Exploring Efficient Hardware Support for Applications with Irregular Memory Patterns on Multinode Manycore Architectures

Marco Ceriani, Simone Secchi, Oreste Villa, Antonino Tumeo, Member, IEEE, and Gianluca Palermo

Abstract—With computing systems becoming ubiquitous, numerous data sets of extremely large size are becoming available for analysis. Often the data collected have complex, graph based structures, which makes them difficult to process with traditional tools. Moreover, the irregularities in the data sets, and in the analysis algorithms, hamper the scaling of performance in large distributed high-performance systems, optimized for locality exploitation and regular data structures. In this paper we present an approach to system design that enable efficient execution of applications with irregular memory patterns on a distributed, many-core architecture, based on off-the-shelf cores. We introduce a set of hardware and software components, which provide a distributed global address space, fine-grained synchronization and latency hiding of remote accesses with multithreading. An FPGA prototype has been implemented to explore the design with a set of typical irregular kernels. We finally present an analytical model that highlights the benefits of the approach and helps identifying the bottlenecks in the prototype. The experimental evaluation on graph based applications demonstrates the scalability of the architecture for different configurations of the whole system.

Index Terms: Computer architecture, irregular applications, high-performance computing, distributed computing, multithreaded architectures, parallel architectures, field programmable gate arrays, prototype

1 Introduction

The interest towards algorithms and applications for data knowledge discovery on big, unstructured data, is rapidly increasing. The most visible use of such algorithms is the analysis of social networks or customer habits for marketing and recommendations. They are also used in various areas of research and business, such as the study of interactions in biological systems [1], [2] or the analysis of logistic networks [3]. In these applications, the data are stored in very large pointer-based structures, such as graphs and unbalanced trees, which can be composed of many millions of nodes and billions of edges, as in the case of social networks [4].

Because of the large size of the data, distributed high performance computing (HPC) systems are required to run the algorithms. However, the irregularity of the data structures makes it very difficult to partition the data effectively across the nodes. In addition, the accesses to these structures usually present poor spatial and temporal locality, causing frequent fine-grained requests to the memory hierarchy and network, with a very irregular mix of long and short latencies [5]. Finally, the dynamic nature of the data structures makes the task of optimizing their layout even more difficult.

Past and current HPC systems mainly target traditional scientific applications, which process highly regular data structures. These applications feature high arithmetic intensity and data locality, and are relatively easy to partition into loosely dependent tasks. Therefore, modern HPC systems are designed as clusters whose nodes include powerful multicore processors, large memories and complex cache hierarchies, and are interconnected by high bandwidth communication networks. However, deep cache hierarchies are beneficial only to algorithms with high data locality. Instead, applications with highly irregular memory patterns and large data sets, from now called *irregular applications*, perform poorly [6]. Furthermore, knowledge discovery applications have large degrees of parallelism exploitable by HPC clusters. However, because of the connectivity of the graphs, the performance on large systems is dominated by the communication time [7].

The Cray XMT [8] is a supercomputer explicitly designed for irregular applications. The three key features of this distributed machine are: massive *multithreading*, fine-grained *synchronization* and *global memory address space*. Each node of the XMT includes a custom hardware multi-threaded processor, named ThreadStorm, that supports up to 128 hardware contexts. The large number of thread contexts allows keeping the processor pipeline busy during long latency operations, such as remote memory accesses (RMA) across the network. The system also includes a remote memory access module, placed between the ThreadStorm processor

- *M. Ceriani and G. Palermo are with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, Italy. E-mail: {marco.ceriani, gianluca.palermo}@polimi.it.*
- *S. Secchi is with ARM Ltd, Cambridge, U.K. E-mail: simone.secchi@arm.com.*
- *O. Villa is with NVIDIA Research, Santa Clara, CA. E-mail: ovilla@nvidia.com.*
- *A. Tumeo is with the High Performance Computing Group, Pacific Northwest National Laboratory, 902 Battelle Blvd, MSIN J4-30, Richland, WA. E-mail: antonino.tumeo@pnnl.gov.*

Manuscript received 12 Dec. 2013;
revised 1 June 2014;
accepted 15 July 2014.
Date of publication 5 Aug. 2014;
date of current version 17 May 2017.
Recommended for acceptance by S. Ranka.

© 2014 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. To access the final edited and published work see http://dx.doi.org/10.1109/TPDS.2014.2345073

and the Seastar network. This module provides a globally-shared address space, which is transparently mapped and uniformly scrambled across all the nodes of the system. This simplifies the algorithms design, by eliminating the need to manually partition the data. The XMT approach is viable because massive multithreading enables latency tolerance in place of latency reduction, and the address scrambling makes the average data access latencies more uniform. Finally, the XMT associates a full/empty bit with every memory word, used by atomic operations and blocking memory accesses. The result is a mechanism for dynamic conflict avoidance with very low overhead, helpful in irregular and unpredictable algorithms. However, the XMT does not run traditional regular workloads as efficiently, and its evolution may not be economically sustainable, since it is a custom machine with a limited market.

Meanwhile, paradigms for novel processor designs are shifting towards many-core architectures, which integrate hundreds of small and low-power cores in a single die. The unprecedented levels of parallelism exposed are ideal for building new HPC systems, especially for running memory bound and massively parallel applications. However, for performance reasons, all these architectures favor distributed programming models, focused on locality exploitation, independently from the actual memory model [9], [10], [11]. Therefore, it is of crucial interest to understand how to evolve these novel architectures without neglecting the characteristics of irregular algorithms.

One of the driving principles of the XMT design was improving the performance of irregular applications without requiring complex optimizations from the programmer. Provided that enough parallelism is available, the XMT's hardware and software enable automatic and transparent latency hiding and workload balancing. In this paper we study the possibility to achieve the same result starting from standard, therefore cheaper, off-the-shelf components, including processors and memory controllers (MC).

The main contributions of this paper are:

- the design of custom hardware and software components that provide a virtual global address space, transparently to the application, and latency hiding through software multithreading;
- the development of a prototyping platform based on multiple field programmable gate arrays (FPGAs), employed to validate the approach;
- an analytical model that correlates single-node and system-wide architectural parameters with system performance metrics.

The paper is organized as follows. Section 2 discusses the related work. Section 3 describes the architecture building blocks and the custom components that we designed, while Section 4 gives an overview of the programming model and the software execution flow. Sections 5 and 6 describe the FPGA prototype that we developed for exploration purposes and an analytical model that links the most relevant system-level parameters to the overall performance, giving useful insights on the architecture strengths and bottlenecks. Section 7 presents and discusses the results of the experimental evaluation and, finally, Section 8 concludes the paper.

2 Related Work

The Cray XMT [8], mentioned in Section 1, is a reference for many research projects on irregular parallel applications, because its peculiar features provide both good performance and a simple programming model. Its main drawbacks are the cost and the excessive specialization. These motivate the search for ways to integrate its features into more traditional architectures and systems.

Multithreading allows tolerating long latency memory accesses, in combination or in opposition to latency reduction techniques such as caches, prefetching and operation reordering. Interleaved multithreaded processors switch from thread to thread on a cycle-by-cycle basis, hiding the stalls due to short distance dependencies. Examples of interleaved multithreaded processors are those in the Tera MTA and Tera MTA 2, the ThreadStorm in the Cray XMT [8], and the Oracle SPARC processors based on the Niagara [12] architecture. The Niagara-based processors have smaller caches than other contemporary architectures, but, thanks to a higher number of threads, can tolerate memory access latencies. In the UltraSPARC T1 each of the eight cores can execute four threads, while the SPARC T3 can run eight threads on each of its 16 cores. Nevertheless, they are not designed to tolerate the network latency due to remote memory accesses, nor they provide a global address space. The ThreadStorm, with 128 hardware thread contexts in a single core, can effectively tolerate network latency, and supports a scrambled global address space in hardware. However, it is not as effective when running regular workloads.

An alternative to temporal multithreading is the simultaneous multithreading (SMT) implemented in superscalar architectures such as the IBM POWER7 and the Intel Sandy Bridge. SMT keeps multiple thread contexts active at the same time, and identifies independent instructions to be issued simultaneously on the available execution units. However, the objective of SMT is to maintain high the utilization of the processor functional units, rather than hiding memory latency.

The use of software-based fast thread switching for latency tolerance has been explored in several works. For example, the Qthreads [13] library implements software fine-grained multithreading, XMT-like full/empty bit semantics, and supports various architectures (POWER, x86, SPARC, Tilera). The APRIL processor architecture [14] in the Alewife machine, instead, exploits software block multithreading to hide memory access latencies. However, these approaches have not been thoroughly explored in multi-node machines, with support for distributed or global address spaces.

