A NoC-based hybrid message-passing/shared-memory approach to CMP design

A NoC-Based Hybrid Message-Passing/Shared-Memory Approach to CMP design

Mario R. Casu[a], Massimo Ruo Roch[a], Sergio V. Tota[b], Maurizio Zamboni[a]

[a] *Politecnico di Torino, Dipartimento di Elettronica, C.so Duca degli Abruzzi, 24, I-10129, Torino, Italy*
[b] *Imagination Technologies, Home Park Estate, Kings Langley, Hertfordshire, WD4 8LZ, United Kingdom*

Abstract

Future chip-multiprocessors (CMP) will integrate many cores interconnected with a high-bandwidth and low-latency scalable network-on-chip (NoC). However, the potential that this approach offers at the transport level needs to be paired with an analogous paradigm shift at the higher levels. In particular, the standard shared-memory programming model fails to address the requirements of scalability of the many-core era. Fast data exchange among the cores and low-latency synchronization are desirable but hard to achieve in practice due to the memory hierarchy. The message-passing paradigm permits instead direct data communication and synchronization between the cores. The shared-memory could still be used for the instruction fetch. Hence, we propose a hybrid approach that combines shared-memory and message passing in a single general-purpose CMP architecture that allows efficient execution of applications developed with both parallel programming approaches. Cores fetch instructions from a hierarchical memory and exchange their data through the same memory, for compatibility with existing software, or directly through the fast NoC. We developed a fast SystemC based cycle-accurate simulator for design space explorations that we used to evaluate the performance with real benchmarks. The various components have been RTL coded and mapped to a CMOS 45 nm technology to build a silicon area model that we used to select the best architectural configurations.

Keywords: Chip-multiprocessors (CMP), Networks-on-Chip (NOC)

1. Introduction

Moore's empirical law has been proved valid even in the nanometric lithographic regime and we will likely continue to behold an exponential increase in silicon transistor counts in the years to come. As a corollary, we will see also an exponential increase in the "core" counts in chip-multiprocessors (CMP) for general purpose computing and in special purpose dedicated on-chip architectures like graphics processing units. Even the high-end sector of embedded computing is moving toward the multi-core realm [1]. Commercial CPU chips are now sold with up to eight cores but sixteen cores are on their way [2]. It is likely that in the shift from *multi* to *many* core we will see less complex cores then they used to be, because many smaller in-order cores seem to produce a better aggregate performance for a given silicon area than less out-of-order larger cores [3]. The communication infrastructure is one of the keys to allow cores pool to deliver its potential. It is now well-established that packet-switched networks on-chip (NoC's) will be the communication backbone of such future chips [4][5].

The move to the NoC paradigm is a response to a scalability problem. Having a scalable network is not sufficient though if other hindrances impede the potential of such a parallel system to fully develop. One of the obstacles is the current programming model, based on the classic shared-memory paradigm. The memory hierarchy becomes the true performance wall for shared-memory based systems [6]. Cluster-based systems historically faced the problem of sharing data by exchanging them through explicit messages. The increasing difficulties in the access to shared memory resources justify the adoption of a similar approach also in the on-chip environment. The so-called *Message Passing* paradigm offers then an alternative for both data exchange and synchronization among different cores that may benefit from the presence of a low-latency and high-bandwidth on-chip network. Nonetheless, there are a number of good reasons not to abandon the old way. First of all legacy issues suggest to keep compatibility with parallel programs written for a shared-memory environment. Second, not all the parallelized applications fully benefit from a message-passing approach. Third, the shared-memory paradigm is "easier" for developers, not that message-passing is difficult but requires an initial learning cost. It must be said that shared-memory is easy for parallel programming as long as there is a memory coherence mechanism transparent for the developer. Unfortunately, hardware cache coherency methods do not scale well and it is likely that they need to be assisted more and more by software in future parallel systems, hence reducing the appeal of a shared-memory programming model.

For all these reasons, we propose a NoC-based *hybrid* shared-memory/message-passing parallel CMP architecture that has the following key characteristics:

1) Generality. It is not tailored to a specific class of applications, but must be able to support both message-passing and shared-memory programming models, without any need to rewrite existing code.

2) Scalability. Hardware characteristics of its constituent elements do not depend on overall system architecture and size (external memories topology, number of processing

nodes), nonetheless their key parameters must be tunable (e.g. L1 cache size and microprocessor features).

This paper illustrates the architecture, gives details about implementation and discusses experimental results obtained executing real benchmarks. It is not the aim of this work a performance comparison with the "best" existing CMP architecture designed for message-passing only, or rather with the optimal CMP configuration for shared-memory. Likely, that kind of comparison will be at a disadvantage, performance-wise, for our work. We believe instead that the support for both programming paradigms is necessary for the non-specialized, cost-effective CMP's of the manycore era.

