RZ 3776 (# Z1005-001) 05/10/2010
Electrical Engineering 11 pages

Research Report

Cache Injection for Private Cache Architectures (Concept Paper)

F. Auernhammer, P. Sagmeister

IBM Research – Zurich
8803 Rüschlikon
Switzerland

Cache Injection for Private Cache Architectures

(Concept paper)

Florian Auernhammer
IBM Zurich Research Laboratory
fau@zurich.ibm.com

Patricia Sagmeister
IBM Zurich Research Laboratory
psa@zurich.ibm.com

ABSTRACT

The memory wall is considered to be one of the biggest challenges for multi- and many-core architectures. Putting more and more cores on the processor die considerably increases the required memory bandwidth far in excess of the available memory subsystem bandwidth. The major bottleneck for achievable memory bandwidth are limited off-chip bandwidth and coherence requirements. Larger caches and deeper cache hierarchies can alleviate the memory bandwidth problem only to some extent.

For I/O-related operations, cache injection has the potential of mitigating the pressure on the memory subsystem by reducing the number of necessary memory transfers. However, the mechanisms proposed so far are mainly aimed at non-virtualized environments. In this paper we propose concepts for cache injection in systems using private cache architectures with special focus on providing support for heavily virtualized devices. Therefore we study different aspects of cache injection, i.e., the device-processor interaction, the injection process in the cache-coherent processor fabric, as well as the cache itself. The key characteristics of our proposed methods are better directivity, less overhead on the processor interconnect, and better control mechanisms for cache injection.

1. INTRODUCTION

Cache injection is a well-known topic in computer architecture research. Nevertheless, little research applicable to upcoming heavily virtualized many-core systems is available. On the one hand, many proposed mechanisms focused on bus-based rather than on today's switched fabric processor interconnect implementations, rendering many propositions such as snarfing [3] impractical. On the other hand, they are focused, for the most part, on single server communication. With heavily virtualized devices, providing user-level interfaces, and large many-core systems, however, new challenges but also new opportunities for cache-injection arise.

Today's systems heavily use virtual machines (VMs) or logical partitions (LPARs) to leverage multi-core processors. The operating systems running in LPARs may not be aware of where they are physically running and may also not be running at the time a network device receives data for it. This makes it difficult for cache injection to determine if and where data should be injected as injudicious injection may jeopardize the system performance.

Another challenge for cache injection is keeping overheads in the processor fabric low. Many-core systems with distributed cache architectures use partitioned cache coherence domains [8, 10], and will very likely resort to them ever more heavily in the future in order to restrict coherence operations to smaller domains of the system. Therefore, cache injection should be able to place data in caches that are close to the final consumer to mitigate the overhead and performance impact [17] of coherence operations.

For traditional devices as well as for devices shared on a virtual machine basis, directed cache injection can be supported efficiently in an IOMMU. For user-level interfaces, however, we think that it is essential that the I/O device be able to provide information for cache-injection purposes when storing data to the system. We therefore propose two mechanisms that allow I/O devices to use the processor interconnect to improve the directivity of cache injection. We further propose concepts aimed at reducing the overhead for cache injection on the processor interconnect as well as controlling the amount of data that may be injected into a cache to limit destructive effects. Finally, we propose a doorbell concept based on the injection concepts.

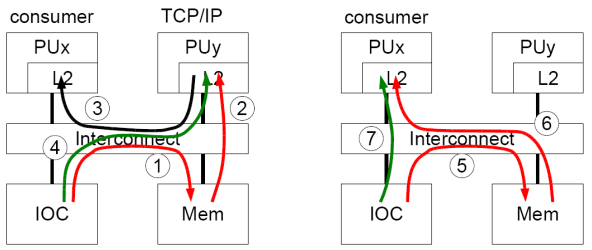
The remainder of the paper is organized as follows. In Sections 2 and 3 we present the principles of user-level interfacing and snooping-based cache coherent fabrics respectively. Section 4 describes the different mechanisms we propose for cache injection in private cache architectures. Section 5 presents related work, and we conclude in Section 6.

2. USER-LEVEL INTERFACES

User-level interfaces based on the Virtual Interface Architecture (VIA) [2] are used for low-latency communication and multi-user support in network interface controllers. VIA was first used in InfiniBand but is now also increasingly being adopted in high-speed TCP/IP off-loading solutions.

The basic concept for user-level interfaces is the use of asynchronous queues for the communication between user and adapter. Therefore, one queue for send and one for receive operations is created in main memory for each user-level interface. The user puts send and receive requests, so-called work requests, on the queues and notifies the adapter of new work requests by ringing its doorbell. The requests contain just address references that indicate where the actual payload data is to be fetched from or stored to.

The adapter maintains separate contexts for all active connections. The contexts contain, on the one hand, information on the state of the network connection and, on the other hand, data related to the software interface. Compared with traditional devices, contexts represent a major advantage for virtualized devices as, by extending the context, additional information for each interface can be stored, for example, by



(a) Socket interface (b) User-level interface

Figure 1: Potential of cache injection

which process it is used and also on which processing unit (PU) the interfacing process is running.

Figure 1 shows the potential benefits of a user-level interface combined with directed cache injection compared with a socket implementation. In the traditional stack-based operation (Figure 1(a)), the I/O device DMAs receive data into main memory (1). It is then fetched by the PU running the TCP/IP process (2), and copied into a user buffer. Next, the consumer process, which usually runs on a different PU, fetches the data into its own cache (3). Cache injection can be used to improve this flow and avoid the detour over memory by injecting the receive data directly into the cache of the PU running the TCP/IP stack (4). This reduces communication latency and avoids one memory transfer (2). Because of cache coherence requirements, the write-back of the data (1) usually cannot be avoided. It is only deferred to a later point in time and effected by the PU. The efficiency of cache injection also depends on whether the line that has to be cast out for an injected one is clean or dirty, and when it is accessed again.

In contrast, a user-level interface with knowledge about where the consumer process is currently running, can inject the data directly into the consumer's cache as shown in Figure 1(b), and thus avoid the transfer between the stack and consumer PU ((3) in Figure 1(a)).