CPU Scheduling is a topic of great importance, especially in parallel and distributed systems. A widely used strategy to enhance load balancing in parallel programs with fine-grained parallelism is Work-Stealing. According to it, each processing element has a dedicated queue, but can steal tasks from other queues instead of stalling. To reduce the algorithm overhead, different techniques are proposed [15], [16]. In this paper we are addressing data-intensive algorithms that execute long latency requests very frequently. Hence, we neglect the details of the scheduling algorithm, instead focusing on identifying the requests and on automatically removing the stalls from the pipeline with fast context switches. We consider load balancing techniques, such as work-stealing, orthogonal and synergic to our proposal.

Partitioned Global Address Space (PGAS) programming models offer a virtual global address space space without neglecting data or thread locality. These programming models are progressively gaining traction in the HPC community, because they can reduce the effort for programming large-scale machines without trading off too much performance. They can be found in languages such as Chapel [17], Unified Parallel C [18], Titanium [19], Co-Array Fortran [20], X-10 [21] or libraries such as GASNet [22] and Global Arrays [23]. However, these approaches still are mostly software-based, and their runtimes can generate significant overheads [24]. An example of hardware support for the global address space is the Cray Gemini interconnect [25], that allows pipelining of remote references. Nevertheless, remote DMA operations in current high performance networks require to be set up through complex memory registration operations, usually involve limited portions of the memory, and are optimized to transfer large chunks of data. Therefore, PGAS models are still more amenable to regular workloads that generate coarsegrained transactions.

Graph based data structures appear in many irregular parallel algorithms. Grappa [26] is a software system which integrates a PGAS programming model and multithreading to crunch large graphs on commodity clusters. The Gobal Memory and Threading (GMT) [27] runtime library adds message aggregation to the PGAS memory model and lightweight software multithreading to further enhance network utilization. Other tools, such as GraphLab [28], Pregel [29] and Giraph, exploit vertex programs that run on each vertex and interacts along edges, exploiting a bulk synchronous parallel model. Interactions either use messages (Pregel and Giraph) or shared states (Graphlab). However, even if we use graph algorithms as target benchmarks, our objective is not to implement a software library exclusively for graphs. Instead, we look at integrating small modules at all system levels (hardware architecture, runtime) to enhance the execution of irregular applications with limited impact on the programming model.

Hardware support for global address space is provided by the RMA modules in the Cray XMT system, which forwards memory requests across the network and also provides fine-grained synchronization by extending the memory semantics. To the best of our knowledge, the XMT is the only commercial system to provide such features. We follow a similar approach for the global address space, but we target commodity cores and memory subsystem. A custom module is proposed also in [30]. The module extends the HyperTransport protocol from single to multiple nodes, forwarding Programmed I/O (PIO) operations over a custom high performance network. The approach is validated through FPGA prototyping. Our approach has similar basis, but is focused on the on-chip design rather than the external network. Furthermore, we look at how to tolerate access latencies through multithreading and include fine-grain synchronization support. Finally, we also evaluate the system-level implications of the design by presenting a model for latency tolerance.

Analytical models for multithreaded processors have been developed in the past. In [31], Agarwal presents an analytical performance model for cache-coherent multithreaded distributed processors, which includes cache interference, network contention, context-switching overhead, and data-sharing effects. In [32] the authors present an analytic framework to evaluate the performance of a spectrum of machines, including interleaved and block multithreaded processors. We follow an approach similar to these works, introducing an analytic model that correlates the degree of multithreading and switching delay to memory/network bandwidth and latency. Unlike the afore-mentioned works, we focus in particular on irregular applications, characterized by a high ratio of remote memory operations with respect to local ones, and we ignore the cache hierarchy, which is ineffective for this kind of workloads [6].

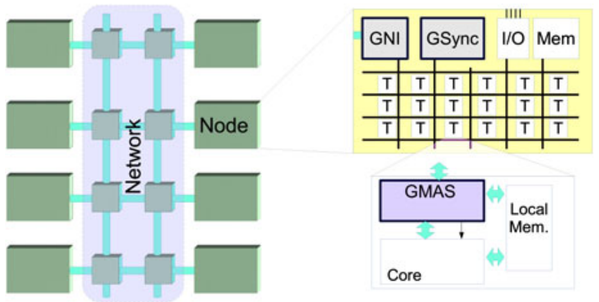


Fig. 1. Hierarchical overview of the architecture: whole system, single node and single tile detail.

3 Architecture Description

This section presents in detail the proposed architecture. We start by providing a high level overview of the system. Then, we delve into the details of the required features, and how the system provides them.

Fig. 1 shows an overview of the HPC system, with different levels of details going from the whole cluster (on the left) to the single core. The cluster is composed of many nodes interconnected through a high performance network. Our architecture is independent of the topology and protocol used for the network. The network design only affects the system performance and its optimal configuration. Every node of the cluster consists of a many-core system, composed of many processing tiles (T) connected by an on-chip network to the memory controller, peripherals and the external network interface. Each tile, in turn, includes a core, a private memory element, which can be a scratchpad or an L1 cache, and a custom memory interface adapter. The purpose of the private memory is to store near to the core the portions of the address space that show locality of references, such as the stack and the code segment.

We have enhanced the system with three on-chip custom components, highlighted in Fig. 1: the global memory access scheduler (GMAS), the global network interface (GNI) and global synchronization (GSYNC). These components jointly provides a set of features that enable efficient execution of irregular applications:

- transparent global address space;
- reduction of hot-spots formation in the network;
- automatic hiding of network latency through multithreading;
- fine-grained global synchronization primitives.

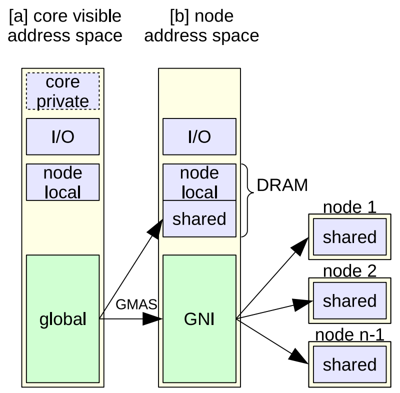


Fig. 2. Address space layout, as seen by (a) the cores and (b) the on-chip network.

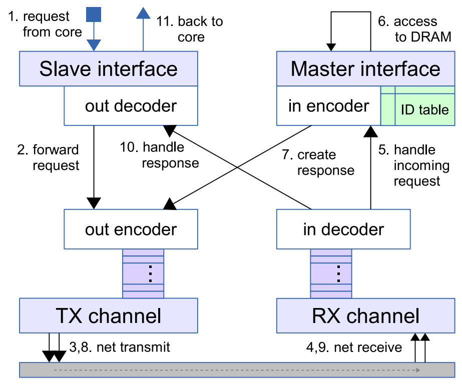


Fig. 3. GNI internal structure.

An instance of the GMAS is attached between the data port of each core and the on-chip network, so that it can intercept all data accesses and translate global addresses to physical ones. The GMAS also connects to the interrupt port of the core, in order to trigger the scheduler and interact with the run-time. Each node includes a single GNI instance, shared by all its tiles. The GNI provides access to the remote sections of the global address space though memory mapping. Lastly, the GSYNC provides synchronization support, by implementing a lock map that each core can access by reading or writing dedicated memory mapped registers inside the GMAS. All the blocks are designed for minimal impact on the pre-existing components, and can be disabled to run applications with high data locality, or based on different concurrency models.

The rest of this section describes how the custom components provide the desired features, and details their implementation.

3.1 Global Address Space

The first feature offered by the designed architecture is hardware support for a global address space across all nodes and application instances. To create a global address space, part of the memory range in each node is reserved for global use. All these regions are then composed in a global address range, logically contiguous but physically distributed on the whole system. The hardware support consists in exposing this address space to the cores, and in automatically forwarding remote memory requests to the corresponding nodes. Fig. 2 shows three different address spaces. On the left, there is the address space visible to each core, which includes the private memory, local to the core, the peripherals, the node-local memory, and the global space. The second address space is composed of the physical addresses seen by the on-chip network. A small part of the global address space is mapped into the reserved portion of the node memory, while the remaining is mapped on the GNI addresses. This first mapping is performed by the GMAS inserted between the data port of a core and the local network, and allows direct access to the local portion. Finally, the GNI maps the remote memories to portions of its own address range, exposing them to the on-chip space.