The processing elements in our CMP fetch their own instructions and load/store their private data from/to a more or less standard hierarchical memory. As for shared data, processing cores may still use a hardware/software assisted shared-memory approach, or a faster point-to-point message-passing mechanism that exploits the features of an on-chip packet-switched network. For this case we developed simple and easy-to-use API primitives which rely on an underlying hardware interfacing with the NoC at full processor speed.

The architectural parameters need to be adjusted around a single or a set of applications. This customization requires a simulation environment fast enough to allow complete design space exploration in an affordable time but that does not sacrifice accuracy for speed. We thus developed a cycle-accurate SystemC based tool that made it possible to simulate more than two hundred different instances of a 16 cores architecture running a real software on it in few days. We used the results to prune inefficient instances dominated by others with same or more performance at equal or lower cost.

The idea of a multiprocessor architecture with hardware support for shared memories and explicit messaging is not totally new and traces its roots back to the nineties. Notable examples are Stanford FLASH, MIT Alewife and ASCOMA projects [7][8][9][10]. In those early cases that aimed to implement board-level multiprocessor systems, a conventional processor with cache and local memories was connected to special purpose complex devices dedicated to handle I/O and interconnection to other nodes in the network. Those architectures were mainly based on the concept of Distributed Shared Memory (DSM), in which address translation is performed locally at each node level, and no main memory is present. Compared to one node of those machines, our on-chip processor core and the switch, the essential elements of our NoC, are incomparably simpler. The on-chip environment poses totally different design constraints than board-level multiprocessor design and we think that to maximize the performance it is better to keep the "tile" of the modular design as simple as needed and to fill the available area with as many tiles as possible.

In the NoC scientific community, some early papers envisioned message passing as the communication mechanism between computational resources [11], others proposed that the network interface had to offer a shared-memory abstraction [12] or that a new parallel programming paradigm was needed [13]. Researchers involved in application specific multiprocessor systems-on-chip, not general purpose homogeneous CMP, recognized the need to integrate both programming models for reasons of flexibility [14]. More recently, after the industry shift toward CMP architectures, it became clear that NoC's are the only answer for scalable intra-core connectivity. Support to shared-memory by the NoC is then necessary, especially for legacy reasons, but the need to include also a support for message-passing in CMP emerged [15]. As far as we know this is the first paper to address implementation aspects of a hybrid shared-memory/message-passing CMP architecture and to propose a detailed analysis of the performance/area tradeoff obtained varying architectural parameters like cache size and number of active cores when executing real parallel code.

Organization of the paper is as follows. The architecture with its components is presented in section 2 while section 3 describes the SystemC based simulator. Section 4 discusses physical implementation issues. Simulation results are reported and analyzed in section 5. Finally, section 6 comments on the obtained results and concludes this work.

2. System Architecture

Figure 1 represents a generic instance of our NoC-based multiprocessor architecture. The network consists of 5-port Switches (SW) connected in a two-dimensional topology. Four of the switch ports connect to four other switches. The fifth one connects to a local Processing Element (PE) or to a Multiprocessor Memory Management Unit (MPMMU) which in turn communicates with an off-chip double-data-rate SDRAM (DDR in figure). Every link in figure is actually made of a pair of busses traveling in opposite directions. (The external DRAM data bus, however, is bidirectional, as usual for standard DDR memories.) Every switch hence has five input and five output ports (north, south, west, east and PE).

The following subsections describe in details components used to build the system.

2.1. The On-Chip Network

Bidimensional mesh and torus are more amenable than other topologies to large-scale silicon implementations primarily because of easiness of layout [11][16], but also for latency and timing reasons [17]. Recent investigations evaluated multidimensional variants of these simple topologies and concentrated meshes which seem advantageous, in some cases, compared to standard topologies [18][19]. The fact that a potential advantage depends on the traffic scenario and possible layout complications drove our decision toward a simpler topology. We were then left with the alternative between 2D mesh or torus. We chose the latter for its larger bisection bandwidth, its better symmetry, and its smaller latency [16]. Although the wrap-around connections in Figure 1 seem longer than the others, an optimal layout arrangement called "folded torus" guarantees that switches connect only through local links [20]. Torus topologies are less power efficient than meshes [16][21] and its local

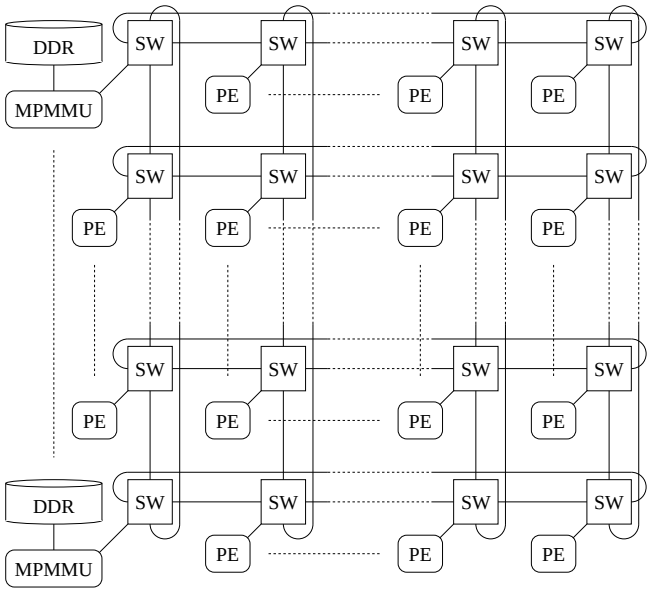

Figure 1: The system architecture.