Table 1 summarizes the number of necessary memory transfers for network payload data per cache line (CL) for different payload transfer mechanisms. The case of a stack with injection and buffer reuse (3.) assumes that receive buffers are reused and that the new cache line arrives before the old is written back to memory. This behavior, however, depends on the temporal locality of the receive traffic, the cache size, and the stack implementation.

The transfers shown in Figure 1 and Table 1 only consider the optimal case using cache line-sized transfers. The transfer picture may change significantly if partial stores are used. Especially for cache injection that would bypass main memory, partial stores may annihilate all advantage from cache injection as the remainder of the cache line data needs to be brought into the cache to combine it with the new data. Furthermore, there is a distinct latency and overhead difference between non-aligned, non-cache line-sized (partial) and aligned, cache line-sized write operations in most processor interconnects. Therefore, I/O devices using cache injection need to take into consideration the processor architecture, especially the cache line size, to achieve optimal performance.

Table 1: Memory transfers overview for receive data

Receive processing	Mem write	Mem read
1. Stack DMA	2	1
2. Stack Injection	2	0
3. Stack Injection + buffer reuse	1	0
4. User-level DMA	1	1
5. User-level Injection	1	0

3. SNOOPING CACHE COHERENCE

In snooping-based cache coherent systems, snoop transactions are used to maintain coherence of data, usually based on a cache line granularity. It guarantees that processing units are not working on stale data. Moreover, snoop transactions are used for system management purposes, for example, to maintain coherence of address translations and routing of interrupt requests.

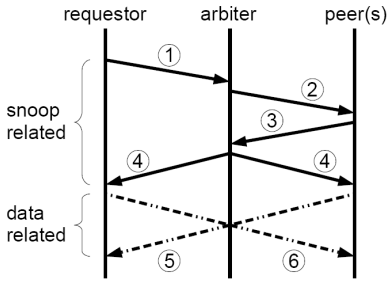


Figure 2: Interconnect transaction

Figure 2 shows the typical sequence of a processor interconnect transaction in a split-transaction protocol. First, the requestor signals the request to the arbiter (1). The snoop request usually contains the unit ID of the requestor, the address of the data referenced and further control information. The arbiter sends the request to all peers (2). Each peer then checks whether it is affected by the request, i.e., if it has the referenced data in its cache or, in case of memory or I/O, if it is responsible for the corresponding address as a last point of coherence. Based on this check and the availability of transfer resources, the peer generates a partial response (3). The arbiter then combines all responses into one and broadcasts the final response (4). Thus, all units within the scope of the transaction are aware of the snoop result. Depending on the request type, that is whether it requires a data transfer, the referenced data is then transferred from the requestor to the destination (6) for a write request or sourced by a peer to the requestor (5) for a read request. The data transfer itself is usually divided into several smaller parts.

The snoop transaction thus serves three purposes: it maintains coherence in the system and, if applicable, determines the counterpart of the data transfer and reserves resources in the form of queue slots.

Figure 3 shows the structure of a typical snooping cache coherent SMP with private caches [8]. Each processing unit has a private L1 and L2 cache. The PUs, the memory controller (M), the I/O controller (IOC) and the processor coupling unit (PCU) are connected through a high-speed interconnect, which is used for both snoop and data transfers.

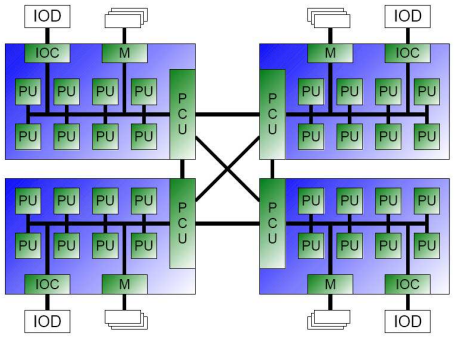


Figure 3: Cache-coherent SMP

Not shown in the figure are victim caches (i.e. L3 and L4) that may be used to offer larger caches with slower access times. The four chips in the figure are coupled together through the processor coupling unit to form the processor fabric (PF). Each unit in the processor fabric has its unique identifier. As shown in Figure 6, this identifier can, for example, consist of a node, chip, and unit field.

To reduce snoop traffic overheads, coherence domains, also called partitions, with a limited scope can be defined. In the presented system, a local snoop would only be sent to any units on the same processor whereas a snoop with global scope would be broadcast to all units.

For requests from I/O devices (IODs), the I/O controller usually needs to use global snoops to guarantee coherence in the fabric. If, however, an I/O device is only interfaced from processing units within the same processor, and only references memory from this processor, it is sufficient to use snoops with a local scope. This restriction to the local domain can reduce the processor fabric traffic that is generated by an I/O device considerably. However, the I/O controller must be configurable to support this mode of operation on a per-device basis, and software needs to guarantee that processes interacting with the device are not running on PUs of a remote coherence domain.

Virtualized I/O devices, on the other hand, can be interfaced directly by any user in the system. Therefore, they always need to use global snoop transactions as long as there is no information available about where the consumer process is running and whether data is cached in a remote processor chip or not.

4. CACHE INJECTION MECHANISMS

For data injection in private cache architectures, there arise two key problems: how to determine the destination for the injection and how to incorporate the injection process into the interconnect protocol. In the following, we present different concepts for these two problems.

In Section 4.1, we show two mechanisms that help in identifying the destination cache for injections. These methods require interaction across the entire system as shown in Figure 4, i.e. from the PU down to the I/O device. Section 4.2 presents concepts for incorporating cache injection in the protocol of a snooping-based cache-coherent interconnect and how to keep its overhead low. We further propose a concept for keeping track of and handling injected cache lines within a cache in Section 4.3. Section 4.4 shows an application that combines those three concepts for efficient

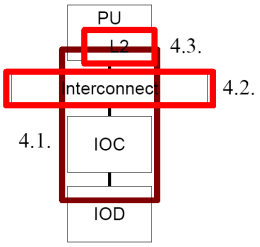


Figure 4: Proposed methods

in-system doorbell processing.