The GNI (Fig. 3) acts as a transparent bridge between the node-local network and the system network, converting the transactions from one protocol to the other. In details, it decodes memory transactions, encapsulates them inside network packets, and performs translation of addresses and identifiers. The GNI handles outgoing and ingoing transactions concurrently, using separate queues and paths, and serves both categories with a first come first served policy. The GNI receives long latency requests as memory mapped load/store transactions through the on-chip network. It extracts from the memory address both the identifier of the destination node and the physical memory address, used at the destination. Then, it creates a network packet that includes as payload the requested operation and its arguments. The GNI also generates a unique transaction identifier, which includes the identifiers of the originating core and thread, as well as the node address. Hence, all the information required to match the responses with the pending threads are included in the packets themselves. The destination GNI decodes incoming requests and converts them into memory accesses towards the local DRAM, then sends the responses along an analogous path to the originating core.

To make programming easier, we require both load and store transactions to be non-posted, i.e., they both require a response from the receiver. Therefore, all types of transactions require a full round-trip over the network. This requirement ensures that all memory writes included in a critical section are concluded and visible by every other node before exiting the section itself, independently from the distance of the destination.

In order to route the responses back to the originating node, the network packet headers have to include both the source node address and an identifier field large enough to associate the transaction to the issuing thread. These two fields together, therefore, should be capable to represent the maximum number of threads supported on the whole system. On the other hand, the on-chip networks and buses are usually designed with much smaller transaction identifiers. Hence, the GNI includes a translation table that dynamically assigns unused local identifiers to the incoming global transactions, for accessing the DRAM and GSYNC.

As previously said, the GMAS is also involved in providing a global address space. Since it is inserted between the cores and the on-chip network, it intercepts all memory accesses and identifies those directed towards the global address space. It can then modify the addresses, and even transform the requested operations into different ones. In particular, accesses to the local portion of the global space are directly forwarded to the memory controller, while the other global accesses are redirected to the GNI, with the addresses rewritten in the format that it can understand.

3.2 Reducing Hot-Spot Formation

One of the problems with parallel and distributed systems is the run-time formation of hotspots due to temporary high contention on single resources, which generate work imbalance and reduce performance. If a significant fraction of global memory accesses are directed to a single node, then the system performance is bound to the bandwidth of its network interface. The desired optimal behavior, instead, is to evenly distribute the accesses on all the nodes. One way to achieve it is to use the complex mappings of memory addresses to nodes to minimize the clashes caused by concurrent visits on the data structure. This approach has been used for optimizing parallel access to memory banks, e.g. in [33], and to distribute the workload in the Cray XMT.

We followed this approach, and added a scrambling function inside the GMAS, which scrambles global memory addresses before mapping them on the nodes. The function is configurable, and preserves very few last-significant bits, in order to distribute the addresses on the nodes with a very fine granularity, e.g. 64 bytes as the Cray XMT [8].

The scrambling has the effect of breaking structures and variables larger than the selected granularity and of scattering them across the nodes of the system. This is useful when accessing data structures such as the adjacency lists used in graphs and trees, because it increases the probability of distributing subsequent requests on multiple nodes, balancing the network traffic. In addition, it allows spreading the data structures on all the nodes in the system, independently on their size. This enables exploiting all the system resources even on small problem instances, and to evenly distribute multiple data structures.

The result of the address scrambling is an uniform distribution (in average) of the traffic over the system [34].

3.3 Latency Tolerance

In addition to providing the global address space, the GMAS includes various features to hide the long latency of network transactions through multithreading. The goal is to automatically exploit task-level parallelism to improve both the processor and network bandwidth utilization, by executing multiple threads while waiting for a response from the network. To achieve this, the GMAS performs the following actions when it identifies a remote request. First of all, it takes charge of waiting for the answer, internally keeping track of the pending status of the current thread. Then, it notifies the event to the core to start a context switch and, finally, it selects the next thread to execute. These actions involve a set of hardware components inside the GMAS and of software routines for the context switch.

Fig. 4 shows the internal structure of the GMAS. It includes a load/store buffer (LSB) that stores the information required to handle the remote requests. Each slot corresponds to a thread that can run on the processor, and includes a status bit and a field to store the result of remote requests when they are received. We are currently targeting

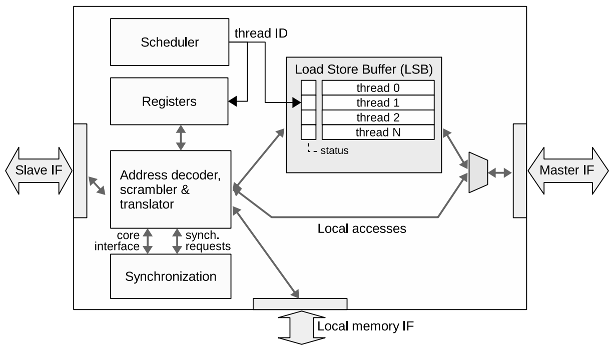


Fig. 4. GMAS internal structure.

cores with an in-order pipeline, which can handle a single pending memory access at a time. Therefore, it is necessary to add a buffer external to the core to free the internal load/store unit and exploit thread-level parallelism.

While the GMAS stores the request for a remote transaction in the LSB, it also triggers an interrupt towards the core, causing a context switch. The core execution is then decoupled from the network transaction, hiding its latency. The specific details on how the core pipeline handles an external interrupt vary from core to core. However, in many architectures, a memory instruction that reaches the memory access stage (or equivalent) is allowed to complete, delaying the interrupt handler. This would prevent the core from servicing the interrupt and perform the context switch. To include in our architecture design also simple cores that have this limitation, the GMAS also sends to the core a fake memory response. When restoring the context of the thread, the program counter is set to the last instruction executed, instead of the next instruction, as it usually happens. This enables the core to re-issue the memory request and to fetch the result from the LSB.

Another key component of the GMAS is the hardware scheduler. The scheduler selects the next thread to execute and exposes its ID to the software through one of the memory mapped registers. The GMAS always triggers the scheduler together with the interrupt signal, hence scheduling is masked by the saving of the old thread context, reducing the context switch latency. In addition, the scheduler does not need to access the memory, because the GMAS stores the statuses of the threads running on its core in the LSB. We selected the Round Robin algorithm for the prototype, because it is predictable and permits fast hardware implementations.

In addition to preempting the threads on remote memory accesses, the GMAS also allows the threads to yield the processor by writing in a specific register. We use this feature in the software synchronization primitives, to block the thread while waiting on local resources.

3.4 Synchronization

Regular data structures, such as dense matrices, allow identifying most of the data dependencies at design or compile time. Conversely, algorithms based on arbitrary graphs and trees require run-time conflict avoidance, because the interference depends on the data instance. Therefore, the synchronization primitives offered by a system have a huge impact on both the applications complexity and performance. Common APIs based on mutex variables incur in excessive overhead, both in terms of memory size and time, because every element in the shared data structure has to be protected. Consequently, there is a need for hardware support for fine-grained synchronization, which permits precise control of the threads with low overheads.

We included in the system a dedicated hardware component to support synchronization, the GSYNC, and an interface inside the GMAS that exposes its features to the software. The GSYNC integrates a lock table, which is exposed on the on-chip interconnection address space. The reason for an interface inside the GMAS is that each GSYNC manages the synchronization on the memory addresses physically mapped on its own node. Hence, accesses from the core to the GSYNCs have to pass through the scrambling and mapping logic included in the GMAS, as described in Sections 3.1 and 3.2.

We have implemented two synchronization operations, the *try-lock* and *unlock*, which are exposed to the application through two registers inside the GMAS. Since each node includes many cores, the accesses to the GSYNC have to be atomic. Therefore, we implemented the two operations as memory requests in the GSYNC address space, respectively as read and write accesses. This keeps the implementation independent from the on-chip interconnection, and bypasses its limitations.

When the GSYNC receives a load request, it sets the lock bit associated to the address to 1 and returns the previous value. Conversely, a write request always sets the bit to 0. We implemented in software a *lock* operation, using a spin loop that yields the processor after every failed attempt. Even if the yields increase the traffic generated, they are required to prevent deadlocks. The reason is that only remote accesses cause a preemption, so a thread spinning on a lock owned by another thread of the same processor would cause a deadlock.

