FPGA Prototyping of Manycore Multinode Systems for Irregular Applications

Marco Ceriani, Gianluca Palermo
DEIB - Politecnico di Milano
Milano, Italy
{mceriani,gpalermo}@elet.polimi.it

Simone Secchi
DIEE - Università di Cagliari
Cagliari, Italy
simone.secchi@diee.unica.it

Antonino Tumeo
Pacific Northwest National Laboratory
Richland, WA, USA
antonino.tumeo@pnnl.gov

Oreste Villa
NVIDIA
NVIDIA, CA, USA
ovilla@nvidia.com

***Abstract*—Knowledge discovery applications are an emerging class of irregular applications that exploit graph-based data structures, present poor locality and analyze very big data sets that require multi-node systems for processing. Current commodity clusters, which exploit cache-based processors, usually perform poorly with these applications. To address their requirements, full-custom machines, like the Cray XMT, or software approaches, such as several distributed graph libraries have been proposed. In this paper we discuss a set of hardware and software components that makes manycore architectures, composed of off-the-shelf cores, more efficient with these applications. The components enable support for latency tolerance, scrambled global address space, fine-grained synchronization, and software multi-threading with hardware scheduling. We validate our approach with a 4-node FPGA prototype that enables exploring tradeoffs among network bandwidth, number of cores per node and threads per core. The prototype has been presented in various FPGA-related venues. We employ two typical irregular kernels (pointer chasing and Breadth First Search), showing that the the proposed components allow higher utilization of the network bandwidth with fine-grained transactions and better multinode scaling.**

I. Introduction

Social network analysis, data mining, semantic databases and in general, knowledge discovery, constitute a new class of irregular applications. Such applications exploit pointer or linked-list based data structures, such as graphs, unbalanced trees and unstructured grids, which present poor locality, require fine grained memory and network accesses and generate unpredictable memory and network transactions. These applications employ very large datasets, which require significantly more memory than that available on a single node server, and are difficult to partition among multiple nodes in a balanced way. Modern high performance systems are optimized for regular workloads. They integrate multicore processors with complex cache hierarchies and high flop ratings, and exploit locality and regular computation to increase performance. Furthermore, their interconnection networks are optimized for large batched transfers and have significantly lower bandwidth with respect to node-local memory. Because of the unpredictable memory access pattern, the performance of irregular applications is substantially limited by the sustained memory and network bandwidths for fine-grained accesses.

Architectures such as the Sun Niagara-based processors [1], the Tera MTA, the Cray MTA2, XMT and Urika [2] are more suited to irregular workloads, because they focus on tolerating, rather than reducing, memory access latencies through multi-threading. In particular, the Cray XMT is a supercomputer specifically designed to address the issues of irregular applications. It integrates custom multithreaded processors (ThreadStorm) with a large number of hardware contexts, allowing it to tolerate not only node-local memory but also network latencies. It also supports a global address space across nodes with address randomization (fine grain scrambling) to reduce network hot-spots in presence of fine-grained network transactions. Finally, it includes fine grain synchronization support through full/empty bits associated with each memory word. These features simplify the development of irregular applications with large datasets using a system-wide shared memory programming model, without the need to worry about data location. On the other hand, several software infrastructures specifically targeted at crunching large graphs on commodity clusters have appeared. The majority of the tools exploits map-reduce approaches, while others integrate support for a Partitioned Global Address Space (PGAS) programming model with lightweight multi-threading to hide communication latencies. However, they are mainly targeted to graph crawling rather than to the wider class of irregular applications, and have limited performance scaling.

In this paper we present an intermediate approach between full-custom hardware systems and full-software tools. We describe an architecture designed starting from off-the-shelf components, and integrate a set of hardware and software custom components that enable efficient support of irregular applications, following an approach similar to many IP-based design companies. Considering the current trend towards manycore processors, we build an architecture that exploits multiple simple cores interconnected together, with memory controllers and network interfaces in a system-on-chip fashion. We propose three components that can be enabled or disabled only when required: the Global Memory Access Scheduler (GMAS), the Global Network Interface (GNI) and the Global SYNChronization module (GSYNC). These components provide support for global scrambled address space across nodes and for fine-grained synchronization. Furthermore, they assist lightweight software multi-threading, which allows tolerating network access latency. To validate our approach, we exploit prototyping on Field Programmable Gate Arrays (FPGAs). We implemented a multi-node FPGA design integrating the off-the-shelf components offered in the Xilinx toolchain with our own components. We use the prototype to explore system trade-offs such as number of threads and number of cores with respect to the network bandwidth and latency, using typical irregular applications. This paper discusses material presented in [3], [4], and [5], providing significantly more details on the architectural design of the components.