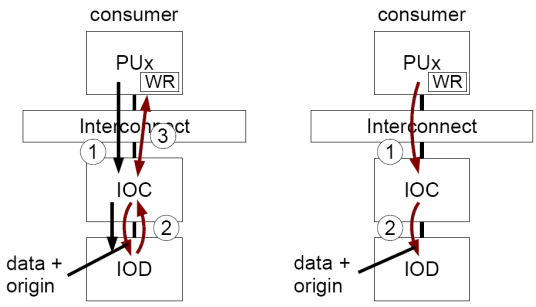
4.1 Identifying the right cache

4.1.1 Processor fabric knowledge forwarding

Our first solution for determining the destination of an injection is to allow the I/O device to gather originator information, i.e. the unit identifier of the requestor in the processor fabric, when a work request is posted. The device can use and forward this knowledge later when data is injected into the system.

There are two possible implementations:

1. The first is initiated by the I/O device as shown in Figure 5(a). In work request-oriented interfacing, the processing unit issues a doorbell request to the I/O device (1). This indicates the availability of a new receive work request. The I/O device fetches the work request, usually comprised in one cache line, and indicates in the DMA read request (2) that it wants to obtain more information on the origin of the data. The origin can be identified by the I/O controller as this information is included in the transfer over the processor interconnect (3). If the I/O device fetches the work request soon enough after it has been issued, the originator is still the processing unit in which the work request was generated.



(a) DMA read (b) MMIO write

Figure 5: PF knowledge forwarding

It can also be beneficial to provide the I/O device with the possibility to request information about the location of a cache line only, i.e. without the need to actually fetch the data. In work request-oriented communication schemes, the consumer frequently posts receive work requests that are fetched from the I/O device only at a later point in time. Thus, at this later point, the cache line has very probably already been displaced in the system, e.g. moved to a victim cache. In this case, the originator information is no longer

of value for the I/O device. A data-less transfer would allow the device to capture the originator information without the need to actually transfer the data. The I/O controller furthermore needs to make a pre-selection on the forwarded originator information.

2. The second possibility for processor fabric knowledge forwarding, shown in Figure 5(b), is to allow the user itself to trigger the forwarding. An MMIO write request is therefore tagged in a special way in the transfer on the processor interconnect (1). When the I/O controller detects the presence of such a transfer, it can automatically include fabric information in the transfer of the data to the I/O device. Preferably, the originator information would not be contained in every doorbell as this would inflict an overhead on the external link as well as in the I/O device. Not providing the information often enough will, on the other hand, reduce the accuracy of the information. The process issuing the work request should therefore keep track of whether it was moved to a different processing unit since it last provided originator information, and request the originator information inclusion accordingly.

Both mechanisms described provide information to the device about where data came from without requiring the device to be able to understand the information it is provided with. However, the device needs to get the originator information in order to make the mechanism scalable. Ordinary non-virtualized devices interact mainly with one process in the system, usually the software driver of the device. Virtualized devices, on the other hand, may provide millions of isolated user interfaces. The I/O controller, no matter whether provided with virtualization features, defined for example for PCI Express [12], cannot be implemented in a way such that it allows efficient directed cache injection for this kind of devices. The scalability therefore needs to be provided by the device itself, and necessitates passing on originator information in one of the ways described above.

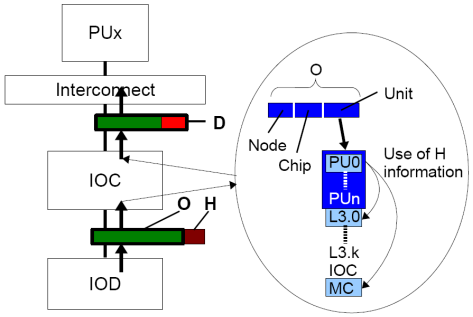


Figure 6: Destination translation

When a device wants to store data, it now knows where the destination process is/was recently running in the system. The device can thus provide this information (O) with the store request in order to indicate where the data is aimed at. Furthermore, as shown in Figure 6, it can also add supplemental information on a more abstract level in the form of a hierarchy information (H). Thus, the device does not require any knowledge about the architecture of the processor fabric. It merely provides the exact originator information and adds an abstract hierarchy information, for example, based on an indication included by software in the associated work request. The I/O controller is then responsible for combining the provided information based on the processor fabric architecture. The hierarchy information can include information about the exact cache level the data shall go into, e.g. L2, L3 or memory, as well as information related to alternative destinations to allow/disallow degradation of the indicated cache level, or inhibit use of system memory. Furthermore, the I/O controller of a system that does not have a requested cache level can determine an adequate destination that should be used instead. Therefore this injection mechanism is independent of the system architecture and can be used for standardized protocols that are used in different processor architectures, such as PCI Express.

Originator information could also be maintained by the hypervisor running in the system. This approach has the disadvantage that the hypervisor has to update the information for all affected connections explicitly every time there is a change. This obviously inflicts lots of overhead. Therefore we think it is more advantageous to transfer the management of this task to hardware or the user himself.

4.1.2 *Destination discovery*

The mechanism described in the preceding section is based on using originator information. The opposite mechanism for directing cache injection would thus be letting the device specify the destination for an injection.

Especially in virtualized systems it is interesting to use a dynamic mechanism for cache injection based on the processes currently running in the system. As mentioned above, keeping the connection contexts in a virtualized I/O device up-to-date can be a tedious task, inflicting considerable management overhead for the hypervisor. It is much easier to merely specify the destination for injections and make the injection decision based on the run state of the receiving process using hardware mechanisms.

Usually, snoop transactions maintain coherence in the processor fabric. We can, however, also imagine using them for different operations. For dynamic cache injection purposes a new request type as shown in Figure 7 can be specified. This request would then contain a destination ID instead of the address usually used for data transfers. The process-specific ID is used to detect whether and where a process is running in the system.

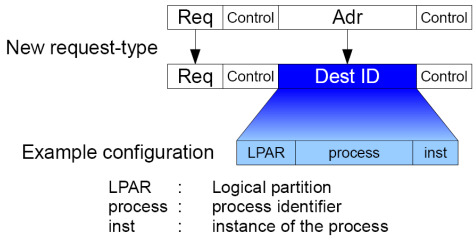


Figure 7: Snoop structure

The destination ID value can be configured according to the hardware architecture. In a multi-core system with hardware virtualization it would contain a field indicating the logical partition or virtual machine the injection is destined for, a field indicating the process, and a field giving the instance of the process. The process ID can be the same as the process ID used by the operating system. As not all

processes use user-level interfaces and the number of bits available for the destination ID is limited by the physical address size, the process ID can also be managed separately.