In the prototype, the lock table internal to the GSYNC is much smaller than the memory it protects. Consequently, multiple addresses share the same entry. The GMAS performs the mapping among addresses and entries by considering a subset of the address bits and encoding the entry number as part of the address used to access the GSYNC. This direct mapping may cause false failures of the lock primitive, increasing spinning times. However, the simplicity of the logic limits the latency of the transactions, providing a reasonable tradeoffs for the purpose of prototype. A more robust implementation could be obtained by replacing the direct mapped table with a content addressable memory (CAM). From the protocol side, this requires to actually send to the GSYNC the whole address to lock. From the hardware side, the on-chip interconnection has to provide adequate support for atomic transactions. While in general this is a reasonable approach, in the very first implementation of the prototype we preferred the flexibility in architectural design over the robustness of the system.

3.5 The Processing Core

The system architecture is completely independent from the instruction set of the cores. It only weakly depends on their internal micro-architecture. We are currently focused on in-order single-issue cores. This prevents the threads from overlapping the execution of multiple memory accesses and simplifies the automatic context switch triggered by them. The core also needs to support precise interrupt handling.

A second requirement for the implementation of the prototype is the availability of a dedicated low-latency memory directly connected to each core, to store the thread contexts and allow fast context switches. If, instead, a cache hierarchy is used, the pressure on the first level caches can produce a high miss rate during the context switch [26]. Hence, fast context switches would be possible only if the memory hierarchy allows pinning the contexts inside the lower levels.

4 Programming and Execution Model

This section describes the programming model and execution flow of the designed architecture. Thanks to the custom hardware modules specifically designed for this architecture, the programming model can replicate the shared-memory semantic on top of a distributed memory system, with POSIX-like multithreading and lock-based synchronization. The applications are designed following the single-program-multiple-data (SPMD) paradigm, where every core of the system runs the same application with different inputs.

During the system boot, every core reads a unique node identifier from dedicated hardware and a unique core identifier (within the node) from its attached GMAS. These identifiers are exposed to the application to distinguish the application instances and distribute the workload. Then, the predefined master core initializes the global data structures used by the run-time, while the remaining cores wait on a barrier allocated at a fixed address inside the global space.

As discussed in Section 3.3, multithreading enables each core to tolerate the latency of remote memory accesses. The applications start with one master thread per core, which can create child threads at anytime during application execution. The switching among the threads in a core occurs at every remote memory reference, or using an explicit yield command. In detail, when a GMAS detects that its core has issued a remote memory reference, it sends an interrupt. Within the interrupt service routine, the core saves the context of the running thread and switches to the next one, if available. The threads are selected by the hardware scheduler inside the GMAS, using a round robin policy among those that have no pending operations in the LSB (see Fig. 4). A thread can yield the processor by writing the command in a GMAS register. The write makes the GMAS raise the interrupt signal as if a remote request had been issued.

Communication among threads should take place only through the global address space, whether they run on different cores or on the same one. However, the node memory can still be used for optimization, e.g. for storing temporary local work-lists. Inter-thread synchronization is performed through explicit *lock* and *unlock* primitives, described in Section 3.4. In addition, barrier primitives implemented in software are provided for bulk synchronization. Since the synchronization happens in the global address space, it can generate remote requests and trigger context switches. Moreover, a failure to acquire a lock also triggers a context switch, preventing deadlocks in case the lock is held by another thread on the same core.

5 Prototyping Platform

This section describes the FPGA prototyping platform that we developed to evaluate the designed architecture. Hardware prototypes and simulators allow joint research on hardware architectures and software algorithms. However, software simulation is still too slow and impractical to evaluate parallel algorithms with large data-sets. On the contrary, FPGAs systems can integrate multiple cores and I/O interfaces, hence they allow to emulate multi-core systems in full detail, while achieving better performance than simulation [35].

We implemented the FPGA design using the Xilinx ISE Embedded Design Suite, version 13.4. The multi-node prototype is composed of four ML605 boards, each mounting a LX240T Virtex-6 FPGA. The boards communicate through the RocketIO GTX transceivers and are interconnected with coaxial cables, providing full-duplex links. To increase the number of transceivers available, every board mounts an FMC XM104 daughter board. Communication between nodes is done via the Aurora protocol, a lightweight link-layer protocol developed by Xilinx for high-speed serial links [36]. In particular, we use the 8B/10B flit encoding.

Since our prototype is based on FPGA, it is limited by the reduced operating frequency for the processing cores, custom logic and the on- and off-chip interconnects. However, the downscaling of processor operating frequency and channel bandwidth does not invalidate the information on performance scaling that we can extract from the prototype. Moreover, we downscaled the network bandwidth to match the reduced frequency of FPGA, maintaining the relative performance of the components coherent with those of commercial processors, on-chip interconnects and channels. The prototype uses off-the-shelf IP cores for the on-chip interconnection, the processing cores and memory controller, and only adds the three modules described in Section 3.

The many-core node architecture is composed of instances of the Xilinx MicroBlaze core. This is a simple but configurable 32-bit RISC soft-core, which can be synthesized on FPGAs. Nevertheless, the proposed approach is independent from its specific ISA, as long as the selected core is in-order and supports precise interrupt processing. The on-chip interconnect sub-system is the ARM AXI4 bus, a common interconnect solution for the design of systems-on-chip and embedded systems in general. Xilinx implementation of the bus supports up to 16 masters in single-layer configuration, along with split transactions, pipelined requests and burst messages [37]. The platform also includes a DDR3 RAM controller to connect to the node memory, i.e. a 512 MB SODIMM module. Finally, each node includes an UART controller and a hardware debug module (MDM) that can connect to up to eight processors.

We synthesized the architecture with up to 32 cores per node. The clock frequency of the cores and other components in the FPGA is 100 MHz, and the GTX transceiver provides a bandwidth of 625 Mbit/s. Internally, the cores communicate with the memory controller, GNI and GSYNC through a tree-shaped hierarchy of AXI4 buses. The reason for adopting a hierarchy is that the Xilinx implementation of AXI supports only up to 16 masters on a single instance. Therefore, the cores are grouped in small clusters that share a first bus level, which, in turn, is connected to the main one. In addition, the bus can only maintain up to 32 read and 32 write operations concurrently towards each slave, becoming a bottleneck when the number of threads per node is higher than 32.

TABLE 1
FPGA Resources Occupation and Maximum Frequency of the Custom Components

	# of slice registers	# of slice LUTs	$f_{max} [MHz]$
GMAS	1063 (0.4%)	1566 (1.0%)	230.3
GNI	450 (0.1%)	1915 (1.3%)	253.3
GSYNC	4711 (1.6%)	8054 (5.3%)	241.1
MicroBlaze	1510 (0.5%)	1457 (1.0%)	155.7
AXI4 (sum)	29694 (9.9%)	28868 (19.0%)	153.4

Table 1 shows the FPGA resource occupation and maximum operating frequencies of the three custom modules presented in Section 3, the MicroBlaze core, and the AXI4 bus, as configured in the prototype. The percentages refer to an LX240T Virtex-6 device. The MicroBlaze is configured for area optimization to increase the number of cores, so it lacks modules such as the floating point unit and optional instructions, not very useful in memory bound applications. The most critical occupation figures among the custom components are those of the GMAS, because every processing core has an attached GMAS module. Each node, instead, needs only one GNI and one GSYNC module. For the GSYNC, almost all the resources are used to implement the lock table, configured to have 4,096 entries. Finally, the interconnection is also quite significant, consuming one fifth of the available LUTs. In terms of operating frequency, none of the custom components is critical.

6 Performance Model

This section presents a mathematical model that correlates the main features of the proposed architecture, listed in Table 2, with system performance metrics. This model provides insights on the impact and importance of each feature. It allows estimating the components utilization and identify the bottlenecks.

The execution time of data-irregular applications is essentially dictated by the frequent occurrence of long latency memory requests. Therefore, the model focuses on the memory and network subsystems, and on the number of accesses to the global address space.

The modeled system is a cluster composed of N nodes, each hosting C computing cores running at frequency f. Every memory access to the global address space may be directed towards the local memory controller or forwarded to a remote note. The delays for these accesses are labeled respectively D_m for memory, and D_{net} for network transactions. In both cases, we consider the average delays. We define the parameter K_r as the ratio of remote requests over all global memory accesses, and $K_l = 1 - K_r$ the ratio of

TABLE 2
Definitions of Terms Used in the Model

	Hw parameters
N	Number of nodes in the cluster
C	Number cores on a node
f	Clock frequency [Hz]
D_{net}	Latency of a network request (average) [seconds]
D_m	Latency of a memory request (average) [cycles]
S	Size of network request (average) [bytes]
	Sw parameters
K_r	Ratio between remote requests and global requests
K_l	Ratio of global requests managed locally
T	Number of threads per core
I_{sw}	Number of processor cycles between global requests
D_{csw}	Context switch delay [cycles]

local requests. In a perfectly balanced system, the ratios are $K_r = \frac{N-1}{N}$ and $K_l = \frac{1}{N}$.