Using a prototype, instead of simulation approaches, provides several benefits. The validation of the hardware components is stronger, because they are directly designed at the register transfer level. This also allows capturing primary performance issues and exposing hardware implementation challenges. The prototype provides higher performances than a simulation infrastructure. It allows faster iterations between hardware and software, so that the software layer can be progressively evaluated with the development of the hardware.

The paper proceeds as follows. Section II presents the related work. Section III provides a general overview of the architecture and the rationale behind its design. Section IV discusses the design of the components that enable efficient execution of irregular applications

on many-core designs. Section VII deals with the experimental evaluation. Finally Section VIII concludes the paper.

II. Related work

Efficiently executing irregular workloads on multi-node machines require support for global address space, for fine-grained synchronization and for tolerating memory and network latency when data are retrieved.

Modern processor architectures exploit large and complex multi-level caches, data pre-fetching and memory operation reordering to reduce data access latencies. These techniques increase the performance of regular applications, which exhibit predictable data access patterns and high spatial temporal locality. However, they may be detrimental for irregular applications, because their accesses cannot be predicted. Multithreading, which tolerates access latencies by switching to other thread contexts while data are loaded or stored, appears more suited to these applications. Among multithreaded machines, only the Tera MTA, and its successors Cray MTA-2, Cray XMT and Cray Urika [2], provide system-wide latency tolerance features for multi-node systems. The ThreadStorm processors in the Cray XMT, with up to 128 thread contexts, can tolerate the latency to cross the network and access memory on the other nodes. Furthermore, the XMT implements in hardware a scrambled global address space across nodes and full/empty bits for each memory word to support fine-grain synchronization.

Software multi-threading also enables latency tolerance. The APRIL processor in the Alewife machine exploited very fast context switching routines to implement software block multithreading. Qthreads [6] is a software library that implements software fine-grained multithreading and XMT-like full/empty bit semantics on various architectures (POWER, x86, SPARC, Tilera). However, these approaches have not been thoroughly explored on multi-node machines with global address spaces.

Partitioned Global Address Space (PGAS) programming models implement a shared address space across multi-node machines without neglecting data or thread locality. Languages such as Unified Parallel C (UPC) [7], Co-Array Fortran [8], Titanium [9], X-10 [10], Chapel [11] or libraries such as Global Arrays [12] and GASNet [13], are progressively gaining traction in High Performance Computing because they can reduce the effort for programming large-scale machines without trading off too much performance. However, these approaches are still mostly software-based, and the runtimes can still generate significant overheads. The Cray Gemini interconnect [14] has direct hardware support for PGAS programming models, allowing pipelining of remote references. [15], instead, describes a hardware communication engine that enables the shared address space across multiple nodes of a commodity cluster. Out approach is different, because it works at the chip level and looks at extending the on-chip interconnect protocol of a manycore design.

The GRAPPA [16] runtime integrates PGAS and lightweight software multi-threading to hide communication latency. It aims at increasing the performance of graph crawling on commodity x86 clusters. Other tools, such as Pregel [17], Giraph and GraphLab, instead, exploit bulk synchronous parallel models (such as map-reduce) through vertex programs that run on each vertex and interact along edges. Interactions either use messages (Pregel and Giraph) or shared states (Graphlab). Our objective is not to implement a software library for graphs, but rather to propose a set of hardware and software components to enable efficient execution of irregular applications on many-core processors. While our approach also exploits software multi-threading, it is only a part of the set of the components that we integrate in the architecture.

