indicated by the red arrows, modern multicore processors are really message-passing systems when viewed at the hardware level. These messages carry requests for cachelines containing the corresponding memory address, responses containing the corresponding memory data, invalidation requests containing the corresponding memory address, and acknowledgments containing the corresponding memory address.

With this overview, we are ready to look into the concerns addressed by Martin et al.'s paper, paying special attention to real-time issues.

# 3 System-Interconnect Traffic

Cache coherence requires additional system-interconnect traffic in the form of acknowledgment messages when a given CPU suffers a cache miss in response to a memory write. These additional acknowledgment messages let the writing CPU making the request know when the cache line has been invalidated from other CPUs' caches, avoiding different caches containing conflicting values for a given cache line. A key concern is that the number of acknowledgment messages will grow as the number of CPUs grows, eventually crowding out the messages that actually carry the data. Because cache-incoherent systems do not require acknowledgment messages, one school of thought argues that larger systems will need to be cache-incoherent.

Martin et al.'s key observation is that these acknowledgment messages are small, containing an address without data, and further that each acknowledgment message pairs with a larger data message. To see this, consider that the acknowledgment messages come only from other CPUs' caches containing the cache line being written to. But the only way that this cache line could be pulled into the other CPU's caches is for the corresponding CPU to have previously suffered a read cache miss. Because the read cache miss requires a small request and a large response, the small cache-coherence acknowledgment will impose a small fixed incremental overhead on system traffic, regardless of the number of CPUs.

This analysis ignores potential degradation to real-time response due to the potentially large number of messages sent in response to a single write request. Although it is possible that the outgoing messages might be replicated in hardware, and that the incoming messages might be combined in hardware, a high rate of writes to cachelines replicated in all caches is likely to impose high worst-case latencies due to hardware queueing, either in the interconnect or at the destination caches. In the future, therefore, it may be necessary for real-time programs to reduce (or perhaps even completely avoid) variables that are written frequently and read by all CPUs even more frequently.

Of course, Martin et al.'s original line of reasoning for this concern relies on the system tracking exactly which CPUs have a given block of memory in their caches, which in turn increases the amount of state that each cache must track. This additional in-cache storage overhead is addressed in the next section.

# 4 In-Cache Storage Overhead

Martin et al.'s assumption that the system would track exactly which caches contain a given cache line requires extra state be maintained in the caches, thus increasing their on-die size. For example, a 4096-CPU system would need a full 4096 bits in each cache line to track which CPUs had copies. 4096 bits is 512 bytes, which is an egregiously excessive overhead to impose on a 64-byte cacheline.

Martin et al. address this concern by placing the extra state in the last-level shared caches and by tracking only the last-level caches, as opposed to tracking all the CPUs. This approach reduces the number of tracking bits from one per logical (Linux-visible) CPU down to one per core. In our 4096-CPU example, assume that each last-level cache served 64 CPUs, so that there were 64 last-level caches. Then each cache line would need only eight bytes for tracking, which is far more reasonable than is 512 bytes. The authors further expect that increases in system size will continue to require increasing numbers of levels of caches, further reducing the number of tracking bits required in each last-level cache.

However, this assumes that the last-level cache knows which cachelines are held by all the caches closer to the CPU. One straightforward way to accomplish this is to enforce *inclusion*, in which any given shared cache must contain copies of all cache-

lines held in the caches closer to the CPUs served by that shared cache. For example, in Figure 1, every cache line held in CPUs 0 and 1 would also need to be also held in the shared cache associated with these two CPUs.

On systems with inclusion, each cache need only track the presence of the cacheline in its siblings within the cache hierarchy, thus greatly reducing the required state. In fact, if system size is increased by adding levels to the cache hierarchy while limiting the hierachy's fanout, it is easy to see that the storage overhead would be a constant fraction of the total. However, the counterargument from cache-incoherence proponents would be that these additional levels of hierarchy are not needed for cache-incoherent systems. Martin et al.'s response is to note that three levels of cache hierarchy are sufficient to keep the storage overhead required for exact tracking of cachelines to within a few percent up to a few thousand cores.

The real-time issue with all of this is of course that adding levels to the cache hierarchy increases worst-case memory latency, which in turn increases overall worst-case latency. That said, a large number of real-time applications are running just fine on systems with three levels of cache hierarchy, so it is quite possible that this real-time concern is strictly theoretical.

Theoretical or not, this scalability argument rests on the property of inclusion, which leads to the question of whether inclusion is practical for arbitrarily large systems. This question is taken up in the next section.

# 5 Maintaining Inclusion

Maintaining inclusion is not free: If the shared cache needs to eject a cacheline for any reason, then all of the subordinate caches are also required to eject that line, even if they were making heavy use of it. This can of course result in expensive cache misses that are in some sense unnecessary. To analyze this situation, we need to introduce the concept of cache *associativity*. To understand associativity, consider that a cache can be thought of as a hash table that is implemented in hardware. When taking this view, a cache's associativity would be analogous to a maximum number of elements in each bucket of the hash table.

Martin et al.'s observation is that if the shared cache is considerably larger than the sum of the sizes of its subordinate caches and if the shared cache's associativity is larger than the sum of the associativities of its immediately subordinate caches, then the shared cache is guaranteed to be able to hold any combination of data that its subordinate caches can hold. Even so, this situation requires that subordinate caches notify shared caches when the eject a cacheline, even if that cacheline was a read-only copy of data held elsewhere. This notification allows the shared cache to preferentially eject cachelines that are not being used by any of the subordinate caches.

Unfortunately, the associativity requirement is a severe one: a shared cache with eight immediately subordinate eight-way set-associative caches must be at least 64-way set associative. From a real-fast perspective, this is not a real problem because reducing the associativity of the cache from 64-way to (say) 16-way will result in but a tiny increase in cache miss rate, especially if the shared cache is proportionately larger.