The software side of the system is composed of the run-time and the application. For the run-time system, we are mainly interested in the time D_{csw} required to perform a context switch, because it imposes a hard limit on the effectiveness of multithreading. The application is characterized by the average number of clock cycles between consecutive global memory requests, I_{sw}, and the number T of threads spawned on each core.

The performance of the system depends on the rate of generated global memory requests. To estimate this value, we start by defining two asymptotic formulas for a single core, distinguishing the case of low processor utilization and processor saturation. Fig. 5 depicts two possible schedules corresponding to these two scenarios.

When a thread (T1) issues a remote request, it remains blocked for a whole round trip over the net (D_{net}). Meantime, the core switches to a different thread, which uses the CPU until it also issues a remote request, after I_{sw} cycles. In a situation with only two threads and low processor utilization, the core becomes idle until the first response arrives. After receiving the response, the core restores the context of T1 in $\frac{1}{2}D_{csw}$ clock cycles. We assume that context save and restore have approximately the same duration, and they constitute the whole context switch delay. On the other hand, local memory accesses simply stalls the core for a time D_m, without switching the context. Accordingly, both threads T1 and T2 are executed every $I_{sw} + K_r(D_{net} + \frac{D_{csw}}{2}) + K_l D_m$ cycles.

The resulting asymptotic equation for the global access rate in low utilization regimen is given by Equation (1), dividing the number of threads per core by the response time

$$Rate^{glob}_{low} = \frac{T}{I_{sw} + K_r(D_{net} + \frac{D_{csw}}{2}) + K_l D_m}. \tag{1}$$

In the case of a saturated processor, whenever a remote request terminates, the core is busy running other threads. When the core issues a global memory access, it has a probability K_l of being stalled for a local memory access, and a probability K_r of executing a context switch caused by a remote request. However, it is never forced

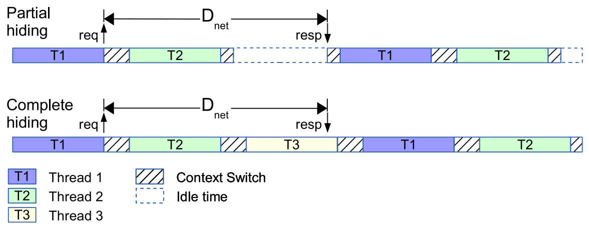


Fig. 5. Latency hiding through multithreading.

to wait for network transactions. Hence, if the number of threads is N, the time required to execute once each thread is $N(I_{sw} + K_r D_{csw} + K_l D_m)$. This is also the demand on the core for executing $N(K_r + K_l) = N$ global accesses. The resulting global access rate is given by Equation (2), and is the maximum rate for a single core

$$Rate^{glob}_{sat} = \frac{1}{I_{sw} + K_r \cdot D_{csw} + K_l D_m}. \tag{2}$$

The number of threads required to completely hide the network latency, and reach this upper-bound, is given in Equation (3). The equation is obtained by equating the response time of a single global request to its demand on the core, so that the core utilization is 100 percent

$$T_{th} = \frac{I_{sw} + K_r(D_{net} + \frac{D_{csw}}{2}) + K_l D_m}{I_{sw} + K_r \cdot D_{csw} + K_l D_m}. \tag{3}$$

In the proposed equations, the network latency is a fixed value, not depending on the congestion of the network. This is acceptable for under-loaded networks, but it is no more true when approaching the saturation point. Anyway, the equations are still useful, if used together with a formula that relates the delay to the traffic injected in a specific network instance. We can then solve the resulting system of equations to estimate the global access rate. As it can be seen from Equations (2) and (3), the maximum rate of requests is independent from the network delay D_{net}, which only increases the number of threads required to produce it. On the other hand, if the network delay increases because of its saturation, it would mean that the bottleneck of the system is the network itself. In such a case, increasing the requests injected by the nodes would be meaningless.

Several system performance metrics depend on the single-core global memory access rate, starting from the network bandwidth used by a system node. We assume that contention on on-chip resources is much lower than contention on the network interface, therefore the throughput of a node is directly proportional to the number of cores C, up to network saturation. A fraction K_r of the global requests by each core is turned into a network request, and each of them involves in average S bytes. Since all the network packets transport similarly small payloads, averaging their sizes is a good approximation. Eq. (4) gives the asymptotic per-node network bandwidth, valid up to saturation of the network interface. As with Eq. (2), this equation can be reversed to compute the maximum rate per-node

$$Band_{node} = C \cdot f \cdot S \cdot K_r \cdot \begin{cases} Rate^{glob}_{low}, T < T_{th} \\ Rate^{glob}_{sat}, T \geq T_{th}. \end{cases} \quad (4)$$

Similarly, it is possible to model the memory bandwidth utilization. The total memory bandwidth of the system is given by the number of nodes multiplied by the bandwidth of a single controller. In the best-case scenario, the requests are evenly distributed among the nodes. In this case, the demand on each memory controller is exactly equal to the request rate generated by the cores in one node. Conversely, on an unbalanced system, a single memory controller can receive more requests than the others, and can become the bottleneck. Anyway, in a large system, the majority of global memory requests that arrive to a memory controller pass through the incoming network interface. Therefore, it is simple to determine whether the memory controllers are capable of sustaining the traffic that passes over the network, or whether they are the system bottleneck.

The core utilization is another interesting metric that is linked to the global access rate. The demand on a core by the application for each global access is the sum of the three components: the average computation time between global requests, the context switch caused by remote requests, and the stalls due to accesses to the local portion of the global address space. This is also the sum that appears at the denominator of Equation (2)

$$U_{cpu} = U_{cpu,user} + U_{cpu,sys} + U_{cpu,stall} \quad (5a)$$

$$= Rate_{glob} \cdot (I_{sw} + K_r D_{csw} + K_l D_m). \quad (5b)$$

7 Experimental Validation

This section discusses the experiments that we run on the architecture prototype described in Section 5. We characterize the system by providing the latencies of the main operations. Then we validate the analytic model with data from the prototype. Using both the prototype and the model, we evaluate the system performance running memory intensive and synchronization intensive benchmarks. Finally we explore the impact of the multithreading overhead on the system throughput.

System characterization. The first step in evaluating the system prototype is to measure the performance of its individual components. The latencies of the main operations performed in the system are reported in Table 3. We measured all the latencies with a single core active, thus without contention on the on-chip bus hierarchy. The overhead for traversing the GMAS is only one cycle for local accesses and three cycles for remote accesses.

From these profiled data, and the equations presented in Section 6, we can obtain an upper bound to the rate of global memory accesses generated by the system. To maximize the rate, we minimize the interval between global accesses, setting $I_{sw} = 0$ in Equation (2), i.e., we assume that every instruction in the application is a global memory access, without interposed computation. In addition, the same assumption applied to Equation (3) provides the number of threads required to achieve the maximum rate and saturate the cores. Using the latencies measured on the prototype, we obtain that the maximum rate per core is 600 K accesses per second, and that three threads are enough to saturate it. Therefore, the whole system composed of four nodes with 32 cores each has a theoretical maximum of 76.8 million references/second.

TABLE 3
Latencies of the Main Operations

Action	Latency [clock ticks]
crossing GMAS	1-3
access to GNI/GSYNC	20/18
local/remote mem. access	28/505
context switch latency	213

Model validation. To validate the model accuracy, we compare the performance estimated by it with the performance measured on the prototype. For the comparison, we use a pointer chasing benchmark, a basic irregular kernel that stresses the global system memory with random walks.

The benchmark initially creates a linked list in the global memory space, with $1.048.576$ nodes allocated contiguously but linked in a random order. Then, it spawns one or more threads per core, and each thread iterates over the whole list. This iteration results in a highly irregular access pattern to the memory, with continuous jumps randomly distributed in both distance and direction. Between consecutive requests, our implementation performs a small number of accesses to the stack, adding a delay I_{sw} equal to 67 clock cycles. This benchmark has no synchronization and is completely homogeneous, both in terms of processor utilization and memory accesses. Hence, it is the most similar to the perfectly balanced situation hypothesized in the model.

