RZ 3759 (# 99769) 03/20/2009
Electrical Engineering 9 pages

Research Report

Architecture and Implementation of an MMU on a Server-Class Infiniband HCA

A. Doering

*IBM Research – Zurich
8803 Rüschlikon
Switzerland

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies (e.g., payment of royalties). Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.

Research
Almaden • Austin • Beijing • Delhi • Haifa • T.J. Watson • Tokyo • Zurich

Architecture and Implementation of an MMU on a Server-class Infiniband HCA

Andreas C. Döring

ado @ zurich.ibm.com
IBM Research GmbH, Zurich Research Laboratory
Säumerstrasse 4
CH-8803 Rüschlikon, Switzerland

Abstract. IBM's System z®and System p® servers use Infiniband® for IO, clustering, and coupling. The recent families z10^{TM} and POWER6^{TM} use several chips in common, including a second-generation two-port Double Data Rate (DDR) 12x Infiniband Host Channel Adapter(HCA). This chip is manufactured in 90-nm technology and implements the InfiniBand standard and a proprietary protocol fully in hardware. The required address translation is implemented in an on-chip Memory Management Unit (MMU). The architecture, logic design, and verification of this unit are described in this paper.

1 Introduction

The Infiniband standard [1] is the result of the combined effort of two initiatives, Futurebus and Next Generation IO, which both inherit the concept of direct communication of an application with IO hardware in the Virtual Interface Architecture. Infiniband-based products are successful as cluster interconnects and for interconnecting servers as well as with storage products in the data center. The high performance is achieved by the use of data exchange through shared memory regions and triggering actions in the hardware by writing to so-called doorbell registers by the applications directly. Sharing memory between a virtual-addressing application and CPU-independent hardware components requires the implementation of address protection and translation mechanisms similar to those found in the processor. A unit implementing this is called a Memory Management Unit (MMU). For the high data rate required by clustering and IO in multiprocessor servers, a high rate of address checks and translations needs to be provided. High-end servers require logic partitioning. This impacts the MMU architecture, because memory needs to be managed on three levels, namely the application, the guest operating system and the hypervisor. Furthermore, the reliability expectations for servers require strong protection against hardware and software errors. To reduce design cost, the described Infiband HCA is used in two IBM server families, System z (Mainframes) and System p (POWER-6 based UNIX ®servers) [2]. Both families have common requirements with respect to performance and quality, but also specific requirements for each individual family. This paper describes the micro-architecture

and logic design of such an MMU. The chip is implemented in IBM's 90-nm copper technology, mainly using a standard cell library design style.

The microarchitecture of MMUs in the IO domain has not been published so far, even though the knowledge of an MMU's architecture, and operation, in particular related to caching has a considerable impact on system performance. Performance studies using Infiniband so far only could speculate about these details.

It is assumed that the reader is familiar with Infiniband, as we can give here only a very short introduction.

Infiniband uses queues to communicate between the network hardware and the application. After setup, the operating system is not needed for sending or receiving data. Data is sent by placing a so called Work Queue Element (WQE) into one of many send queues. The send queues are organized as contiguous memory region in the virtual address space, where for reading and writing the address is wrapped at the end to the beginning. To trigger sending, the application writes the number of new WQEs in the send queue into a register of the hardware device, the so called doorbell. By address mapping in the host processor each application has a separate doorbell register. The fifos may not be contiguous in real memory, but by using data types such as linked lists, the MMU presented here is not needed for queue access. The WQEs contain information about the packet, such as operation, destination node and queue, and one or several descriptors of data segments that contain the payload of the packet. Each data descriptor contains a virtual start address, length and a key. This memory key is obtained once by registering a memory region with the Infiniband device for use for sending or receiving. The registration "pins" the memory (inhibits swapping it out) and builds the address translation data structures described below. Receiving works similarly, a WQE can be used to determine where received data should be stored. For RDMA (Remote Direct Memory Access) operations the data descriptors are provided by the send side and do not come from a receive WQE. In summary, the more and smaller the data segments are the more address translation have to be carried out. The system design target of the described Infiniband HCA chip was to make maximum use of the available memory access bandwidth. Using 4K data segments this results in 1.75 million address translations for send and for receive traffic.