Of course, this argument does not necessarily hold for real-time systems, especially for hard real-time systems, where even a tiny increase in the number of cache misses could dangerously degrade worst-case latency.

Of course, the number of cache misses is not the only contributor to overall latency. This overall latency is also degraded if memory latency increases, which is addressed by the following section.

# 6 Memory Latency

As the number of CPUs increases, all else being equal, the time required to move data from one CPU to another must also increase. The authors note that this increase is smallest in the absence of widely-read variables, and also note that careful use of hierarchy in the hardware interconnect (including the cache hierarchy) can minimize memory latency. In other words, the average latency normally does not increase as a linear function of the number of CPUs.

This of course does not make any guarantees on the worst-case memory latencies, which are critical to real-time applications.

# 7 Energy Efficiency

It is tempting to dispense with energy efficiency on the grounds that one of the first things done to any real-time systems is to disable any and all energy-efficiency features. On the other hand, the level of energy-efficiency concern has been steadily rising, so

it is not completely unreasonable to assume that this concern will eventually reach the domain of real-time computing.

Martin et al. argue that energy consumption will continue to be a function of the number of transistors and the number of messages generated by memory traffic, including any cache-coherence messages. Since they demonstrated that both the storage overhead (which translates to additional transistors) and the cache-coherence traffic scale sublinearly with the number of CPUs, they argue that the energy consumption will also scale sublinearly, so that energy efficiency will not be an obstacle to scaling of cache-coherent systems.

For the most part, real-time systems should be limited by average energy consumption rather than worst-case energy consumption, so that this analysis should apply to the real-time as well as real-fast applications. One possible exception is situations where real-fast systems could impose thermal throttling to handle momentary over-temperature conditions. However, the appropriate real-time response might be to overprovision cooling technologies or to limit the clock rate to a value that avoided over-temperature conditions.

# 8 So, What To Do?

For real-fast applications, cache coherent systems should be able to scale up throughout the foreseeable future. It is quite possible that almost all real-time applications will continue to scale as well.

However, it is possible that real-time systems will face some obstacles that are not a problem for real-fast systems:

1. The worst-case memory latency of writes to heavily read-shared variables is likely to increase with system size.
2. Increasing the depth of the cache hierarchy to keep cache storage overhead bounded will further increase memory latency.
3. Underprovisioning the associativity of shared caches will result in increased numbers of expensive cache misses.

Of course, if these problems do not arise, nothing need be done. Nevertheless, it is worth investing some thought into how to solve these problems.

If the worst-case memory latency to heavily read-shared variables proves problematic, real-time developers could avoid software designs featuring data that is written often and read widely. Where oft-written widely-read data cannot be avoided and where its latency proves problematic, one approach is to minimize its scope, for example, by hiding the data behind a single pointer and ensuring that freed memory blocks are flushed from all caches before being reallocated. This way only one cacheline will suffer outsized latencies. Another approach is to replicate the data, replacing the single long-latency write with multiple writes each having smaller latency. It is worth noting that for the foreseeable future, many other software considerations, including lock contention and memory contention due to lockless algorithms, are likely to be more problematic for real-time applications than are the latency of writes to heavily read-shared variables.

If memory latency is a problem even in the absence of writes to widely-read data, confine a given realtime application to some portion of the system. For example, if cross-system CPU-to-CPU latency is problematic, confine the application to a small subtree of the system's cache hierarchy. This approach ensures that the application's data can only be shared within that subtree, thereby reducing the write latency. The other subtrees of the system might be used to run other applications, both real-time and real-fast.

On the other hand, if the memory-latency problem is instead due to the limited associativity of the shared caches, use only a few of the CPUs serviced by each shared cache. This latter approach is similar to the current common real-time practice of disabling hyperthreading, and does represent underutilization of the system unless a non-interfering workload can make good use of the rest of the system. Underutilization smacks of inefficiency, but computing systems are now inexpensive enough that underutilization is nowhere near as great a sin as it was in past decades.

Finally, it is important to understand that scalability is not the only threat to real-time applications running on modern hardware [5].

# 9 Conclusions

Martin et al. [4] make a good case that computer systems will retain cache coherence despite increasing CPU counts. Several of their arguments do not apply to real-time systems, however, it is likely that the software and configuration workarounds listed in Section 8 will allow real-time applications to nevertheless run reasonably gracefully on such systems.

These workarounds will not be pain-free, but then again, neither would software handling of hardware cache incoherence.

Software issues currently faced when scaling real-time systems, including various forms of contention as well as the inherent randomness of modern hardware, are likely to be more pressing than hardware scalability issues for the foreseeable future. It is nevertheless worth putting some thought into how best to handle the increasing scalability of post-modern hardware.

## Legal Statement



## References

[1] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 20$^{th}$ International Conference on Parallel Architectures and Compilation Techniques (Galveston Island, TX USA)*, pages 155–166, Washington, DC USA, October 2011. IEEE Computer Society.

[2] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vanagal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntly, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the 2010 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 108–109, Washington, DC USA, February 2010. IEEE.

[3] John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel. Cohesion: An adaptive hybrid memory model for accelerators. *IEEE Micro*, 31(1):42–55, Jan./Feb.) 2011.

[4] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip coherence is here to stay. *Communications of the ACM*, 55(7):78–89, July 2012.

[5] Nicholas Mc Guire, Peter Odhiambo Okech, and Qingguo Zhou. Analysis of inherent randomness of the linux kernel. In *Eleventh Real Time Linux Workshop*, Dresden, Germany, September 2009.