LA-UR- 09- 10-07974

Approved for public release; distribution is unlimited.

Title:	Tuple spaces in hardware for accelerated implicit routing
Author(s):	Zachary K. Baker, Justin L. Tripp
Intended for:	2011 Reconfigurable Architectures Workshop at IPDPS

Los Alamos
NATIONAL LABORATORY

Form 836 (7/06)

Tuple spaces in hardware for accelerated implicit routing

Zachary K. Baker and Justin L. Tripp
Los Alamos National Laboratory
Los Alamos, NM 87545
Email: {zbaker, jtripp}@lanl.gov

Abstract

Organizing and optimizing data objects on networks with support for data migration and failing nodes is a complicated problem to handle as systems grow. The goal of this work is to demonstrate that high levels of speedup can be achieved by moving responsibility for finding, fetching, and staging data into an FPGA-based network card. We present a system for implicit routing of data via FPGA-based network cards. In this system, data stuctures are requested by name, and the network of FPGAs finds the data within the network and returns the structure to the requester. This is acheived through successive examination of hardware hash tables implemented in the FPGA. By avoiding software stacks between nodes, the data is quickly fetched entirely through FPGA-FPGA interaction. The performance of this system is orders of magnitude faster than software implementations due to the improved speed of the hash tables and lowered latency between the network nodes.

1 Introduction

Managing the parallelism provided by thousands of cores is extremely challenging. Legacy codes in particular are not designed to take advantage of the large parallel resources provided by current and upcoming system. New approaches to programming these systems are required to harness their power. Increasing the productivity of our scientists and our computer systems is the most important goal of this effort. The execution speed of a given code is of little value if it takes an inordinate amount of time to develop, debug, modify, and maintain. When we raise the level of abstraction supported by the programming environment we let developers work in a domain specific to the problem at hand, not the domain of the machine. This approach provides a more natural and comfortable interface to the machine there is no need to learn the semantics of an all-purpose environment or to "wedge" the problem into the model of traditional programming languages.

Communication between nodes in HPC systems is a performance bottleneck. Experience at LANL has shown that 50% of application execution time is actually just waiting on the network [2]. The network topologies in HPC systems are typically designed to accommodate the communication patterns of any generic application, but are not optimized to maximize the performance of any specific application. Ideally the network behavior and application would be co-designed to optimize performance by matching the topology to the communication needs of the application.

We are currently developing the idea of the "RDI" or Reconfigurable Data Interface. This is a fancy name for FPGA-based "Smart" network card connected via PCI-E that can make decisions about data movement and migration in order to optimize a computation. The programmer then interacts with the network via abstracted calls that provide them with some separation from the dirty details of working with an FPGA. The code does have to support the abstraction of the global name space, but this is similar to using a virtual address space. If the user can adopt a new paradigm but not have to worry about why the performance is better, then they will likely continue to use the new system.

NoSQL databases have recently increased in popularity, since they are continue to scale for large amounts of data or across large clusters of computers. These databases are less structured, typically lack tables and are not relational. This decreases the overhead needed for storing data and eases the difficulty of scaling. An example of storage of this type would be key/value pairs. Key/value pair datastores organize the data using a key to store the data or *value*. Thus, the data can be recalled using the key that was used to store it. Memcached [8] is a prime example of the key/value datastore, that is used by several major websites (Facebook, Twitter, YouTube, Wikipedia, etc. [6]) to cache data. Amazon uses a similar approach in their proprietary system called Dynamo [5], which powers the Amazon Web Services.

Another approach used by NoSQL databases are distributed hashtables (DHT). DHTs are commonly used in Peer-to-Peer systems (such as Bittorrent [19]), to store and

distribute the data. Pernicious systems, such as botnets, use DHTs to store and distribute data as well. DHTs have three properties that make them useful and a potentially a key technology to high performance computing problems: Decentralization, Scalability and Fault Tolerance. Current investigation [12] shows how useful this approach could be for high performance computing.