links are longer [17]. However, [21] shows that when adaptive routing is used – and that is also our choice as will be later explained – the power gap diminishes. In a previous work we evaluated that the NoC consumes significantly less than the processing elements connected to it [22], and so benefits outweigh the relatively small power disadvantage. As for the longer links, they may become an energy issue, because buffers inserted to reduce delay are power expensive. But our experience with synthesizable cores is that cores limit the clock frequency in a fully synchronous design, while the NoC is not in the critical path. Timing margins can then be used to relax NoC links design. This conclusion may be upset if custom, optimized processors were used. But, for the case at stake, the added power for longer torus links was marginal at chip scale.

After topology selection, the second relevant choice was the routing strategy implemented within switches. High-performance switches consume a large share of silicon real estate as well as power budget [4]. The main reason is their large use of memory buffers. We believe that the largest part of chip area and power budget should be dedicated to computational elements and their local caches. Moreover, larger caches reduce traffic due to less cache misses and alleviate performance requirement of the network. We adopted an adaptive routing strategy called "deflection-routing" that permits to keep switch size small at a moderate and tolerable performance cost [23][24][25]. It consists in choosing the best route for every incoming packets and in case two or more of them contend for the same output port, a winner is chosen, according to a priority rule. The others are "deflected" elsewhere. No packets are ever kept inside the switch in any buffer, thus the alternative name of hot-potato routing. The fifth input from the PE has the smallest priority and enters the NoC only if at least one of the switch output ports is not already taken. It thence follows that PE needs to be stopped whenever it did not manage to send a packet. Conversely, if the switch port toward a PE is not open because the processing element is busy, packets addressed to it will get deflected. It must be noted that only PE's can be stopped and no other form of backpressure is present in the switch-to-switch connections, thanks to the hot-potato rule. It follows that there's always a way out from a switch for an incoming packet and so deflection-routing is inherently deadlock-free (see [26] for a thorough analysis).

In our implementation, the basic routing unit is the *flit* (flow control unit) which coincides with the *phit* (physical unit). Every injected flit is treated as a complete packet by the switches and is given a relative address in (X,Y) form that is decremented as the flit gets close to its destination or incremented if it gets deflected astray. Such address and a validity bit form the flit header, followed by the NoC-level payload which in turn encapsulates other higher-level packets described later. The price for using hot-potato is that deflected flits of a larger message may experience different latencies and arrive out of order at destination. Reordering mechanisms must be then implemented in hardware or in software and assisted by sequence numbers printed in the packets' header. In theory, packets could be deflected forever giving rise to a livelock issue. However the probability that a packet is not delivered vanishes rapidly as a function of packet age. In one of our previous experiences with deflection-routing and a benchmark suite of parallel applications, we observed sporadic cases of single flits delivered with high latency (larger than average) that did not significantly hamper execution times [24]. We anticipate that in this work we did not observe any significant overhead, too. Should livelock become an issue, a flit-age prioritization mechanism must be implemented at router level that guarantees that the oldest flit is not deflected and always arrives at destination [25]. Reserving bits for packet age implies some performance reduction (less wires used for payload) or some higher area and power cost (more wires for a given payload).

2.2. Processing Element and NoC Interface

Building a full-blown CMP architecture from scratch and experimenting with it using realistic programs as benchmarks is an almost prohibitive task for a university research group. Therefore instead of building our own processing elements we resorted to a third party core. We selected the highly configurable Tensilica Xtensa-LX processor which comes with a stable and powerful tool-chain for compilation, profiling, hardware/software co-simulation and emulation on FPGA. Another research group in Stanford recently proved useful the same approach [27]. But the main reason of our choice was the possibility to add custom hardware to the processor baseline and to extend its instruction set to deal with such add-ons. This allowed us to configure the Xtensa cores with a high-speed message-passing interface using TIE (Tensilica Instruction Extension) ports. Such I/O directly connects to the processor register-file and behaves as a FIFO queue interface with push/pop commands and full/empty flags. The scheme in Figure 2 depicts the Xtensa processing element and the way it connects to a switch through a NoC interface.

This figure reveals that TIE ports used for message-passing use the same FIFO signals used in the PE-Switch connection.