At the time when the high-level design was done no information on expected workloads was available. Even now, we have only limited insight how the requirements of important applications are for the MMU.

For the descriptions of data structures and algorithms a C-like pseudo-code is used.

In Section 2 the data structures and architected operation of the MMU are outlined. Section 3 describes the detailed design and verification approach. In the final section, lessons learned from the project are summarized.

2 Address Protection and Translation

Infiniband in a partitioned server requires three levels of protection management, namely, the hypervisor, the guest operating system, and the application. This is reflected in the data structure and operation of the MMU illustrated in Fig. 1: On the hypervisor level, an on-chip memory contains a set of registers. These registers describe a set of pages by providing the page address, error state, and owning partition. Each guest OS manages the keys contained in the pages they receive from the hypervisor. Each page contains the information for 64 keys. The keys can be used as R-keys or L-keys (Remote/Local), for memory regions or windows.

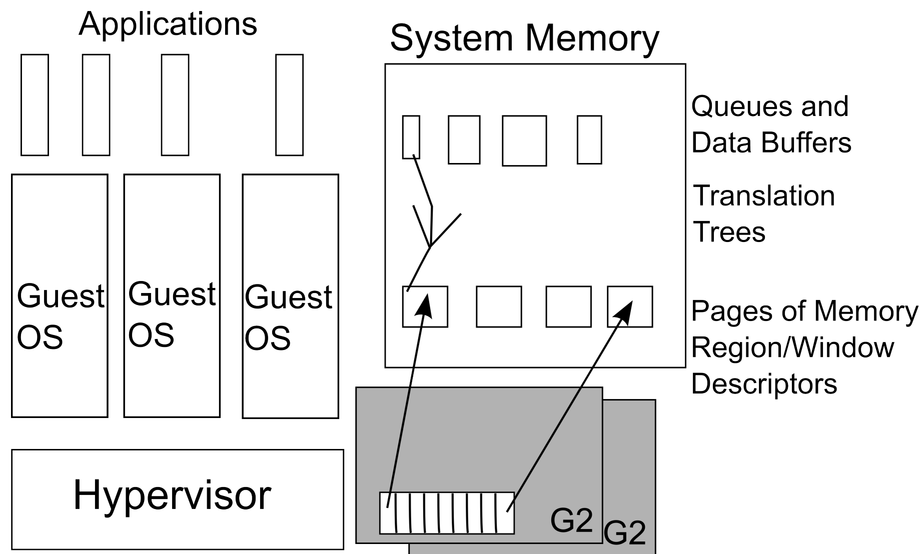

Fig. 1. System hierarchy and data structures for address translation, the data structures correspond to the management levels

The following registers are implemented inside the MMU (uint$n$ indicates an n-bit unsigned integer):

```
const int pagenum=2048;
uint8[pagenum] LPAR_IDs;
uint8 CTRL;
uint3 keyctrl[64];
bool[pagenum] LPAR_valid;
uint8[pagenum] SW_ERROR;
uint64[pagenum] HW_ERROR;
keydescriptor*[pagenum] page_ptr;
```

The descriptor of a memory region or window is as follows:

```
struct {
  uint64 base_addr, length;     // in Bytes
  uint64 region_base;           // only used for windows
  uint32 protection_domain;     // defined by Infiniband
  uint8 key_instance;
  uint8 format;   // number of tree levels, page size
  bool[5] permissions;          // e.g. write, bind
  bool    valid; bool window;
  bool LorR;
  char *tree_root[4];
  uint region_key; // needed only for windows
 } keydescriptor;
```

A descriptor consumes 64 Bytes of main memory, thus aligning well with the cache line granularity (128 Bytes for p- and 256 Bytes for z-Series). A set of 512 keys, called static keys, plays an important role for performance: As each static key has a dedicated cache entry, their `keydescriptor` needs to be read from memory only once after creation. To allow distribution of these keys to several guest OSs, they are distributed to the 64 first pages with 8 keys each. The remaining 48 keys in these pages are invalid. Static keys can be be used for regions, not for windows.