Having run information it is possible to make a good decision on whether and where to inject data. On the one hand, it is not very efficient to inject data for a process whose LPAR is currently not running. On the other hand, if the LPAR is running but the process is not, it may be better to inject into a victim cache (L3) rather than the L2 cache of a PU, also depending of course on their respective size and whether the destination process is waiting to be woken up upon reception of the data.

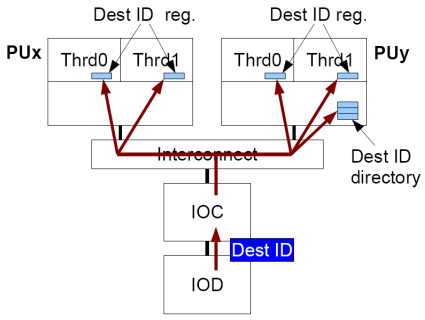


Figure 8: Process snoop schematic

For the destination discovery snoop transaction, shown in Figure 8, the I/O controller receives the destination ID from the I/O device along with the data. The destination ID usually stays the same for the entire lifetime of a network connection and is stored in the connection context in the device. The I/O controller then issues a snoop request with the destination ID and global scope to the processor fabric. All processing units in the system therefore implement a local register for each of their hardware threads. These registers hold the ID of the process currently running, and are thus part of the processor context that needs to be saved on a context switch. The processing units then compare the destination ID of the incoming snoop request with the value contained in the register(s) and generate a response depending on the matching degree. For the example configuration in Figure 7, they generate different partial responses depending on whether there is an "LPAR", an "LPAR+function", or an "LPAR+function+instance" match. As in the normal snoop process, each processing unit returns its partial response, which is then combined by the arbiter into a single final response. The priority used for combining the responses is inverse to the sequence given above, an "LPAR+function+instance" match having highest, and a simple "LPAR" match having the lowest priority. Using this mechanism, the I/O controller is now able to inject the received data into the most appropriate cache in the system.

To increase the preciseness of injections, a directory of scheduled but not running processes can be added to the PUs, as shown in Figure 8. Therefore, the location of a destination process can be determined even if it is not being executed at the moment of the injection request.

The instance field enables load spreading for processes that use a varying number of serving threads in many-to-few relationships. This can, for example, be the case for a TCP/IP process serving many connections. In a normal load situation, it may use two threads to process the network

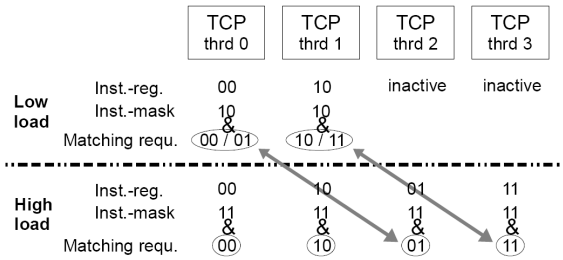


Figure 9: Dynamic load spreading

traffic, whereas, under high load, more threads are started as shown in the upper and lower part of Figure 9.

Instead of changing the configuration in the device for half of the connections to make best use of the available number of threads, the instance identifier is used to distribute the load among the threads. It is initialized when a connection is initialized. The available instance identifiers may, e.g., be evenly distributed over all connections or reflect different priority levels. When there are fewer threads running than there are instance identifiers assigned in the device, the requests need to be mapped to the number of threads available. This can easily be implemented by applying a mask or compare operation to the instance identifier part of an injection request. In the upper part of the example in Figure 9, only two TCP threads are running in the system, but a 2-bit instance identifier is used. Therefore, each thread processes the requests of two instances. Data of injection requests with an instance value of "00" and "01" are injected into the cache of thread 0, whereas those using "10" and "11" are injected into the cache of thread 1. If more threads are scheduled, the load can be distributed to all of them by adapting the mask value as shown in the lower part of Figure 9.

4.2 The Injection process

The mechanisms described so far help determine where data should be injected. However, we not only need to decide where to inject data, but also to invalidate stale data in the system and reserve queue slots in the destination unit of the transfer. As injection requests are much less frequent than read and write operations from the PU, the additional cost for an injection queue is not justifiable in most cases. Therefore, our approach would be to design the snoop protocol such that it can reuse read queue slots for cache line injections.

4.2.1 Two-step injection process

For injecting data based on the mode of operation of a snooping-based fabric and without necessitating an increased size for snoop requests, we propose a two-step mechanism as shown in Figure 10. The first step is a data-less, short snoop transfer, the second a transfer including the injection data. In the first step, the inject snoop (1), the destination for the injection is determined based on one of the mechanisms described above. This snoop transaction is used to notify the injection destination that the data of the subsequent transfer is destined for it. If there is no dedicated injection queue, the destination unit can already initiate the reservation of a read queue slot during this phase, as well

as effect necessary castouts to avoid retries during the data transfer. After determination of the destination, a second request (2) is sent with the address of the data, which (i) invalidates all copies of the data in system caches and (ii) is used to transfer the data to the destination unit.

If this two-step process is used for every cache line, cache injection inflicts overhead on the processor interconnect. Compared with a DMA-based operation, it requires one additional data-less snoop transaction (1), but in turn accesses memory only once to write back the receive data to memory (3). As this memory access happens during the normal castout operation, the burstiness of I/O traffic that is observed by the memory subsystem is alleviated to some extent. However, if the data is not placed in the correct cache during the data transfer (2), another additional snoop transaction compared with the DMA-based operation is needed.

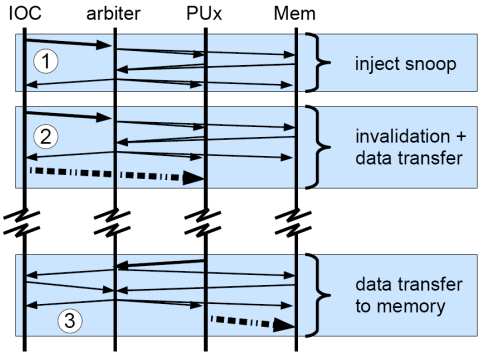


Figure 10: Data transfer sequence using injection

We can exploit the distinct characteristics of I/O traffic, especially network traffic, to reduce the overhead: network traffic usually exhibits very good spatial locality. This means that the data to be injected usually spans a consecutive address range that is larger than one cache line. Bringing data into the processor fabric intelligently should therefore also take advantage of this fact by using block invalidations and block transfers. A block transfer in this sense is a transfer of several consecutive cache lines over the processor interconnect with a reduced need of additional snoop requests.