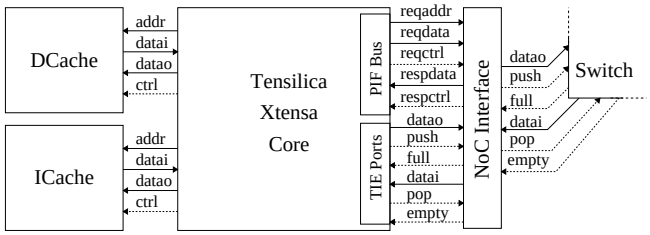

Figure 2: The processing element and its interface with the network-on-chip.

Hence the message-passing interprocess communication between cores is nothing else but a simple FIFO read/write process, as Figure 3-a shows. A message is split into flits when it is sent and the TIE interface puts sequence number and (X,Y) address into all flits. In order to speed-up the operation and to sustain the maximum throughput of one flit per clock cycle, we extended the processor core with a counter for the sequence number and a LUT for addressing. The sequence number removes the need to instantiate buffers to sort out-of-order received flits. When a flit arrives, the PE reads it from the NoC interface with a pop operation and stores it into a register. Then the flit gets written in memory by using its sequence number as an offset address while another register contains the base address. Received messages can be of *data* or *request* type and a bit of the flit's header is used as selector of the appropriate base address. A double buffer technique allows one clock cycle read operations. Additional hardware for reception of messages, whose overhead is around 5k gates for a 64 bit wide flit, is directly supported by custom TIE instructions. Figure 3-b describes the reception operation.

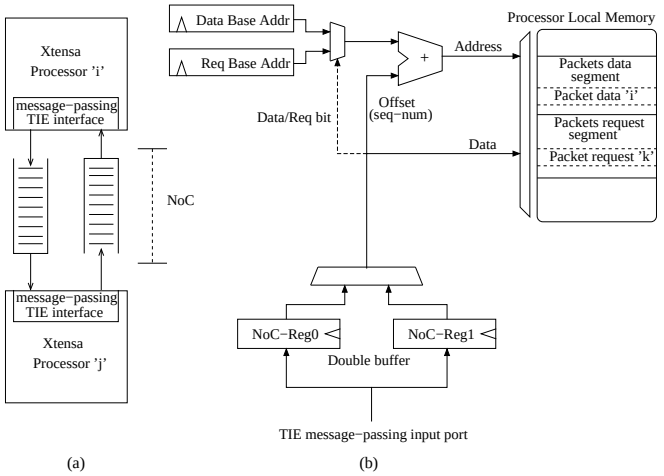

Figure 3: FIFO-like inter-processor communication model and details of the receiving interface.

Figure 2 showed that TIE ports assisted message-passing is not the only way PE's interact with the NoC. Every time a data or instruction miss occurs in local caches, a processor interface bus (PIF) transaction takes place as if a main memory was connected to that bus. The NoC interface, whose a scheme is reported in Figure 4, intercepts such requests and translates them into various types of NoC packets corresponding to single read/write operations or block transfers. Final destinations of such packets are the MPMMU's. Translation of a specific shared-memory address into a NoC address depends on a configuration memory inside the interface and can be directly configured by the microprocessor (MPMMU LUT in Figure 4). In block-read transactions, which always occur during cache misses, different flits containing words read from the MPMMU may arrive out-of-order. The usual processor configuration supports a cache line of 16 bytes thus a miss causes a block read of four 32 bits words. For this reason, the NoC interface contains a reordering buffer with a depth of four flits (every word is encapsulated in one flit). Even though there are no limitations in the number of MPMMUs in the whole system, in all our experiments we used a single MPMMU which receives all the memory transactions. Thus the interface prints a fixed NoC (X,Y) address for the whole memory address space in all flits of this type of transactions. In case of a distributed shared-memory, a given address range will correspond to a given MPMMU.

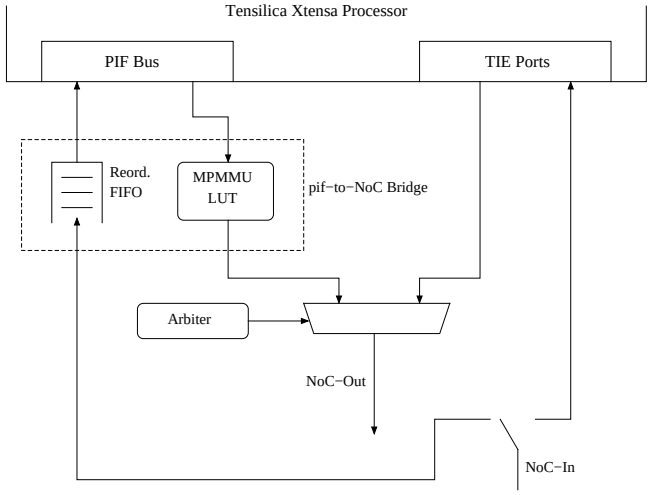

Figure 4: Shared-memory and message-passing interface to and from the NoC.