Hardware accelerated Tuple Spaces are a potential answer to the problem of keeping track of data and computation on a large network. Performing load balancing on a large system causes data and compute to migrate, making it difficult for a user to manage their computation effectively. By providing a fast mechanism to manage data and computation, we alleviate some of the burden from the user. This is accomplished through a distributed associative memory technique called "Tuple Spaces". The attractiveness of the Tuple Space paradigm is that a user requests data by its name, rather than by its location. This allows the system to maintain the location of data, similar to the way virtual memory works in any common microprocessor.

In [7] real performance of a Tuple Space implementation on a small cluster is approximately 300 tuple lookup per second, with a latency of about 60ms. Granted, this implementation is in Java, with associated performance degradations over coding at the metal. We believe these aspects of performance have reduced interest in the Tuple Space approach to managing data. By moving the details of managing the Tuple Space to hardware and allowing it to be accessed via standard API calls, we believe that the interest in implicit data routing may be rejuvinated.

1.1 Field Programmable Gate Arrays

FPGA's provide a fabric upon which applications can be built. FPGAs, in particular, SRAM based FPGAs from Xilinx [17] or Altera [9] are based on a look-up tables, flip-flops, and multiplexers. In these devices, a SRAM bank serves as a configuration memory that controls all of the functionality of the device, from the logic implemented to the signaling standards of the IO pins. The values in the look-up tables can produce any combinational logic functionality necessary, the flip-flops provide integrated state elements, and the SRAM-controlled routing direct logic values into the appropriate paths to produce the desired architecture. The device is composed of many thousands of basic *logic cells* that include the basic logic elements, and based on the device variety, includes fast ASIC multipliers, ethernet MACs, local RAMs, and clock managers.

There are many tools for automatically converting a high-level language program into a low-level design. With recent advances in compiler and synthesis technology, it is now possible to map the computationally intensive modules of a program in C (SRC Carte [14] and Celoxica [4]), or through graphical tools such as Xilinx System Generator [15].

While the ease and popularity of using FPGAs for application design has increased, the automatically generated architectures tend to be inefficient compared to well-researched and thought-out architectures for complex applications. In these situations, the creativity and domain knowledge relevant to a design provided by a human designer is a valuable asset.

2 Reconfigurable Data Interface

The Reconfigurable Data Interface (RDI) steers an application's computation and communication. The RDI is not just a network interface card. The RDI is network. By integrating the data management with the network infrastructure, latency goes down and bandwidth goes up. Adding a small local router to each processing node extends the ideas that are currently being applied to multi-core processing chips. Distributed routing eases the burden placed on higher-level routers required to build up very large machines. In multi-core nodes, a router is available for a small number of cores enabling the transfer of data to be optimized at a local level. In this same way, the PFRouter's programmable local router provides the ability to off-load the transfer of data between local nodes and optimize the problem between the nodes within a given shelf of processing nodes. Programmable NICs have demonstrated speed ups for various operations (e.g., [20], [13]) FPGAs have shown a 10x speed up for certain MPI collectives [10], and 1000x for network processing [1], which demonstrate the potential for custom operations within a programmable network to provide system speed up.

As larger systems spend more time waiting for network operations, it will be necessary to move some operations closer to the data on the network. Using configurable network processing allows this to be molded to the application and provide the customization necessary to handle different kinds of operations.

Because the RDI is based on FPGA technology, particular instructions can be performed vary quickly in the programmable hardware. For instance, customized network collectives can be implemented in hardware. These are operations that use a large number of nodes, and often are a determining factor in the overall performance of the application. Potential operations we plan on addressing include collectives such as all-reduce, distributed tree traversals, and scatter-gather primitives. A potential candidate for movement into the RDI is distributed tree traversal, where the network card automatically fetches the child nodes of a tree and presents them as a package for processing to the CPU or GPU.