A translation request can be received either from the send or the receive processing units. The operation of a translation is as follows:

```
    translate(uint32 key,
       bool is_read,
       bool is_rkey,
       char* vaddr, // virtual address
       uint16 length,
       uint8 LPAR_ID,  // which guest partition
       uint32 protection_domain)
       return
        (char* page0,      //  page address on 4K granularity
         char* page1,      //  optional second address
         uint8 error_code) //  0 for successful translation
  {
  bool static_key = key < 0x100000;
  uint ix=key/0x4000;
  if (key > MAX_KEY ||
      (static_key && (key & 0x3F00 >800)) ||
      (!LPAR_valid[ix]) ||  (LPAR_IDs[ix] != LPAR_ID) ||
      (CTRL[ix]&6 != 4)) // not enabled or error state
     return error;
  keydescriptor kd= *page_ptr[key/0x400];
  if (!kd.valid ||
```

```
      (static_key & kd.window) ||
      (vaddr<kd.base_addr) ||
      (vaddr+length < kd.base_addr + kd.length) ||
      (key & 0xFF != kd.key_instance) ||
      (protection_domain != kd.protection_domain) ||
      (is_rkey != kd.LorR) ||
      !match(kd.permissions,is_read))
    return process_error(key); // updates SW_ERROR,HW_ERROR, CTRL
   levels=kd.format&7; // 0 to 3 levels
   pagebits = 12+4*(kd.format/8); // log of page size
   pagemask = !(1<<pagebits-1);
   pageoffset = (vaddr & pagemask) - (kd.base_addr & pagemask);
   // including range check:
   p=kd.tree_root[pageoffset/(1<<(pagebits+9*levels))];
   for (int l=0;l<levels;l++)
   // 512 pointers per tree node
     {p=*p[pageoffset/((1<<pagebits+9*i))];
      p1=*p[pageoffset/((1<<pagebits+9*i))+1];} // if not last
   p1=next4K(vaddr,p,p1,pagebits);
   return (p,p1,0);
}
```

As can be seen, there are authentication tests on the hypervisor level (correct partition) as well as on the guest OS level (correct protection domain). Up to four memory accesses are needed for one translation: the first for reading the keydescriptor, and up to three to walk through the translation tree. Each access can cause further errors, such as uncorrectable errors in the memory. Details of error processing are omitted here.

The second operation of the MMU is `bind`, which fills a `keydescriptor` such that it represents a window within a region. Information from both the region's key descriptor and from the bind work request are combined. As bind writes to a page of key descriptors, consistency with subsequent translations has to be ensured.

To achieve a high performance, several caches inside the MMU reduce the number of memory reads. As mentioned above, the static keys are directly mapped to a cache area. The content is read during the first translation, and an entry becomes only invalid by disabling (write 0 to CTRL register). For the remaining key descriptors, a cache of 1K entries is provided. The cache is fully associative and uses a random cast-out method that protects the entries most recently used. Explicit flushing by writing to a control register is possible, which is needed for resizing a memory region or software binding of windows. Another cache associates virtual addresses with two successive physical addresses that each saves entire tree walks. For the static entries four virtual addresses are provided for each key descriptor, whereas for the non-static keydescriptors only

one virtual address is provided. The addresses are on page-size basis, and can therefore cover a large address range if large page sizes are used.

Finally, a small cache saves some of the inner nodes of the translation tree, based on the requesting unit. Therefore, the inner nodes of the tree can be reused from the cache when a large memory region is being transferred, for instance, by an RDMA transfer. The cache relies on the policy that a given queue is processed on the same receive or send engine, if possible.