Since both message-passing and shared-memory transactions use the same NoC, the interface contains a simple arbiter that can be configured in three possible ways, depending on required system performance and area availability. In the simplest configuration the two interfaces connect to the NoC with a simple multiplexer and no buffers, as shown in Figure 4. In case of contention, the arbiter grants the NoC access to one interface while the other waits until the release of the resource. A second implementation uses a single FIFO queue that temporary stores packets from the two interfaces if the switch cannot accept them because of a local congestion. In the last implementation, two FIFOs are used, one for High-Priority and one for Best-Effort traffic. In this case the arbiter will read the best-effort queue only if the high-priority one is empty.

Concerning the core, apart from extensions that we used for the message-passing interface, there are a number of standard options that the user can customize before creating an instance of the core, ranging from pipeline stages (5 or 7), to cache size and associativity, integer unit options (like hardware support for

16x16 or 32x32 multiplications) and many others. Our system accepts cores with any of these options set. Since the architecture can be used for parallel scientific computations, a double precision floating point acceleration provided by Tensilica has been included in the core [28]. A small set of TIE instructions and states can be used for speeding up the existing double-precision software emulation. With the addition of just 4k-7k gates, an Xtensa processor performs double precision adds and subtracts in an average of 19 cycles while multiplies take an average of 60 cycles using 16 or 32 bit multipliers and only 26 cycles for configuration including the "Multiply High" option.

2.3. Multiprocessor Memory Management Unit (MPMMU)

An MPMMU is an Xtensa processor dedicated to shared-memory transactions. Differently from the PE of Figure 2, its PIF bus connects to a DDR-SDRAM controller and not to the NoC interface. The latter is also different because it hosts two FIFOs for incoming and one for outgoing packets. Incoming packets can be of *Pif-Requests/Control* or *Pif-Data* type. The *Pif-Request/Control* FIFO receives "request-for-transaction" tokens generated by cores which aim to perform read/write (single/block) shared-memory transactions. (The MPMMU can be seen as a slave, i.e. it always answers to transactions initiated by other processors.) The depth of this queue is as large as the number of processors, therefore once a core's request token arrives at the MPMMU's switch, the request leaves the NoC and gets queued even if the MPMMU is busy. This choice reduces greatly the probability of traffic congestion. The request token flit contains source-id of transaction initiator, memory address and type of transaction (write/read). In case of write request, the MPMMU issues a grant to the sender which is then enabled to send its data packet. The *Pif-Data* queue absorbs a temporarily speed mismatch between the switch and the MPMMU that reads from the queue and stores into memory, and is then useful to reduce NoC traffic, like the request queue does. At the end of this operation a second acknowledge is sent to the transaction initiator (Figure 5.a). Even though it might seem a better idea to send data immediately with the first request, and not waiting for the acknowledge, our choice reduces the need for more buffer area in the NoC interface of the MPMMU and reduces traffic congestion in the on-chip network. Moreover, once the core initiates a PIF transaction because of a write miss, the time spent in the network for the first round of request and acknowledge is a negligible fraction of the whole transaction time.

In case of a read transaction request, the MPMMU immediately sends requested data through the outgoing FIFO (Figure 5.b). Since the MPMMU has a local cache for both instructions and data, latency of read operations strongly depends on availability of the given word inside the cache.

The global shared-memory which can be banked and attached to different MPMMU's is divided into logic segments, i. e. a shared area and as many private areas as the processing elements. Since the private area can be accessed only by one PE, no coherency is required between L1 cache and system memory. In order to support atomic operations like critical sections, a lock/unlock mechanism of a given word in shared-memory has been implemented in the firmware of the MPMMU. Every

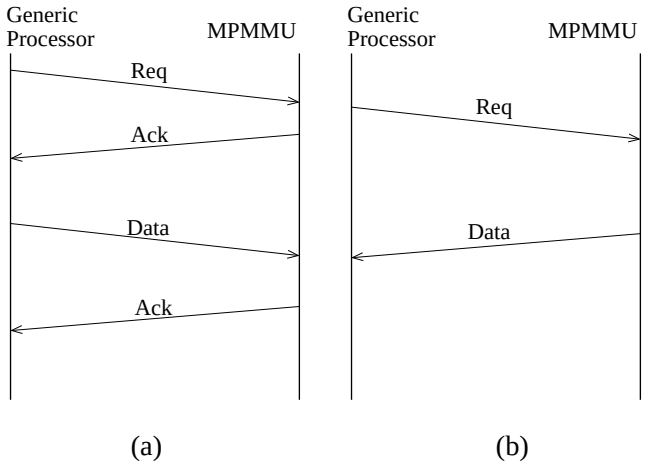

Figure 5: Write (a) and Read (b) protocol between a processor and the MPMMU.

processor which aims to access the shared memory segment for read/write operations must first request lock. If granted, the line can be read/written. Before releasing the locked line with an unlock command, the processor must flush the locked line in its L1 cache so as to keep coherency. All the lock/unlock requests are stored in the Pif-Requests/Control queue.