3 Tuple Spaces

Tuple Spaces are based on giving data names that are separate from their address in memory. This is similar to a virtual address with fully associative mapping. However, it can be extended across network nodes, and support more complex data structure than a flat page of memory.

In past implementations, the secondary lookup of the data's name has hindered the performance of a Tuple Space. In the PetaFlops Router system, the FPGAs actually maintains the tables that keep track of the current location of data blocks as well as ongoing requests for computation. This provides many benefits. First, the RDI is directly connected to the network and can keep track of data movement more closely than can a CPU that is several stages from the network. Second, the RDI can fetch data directly from the GPU without interfering with the CPU's computation work. Third, the distributed nature of the Tuple Space means that data can be stored across the network, rather than solely at a given node. This positively impacts the reliability because the failure of one node does not bring down the entire computation. The ability to load balance across the network means that scaling and usability will be improved over current MPI systems.

We have implemented the lookup features of the Tuple System in hardware system. We are not, at this point, considering the larger problems of inserting computation requests in the Tuple Request itself. For kernels and a limited set of comptuation, it would be efficient but for requests that include patterns (for instance, regular expressions), the process of handling the patterns would be very expensive to implement in hardware. This is particularly true with the memory-based hashing system we have devised, a more sophisticated pattern matching system would require Content-Addressable Memory. This would significantly increase the per-entry cost of the hash table.

3.1 Tuple Architecture

We have demonstrated a prototype distributed hardware-based Tuple Space system. Similar in spirit to the software-based LINDA tuple system, the system builds a distributed hardware hash table across several FPGA nodes. The FPGAs then communicates across multi-gigabit links to move data around automatically. This is potentially very exciting as it emphasizes a co-design approach to large cluster design: Allow certain data management and collection functions to migrate to hardware, while avoiding full-custom hardware design of application kernel.

The hash architecture is the core of the tuple system. The hash key is the name of the tuple data element. The hash lookup results in the address of the data in local DD3 memory store. The architecture consists of the blocks in Figure

Figure 1. Photograph of the three node system currently in test. The red and white cables are the inter-FPGA network implemented with SATA and SMA cabling. In practice, these cards would be plugged into the PCI-E interfaces of the CPUs

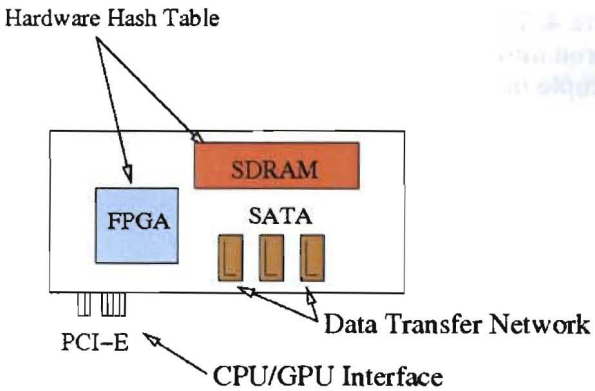

Figure 3. The tuple system is based on a FPGA implementing a hardware hash table, data storage, PCI-E interface to the CPU host, and independent network connections to the other FPGAs in the network