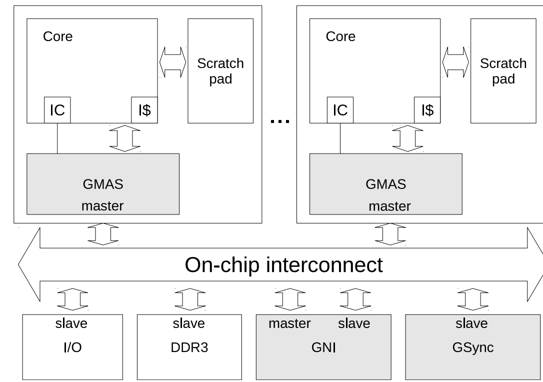

Fig. 1. Node architecture overview

Our prototyping methodology is inspired to RAMP [18]. The RAMP project developed various FPGA-based systems to evaluate transactional memory systems [19], thousand-core designs (RAMP Blue) [20] and to accelerate architectural simulation (RAMP Gold) [21]. Our approach, in particular, follows RAMP Blue and Atlas, where FPGA-based prototypes allow the evaluation of interesting modifications to multi- and many-core designs and early software implementation.

III. Architecture design

We identified the set of features that can enable better support for irregular applications on general purpose architectures. First, a global address space across nodes eliminates the need to partition the data. If the global address space is scrambled, it also makes the network traffic more uniform, thus reducing the occurrence of hot-spots. Second, multithreading allows tolerating the latency of remote memory accesses by switching to other threads while the memory operation is performed. Finally, hardware support for synchronization allows implementing faster and finer grain synchronization. These features are also implemented in custom machines for irregular applications such as the Cray XMT, which supports a scrambled global address space, uses massively multithreaded processors (128 threads) and implements fine grained synchronization through full/empty bits associated with each memory word.

Figure 1 shows an overview of the proposed architecture. We start from an off-the-shelf architecture composed of simple, in-order cores and an on-chip interconnection. The on-chip interconnection interfaces the processing core with the memory controller for the external memory (DDR3) and the shared I/O peripherals. We add three custom components: the Global Memory Access Scheduler (GMAS), the Global Network Interface (GNI) and the Global SYNChronization module (GSYNC). The GMAS enables support for the scrambled address space. It also implements part of the support for latency tolerance, storing remote memory operations, and acts as a scheduler for the lightweight software multithreading. The GNI interfaces the on-chip interconnection with the external network. The GSYNC enables support for fine grain synchronization. Each core in the architecture attaches to a GMAS. The GMAS also needs to connect to an interrupt port of the core. Each node only includes one GNI and one GSYNC. Part of the external DDR memory is exposed as node-local memory, shared among all the cores in the node, while part is exposed as global address space, accessible by all the nodes of the system. Cores can include instruction caches, as they are read

only and do not require coherency. Data caches, instead, works only for private data or, if coherency is supported, for data shared at the node level. The global address space is, instead, uncacheable, as accesses of irregular applications are unpredictable and coherency across nodes would be too expensive in terms of overheads. The cores also support private scratchpads, useful for storing the contexts of the lightweight threads, enabling fast switching. If scratchpads are not present, pinning the thread contexts in the L1 caches may achieve similar effects.

A. Prototype

We implemented a prototype of the proposed architecture exploiting the Xilinx ISE Embedded Design Suite version 13.4. We employed four Xilinx ML605 boards, each mounting a Virtex-6 LX240T FPGA. Each board prototypes a node of the system. The boards communicate through the RocketIO GTX transceiver in a fully interconnected topology (i.e., each board is directly connected to all the other three). Since the boards only feature one pair of differential signaling coaxial interconnections, we added a daughter board to each of them with two additional pairs of coaxial interconnections. Data is exchanged on the links using the Aurora protocol, a lightweight link-layer protocol, exploiting an 8B/10B encoding. Each node hosts multiple 32-bit MicroBlaze cores, the highly customizable Xilinx soft-cores. We configured the cores with a 3-stage pipeline, including the hardware multiplier and the barrel shifter. As this work looks at irregular applications, which are mainly memory intensive, we did not implement the floating point unit and the branch prediction logic. We also did not implement the memory management unit, as functionalities for managing the global address space can be implemented separately from it. These choices limit the area footprints of the cores to the minimum, allowing to increase the number of cores, and consequently the injection rate of memory operations, inside a single node. We enabled L1 instruction caches for the external memory, allowing to use applications of arbitrary size without performance penalties. However, we did not enable the L1 data caches because of the lack of cache coherency. The MicroBlazes are connected together, to the shared DDR3 RAM controller and to the other components through the ARM AXI4 bus included in the Xilinx toolchain. As this implementation only supports up to 16 masters on a single bus segment, we implemented a two-level hierarchy of buses to allow scaling to higher number of cores. We found that AXI segments with up to 4 cores each offer the best trade-off in terms of area and timings. We included the proposed components in this basic architecture. The GMAS is interposed between the data port of each core and the AXI bus, to analyze every memory operations exiting the core. The GNI and the GSYNC, together with the DDR memory controller and the other peripherals, are instead connected to the last, shared level of the AXI hierarchy.