Even if not mandatory, all the processors access the same address-space, a choice that makes easier and faster to use the system. The MPMMU will thus add a base address different for each core on the basis of the source-id.

2.4. Network Protocol

The network protocol of the system can be divided into three levels: network, bridge and application. Network-level is used by NoC switches to route flits through the network. As we anticipated above, the network payload comes along with a simple header made of a validity bit and a (X,Y) destination address field the size of which depends on network size. For a 4x4 topology two bits are required for each coordinate. The bridge-level concerns PIF memory-mapped and message-passing transactions, which the interface transposes into the NoC acting as a *bus bridge*. The header for this level includes type, sub-type and sequence-number fields. The first one is a three bits field and expresses seven possible types of packets: single-read, single-write, block-read, block-write, lock and unlock for shared-memory transactions plus another one for generic message-passing packets. The sub-type two bits field used in shared-memory transactions defines if the packet is an Ack/Nack or has an Address/Data in the payload. In case of a message-passing flit, it is used to distinguish requests from generic data packets. The third one, sequence-number, is used at the receiver to perform the re-ordering process of incoming packets in case of out-of-order delivery. In the current implementation we set the size of this field to four bits, so that the maximum packet size is sixteen flits.

All the protocol fields of the application-level are written and used by the software layer. Source-id and burst-size are an example. For instance, in a 4x4 implementation the source-id is a

four bits field. The burst-size is used by the receiver and indicates how many flits, belonging to the same message, must be expected before closing the transaction.

Figure 6 is an example of encapsulated packets for the three levels in case of a 4x4 implementation with a single MPMMU.

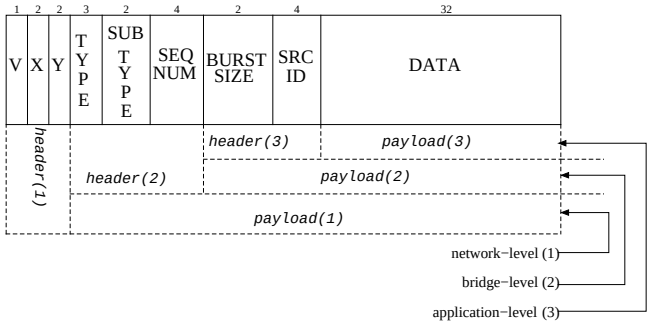

Figure 6: Three-levels packet format description.

2.5. Programming Model

As clear from discussion, the architecture supports a hybrid shared-memory/message-passing approach. To properly use the system it is important to fully understand its programming model. At startup, the code is placed in external DDR memories, partitioned according to the number of MPMMU instances. After reset all processors start to fetch initialization routines (C runtime initialization, interrupt vectors etc.). The private segment owned by each core is completely cacheable and no particular precautions are needed for cache coherency if cores only access that area or if the shared segment is read-only. More attention is required for writing shared-data placed in the shared-memory segment. First of all, data structures of this type must be declared as volatile to alert the compiler that optimizations involving those variables may potentially have side-effects. Second, when a producer wants to write data in this shared segment, a cache flush of the line must be performed to make sure that coherency exists between the local L1 cache and the global system-memory. An Xtensa custom instruction, *DIWBI*, is used for this purpose. Also the consumer of a given data in the shared segment must avoid incoherency making the corresponding address not cacheable. For small memory regions the *DII* instruction can be used, which invalidates a specific address of the cache and forces a load from the system memory. For wider segments of at least 512MB it would be a better choice to set all the segment as uncacheable bypassing completely the cache, mainly in case of frequent accesses.

For the message-passing model, we used a subset of message-passing interface (MPI) [29] APIs that we called embedded-MPI (eMPI), developed in a previous project in which cores were allowed to communicate only via message-passing and did not access a global shared memory [22]. Three basic primitives, MPI_send(), MPI_receive() and MPI_barrier(), are sufficient for direct communication between cores and can be used for synchronization and for data exchange. The MPI way of communicating performs best when data to be sent are resident in the local L1 data cache memory, otherwise the time to retrieve them from external memory after a cache miss sums up to total communication time, making MPI less convenient in this case than a coordinated shared-memory access. We will see in the results section that L1 cache size is a critical parameter and must be well tuned around the application or range of applications. Choosing a "one-size fits all" big L1 cache is not a good choice, as large memories steal silicon area that could be allocated to more computing elements [30].

A relevant point is that the hardware architecture of the communication infrastructure and the memory hierarchy is totally transparent to the processor cores. Hence we can make use of the standard development tool-chain including the compiler that does not have to be modified under any aspect.