The second snoop transaction in the scheme shown in Figure 10, which includes the invalidation request, therefore not only invalidates the first cache line of the transfer, but also all consecutive cache lines used by the entire payload transfer. For a maximum transfer unit-sized (MTU) data transfer, this can typically be 12 cache lines for TCP/IP and 32 for InfiniBand for a cache line size of 128 bytes. Thus, invalidations for every single cache line are avoided.

Unfortunately, the second snoop request is also used to ensure that either an injection or a read queue slot in the destination cache is reserved for the data transfer. Therefore, those snoop transactions are still needed for reservation purposes.

4.2.2 *Buffering at the partition boundaries*

To reduce the number of global snoops for block transfers, a buffer in the processor coupling unit can be used such that local snoop transactions are sufficient to get the data into the cache of the destination unit.

For injections using multi-cache line invalidation, the invalidation and data transfer request of the first CL ((2) in Figure 10) can, at the same time, be used to reserve buffer space in the remote processor coupling unit (rPCU) of the destination unit. Therefore, the destination PU signals how many read queue slots it was able to reserve (2) in the response to the multi-cache line invalidation request. The processor coupling unit then checks its buffer resources, reserves the remaining number of cache lines in its buffer (+4), and signals back to the requesting I/O controller the total number of cache lines that can be transferred without the need for further snoop transactions (6). The first two cache lines of the transfer are therefore directly forwarded to the destination PU (A), whereas the others are temporarily stored in the PCU buffer (B). Meanwhile, the destination PU schedules further read slots and fetches the data from the buffer using local transfers (C) until the data for the entire injection has been transferred. To retain coherence during the injection process, the coupling unit fences off all requests for data that has been invalidated and not yet arrived from the I/O device.

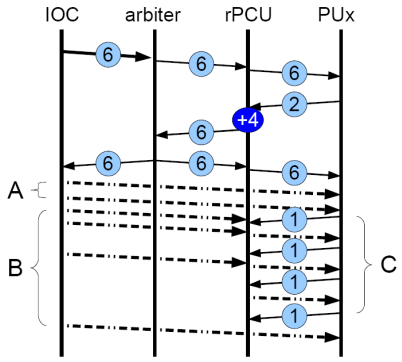


Figure 11: Multi-cache line invalidation and data transfer transaction ((2) in Figure 10) using CL buffering

Especially in High Performance Computing (HPC), consumers frequently poll the last cache line of the expected receive data to start processing immediately after it is received. To avoid that they work on stale data, the last receive data store therefore has to be kept in order. The same is true for a completion notification, which has to be issued only after all payload writes have been finished.

On the external bus and thus also in the I/O controller, on the other hand, the stores are related only by relatively simple store ordering mechanisms, based on traffic classes or tags. Especially for payload data, the available ordering rules are too strict as all but the last received payload cache lines usually have no dependency and can be stored unordered if contention is encountered by some stores.

Using multi-cache line transfer support in the processor, which fences off read requests to invalid data, store ordering problems for payload data can be avoided completely.

Therefore, if more of the request affiliation knowledge is passed on to the I/O controller, the latter can optimize coherence transactions and resource reservation in the processor fabric much more efficiently. This would be possible with the maximum payload size (MPS) specified in PCI Express. The problem is however that most root complexes do not implement enough buffer space to allow transfers of

complete payload data chunks with one request. Thus, it is again not possible to take full advantage of the complete transfer knowledge.

Instead of for cache injection, the PCU buffer can also be used for efficient remote memory stores. The problem that arises in this context is the increasing pressure on the memory subsystem due to the increasing number of PUs in a processor. Together with the bursty nature of I/O traffic, larger network transfers that are stored into consecutive memory addresses are therefore increasingly susceptible to experience backpressure in the form of retry responses from memory controllers because of overflowing store queues. To prevent the backpressure from propagating over the external bus to the I/O device, the proposed PCU buffer can thus be used for memory stores in the same way as was described above for cache injection.

4.3 Controlling cache injection

Optimal for cache injection would be a method that allows dividing a cache into I(nstruction)/D(ata)-cache and injection cache (e.g. at least 80% I/D and max. 20% injections). This division could effectively limit cache thrashing. However, most caches are N-way set-associative and thus divided into congruence classes (sets) with equal numbers (ways) of cache lines, of which only one set can be used, based on the memory address.

Therefore a mechanism is desirable in the caches that allows the amount of data injected to be controlled. Taking into account the cache architecture, cache injection can be limited on a cache and/or set level. To be able to check whether the limits are reached, however, we need to keep track of the number of cache lines injected.

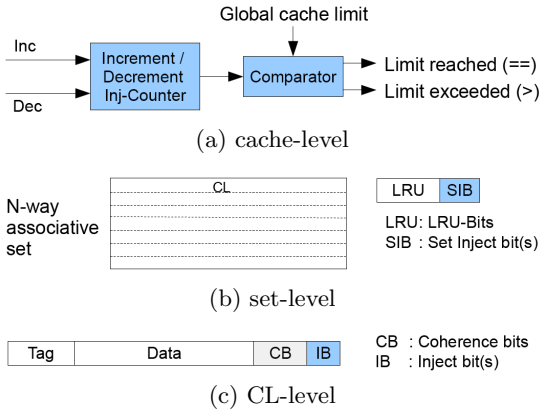


Figure 12: Cache architecture enhancements

Accordingly, injected cache lines need to be marked in the cache. This mechanism is shown in Figure 12(c). A cache line is stored in a cache basically with two identifiers, the tag/address, and the state of the cache line in the system, indicated by the coherence bits. To indicate injected cache lines either an explicit bit can be added to the available coherence bits (Inject bit(s) (IB)), or a new state coded with the available coherence bits can be used. Injected cache lines are, by definition, modified and exclusively owned by the cache. Therefore one new state coded with the available coherence bits would be sufficient. Adding one bit can however offer the possibility to use more sophisticated methods for handling injected cache lines by storing additional information.