Fig. 6 shows the remote request rate of a single node, using different numbers of cores and threads. The darker dots of the chart shows the prototype performance. They overlap the lines that represent the performance as obtained from the model. The comparison demonstrates that the model is highly accurate, with an average error of 2.3 percent and a maximum error of 6.1 percent.

Pointer chasing. The pointer chasing kernel is also ideal to evaluate the maximum performance scaling, given the absence of synchronization conflicts.

Fig. 6 shows that the throughput increases linearly with the number of threads, until it saturates when using three threads per core, as predicted by the model. Adding further threads neither improves nor reduces the performance. This happens because the thread stacks reside in the scratchpad memories, instead of a cache hierarchy. Therefore, the only contended resource is the processor pipeline and, once it is saturated, additional threads do not decrease the performance. This puts a hard constraint to the number of threads executable on each core, but does not affect the prototype evaluation, because the low diameter of the network requires only three threads for hiding the latency.

We measured a perfectly linear scaling also with respect to the number of cores. In fact, the 16 cores configuration has a speedup of 3.96 over the four core one. This confirms

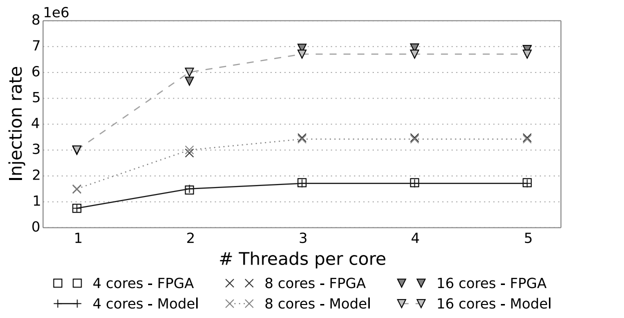


Fig. 6. Injection rate per node vs. number of threads per core, for different number of cores per node.

the assumption that the on-chip bus is not among the possible bottlenecks for the prototype.

The equations presented in Section 6 provide an insight on the utilization of the individual components. Their application to the pointer chasing benchmark identifies the bottlenecks when running a perfectly parallel algorithm, with little computation and no synchronization. Fig. 7 shows the utilization of four components: the core, the outgoing interface of the GNI, the network links and the DRAM. The limit of 32 pending operations, enforced by the Xilinx implementation of the AXI4 bus, sets the upper bound for the GNI interface. Because of the duration of remote transactions, this limit is reached before actually saturating the GNI internal logic.

To provide more details, we divide utilization of the cores in three parts: the time used by the application, the time required by the system to switch contexts, and the stalls due to accesses to the local memory. The two rows in the picture respectively represent systems with eight and 16 cores. The number of threads increases from one to three from the left to the right. With eight cores per node, the performance is limited by the saturation of the cores, while memory utilization caps at 16 percent. The average number of network transactions reaches 16.3. On the 16 cores configuration, instead, the number of pending operations approaches the limit with just two threads per core. With three threads both the cores and the GNI interface saturate, resulting in a better overall system utilization. Regarding cores utilization, when running three threads only 28.7 percent of the time is spent on the kernel execution, while 68 percent goes in context switches. Anyway, even if multithreading has a high overhead, it allows exploiting part of the time that would otherwise be wasted, increasing the application quota from 12.6 to 28.7 percent.

Breadth First Search (BFS). The Breadth First Search is a typical irregular algorithm used to evaluate HPC systems [38]. This kernel is widely used because many problems in graph theory can be solved by algorithms based on BFS, such as finding the shortest path between two vertices, testing the graph for bipartiteness or computing centrality measures. We ported a version originally designed for the Cray XMT[39], adapting it to the synchronization and threading interfaces of the prototype. The graph explored by our benchmark has 100.000 nodes, with an average of 39.9 edges per node, and diameter 6.

The BFS is not only memory intensive, but also synchronization intensive. In fact, it has two critical sections in a small loop body. The first one prevents different

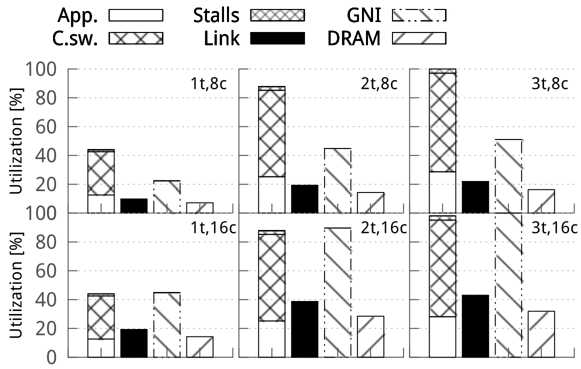


Fig. 7. Utilization of system components by the pointer chasing, on different configurations of threads and cores. CPU utilization is split into application time, context switch and stalls.

threads to concurrently test and set the visited flag of the same vertex. The second is required to insert the newly visited vertices in the shared work-list. Therefore, this benchmark significantly stresses the components providing synchronization support.

First of all, we try to estimate the throughput using a model similar to the one described in Section 6. We modified the model to computes the rate in terms of edge visits, instead of memory requests. Hence, in this version the parameter I_{sw} identifies the average number of instructions executed per edge, and the number of memory requests and context switches are scaled accordingly. The model takes into account synchronization only partially: it counts the lock and unlock operations as global requests, but ignores the run-time contention on the locks. Fig. 8a shows the result, which has a behavior very similar to pointer chasing, with saturation at three threads per core, and a peak of 7 M edges/second with 24 cores.

Fig. 8b, instead, shows the performance obtained running the benchmark on the prototype. First of all, the execution speed on the prototype is greatly reduced with respect to the model, in particular on configurations with 16 and 24 cores. The system with eight cores and one thread per core is the most similar to the model, with a performance 1.4

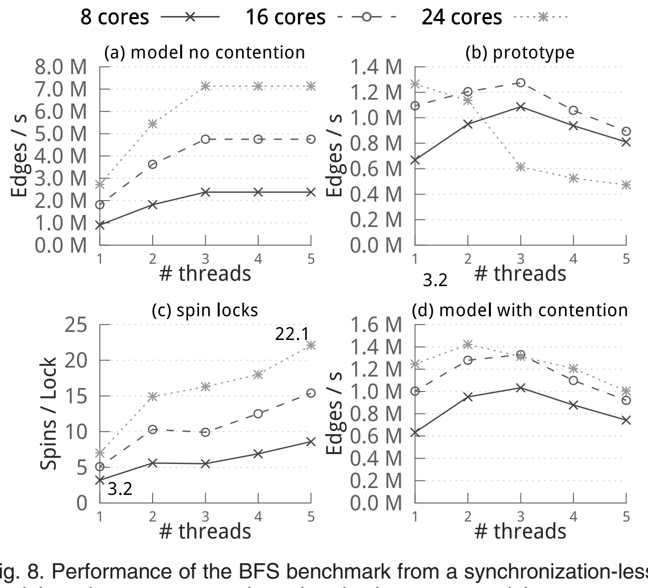


Fig. 8. Performance of the BFS benchmark from a synchronization-less model, on the prototype, and synchronization aware model.

slower than estimated. On the other hand, the configuration with 32 cores and five threads each is 15 times slower. Second, adding threads to an already saturated system has a huge negative effect on the prototype performance, while the modeled performance remains constant. These differences are due to the overhead caused by the contention on the shared data.

Fig. 8c shows in details the overhead caused by the synchronization, presenting the average number of spins required for obtaining a lock, as measured on the prototype. Even in the small configurations, each lock needs to spin between three and seven times before taking ownership, and the number increases with the degree of parallelism. The main causes are the accesses to the shared work-list that stores references to the nodes to be visited by the algorithm. Each insertion in the list is a critical section that takes approximately 2,000 clock cycles ($20\,\mu s$) because of the latency of remote data and synchronization operations. The probability of contention during this interval is significant, because there are many threads trying to enter it, with high frequency. Also, this overhead grows with the number of threads because of two trends. When the system is underused, it behaves accordingly to a modified Amdahl's law [40] that accounts for serialized critical sections. Because of the serialization, the waiting time grows linearly with the number of threads, limiting the speedup. In the prototype, this is demonstrated by the increasing number of spins on the locks from one to three threads. The saturation of the cores adds a second effect. When all the core time is used, the ratio available to each thread shrinks with the addition of others. Consequently, the time required to complete a critical section increases. This second effect is evident in Figs. 8b and 8c, when more than three threads per core are used.