3. System-Level Exploration

One of the issues with many-core designs is performance evaluation. The challenge arises from the necessity to co-simulate newly developed hardware, like PE's, NoC and related interfaces, with real software benchmarks running on, and from the multiplication of this problem by the number of core instances. Moreover, finding the right balance of all the architectural parameters involved, like number of active cores, cache size, packet lengths etc., requires many simulation batches.

Pursuing a full HDL modeling approach is not affordable in terms of simulation time, even though it would be preferable for its accuracy and matching with the implementation flow. Limiting usage of HDL modeling to just the NoC and glue logic and using a cycle-accurate Instruction Set Simulator (ISS) for the PE is possible and gives the same accuracy of a full HDL model as long as the ISS is cycle-accurate. Some commercial simulation tools like Mentor Graphics Seamless usually link ISS to HDL blocks through Inter Process Communication (IPC) at operating system level [31]. Although this may work for limited size architectures, the high number of nodes instantiated in our case, and the consequent large overhead of operating system task switching, makes this choice prohibitive if we intend to simulate in an acceptable time different instances of the architecture and to run various benchmarks on it.

As a consequence, we used an alternative modeling technique, based on the SystemC co-simulation language [32]. From the point of view of microprocessor cores, many ISS models written in SystemC are available, and for our architecture, the existence of a cycle-accurate ISS of the Tensilica core is a key point. Moreover, some commodity classes are available to build a Tensilica based computation node. They model local memories, attached to three different busses of the core, and a TIE queue, i.e. the communication port for message-passing. As for the custom developed hardware parts of the architecture, we modeled them both at RTL using a synthesizable VHDL coding style and with SystemC. We took care of checking their behavior, so as to be sure of perfect compliance. This allowed us two things:

- We built an FPGA emulator for small size versions of the architecture and then checked the SystemC simulator against it.

- We prototyped the layout of the systems blocks in a CMOS 45 nm technology and so we accurately estimated the area cost of the various configurations of the architecture (see section 4).

Once selected the modeling technology, we developed a basic framework, which we used to perform debug and execution tracing. The debug subsystem supports methods and classes used during the test of the designed architecture to guarantee model correctness. It is built up two interfaces, one template and one module (in SystemC semantics). They are designed as abstract objects, which must be extended to build the real versions. The template is just a general purpose counter, used in any profiling or performance counter of our system. The only class defined is Parameter, which specifies arguments passed to Debug interfaces and aimed to prepare and collect execution results. The two interfaces, Dump and Debug, must be implemented by every object in the modeled system, and perform the real work. Declared methods have the following meanings:

- **prepare_debug()** - Set debugging environment and initialize event counters.
- **start_debug()** - Start the debugging engine inside the module.
- **stop_debug()** - Stop the debugging engine inside the module.
- **is_debug_started()** - Check if the debugging engine is currently running.
- **get_debug_result()** - Extract event counters values from the current module.
- **dump()** - Save current module state for later reuse.
- **restore()** - Restore module state saved from a previous dump.

Based on these basic objects, three new abstract classes were developed. They are *Switch*, which represents the data switching portion of a NoC node, *Router*, a block describing the routing algorithm adopted, and *Resource*. Every possible data source/sink object connected to the NoC must be derived subclassing the last one. A resource can be either a synthetic traffic generator or a real Xtensa ISS, according to the type of simulation required.

This environment has been used to analyze the performance of the architecture running real benchmarks, as we discuss later on in section 5. Concerning the comparison against a HDL-ISS co-simulation, on average we achieved a speedup of 15x and perfect overlap of behavior. Such speed enables accurate design space explorations of many potential candidate architectures in hours, a relatively small time compared to days for the HDL-ISS version.

4. Physical Implementation

When different system architectures are evaluated, performance is often used as the only criterion of comparison. We think that a better figure of merit is performance for a given cost. Thus, we evaluated the cost in terms of silicon area of some possible architectural configurations. In particular, we configured Xtensa processors as described in Table 1 and limited design exploration to the number of active cores and the size of their L1 caches. Then we mapped the various architectural components described in section 2 on a CMOS 45 nm technology standard-cell library and designed the layout of a "tile" including the processor (core or MPMMU), its L1 data and instruction caches with their tags, the interface with the NoC and the switch. The areas of the different types of tile obtained varying the cache size are reported in Table 2.

Table 1: Principal parameters of the Xtensa processor configuration.

Pipeline stages	5
Number of physical registers	32
16x16 integer multiplier	yes
Zero-overhead loop instructions	yes
Count of Load/Store units	1
Write buffer entries	4
Width of Instruction Fetch interface	32 bits
Width of Data Memory/Cache interface	128 bits
Width of PIF interface	32 bits
Width of interface to instruction cache	32 bits
Instruction Cache size	2, 4, 8 or 16 kBytes
Instruction Cache Line size	16 Bytes
Instruction Cache Associativity	direct mapped
Data Cache size	2, 4, 8 or 16 kBytes
Data Cache Line size	16 Bytes
Data Cache Associativity	direct mapped
Write Back	yes
Debug	yes
Data address breakpoint registers	2
Instruction address breakpoint registers	2
On Chip Debug(OCD)	yes
External Debug Interrupt	yes
System RAM size	128 MBytes

Table 2: Area (mm^2) of CORE and MPMMU tiles in CMOS 45 nm technology as a function of data and instruction cache size.