The reused coherence bits can be used to distinguish the nature of injected data, e.g., to differentiate between header and payload data. If a device wants to use system caches to cache some of its data (e.g. contexts), also this data can be specially tagged. However, the characterization of the data needs to be supplied by the originator of the injected data, i.e. usually the I/O device. For receive data buffers, it may also be interesting to specify that it will not be tagged as dirty after it has been read by a PU because then the payload has been copied into a user buffer and is no longer needed. The buffer space can then be seen as a scratch-pad region that, under normal circumstances, does not consume memory bandwidth. Therefore it helps in making case (3) in Table 1 the normal case for receive stack memory transfers.

A cache line conserves its inject state while it remains in the cache and is not touched/read by a processor. Its state can also be inherited by victim caches after castout from the cache that was originally used. However, it is deleted when the cache line ends up in main memory or a processor, possibly also a remote one, reads the cache line. Then the coherence bits are set according to the new state of the cache line. Counters can be used to track the ratio of used to unused injected data to allow for more dynamic control over the injection process.

The information on injected cache lines can also be taken into consideration for the castout and LRU algorithm. Cache lines marked as injected can e.g. be included in or excluded from the normal castout selection, depending, for example, on the spatial locality of the code currently being executed. On a read from the PU, different actions may be taken regarding the LRU position of the cache line. The LRU can be updated as on a normal cache miss so that the cache line will be at the end of the (pseudo-)LRU queue, the LRU can be left unaltered, or the CL can even be moved to the front of the queue [13].

To limit the total number of injected CLs, a global counter as shown in Figure 12(a) can be used per cache to be able to set a global inject limit for the respective cache. The counter is incremented (inc) on injection of a CL, and decremented (dec) when an injected CL is touched by a PU or cast out of the cache without being replaced by a new injected CL. Injection requests are thus only accepted as long as the injection limit has not been reached.

On a per-set basis, a set injection bit can be stored along with the LRU information as shown in Figure 12(b). This per-set injection bit indicates that the maximum of injected cache lines configured for the cache has been reached for the according set. This enables a fast look-up for snoop operations to determine whether an injection should be allowed. It only needs to be updated when a cache line is injected or unmarked.

Two policies can be defined for inject requests if a limit has been reached: either an older inject-marked CL is cast out in favor of the new line, or the new one receives a “retry”/“try other cache” response.

The lookups should already be made during the first step of the injection process, i.e. the inject snoop phase shown in Figure 10. This helps in avoiding late “retry”/“try other cache” responses during the data transfer phase. Therefore the inject snoop needs to already contain the necessary address information to check whether the addressed cache set

has space left. As the number of sets may differ among the caches of a system, the address part needs to be large enough so that the cache with the largest number of sets can determine the set the data will be stored into.

The global and per-set injection limits can be combined to achieve more precise control over how the cache is used for injected cache lines. Figure 13 exemplarily shows how different settings for the two limits can influence the number of injected cache lines. Setting a global limit only may have destructive effects as it allows single sets to be completely used for injected cache lines (see set 7). Only setting a small per-set limit, on the other hand, may prevent reaping all the benefits of cache injection if the injected data is poorly distributed among the sets.

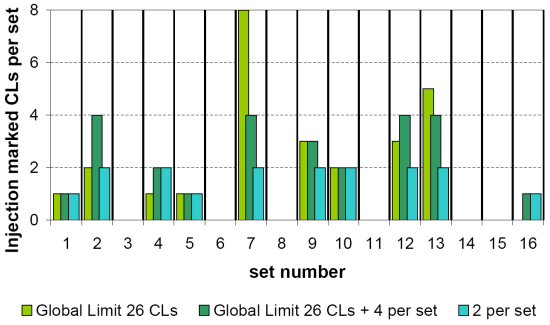


Figure 13: Injected CL distribution examples

In a system with a variety of caches, either attached to PUs or in the form of victim caches, injection control can be configured for each cache individually to best match their respective needs. The cache of a PU used for TCP/IP processing can therefore take many injected CLs, whereas that of a PU running calculation-intensive tasks is configured to accept only a very limited number. Furthermore, with multiple levels of caches, victim caches may be configured to accept more inject data than higher-level caches.

4.4 Putting it all together

Apart from bringing I/O data into the system efficiently, cache injection can also be leveraged for in-system purposes.

Today's systems offer more and more cores to increase the performance while the single-core performance almost stays the same. Still it is not yet clear how to use the new resources efficiently. One very popular approach to profit from the increasing number of cores is onloading tasks that were formerly offloaded to dedicated devices, e.g. network packet processing. To cope with the increasing packet processing needs for higher link speeds, concepts have been developed that profit of the use of dedicated cores for network packet processing. The biggest drawback of those concepts is that the doorbell mechanism can so far not be implemented efficiently in software.

In most cases, either a very expensive mechanism using system-calls or constant polling on notification cache lines is used. Polling is however inefficient in terms of processing unit usage and power consumption and, at the same time, inefficient if the number of consumers is large. Some processors today offer the possibility to wait on a CL modification of the local cache and use it as a wakeup instruction. This mechanism can be especially efficient in multi-threaded processing units. On the other hand it can only be used for a

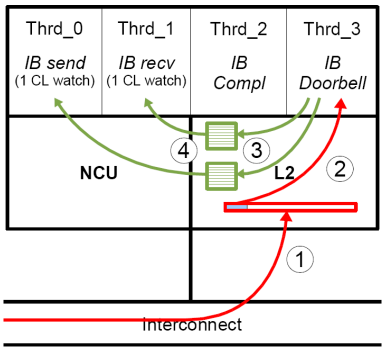


Figure 14: InfiniBand onloading using work request injection

single cache line. For doorbell processing, the serving process would thus be limited to using a single doorbell cache line, which is not practical for protected multi-user serving, or it has to know exactly where the next request will arrive, which is again not possible in many-to-one communication situations.