Here is an example:
A send operation needs the translation of address 0x72510300 for 256 bytes with key 0x0141733, a protection domain value of 0x77460ac1 is also provided. This key indexes the page 0x50, within the page entry 0x17 (6 bits). The lower 8 bits 0x33 are the key instance and allow the reuse of the same key descriptor. When the MMU receives the request it first checks that the key is valid, i.e. that the page index (0x50) is lower than the highes implemented page number (0x7ff for 2048 pages), and that the key is not 0x0. As the next step the MMU checks whether the page with index 0x50 is enabled and not in error state and whether the entry within the page is enabled. This is done by reading on-chip registers CTRL and keyctrl. Since the key is not static, as a next operation the CAM of the protection cache is searched using the key without the instance field (0x01417). Assume that the key_descriptor is found in the cache because of a previous address translation with the same key. The given virtual address (0x72510300) is compared to the starting address (say 0x7250080), and the sum of the virtual address and the length (0x72510400) is compared with the end of the region. The latter is determined by adding the starting address of the region and the length of the region. Furthermore the protection value from the key descriptor is compared against the parameter of 0x77460ac1. As a next step the page index needs to be found. For this example we use a page size of 64K. Both, the starting address of the region and the virtual address are truncated to 64K alignment and the difference is calculated (0x72510000 - 0x7250000 = 10000). This means that the required data is in the second page of this region. If the given region's length is smaller than 64K*4-0x80, four pages are sufficient and can be stored directly in the key_descriptor. Given this case, the offset within the page (0x300) is added to the real address of the second page (`tree_root[1]`). This value is returned to the send processor, thereby finishing this translation operation.

3 Design and Verification

To achieve high performance, up to eight operations can be carried out in parallel, where an operation can be either a translation, a bind, or a cache entry flush. Independent of this, registers of the MMU can be read or written. Parallel operation requires additional measures for consistency. For simplicity, parallel translations have to operate in disjoint keys, where for the bind operation both the region and the window are considered as "used". Before a requested operation can be started, a check is therefore carried out whether the key(s) of the

requested operation are currently in use. The parallel operations are organized as slots that consist of a set of approximately 200 latches. This slot stores the requested operation and its parameters, intermediate values such as the current level, and error codes. A new operation can be started if a slot is free, no currently used slot has the current key in use, and the key is not currently cast out. Only one request is tested per cycle. If an operation is accepted, the slot values will be filled in. The individual steps of an operation are structured in sub-units, organized along major steps or resources, such as search in the CAM for a key, or processing an error, which includes determination of the values for the HW_ERROR and SW_ERROR registers. Between some subunits direct connections exist, but whenever different paths can lead to the same operation (e.g., reading of the key descriptor cache array either after CAM lookup or directly after key validity checks) an arbitration is done, see Fig. 2.

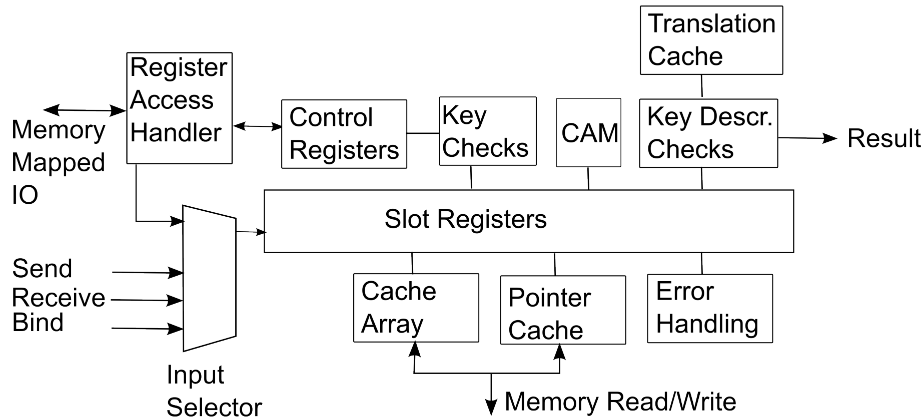

Fig. 2. Micro Architecture of the MMU

This approach has the advantage of offering easy scalability of the number of parallel operations, comparatively easy debugging during design and in the lab during hardware bringup, and good design partitioning to several designers. However, if the MMU is used mainly with the same type of operations, such as translation with non-static keys, several redundant arbitration cycles are inserted. If an external memory read is needed, the memory latency will dominate the translation latency, and extra arbitration cycles will be negligible. However, if most operations are operations with static keys, with either zero translation levels or with a hit in the translation cache, 3 of 13 cycles are arbitration. If higher emphasis is given to this best case, the design can be modified such that more sub-operations are done in parallel, reducing the translation latency to five cycles. The consequence would be an increased design complexity and a reduced throughput in case of translation errors, but the area would be nearly unchanged. Similar considerations apply to translations with non-static keys running from

caches, but the impact is smaller as the sequential dependency of the cache array read on the result of CAM lookup cannot be avoided.