B. Programming model

The proposed system provides a global address space, implementing a shared-memory programming model on top of a distributed memory machine. The application developer can transparently allocate and free memory areas in the global address space by using standard memory allocation primitives. We specialized the standard *malloc* and *free* primitives to support both allocation in the shared global memory space and the node-local memory space of the system. After data allocation, the developer simply uses pointers as she or he would normally do on a shared memory system. The system supports multithreading and lock-based synchronization. We implemented an Application Programming Interface (API) that provides basic POSIX-like thread management and synchronization routines. Thread management primitives include thread creation, thread joining, and thread yield. Thread yield allows forcing a context switch at the software level. In all the other cases, thread switching is triggered only to tolerate the latency of remote memory accesses. Applications are developed in a Single Program Multiple Data (SPMD) fashion, where each thread executes the same code on different elements of the datasets. The synchronization primitives provide try-locks (spinning locks), unlock and barriers that exploit the GSYNC. To minimize thread switching latency, every core has its own private set of thread contexts. In the current prototype, contexts of the threads are stored in private scratchpads and do not migrate across nodes and cores. Load balancing across threads and cores is thus duty of the application developer. However, implementing work stealing is feasible, by exploiting the same communication features that enable the global address space at the price of slower context switching delays. This can be mitigated by employing context prefetch in the private memories.

IV. Component design

In this section, we present the implementation details of the hardware components that enable support for irregular applications.

V. GMAS

The GMAS enables the support of the global address space across all the nodes of the system, manages long latency memory operations and provides hardware support for scheduling the lightweight threads. Figure 2 shows the structure of the GMAS. The GMAS connects as a slave to a core and as a master to the AXI bus. It intercepts all the load/store operations that a core issues towards the external memory and the on-chip peripherals, and distinguishes which ones are node-local and which ones, instead, are operating on the global address space. As previously described, the cores also access privately a scratchpad for the thread contexts. Thread contexts can also be allocated in the node-local memory and pinned into the caches, as they are private per core and do not require cache coherency, which the MicroBlazes do not support. The global address space is scrambled across the DDR memories of all the nodes in the system: logically consecutive addresses are physically distributed on the various boards. The size of the scrambling is configurable. The GMAS also contains some registers for interfacing with the GSYNC, because the AXI bus in the Xilinx implementation does not support atomic transactions.

When a core issues a memory reference, the GMAS analyzes it. If it is directed towards the node-local address space (node-local portion of the DDR3 memory or node-shared peripheral), the GMAS simply forwards the memory reference to the AXI bus. Instead, if it is directed towards the global address space, the GMAS first de-scrambles the address, identifying the destination node. If the destination is the local node, it forwards the operation to the local memory controller without further processing. If the destination is a remote node, the GMAS sends the reference to the GNI for transmission into the external network. At the same time, the slot of the load/store queues (LSQs) pertaining to the thread that emitted the reference is updated with the related information. The LSQs have as many slots as the supported software threads. The *pending* bit of the slot is also set. The GMAS then sends an interrupt to the core, triggering the thread switching routine. Since the MicroBlaze pipeline remains stalled if a memory operation does not complete, the GMAS also replies with a canary value, allowing the interrupt routine to start. To signal that the GMAS returned a canary value, the *redo* bit