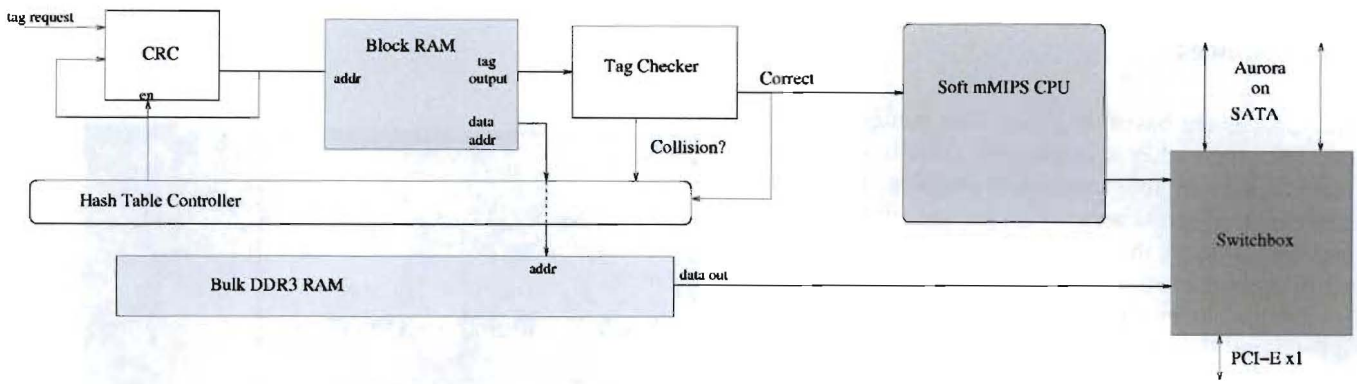

Figure 2. Hash table architecture

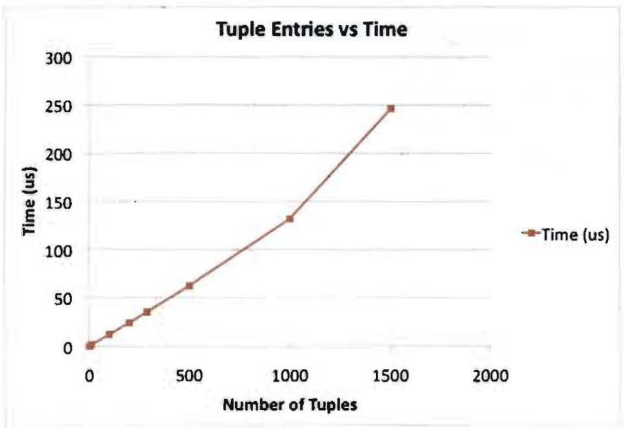

Figure 4. Tuple entry rate vs time. Tuples are entered into table in roughly 130 ns vs 1.2 us per tuple on CPU host

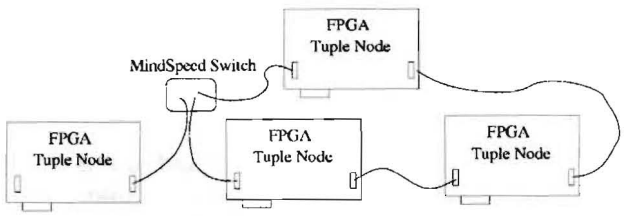

Figure 5. Tuple engines implemented in ML-505 and ML-507 boards are connected via SATA cables

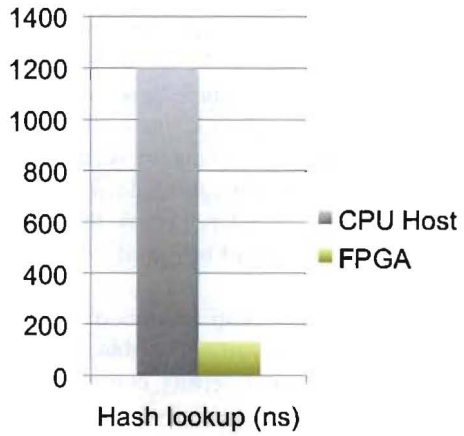

Figure 6. Hash lookup rate

2. The main block of the hash system is a CRC generator and a block memory for the keys. The system would be simple if we did not handle collisions, but any real application would have enough keys that collisions would be unacceptable.

The state machine starts by resetting the CRC generator and inputting the requested key. The CRC produced is used as an address in the key block RAM.

The output data is has the following format:

1	occupied
1	deletion flag
key_width-1 downto 0	stored key for comparison
addr_width-1 downto 0	addr of data element in the bulk memory
data_length-1 downto 0	length of data block stored in bulk memory

If the requested operation is a "write" operation on a new key, the *occupied* bit is checked. If it is unset, then that address is the new location of the hash key and the key, and

the length and address of the data in bulk memory is written into the block memory. If the *occupied* bit is set, implying a collison, then the output CRC is fed back into the CRC generator to try again. Another approach is to increment the address and check again. This would save one cycle in the lookup, but could create clusters of data in the hash table, violating the requirements for randomness in the O(1) analysis.