To increase performance, the MMU can do a look-ahead processing after a translation. If a translation reaches the end of a page and uses at least one level of indirection, the MMU performs a translation of the next address to fill its caches. Therefore, if a later packet continues with the next virtual address soon, it will find the translation in the cache (either in the virtual-to-physical or the indermediate node cache).

The design is implemented in VHDL using IBM internal tools for synthesis, simulation, and physical design. Verity was used for ensuring synthesis correctness. Verification combined random-driven simulation and deterministic test cases, the latter mostly for error scenarios. The HCA portion of the chip runs at 250 MHz. Timing closure was reached in two steps: synthesis and a first timing check was done with a timing target of 3.2 ns to allow for clock skew and wiring, and a second timing analysis was done after placement and wiring. Only few paths in the MMU violated the first timing check. They related to a SRAM-read and Error-Correcting Code(ECC). As the ECC logic can be placed very close to the SRAMs, these violations were compensated in the wiring step.

The bringup was performed using bare-metal Linux®on p-Series and 370-architecture-based Linux on z-Series machines.

4 Conclusion

Several approaches to the implementation of a complex protocol, such as Infiniband, have been proposed in literature and used in products, ranging from software on a more or less specialized processor to a full hardware implementation as presented in this paper, e.g. [3]. Some well-known or obvious rules were once again confirmed by this design:

- Knowledge of the application and the architecture can greatly improve performance [4]; in this case the use of static keys for frequently used regions improves performance, even in cases where the cache capacity of the non-static keys is not exceeded and no cast-outs are needed.
- The use of large page sizes greatly increases the effectiveness of caches. Already the move from 4K to 64K pages provides a significant advantage. Note that between processor and IO the page sizes do not have to be the same.
- The hardware assumes that data is typically transferred in increasing order. SW can profit from it by also doing so.
- Address translation is faster if buffers do not cross page boundaries.
- To allow sufficient parallelism in the MMU, demanding applications should use a set of keys, e.g., three or four keys. Using too few keys causes stalls in the request acceptance step, whereas too many keys will reduce cache effectiveness.

Several further improvements were considered, but not implemented:

Allowing parallel translations on the same static key requires a restriction to the application. A single translation has to be used first to trigger reading the key descriptor into the cache before multiple translations can be allowed. The reason is that the tag management in the MMU and memory interface logic cause problems when multiple reads to the same address are simultaneously active. Although the design effort involved was not high, the extra verification effort did not justify the advantage.

A second proposal was considered following concerns about the performance when a server is connected to a storage device, such as IBM System Storage™ DS8000. As a server has many critical connections, it is not possible to rely on the use of static keys for the storage communication. However a fast reaction of the server is necessary to allow many parallel connections to the storage device. Therefore we proposed the partitioning of the cache holding the key descriptors, based on several inputs. As inputs, the LPAR, a hash from the key, information contained in a page descriptor register, and in the key descriptor itself, are assembled in a configurable way into a partition identifier. As the cache is fully associative, the partition identifier needs to be applied only during allocation of a new cache line, including cast-out of a previous line. The decision not to implement this feature was primarily based on the required support by the hypervisor, drivers, and applications. In particular, the latter are already complex, and further specialization to one hardware interface is not justifiable.

Acknowledgement

Many people contributed to the work presented here. With high respect for the work of all others only two architects, David Craddock, Poughkeepsie, and Thomas Schlipf, Böblingen, can be named here.

References

1. IBTA: Infiniband architecture specification 1.1 (November 2002)
2. Schlipf, T., Helms, M., Ruf, J., Klein, M., Dorsch, R., Hoppe, B., Lipponer, W., Boekholt, S., Röwer, T., Walz, M., Junghans, S.: Design and verification of the IBM System z10 I/O chips. IBM Journal on Research and Development **53**(1) (January 2009) in press, available at http://www.research.ibm.com/journal/rd/.
3. Georgiou, C.J., Salapura, V., Denneau, M.: A Programmable, Scalable Platform for Next-Generation Networking. In: Network processor design: issues and practices. Morgan Kaufmann (2003) 11–28
4. Hoefler, T., Rehm, W.: A communication model for small messages with infiniband. In: PARS Mitteilungen. Volume 22. (2005) 32–41