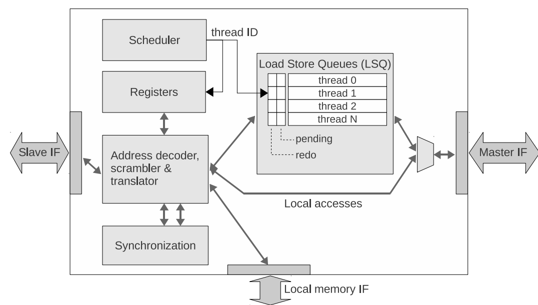

Fig. 2. GMAS internal structure

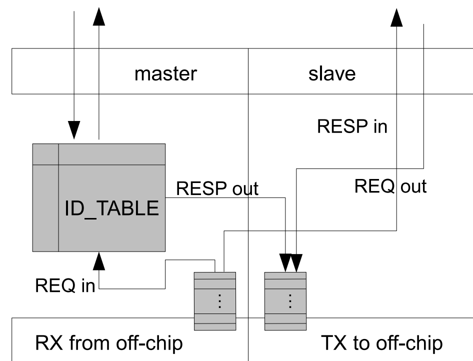

Fig. 3. GNI internal structure

in the related slot is also set. The thread switching routine selects another thread, getting its identifier from the GMAS scheduler. The scheduler chooses the new thread following a round robin policy among the active threads that do not have or have completed any pending remote memory operation.

When the remote memory reference completes, the response (acknowledge in case of a store, data in case of a load) reaches the GMAS. The GMAS stores the reply in the related slot of the LSQs, and resets the pending bit. The thread that generated the remote reference can now be scheduled again. When the scheduler selects that thread, the interrupt routines forces it to repeat the remote memory operation, allowing the core to read the correct value from the LSQs. After the memory operation completes, the redo bit is reset and the GMAS can reuse the slot for another remote memory operation of the same thread.

We designed the GMAS trying to minimize the added latency overhead for memory accesses. To do so, we used only combinatorial logic for all the components that a reference traverses before the GMAS generates the interrupt towards the core. The time interval between the issuing of a memory reference and the sensing of the interrupt adds to the overall latency of the memory reference. The rest of the time spent that a reference spends in the GMAS is instead already hidden through multi-threading, and thus it does not requires specific optimizations.

VI. GNI

The GNI interfaces the on-chip interconnection with the external network. It decodes and encodes memory transactions, encapsulates them in network packets, routes the packets across the network and translates the addresses and identifiers between the on-chip interconnection and the network format. We decided to implement translation because we want to completely decouple the on-chip interconnection protocol from the external network interconnection. In the case of AXI, we cannot directly forward AXI transactions as they are on the other nodes. In fact, transaction identifiers on the AXI bus are limited only to 16 bits (65K transactions), but we want a system able to scale in the number of nodes, potentially managing millions of references flowing into the external network.

Figure 3 shows the organization of the GNI. The GNI connects to the AXI bus both as a master and as a slave. Thus, it can redirect memory references from/to the on-chip interconnection and to/from the external network. The GNI receives outgoing requests on the on-chip slave interface, and sends them to the external network through the *sender* channel (TX). It receives incoming requests from the external network through the *receiver* (RX) channel and routes them into the on-chip interconnection through the master interface. On the other hand, the GNI receives outgoing responses from the on-chip master interface and sends them through the TX channel. It receives incoming responses through the RX channel and routes them into the on-chip interconnection through the master interface. The TX and RX channels of the GNI connect to a switch, which routes the packets through the nodes of the system. The GNI performs the translation on the data crossing the master interface (i.e., on incoming requests from the external network and on outgoing responses to the external network), through an integrated translation table. The switch interfaces the GNI with the three network ports, and performs the requested routing depending on the destination node.

When a core issues a load/store operation toward an address in the global space, the GMAS de-scrambles and determines its destination node. If the operation is directed towards an address in the memory space of another node, the GMAS generates an AXI transaction with the GNI as the destination. The transaction is associated with an identifier that contains the identifiers of both the core and the thread emitting the operation. When the GNI receives the transaction, it starts creating the network packet. It inserts the network address of the source and of the destination node into the packet header, adds the received transaction as the payload, and then sends the packet to the switch, which routes it accordingly to the address of the destination node into the external network. At the destination node, the destination GNI receives the transaction, and prepares to generate an AXI transaction towards the memory of the node exposed as global address space. First, the GNI stores the transaction identifier of the received memory operation, along with the network address of the source node, in the translation table, and generates a new AXI transaction identifier, which is a composition of the identifier of the GNI itself with a unique pending transaction counter. It then extends the physical memory address found in the network packet with the necessary bits to address the memory of the node, and injects the transactions into the on-chip interconnect. When the memory access completes, the GNI at the destination node receives the reply. It then retrieves the source node of the memory operation and its original identifier from the translation table, composes the network packet with the reply using this information, and sends the reply back to the source node. The GNI at the source node receives the network packet, disassembles it, and injects it directly in the on-chip interconnect, because the proper transaction id is already contained in the memory operation, thanks to the translation. Thus, the reply can finally reach the GMAS, concluding the remote memory access.