If the requested operation is a read or delete, the CRC output is checked for the *occupied* bit as well as comparing the input key with the stored key. If the *occupied* bit is set but the keys do not match, the entry is assumed to be from an earlier collision. By following the collision chain until the keys match, the collision can be resolved. In the case of a delete operation, the key is zeroed and the *occupied* bit is unset. If an unset *occupied* bit is found while resolving a collision chain, this implies that the key is no longer in the system, or was never there in the first place.

A collision chain with deleted entries is somewhat problematic because a chain of collisions is determined to be in error if an unset *occupied* bit is found. however, if delete is intended to not corrupt the table, then there are two options. One, all deleted key entries are set to an arbitrary value, perhaps -1. Thus, the any collision chains that happen to pass through the deleted entry will continue. However, this also prevents any new key from being entered in the location, effectively creating a memory leak.

The second option is to add another bit to the hash data structure to imply a *delete flag* bit. An entry with this bit set is not a valid entry, but can be used for new data. This has the trade-off of increasing the size of the hash memory footprint. Because we envision the hash system running for extended periods of time with extensive New and Delete operations, exchanging an extra bit per entry was deemed a better option that an intentional memory leak.

Figure 6 illustrated the speed in which new entries are added to the system. This data was collected for a relatively small hash table to better illustrate the effect of collisions on the hash table performance. The initial entries generally do not encounter collisions, which means that they can be entered into the first addresss generated by the CRC. As the table becomes congested, the average time to find an empty slot increases. There is no quality of service requirements, except a desire for it to be fast, so the system will continue to search for a location until it has searched every slot. This would generally be unacceptable in a real system, but can only be addressed in hardware through allocating a larger memory at design time to the hash table, or moving the data somewhere else, be it a backing store in a larger DRAM or on another host entirely. We will address the second option in Section 3.3.

3.2 Tuple Performance

In previous research, real performance of a tuple space implementation on a small cluster is approximately 300 tuple lookup per second, with a latency of about 60ms. This performance directly impacts overall application performance. In the current prototype system, tuples are entered into table in roughly 130 ns vs 1.2 us per tuple on a single CPU host.

Using a ring architecture, we predict the lookup latency in a 10 node prototype cluster to be 2.5us, a 1000x performance improvement in locating and fetching remote data.

The Xilinx ML-50x boards support single lane PCI-E connectivity, meaning a maximum of X Gbps between the host and the FPGA. In practice, the latency was more relevant, at approximately 1us per transaction. The hash table operates at 100MHz. It could easily operate faster but that is of limited value given that the hash lookup is generally not the performance bottleneck. The system processes one hash lookup at a time, although the pipeline structure can accomodate three simultaneous lookups with minor changes.

The PCI-E interface is based on the BMD design provided by Xilinx. The driver was developed using the Jungo WinDriver [18] package for Linux, but should port to Windows fairly easily. The WinDriver package handles the setup and provides a simple API for talking to a hardware device. We have measured the PCI-E latency from user software to the FPGA to be approximately 1us per transaction. The latency of the other transactions is quite short in comparison with the PCI-E latency $T_{pci} = 1us$. The hash lookups T_{hash} – ignoring the occasional multiple-collisions lookup – take less than ten cycles, or about 100ns. The time to fetch an address from the SDRAM is approximately **. The hops between FPGAs $nodes$ via MGT links running Aurora are in the tens of cycles T_{hop}.

$T_{total} = T_{pci} + nodes(T_{hash} + T_{sdram} + T_{hop})$

The ring structure reveals its limitations in this analysis. While the total time at any given node is low the aggregation of hash lookups in every node the search passes through builds up becomes expensive. Clearly a ring structure is not ideal in most cases, but is convenient when prototyping with the ML-50x series of boards. These boards feature two SATA connectors, allowing for easy building of ring structures. The board also has a set of SMA connectors, allowing for a third link, so a tree structure is also quite feasible but somewhat less elegant.