We extended the model to take into account the number of lock attempts due to contention, obtaining a better approximation shown in Fig. 8d. In this version, the number of global transactions per edge is increased with the number of cores and threads used. Hence, the model does not distinguish single critical sections, but captures the resulting average overhead. Even so, the accuracy is greatly improved, and captures the same trends observed on the prototype. This hints that the main limiting factor in the proposed architecture is the increased resource utilization due to synchronization conflicts. This is also demonstrated by the utilization of the components, shown in Table 4. When running the BFS, the core utilization is much higher with respect to the other components than when running the pointer chasing. However, only 4 percent of it is due to the application. The remaining is used by the context switches that are forced by every *try_lock*, even when the request fails. In column *SyncBW* we show the percentage of network utilization due to synchronization requests. In the best case, it equals the bandwidth used by memory accesses, but increases with the number of threads. This suggests that the most effective improvement to the architecture would be a hardware implementation of the spin lock.

SSCA#2. Another benchmark representative of graph theory computational kernels is SSCA#2 [40]. It generates scale-free graphs, using a generator algorithm based on R-MAT [42]. The benchmark is composed of four kernels, but

TABLE 4
BFS System Utilization

Cores, Threads	CPU		Network			Memory
	Total	App.	Link	GNI	SyncBW	DDR
8, 1	43.5	2.6	1.0	2.4	51.1	7.2
8, 2	93.2	3.9	1.5	3.5	62.0	13.6
8, 3	100.0	4.2	1.7	3.9	61.8	14.6
8, 4	100.0	3.6	1.4	3.3	66.3	13.9
16, 1	46.1	2.0	1.6	3.7	60.3	13.7
16, 2	99.3	2.6	2.0	4.8	73.7	25.6
16, 3	100.0	2.7	2.1	5.0	73.1	26.0
16, 4	100.0	2.2	1.8	4.1	77.0	24.9

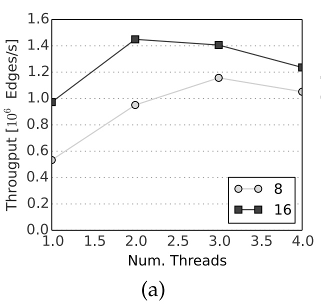
utilization [%]

we will focus on the betweenness centrality computation, which accounts for 99 percent of the execution time. The kernel involves multiple breadth first visits of the graph, and assigns a weight to each node according to the number of shortest paths it belongs to. For our experiments, we ported the reference OpenMP implementation to the prototype APIs, and configured it to approximate the betweenness centrality by running 256 BFS from random nodes. For the main BFS loop we used a dynamic scheduling of loop iterations to threads, similar to the OpenMP *guided* schedule, and a fixed round robin for the other loops.

Fig. 9 shows kernel throughput and core utilization, averaged over the entire execution. The performance scales very well in the unsaturated systems (1-2 threads), and shows the same degradation observed in the BFS when the cores are fully used. The saturation occurs at 88 percent average CPU utilization, because the algorithm has a small serial section, and uses multiple barriers for synchronization, with a higher frequency with respect to the previous BFS benchmark. Even if the barriers prevents full utilization of the processor, the dynamic scheduling distributes evenly the workload across the entire set of cores, as shown by the min-max bars in the plot. On the memory side, the number of remote memory requests generated by the various cores presents a relative standard deviation lower than 7 percent.

Multithreading. By running the benchmarks on the prototype, we saw that the cores spend most of the time switching contexts. This limits the number of runnable threads before saturation and the maximum throughput. Consequently, we evaluated how the performance would change with different multithreading approaches.

First of all, we considered how much the performance would degrade when using a software version of the same scheduler included in the GMAS. The scheduler stores the ready/waiting flag of the threads in a single word, with the most significant bit corresponding to the thread with identifier zero. The MicroBlaze ABI includes an instruction (*clz*) that counts the number of leading zeros in a 32-bit word, which can be used to retrieve the identifier of the first available thread, using a fixed priority. We added a second bit-mask to cancel the flags of the most recently scheduled threads, and rotate the highest priority position in a round-robin fashion. The whole scheduling sequence reads the pending status word from



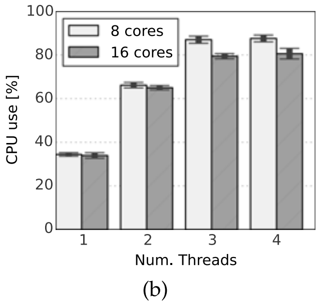


Fig. 9. Betweenness centrality performance: CPU utilization (a) and throughput (b).

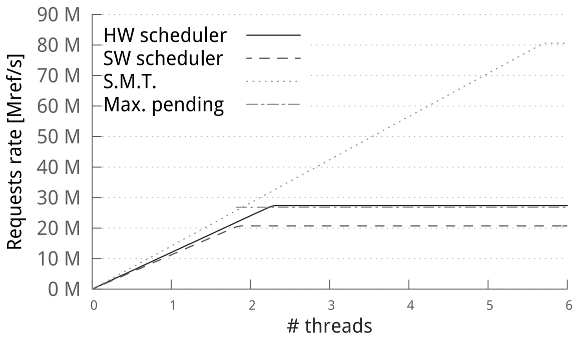


Fig. 10. Global request rate generated by the system with different context switch implementations: hw scheduler, sw routine, simultaneous multithreading.

the local scratchpad, searches for a valid thread, updates the status, and eventually rotates the mask. This code requires between 45 and 50 clock cycles for completion, according to the thread statuses. Therefore, the software scheduler (SW sched) would increase the context switch time from 213 cycles to approximately 260 cycles.

However, the GMAS not only selects the next thread, it also receives the responses from the network, and manages the thread status flags without intervention from the core. A pure software support for multithreading requires the scheduler to interrogate the network interface, to identify the completed requests and to map them to the respective threads, thus increasing the overhead. Therefore, we considered an hypothetical software scheduler that requires 100 clock cycles to interact with the GNI, select the next thread and switch context. Fig. 10 shows the estimated performance, and compares it to the prototype performance when running pointer chasing on 16 cores. The maximum request rate of the cores drops by 24 percent, from 27.4 to 20.7 Mref/s. At this rate, the number of concurrent network transactions is no more maximized, reducing network utilization and system efficiency.

The plot in Fig. 10 also shows the estimated performance of a system that executes the context switch in zero cycles. This system would require to modify the cores to add multiple register files, as in the Cray XMT. In this scenario, pointer chasing would reach 80.6 Mref/s, three times more than the rate obtained with the prototype. However, this rate requires the ability to keep six thread contexts concurrently active in hardware, while the proposed architecture only stores the thread statuses. Hence, the considerable slow down of our prototype with respect to hardware multithreading comes with large savings in term of hardware complexity and cost, and more flexibility in increasing the number of threads, making it a valid trade-off.

8 Conclusions and Future Work

This paper presented a distributed multi-core system architecture, specifically designed to execute applications with irregular memory patterns. We presented the design of the system, starting from highlighting the issues in the optimization of irregular applications and proposing architectural solutions for each one of them. The architecture is based on off-the-shelf processing cores, and includes custom components that provide support for a scrambled global address space, fine-grained synchronization, and a hardware scheduler for fast multithreading. To evaluate its feasibility and effectiveness, we prototyped the architecture on a FPGA. We also presented an analytic model specifically focused on the features provided by the architecture, to help evaluating its effectiveness in hiding network latency and improving resource utilization. The experiments show that the performance of irregular, memory bound applications scales with the number of cores, and that fast multithreading allows hiding memory and network latency. The proposed model enables computing the optimal number of threads to achieve full latency hiding. We also proved that support for both efficient synchronization and multithreading are basilar to make the proposed architecture effective. Further improvements to these two aspects will be the key for future research on hybrid hardware-software support for irregular applications. We plan to extend the architecture to integrate aggregation of network requests in order to increase the payload vs header ratio, improve network efficiency and therefore achieve higher throughput. We also plan to apply work-stealing techniques at the node and system level, in order to improve the load balancing on the cores and to provide better latency hiding.

References

[1] A. M. M. González, B. Dalsgaard, and J. M. Olesen, "Centrality measures and the importance of generalist species in pollination networks," *Ecological Complexity*, vol. 7, no. 1, pp. 36–43, Mar. 2010.

[2] E. Estrada, "Characterization of topological keystone species: Local, global and "meso-scale" centralities in food webs," *Ecological Complexity*, vol. 4, no. 1–2, pp. 48–57, Mar. 2007.

[3] J. Wang, H. Mo, F. Wang, and F. Jin, "Exploring the network structure and nodal centrality of China's air transport network: A complex network approach," *J. Transport Geography*, vol. 19, no. 4, pp. 712–721, Jul. 2011.