A. GSYNC

The GSYNC, together with the synchronization interface in the GMAS, implements fine-grain synchronization for the global address space. The GSYNC is a memory-mapped slave component, which integrates a lock table of configurable size. Each entry has a size of one bit. Each node of the system hosts a GSYNC, and each GSYNC manages synchronization only on the memory addresses, part of the global address space, of its own node. The entries of the lock table are associated with the local addresses of the global address space through direct mapping. Multiple memory addresses share the same lock bit, because the number of entries in the lock table is small with respect to the size of the memory. Thus, a lock request for an address can fail if the location is already locked for an aliased address.

To lock/unlock an address, the programmer employs a lock routine that writes the address into the dedicated memory-mapped lock/unlock registers of the GMAS. The write returns immediately. The GMAS identifies if the requested address pertains to the local node or to a remote node. If the address resides on the local node, the GMAS issues an operation towards the local GSYNC, addressed to the lock bit associated with the corresponding memory address. When the requested synchronization operation is a lock, the GMAS issues a load towards the GSYNC. The GSYNC handles the load as a 1-bit swap, replacing the value in the addressed lock bit with a 1, and returning the previous value. If the returned value is 0, the lock is taken. Otherwise, the lock is not taken. If the requested operation is an unlock, the GMAS simply issues a store with value 0 for the addressed lock bit. This approach allows implementing lock/unlock operations without requiring support for specific atomic instructions in the cores, the on-chip interconnect and the external network. If the requested address of the global address space is located on a remote node, the load/store operation associated with the lock/unlock operation is treated like any other remote memory operation, only addressing the memory-mapped registers of the GSYNC on the remote node. The operation is thus sent to the GNI of the source node, which encapsulates and injects it into the network. The GNI on the receiving node disassembles the packet, and sends the operation to the GSYNC of the node, which provides the reply. The reply is then sent back to the source node. In presence of a remote synchronization operation, the GMAS also sends to the core the interrupt that triggers thread switching for hiding the latency, like all remote memory operations. In software, the lock routine is implemented with busy waiting. If the lock is not taken, the routine keeps repeating the operation. The unlock is, instead, non-blocking.

The GSYNC also includes two memory-mapped registers set to zero, in hardware, at system boot. These registers allow implementing hardware barriers for initial inter-node synchronization.

VII. Experimental evaluation

We synthesized four-node architectures including 1 to 32 cores per node, each one implementing up to 4 threads. We set the basic frequency of the prototype at 100 MHz. Each node hosts 512 MB of DDR3 memory. Of these, 32 MB are node-local, while the rest is exposed in the global address space, for a total of 1920 MB of shared-memory across the four nodes. The scrambling is at a granularity of 8 bytes (2 memory words, as the MicroBlaze is a 32-bit core). The GSYNC lock tables have 8096 entries. The GTX transceivers provide a usable bandwidth of 500 Mbit/s each. We measured a non-contended round-trip time of 403 cycles, which is the same for all the nodes, because they are in a fully connected mesh. Thread contexts are allocated in private scratchpads for each MicroBlaze. The thread switching time is 232 cycles, composed of 41 cycles to launch the

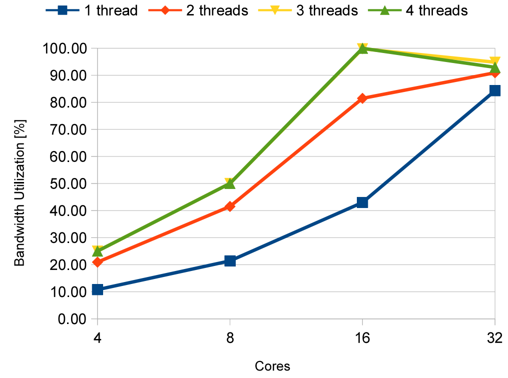

Fig. 4. Pointer chasing bandwidth utilization

Interrupt Service Routine (ISR), 65 cycles for saving the old context, 20 cycles for loading the scheduler, 50 cycles to load the new context, 24 cycles for resetting the interrupt and exiting the switching function, and 32 cycles for exiting the Interrupt Service Routine.