The hash table system as implemented occupies 60% of the block RAM resources in the LX110T. The soft processor's memory occupies the balance of the block RAM. The hash table, including the CRC, operates at 100 MHz. This is not the maximum acheiveable speed, but decreases the time required for place and route. The mmips processor operates at 50 MHz and is responsible for steering data between the

PCI-E connection, the hash table, and the two Aurora links. The Aurora links operate at 1.5Gbps. The Aurora links can go much faster, up to 6 Gps with Gen 3 cabling. The single-lane PCI-E connectors on the ML-50x boards are limited to 2Gbps. Simply upgrading to the ML-605 would bring a x8 connector for 8x the bandwidth and block RAM for the hash table.

The largest performance improvement comes with knowledge of data structures. For instance, when traversing a tree or a linked list, the FPGAs can be given knowledge of the data structure in question. Traditionally, the traversal would require pulling data from the network, pushing it up through the PCI-E bus to decode the data structure and then back through the PCI-E bus to the network card for the next request. In our simple, single node tests with memcached [6], traversing a 7 element linked list took roughly 125us. In the hardware system with a single node, this same operation took 3us, where 2 us is the PCI-E latency and the pointer chasing essentially falls into the noise. The speedup is entirely due to traversing the PCI-E bus twice for the entire operation: first, to make the initial request, and second, to fetch the result. The pointer chasing through the hash table happens entirely in hardware. The data structure is handled by the soft processor, so adding knowledge about a particular struct is software, not hardware.

3.3 Handling Larger Data Sets

In the hash table architecture discussion, we came upon an important question. What happens when the hash table is full, or full enough that the hash performance is degraded? In a software-based system, it is not difficult to expand the size of the hash table dynamically. This is somewhat more difficult in hardware unless we are willing to degrade performance by switching out of the limited on-FPGA block memory resources and into the essentially limitless external DRAM. However, given the distributed nature of the system, the latency to store entries on another node's block RAM can actually be less than going to external SDRAM locally. This can be generalized into a system where hashed addresses can point anywhere in the system.

Unfortunately, this global addressing has many characteristics of the systems we are trying to improve upon. The system is intended to be resistant to node or link failure (also addressed in the 3.3 section). Thus, keeping a static pointer to a particular node is self-defeating. This is easily avoided by not using the node's address per se, but rather a secondary key structure that a node can claim. In this mode, the request traverses the network structure until it finds a matching node, then performs a hash lookup to find an index into the local DRAM.

3.4 Migration

One problem with the global access to locally requested keys is that repeated accesses to a location can cause unnecessary traffic on the network and extra work for the FPGAs. Migrating the data and hash table entries to the local node can be a solution, but can also increase the complexity of managing the data in the FPGA-based system.

There are two potential pathways to acheive the goal of migration. First, repeated requests to a particular tuple can be tracked in the tuple data structure. Requests that exceed a threshold can cause the entire data structure to be moved, including a deletion of the tuple from the original tuple location. In this mode, a single copy of the data is accessed by all nodes in the system. This allows for simple coherence, but hurts performance. Migration allows for the copy to migrate to where it is most often used, but multiple readers can cause excessive migration as well as the equivalent physical separation of the reader and owner of a piece of data. One strength of this approach is the ability to change values in the data structure pointed to by the tuple in-situ, rather than reading it out and completely re-writing it in memory.

A more traditional approach is to require single static assignment. In this mode, every data element is written once, and no re-writing is allowed. This allows references to be tracked more easily, as any changes to the data structure requires a completely new tuple entry and allocation in the bulk storage. In this mode, any requestor of a particular data structure can create a local copy of the data. Because it cannot be changed by another node, the data node remains valid. Any updates to the data will take place in a newly created tuple entry. Then, the updated tuple name/data combination can be propagated as nodes request the new tuple name.