[4] S. A. Myers, A. Sharma, P. Gupta, and J. Lin, "Information network or social network?: The structure of the twitter follow graph," in *Proc. Companion Publication 23rd Int. World Wide Web Conf.*, 2014, pp. 493–498.

[5] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2009, pp. 65–76.

[6] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2010, pp. 1–11.

[7] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on BlueGene/L," in *Proc. ACM/IEEE Conf. Supercomput.*, 2005, p. 25.

[8] J. Feo, D. Harper, S. Kahan, and P. Konecny, "ELDORADO," in *Proc. Int. Conf. Comput. Frontiers*, 2005, pp. 28–34.

[9] J. Matienzo and N. Jerger, “Performance analysis of broadcasting algorithms on the intel single-chip cloud computer,” in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2013 pp.163–172.
[10] B. de Dinechin, D. van Amstel, M. Poulhies, and G. Lager, “Time-critical computing on a single-chip massively parallel processor,” in *Proc. Conf. Des., Autom. Test Eur. Conf. Exhib.*, 2014, pp. 1–6.
[11] I. Choi, M. Zhao, X. Yang, and D. Yeung, “Experience with improving distributed shared cache performance on tilera’s tile processor,” *Comput. Arch. Lett.*, vol. 10, no. 2, pp. 45–48, 2011.
[12] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-way multithreaded sparc processor,” *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Mar./Apr. 2005.
[13] K. Wheeler, R. Murphy, and D. Thain, “Qthreads: An API for programming with millions of lightweight threads,” in *Proc. IEEE 22nd Int. Symp. Parallel Distrib. Process.*, Apr. 2008, pp. 1–8.
[14] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz, “APRIL: A processor architecture for multiprocessing,” in *Proc. 17th Annu. Int. Symp. Comput. Arch.*, May. 1990, pp. 104–114.
[15] D. Sanchez, R. M. Yoo, and C. Kozyrakis, “Flexible architectural support for fine-grain scheduling,” in *Proc. 15th Edition ASPLOS Arch. Support Program. Lang. Oper. Syst.*, 2010, pp. 311–322.
[16] Y. Wang, W. Ji, Q. Zuo, and F. Shi, “A hierarchical work-stealing framework for multi-core clusters,” in *Proc. 13th Int. Conf. Parallel Distrib. Comput., Appl. Technol.*, 2012, pp. 350–355.
[17] 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.
[18] “UPC language specifications, v1.2,” UPC Consortium, Lawrence Berkeley Nat. Lab, Berkeley, CA, USA, Tech. Rep. LBNL-59208, 2005.
[19] 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 Experience*, vol. 10, no. 11-13, pp. 825–836, 1998.
[20] Co-array Fortran [Online]. Available http://caf.rice.edu, Aug. 2015.
[21] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, “X10 language specification version 2.2,” Jul. 2012.
[22] D. Bonachea, “*GASNet Specification, v1.1*,” CS Division, EECS Department, Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/CSD-02-1207, Oct. 2002.
[23] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apà, “Advances, applications and performance of the global arrays shared memory programming toolkit,” *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 203–231, 2006.
[24] O. Villa, D. Scarpazza, F. Petrini, and J. Peinador, “Challenges in mapping graph exploration algorithms on advanced multi-core processors,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Mar. 2007, pp. 1–10.
[25] A. Vishnu, M. ten Bruggencate, and R. Olson “Evaluating the potential of cray gemini interconnect for PGAS communication runtime systems,” *IEEE 19th Annual Symp. High Performance Interconnects (HOTI)*, pp. 70–77, Aug. 2011.
[26] 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 *Proc. 3rd USENIX Conf. Hot Topic Parallelism*, 2011, p.10.
[27] A. Morari, A. Tumeo, D. Chavarría-Miranda, O. Villa, and M. Valero, “Scaling Irregular Applications through Data Aggregation and Software Multithreading,” *IPDPS28: IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 1126–1135.
[28] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, K. Aapo, and J. M. Hellerstein, “Distributed graphLab: A framework for machine learning and data mining in the cloud,” in *Proc. VLDB Endow.*, vol. 5, no. 8, Apr. 2012, pp. 716–727.
[29] 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 *Proc. ACM Int. Conf. Manage. Data*, 2010, pp. 135–146.
[30] H. Froning, and H. Litz, “Efficient hardware support for the partitioned global address space,” in *Proc. 10th Workshop Commun. Arch. Clusters, with 24th Int. Parallel Distrib. Process. Symp.*, 2010, pp. 1–6.
[31] A. Agarwal, “Performance tradeoffs in multithreaded processors,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 5, pp. 525–539, Sep. 1992.
[32] P. Dubey, A. Krishna, and M. Squillante, “Analytic performance modeling for a spectrum of multithreaded processor architectures,” in *Proc. 3rd Int. Workshop Model. Anal., Simul. Comput. Telecommun. Syst.*, Jan. 1995, pp. 110–122.
[33] S. Weiss, “An aperiodic storage scheme to reduce memory conflicts in vector processors,” in *Proc. 16th Annu. Int. Symp. Comput. Arch.*, 1989, pp. 380–386.
[34] O. Villa, A. Tumeo, S. Secchi, and J. Manzano, “Fast and accurate simulation of the cray XMT multithreaded supercomputer,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2266–2279, Dec. 2012.
[35] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun, “A practical FPGA-based framework for novel CMP research,” in *Proc. ACM/SIGDA 15th Int. Symp. Field Programm. Gate Arrays*, 2007, pp. 116–125.
[36] Xilinx Aurora [Online]. Available: www.xilinx.com/products/intellectual-property/aurora8b10b.htm, Aug. 2015.
[37] Xilinx AMBA AXI4 Interface Protocol [Online]. Available: www.xilinx.com/ipcenter/axi4.htm, Aug. 2015.
[38] (2014). Graph 500 [Online]. Available: http://www.graph500.org/specifications#sec-6
[39] D. A. Bader, and K. Madduri, “Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray MTA-2,” in *Proc. Int. Conf. Parallel Process.*, 2006, pp. 523–530.
[40] S. Eyerman and L. Eeckhout, “Modeling critical sections in Amdahl’s law and its implications for multicore design,” *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 362–370, 2010.
[41] (2014). Ssca#2 v2.0 specification. [Online]. Available: http://www.graphanalysis.org/benchmark/
[42] D. Chakrabarti, Y. Zhan, and C. Faloutsos, “R-MAT: A recursive model for graph mining,” in *Proc. 4th SIAM Int. Conf. Data Mining*, 2004, pp. 442–446.

Marco Ceriani received the MS degree in 2009 from Politecnico di Milano, Italy, where he is working toward the PhD degree. During 2012, he was a post master research associate at Pacific Northwest National Laboratory. His research interests include modeling and simulation of multiprocessor computer architectures and FPGA prototyping.

Simone Secchi received the PhD degree in electronic and computer engineering from the University of Cagliari in 2011. He is currently a research associate at the University of Cagliari, Italy. Previously, he was a postdoctoral research associate at Pacific Northwest National Laboratory. His research interests include modeling and simulation of high-performance computing architectures, FPGA-based emulation of multiprocessor systems, and advanced network-on-chip architectures.

Oreste Villa received the PhD degree in computer engineering from Politecnico di Milano. He is currently a senior research scientist at NVIDIA Research in the Architecture Research Group. Previously, he was a research scientist at Pacific Northwest National Laboratory. His research interests include computer architectures and simulation, accelerators for scientific computing, GPGPU, and irregular applications.

Antonino Tumeo received the MS degree in informatic engineering, in 2005, and the PhD degree in computer engineering, in 2009, from Politecnico di Milano, Italy. Since February 2011, he has been a research scientist at the PNNL's High Peformance Computing group. He joined PNNL in 2009 as a post doctoral research associate. Previously, he was a post doctoral researcher at Politecnico di Milano. His research interests include modeling and simulation of high performance and embedded architectures, hardware-software codesign, FPGA prototyping and GPGPU computing. He is a member of the IEEE and ACM.

Gianluca Palermo received the MS and PhD degrees from Politecnico di Milano, Milan, Italy, in 2002 and 2006, respectively. He is currently an assistant professor at the Department of Electronics, Information and Bioengeneering, Politecnico di Milano. Previously, he was a consultant engineer with the Low Power Design Group of AST-STMicroelectronics and a research assistant with the Advanced Learning and Research Institute (ALaRI), University of Lugano. His current research interests include design methodologies and architectures for embedded computing systems.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.