We executed two typical irregular kernels on the prototype. Given the unpredictability of network accesses, and the memory scrambling, reaching the maximum performance for irregular applications on multi-node systems means maximizing the network bandwidth utilization.

The first kernel is pointer chasing. In this benchmark, every thread performs a full exploration of a vector of pointers that randomly redirect one to the other. We explore configurations from 4 to 32 cores per node, each with 1 to 4 threads. Figure 4 shows the bandwidth utilization. As previously stated, the available network bandwidth is 1.5 Gbps (500 Mbit per channel). However, there is a 1/3 overhead due to the network packet headers, so the effective bandwidth is 1 Gbps. The graph shows that the bandwidth utilization increases with the number of cores implemented in the architecture, because more cores increase the injection rate into the network. Thread switching also allows increasing bandwidth utilization up to three threads per core. When increasing the number of threads from one to two, the increase in bandwidth utilization is more significant, because the ratio between the RTT and the switching delay allows issuing another reference. However, the system still allows an increase in bandwidth utilization with three threads per core, as they allow a better distribution of the network accesses on the three available network channels (one for each other node). The utilization provided with four threads is, instead, substantially identical to the utilization with three threads per core. With three and four threads, there is a slight decrease in the utilization when moving from 16 to 32 cores, because the contention in the internal interconnection is higher.

The second kernel is the Breadth First Search (BFS) exploration of a graph. We adapted the code from a standard queue based, level synchronous implementation for the Cray XMT [22], exploiting the threading and synchronization API of the prototype. The graph has size 100,000 vertices, each with 80 neighbors in average, for a total of 3,998,706 traversed edges. Figure 5 shows the throughput (in Traversed Edges Per Second - TEPS) of the algorithm when increasing the number of cores from 4 to 32 per node and the number threads from one to four per core. The throughput increases while increasing the number of cores, because they increase the injection rate of references into the network, maximizing network utilization. We see the biggest performance increase when increasing the cores from 4 to 8 per node. The system keeps increasing the

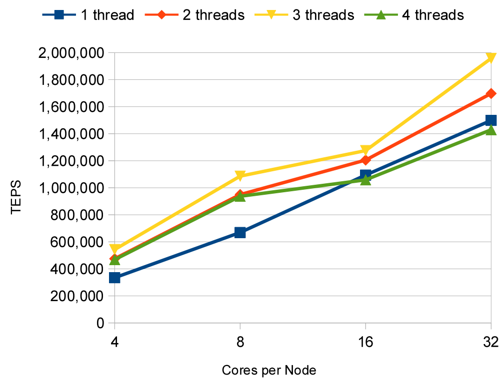

Fig. 5. Performance of the BFS in Traversed Edges Per Second (TEPS)

throughput up to 32 cores, but the slope of the curves from 8 to 32 reduces. Increasing the number of threads from one to two increases the throughput, because thread switching allows injecting further memory references into the network when waiting for remote memory operations to complete. Like in the pointer chasing, there still is a slight benefit in increasing the number of software threads to three, because of a better distribution of the network accesses. However, when increasing the number of threads to four, the throughput reduces, and is identical to two threads for 4 and 8 cores, and to one thread for 16 and 32 cores. The BFS implementation exploits a lock for the queue containing the next vertices to explore that is contended by all the threads. Thus, increasing the number of threads increases the contention on the GSYNC containing such lock. From an inspection of the system counters, we also see a slightly higher amount of network traffic directed towards node 2, because of the presence of such a lock and of the lock used to implement the level-synchronous barrier.