The issues with migration are theoretical at this point as they have not been implemented or tested in real hardware. However, the work is largely in software as the cases are special enough to implement solely in software in the embedded processor.

3.5 Improvements to Network

We also plan on exploring the issues of network topology in the future. Our second implementation will allow the network topology to dynamically change by creating and closing dedicated circuit-switched links as needed during program execution. This is akin to the operation of the original circuit-switched telephone network where switchboard operators were in place to physically connect and disconnect callers based on customer demand and connection availability as opposed to having dedicated lines between customers. In our case, unused circuit-switched links would immediately become part of the packet-switched network,

providing more network capacity for generalized communications. The basic system structure is shown in Figure 5. Each component of the system is commodity hardware. This will allow us to estimate the performance of the initial system without a large outlay for custom hardware. The network is logically built around a central MindSpeed crossbar switch [11]. This crossbar acts as the bulk circuit switching resource in the system, providing the ability to configure the network topology as needed, including hardware-based multicast. As shown in Figure 5, crossbar forms an essentially direct electrical connection between ports, forming the desired network topology. The network interface cards (NIC), inside each PC, are connected to the crossbar. The NICs are responsible for two main functions. First, they provide the fast PCI-E interface with the CPU and memory of the computer. Second, they provide any packet and circuit switching capability that is not covered by the crossbar. Because the crossbar can only provide connectivity that is a subset of that enabled through physical wiring, the NICs provide any other needed connection paths and alternate routes. Using Aurora as a starting point, we will add a protocol layer providing flow control and packet headers for switch-mode links. This protocol layer will support both packet- and circuit-switched communications, meaning that the hybrid network can fluidly transition between operating modes without change of protocol. More sophisticated protocols could be implemented, but a protocol trade study would only serve to detract from the core purpose of the proposed effort.

4 Interesting FPGA tidbits

Because we developed the system using a combination of Xilinx ML-505 and ML-507, we must target a combination of "GTP" and "GTX" multi-gigabit transceivers. This is inconvenient, as the Aurora source trees are somewhat different. One convenience we added is a merging of the source trees so that a particular MGT on a particular V5 device can be targeted without delving deeply into the source code.

The execution and behavior of the system is controlled by a small embedded processor. Initially this was controlled via the serial port rather than the final implementation including PCI-E. Via RS-232, we can interact with the processor, as well as updating the software on the processor. This proved to be difficult to scale as the system increased in size. Past four FPGA boards the overhead of serial cables became an annoyance. Moving to Xilinx ML-605 boards would ammeliorate this problem somewhat as it has USB-emulated serial, but even then the total number if USB cables becomes a burden.

Our solution to this was to move to a new administration system built on I2C. I2C [16] allows many devices to be connected to the same bus. It's fairly slow compared to most modern interfaces, but a) it is faster than serial bus (400kbps vs 115200bps on RS-232) and b) it requires very little hardware support and two signal lines. Out of the box there is no support for external i2c connections on the ML-605, that is to say, there is no convenient set of headers that bring the signals to the surface. However, following the model in [3] we fabricated an i2c cable out of a DVI cable. DVI includes an i2c bus so that a GPU can communicate with the attached monitors. This allows us to use one of the intended i2c networks on the board without building anything expensive/custom. This is a great idea for single boards, but with multiple boards there are two problems: too many i2c pull-up resistors, and too much capacitance. Therefore, other headers including the FMC connectors must be used for larger networks.

The software driver splits the output from the i2c bus into several windows, allowing a single server to view all the output of all of the FPGAs with a single cable. This is a relatively easy mechanism to debug large systems of FPGAs without using the main data channels. The system can upload the average new software package for the soft processor in about 5 seconds per FPGA.