Therefore, an efficient doorbell and notification mechanism for in-system use is desirable. The requirement for such a mechanism is that it allows monitoring hundreds or thousands of cache lines simultaneously. The only way to achieve this scalability is a user-driven scheme: instead of explicit doorbell address monitoring by the server, the user therefore signals new requests using cache injection. An application where this method can be used very efficiently is onloading of InfiniBand packet processing. This example, illustrated in Figure 14 is a typical N-to-1 communication. The serving process is the IB doorbell process and the consumers are processes that want to send or receive data.

Using in-system notification, the consumers can now inject their cache line-sized requests directly into the cache of the server processing unit without any operating system interaction using one of the two injection concepts proposed in Section 4.1, and the two-step injection process described in Section 4.2.1. While there is no work to be done, all threads in the serving processing unit are sleeping to save power. On injection of a work request into the cache (1), the cache line is marked as injected and containing a notification, and subsequently presented to the doorbell thread (2). The doorbell thread analyzes the first part of the data of the request that is presented to it and attaches the request to the correct queue for send or receive operation (3). The cache line is then unmarked and, if present, the next request can be handled by the scheduler. The send process watches the head of the send queue and processes the request (4). It therefore only needs to monitor 1 CL, as already possible in today's processors. At the same time, there is no cache miss for the request as it usually still resides in the cache. Therefore there is also no latency and no further overhead inferred on the processor interconnect.

The fundamentally new concept in this approach is the presentation of the notification to the destination thread or threads. It is shown in more detail in Figure 15. Two presentation modes are possible depending on the mode of operation of the different threads. If a thread is running in user-mode, i.e. it uses virtual addressing, the first part of

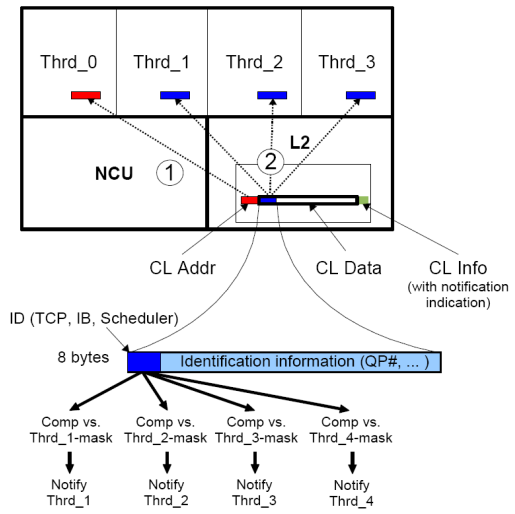


Figure 15: Thread notification

the cache line (for example 8 bytes) is presented to the processor in a read-only register (2). If a thread is running in real-mode, that is it can access physical addresses, it is also possible to present the physical address (1) of the cache line containing the notification information.

To reduce the number of threads the injected data is presented to, a pre-selection mechanism can be implemented, which is for example based on some of the bits of the presented information in order to differentiate between notification types. Possible notifications are then encoded by software and can include notifications for TCP/IP, InfiniBand or the OS scheduler. Using different masks for the different threads, representing the type of application that is currently running on a thread, the presentation of the notification data can be limited to threads that are possible destinations for the notification.

Alternatively, the thread affiliation of a notification might already be determinable during the destination processing unit injection phase based on a matching destination ID register. This information can then be stored along with the cache line and the notification be limited to the actual destination thread.

The presented notification data should allow the thread to clearly determine the cause of the notification. For example, in InfiniBand and using this mechanism as doorbell mechanism, the presented part of the notification would include the queue pair number a work request is attached to. This actually complies with the format of the work-requests used in today's InfiniBand implementations. Using the information of the notification, the destination thread can then access the notification using virtual addresses. For threads running in user-mode this is essential as it is not possible to translate the physical address of the notification cache line back into a virtual address that would be needed by such a destination thread to finally access the notification. For threads running in real mode, however, presentation of the physical address of the injected cache line is an option as the thread can also use this information directly. In some cases it might however be favorable to present the first part of the notification to the destination thread instead of the physical address as the cache line then does not necessarily need to be brought into the L1 cache of the processing unit at once, but it can be fetched later-on when the rest of the notification data is processed.

A cache line marked as notification is unmarked either when it is accessed by a thread or it can be unmarked using a special instruction/a write to the read-only register. It is then not necessary to actually access the cache line.

A cache can be parsed efficiently for notification cache lines if more than one cache line marked as notification is present in a cache. Parsing for those cache lines should be in reverse order of the congruence classes. Therefore, consecutive notifications of one requestor that will, in the normal case, hit consecutive congruence classes are not served back-to-back, increasing fairness for the serving of requests.

A major advantage of this new concept is further the possibility to avoid system calls for notifications in asynchronous communication models. System calls are one of the main latencies in TCP/IP processing [7]. Therefore there is great interest at the moment in reducing the need of those. Furthermore, notifications can be processed quickly as cache misses for requests or notifications are avoided as they are at the same time used as doorbells. The main advantage over today's methods however is the possibility to monitor an unlimited number of notifications while the number of simultaneous notifications is only limited by the cache size and architecture (set-associativity). Last but not least, cache line notifications allow every processing unit in the system to act efficiently as an N-to-1 server, rendering specialized units unnecessary.

The notification scheme can also be used for I/O to scheduler interaction. In many network-dependant applications a process is descheduled when waiting for network data and rescheduled when the data has arrived. Therefore, using the presented method, a cache line can be defined that will be written by the I/O device if data was received for the according process such that it has to be woken up. The device injects a notification with an indication that the notification is addressed to the scheduler. Every time the scheduler is called, it checks if there are outstanding notifications by simply checking if there is any data presented in the according register. If there was a wakeup notification, the according process can be woken up and considered for scheduling. Otherwise scheduling is continued as normal. The advantage of this mechanism is that it supersedes the need for interrupts and thus avoids interrupt overhead and direct OS interaction. Nevertheless, any process that is woken up by receive data can be considered for scheduling during the next scheduling cycle.

5. RELATED WORK

Different mechanisms for cache injection have been studied so far.

The first and easiest way for cache injection is an update-based cache coherence protocol. Actually this mechanism would be suited best for user-level interfaces as the user can easily get the data into its local cache [11]. There are however also two major disadvantages. First, the user has to touch the cache lines to be updated and therefore potentially thrashes data that would be needed at an earlier point in time for processing than the expected inject data. Second, if the data the user wants to be injected into the cache does not arrive in a timely manner, i.e., it does not have sufficient

temporal locality, the CLs will probably have already been cast out, and no update will take place.