VIII. Conclusion

In this paper, we presented an architecture targeted at efficiently executing irregular applications. Rather than focusing on full-custom architectures or on completely commodity systems with ad-hoc runtimes, we propose a set of hardware and software components that can extend many-core architectures designed with off-the-shelf cores and IPs. The hardware components include a module for enabling a scrambled global address space across nodes and support for long latency remote memory operations (GMAS), a module for implementing fine-grained hardware synchronization (GSYNC) and an integrated network interface that translates between address domains and injects network packets into the internal network (GNI). The software components implement multi-threading, supported by a hardware scheduler integrated into the GMAS. We validated these components and explored basic trade-offs in terms of bandwidth, cores-per-node and threads-per-core with a 4-node FPGA prototype. We executed two typical irregular kernels on the prototype (pointer chasing and BFS) and verified scaling in terms of bandwidth utilization and performance when increasing the number of cores and the number of threads.

References

[1] U. Nawathe, M. Hassan, K. Yen, A. Kumar, A. Ramachandran, and D. Greenhill, "Implementation of an 8-core, 64-thread, power-efficient sparc server on a chip," *Solid-State Circuits, IEEE Journal of*, vol. 43, no. 1, pp. 6 –20, jan. 2008.

[2] J. Feo, D. Harper, S. Kahan, and P. Konecny, "ELDORADO," in *CF '05: International Conference on Computing Frontiers*, 2005, pp. 28–34.

[3] M. Ceriani, S. Secchi, A. Tumeo, and O. Villa, "Prototyping hardware support for irregular applications," in *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, ser. RAPIDO '13, 2013, pp. 4:1–4:8.

[4] M. Ceriani, G. Palermo, S. Secchi, A. Tumeo, and O. Villa, "Exploring manycore multinode systems for irregular applications with fpga prototyping," in *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 2013, p. 238.

[5] S. Secchi, M. Ceriani, A. Tumeo, O. Villa, G. Palermo, and L. Raffo, "Exploring hardware support for scaling irregular applications on multi-node multi-core architectures," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, 2013, pp. 309–313.

[6] K. Wheeler, R. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *MTAAP '08: Workshop on Multithreaded Architectures and Applications*, 2008, pp. 1–8.

[7] "Upc language specifications, v1.2," UPC Consortium, Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.

[8] "Co-array fortran. available at: http://www.co-array.org."

[9] A. Krishnamurthy, A. Aiken, P. Colella, D. Gay, S. L. Graham, P. N. Hilfinger, B. Liblit, C. Miyamoto, G. Pike, L. Semenzato, and K. A. Yelick, "Titanium: A high performance java dialect," *Concurrency: Practice and Experience*, vol. 10, no. 11-13, 1998.

[10] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, "X10 language specification version 2.2," July 2012.

[11] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[12] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006.

[13] D. Bonachea, "GASNet Specification, v1.1," CS Division, EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-02-1207, October 2002.

[14] "Cray gemini interconnect. available at: http://www.cray.com/products/xe/technology.aspx."

[15] H. Froning and H. Litz, "Efficient hardware support for the partitioned global address space," in *CAC 2010: 10th Workshop on Communication Architecture for Clusters, in IPDPS '10*, 2010, pp. 1–6.

[16] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin, "Crunching large graphs with commodity processors," in *HotPar '11: 3rd USENIX conference on Hot topic in parallelism*, 2011, pp. 10–10.

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD '10: ACM International Conference on Management of data*, 2010, pp. 135–146.

[18] D. Patterson, "RAMP: research accelerator for multiple processors - a community vision for a shared experimental parallel HW/SW platform," in *ISPASS 2006: International Symposium on Performance Analysis of Systems and Software*, 2006, p. 1.

[19] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun, "ATLAS: a chip-multiprocessor with transactional memory support," in *DATE '07: Conference on Design, Automation and Test in Europe*, 2007, pp. 3–8.

[20] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P. yves Droz, "Ramp blue: a message-passing manycore system in FPGAs," in *In FPL'07: International Conference on Field Programmable Logic and Applications*, 2007, pp. 27–29.

[21] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanovic, and D. A. Patterson, "A case for fame: Fpga architecture model execution," in *ISCA*, 2010, pp. 290–301.

[22] D. A. Bader and K. Madduri, "Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the cray MTA-2," in *ICPP '06: International Conference on Parallel Processing*, 2006, pp. 523–530.