The mMIPS soft processor is a 32 bit CPU that uses a small subset of MIPS instructions. Use of the soft CPU has allowed us to quickly prove out ideas in hardware at hardware speeds rather than develop full hardware implementations. Adding a standardized bus system to the CPU has allowed other IP to be quickly developed and combined to make larger and more complex systems. The C compiler for the CPU was modified to allow the use of larger instruction stores, so that we can have more complex programs. These features combined have made the mMIPS a useful vehicle for testing ideas and exploring the space of the PFRouter hardware system.

5 Conclusion

We have presented a system for implicit routing of data via FPGA-based network cards. In this system, data stuctures are requested by name, and the network of FPGAs finds the data within the network and returns the structure to the requester. This is acheived through successive examination of hardware hash tables implemented in the FPGA. By avoiding software stacks between nodes, the data is quickly fetched entirely through FPGA-FPGA interaction. The performance of this system is orders of magnitude faster than software implementations due to the improved speed of the hash tables and lowered latency between the network nodes.

References

[1] Z. Baker and V. Prasanna. Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs. IEEE Transactions on Dependable and Secure Computing, Vol. 3, No. 4, October 2006.

[2] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[3] P. Burgess. The 25¢ I^2C Adapter. `http://www.paintyourdragon.com/?p=43`.

[4] Celoxica MATLAB/Simulink Interface, 2004. `http://www.celoxica.com/methodology/matlab.asp`.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of SOSP'07*, pages 205–220, Stevenson, WA, October 2007. ACM.

[6] Dormando. memcached - a distributed memory caching system. `http://www.memcached.org`, Nov. 2010.

[7] D. Fiedler, K. Walcott, T. Richardson, G. M. Kapfhammer, A. Amer, and P. K. Chrysanthis. Towards the measurement of tuple space performance. In *ACM SIGMETRICS Performance Evaluation Review*, 2005.

[8] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, (124):72–74,76,78, August 2004.

[9] FPGA CPLD and ASIC from Altera. `http://www.altera.com`.

[10] S. Gao, A. Schmidt, and R. Sass. Hardware implementation of MPI_Barrier on an FPGA cluster. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 12 –17, September 2009.

[11] Mindspeed 72x72 3.2 Gbps Crosspoint Switch with Integrated CDRs, Input Equalization & Pre-Emphasis. `http://www.mindspeed.com/web/product/info.html?id=599&trail=2001,2022,4039`.

[12] R. Minnich and D. Rudish. Ten million and one pengiuns, or lessons learned from booting millions of virtual machines on hpc systems. In *Proc. of Workshop on System-level Virtualization for High Performance Computing in conjunction with EuroSys '10*, Paris, France, April 2010. ACM.

[13] F. Petrini, A. Moody, J. Fernandez, E. Frachtenberg, and D. K. Panda. Nic-based reduction algorithms for large-scale clusters. *International Journal of High Performance Computing and Networking*, 4(3-4), 2006.

[14] SRC Carte Programming Environment, HLL FPGA Programming. `http://www.srccomp.com/techpubs/carte.asp`.

[15] System Generator for DSP, 2010. `http://www.xilinx.com/tools/sysgen.htm`.

[16] telos EDV Systementwicklung GmbH. I2c-bus: What's that? `http://www.i2c-bus.org/i2c-bus/`.

[17] Virtex-5 FXT FPGA ML507 Evaluation Platform. `http://www.xilinx.com/products/devkits/HW-V5-ML507-UNI-G.htm`, Nov. 2010.

[18] WinDriver: Driver Development Tools for USB/PCI/PCI-Express. `http://www.jungo.com/st/windriver_usb_pci_driver_development_software.html`.

[19] Workshop on Economics of Peer-to-Peer systems. *Incentives build robustness in BitTorrent*, 2003.

[20] W. Yu, D. Panda, and D. Buntinas. Scalable, high-performance NIC-based all-to-all broadcast over Myrinet/GM. In *Sixth IEEE International Conference on Cluster Computing (CLUSTER'04)*, pages 125–134, 2004.