IRQ-directed [1] cache injection is mainly applicable for centralized TCP/IP processing stacks where one or more processing units in the system are dedicated to processing of incoming TCP/IP packets. The processing unit receiving the interrupt is thus also the one that will work on the data.

Finally, static cache injection [9], i.e. cache injection into a predefined cache is well suited if an I/O device is served by one process and the process is pinned to a PU. The work by Leon *et al.* nevertheless shows how cache injection can improve application performance and, at the same time, help reduce memory bandwidth requirements.

The methods presented above are not well suited to support directed cache injection for user-level interfaces in a private cache environment.

The work of Huggahalli *et al.* [4, 6] is the current state-of-the-art in terms of cache injection implementation in a commercial processor. This mechanism called Direct Cache Access (DCA) is currently implemented as a hardware-initiated prefetching assist. Therefore, the I/O device makes use of the transfer tag in DMA write transactions to use it as a hint for the destination of data injection. PCIe Gen3 will also include such a hint mechanism in its specification. So far, DCA hints are only used by Intel NICs and its full functionality is slowly released into general availability. The drawback of the current implementation is that it still writes the data to main memory first and therefore does not make full use of the potential of cache injection. Recent patent disclosures [16, 14] however show that Intel is actively working on cache injection with memory bypass. It will then be combined with consumer-controlled write-back avoidance using a cache line invalidation command, enabling scratch pad-like use of cache space for TCP/IP receive data buffering.

For injection control, way limitation has been proposed [6, 15]. Its disadvantage is its inability to exploit the possibility of reducing cache thrashing by replacing invalid cache lines present in a way other than the dedicated one. It also allows for more flexible distribution of the injected data over cache sets. We therefore believe that our proposed tagging method offers more flexibility for controlling and limiting cache injection.

6. SUMMARY

In this paper we present different aspects of and architectural concepts for cache injection in private cache architectures, especially with respect to the requirements of user-level interfaces. As the number of cores, virtualization requirements, and dependence on network traffic increases, cache injection will become a key enabler to allow low latency communication and to reduce the pressure on the memory subsystem. It can be used to reduce memory bandwidth requirements for both stack-based and user-level interfaces. At the same time, device requirements, processor fabric particularities, and effects on the caches have to be taken into account. Therefore we define three major requirements for cache injection from a processor perspective:

Directing mechanisms They need to support a large number of private interfaces and private caches without incurring additional management cost for the hypervisor.

Bulk data transfer support For efficient use of cache injection, directing mechanisms need to be combined with processor interconnect bulk transfers. With the increasing dependence on network traffic [5], more attention needs to be paid to the requirements of bulk data transfers for network traffic in the design of cache coherence protocols.

Injection control It is needed in the caches to be able to adapt the system to different workloads.

To analyze the effects, possibilities and pitfalls of cache injection, not only a simulator for a system of adequate size is needed but, more importantly, real-world applications and workloads have to be analyzed to evaluate the potential benefits. This is especially important as efficient cache injection heavily depends on precise indications from the consumer process, i.e. an efficient HW/SW interaction.

7. REFERENCES

[1] P. J. Bohrer, R. Rajamony, and H. Shafi. Method and Apparatus for accelerating Input/Output Processing using Cache Injections. *United States Patent 6,711,650 B1*, March 2004.

[2] Compaq, Intel, and M. Corporation. *Virtual Interface Architecture Specification, Version 1.0*. December 1997.

[3] F. Dahlgren. Boosting the Performance of Hybrid Snooping Cache Protocols. *ACM SIGARCH Computer Architecture News*, 23(2):60–69, 1995.

[4] R. Huggahalli, R. Iyer, and S. Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the 32nd Annual Int'l Symposium on Computer Architecture (ISCA)*, pages 50–59, June 2005.

[5] M. Ko, D. Eisenhauer, and R. Recio. A Case for Convergence Enhanced Ethernet: Requirements and Applications. In *Proceesings of the 44th Annual Int'l Conference on Cummunications*, pages 5702–5707, May 2008.

[6] A. Kumar and R. Huggahalli. Impact of Cache Coherence Protocols on the Processing of Network Traffic. In *Proceedings of the 40th Annual Int'l Symposium on Microarchitecture (MICRO)*, pages 161–171, December 2007.

[7] S. Larsen, P. Sarangam, and R. Huggahalli. Architectural breakdown of end-to-end latency in a tcp/ip network. In *Proceedings of the 19th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 195–202, October 2007.

[8] H. Q. Le, W. J. Starke, S. J. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 Microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.

[9] E. Leon and A. Maccabe. Reducing the Impact of the Memory Wall for I/O Using Cache Injection. In *Proceedings of the 15th Annual Symposium on High Performance Interconnects (HOTI)*, pages 143–150, August 2007.

[10] M. R. Marty and M. D. Hill. Virtual Hierarchies to support Server Consolidation. In *Proceedings of the*

34th Annual Int'l Symposium on Computer Architecture (ISCA), pages 46–56, 2007.

[11] S. S. Mukherjee and M. D. Hill. Making Network Interfaces Less Peripheral. *IEEE Computer*, 31(10):70–76, 1998.

[12] PCI-SIG. *Single Root I/O Virtualization and Sharing Specification Revision 1.0*. September 2007.

[13] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual Int'l Symposium on Computer Architecture (ISCA)*, pages 381–391, June 2007.

[14] D. Srivastava and J. Gilbert. Direct cache access in multiple core processors. *United States Patent 7,555,594 B2*, June 2009.

[15] D. Tang, Y. Bao, W. Hu, and M. Chen. Dma cache: Using on-chip storage to architecturally separate i/o data from cpu data for improving i/o performance. In *Proceedings of the 19th Int'l Symposium on High-Performance Computer Architecture (HPCA)*, January 2010.

[16] A. Vasudevan, S. Sen, P. Sarangam, and R. Huggahalli. Performing Direct Data Transactions wit a Cache Memory. *United States Patent Application 11/823,519*, June 2007.

[17] B. Veal and A. Foong. Performance Scalability of a Multi-Core Web Server. In *Proceedings of the Third Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 57–66, December 2007.