	2 kB	4 kB	8 kB	16 kB
CORE tile area (mm^2)	0.28	0.30	0.35	0.45
MPMMU tile area (mm^2)	0.37	0.39	0.43	0.51

An exemplar floorplan and post-route layout of a core tile with 4 kB caches is shown in Figure 7. The pins connecting it to other tiles, not visible in figure, are positioned on tile edges in such a way that a simple abutting connection can be made. Therefore the area of an architecture made of P cores and M MPMMU's can be estimated by multiplying CORE tile area in Table 2 by P, MPMMU tile area by M and summing the two contributions together. As we will see in the next section, a larger area does not necessarily correspond to better performance. We then used the calculated areas together with performance evaluations obtained through the SystemC simulation environment to select the configurations for which an increasing area corresponds to a decreasing execution time.

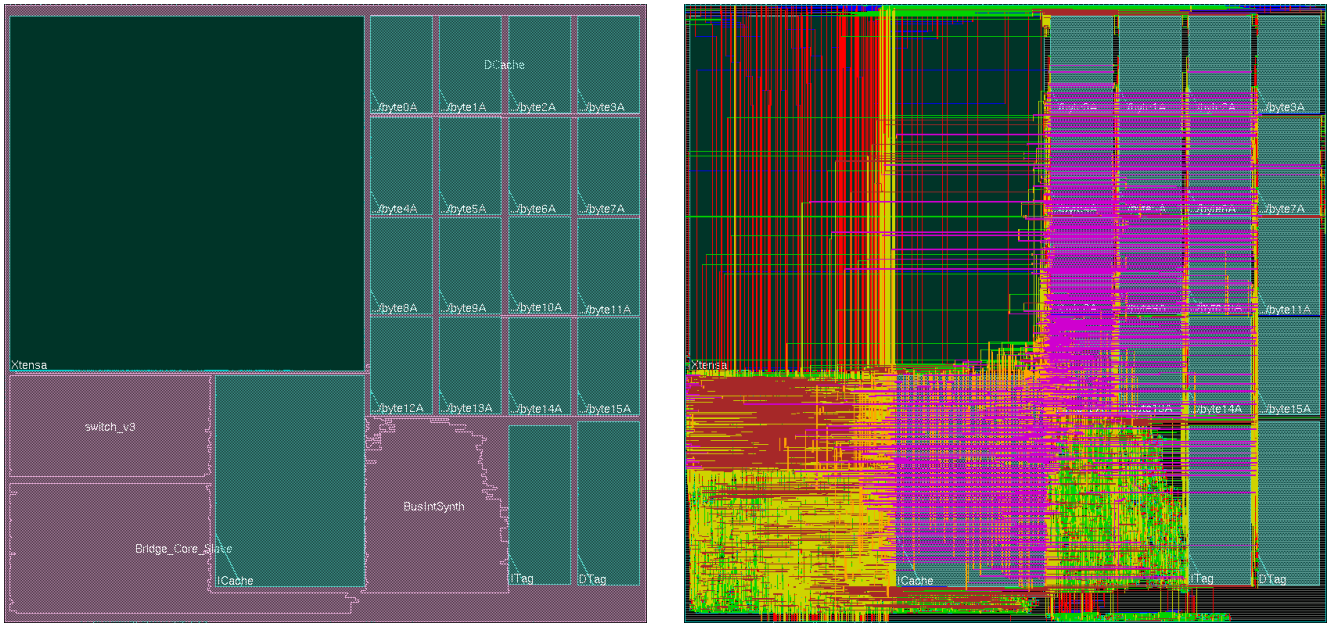

Figure 7: Floorplan (left) and post-route layout (right) of a core tile with 4 kB caches.

5. Experimental Results

We selected a few programs as benchmarks for the system, trying to pick some representative templates from different fields. From the scientific computation area, we chose a parallel version of the iterative Jacobi algorithm for the solution of 2D partial differential equations [33]. In the following we will refer to it simply as "JACOBI." The well-known Advanced Encryption Standard (AES) encoding algorithm was our second choice [34], in the following called "AES", which differently from JACOBI does not make use of floating point data. The third one was the so-called *marching cubes* algorithm – "MC" from now on – which is used in computer graphics applied to medical imaging for extracting a polygonal mesh of an isosurface from a three-dimensional scalar field [35]. MC makes a limited use of floating point operations and is characterized by rather frequent I/O accesses. The three algorithms have been chosen for their different characteristics which make some of them more suited for a message-passing approach rather than a shared memory one, or vice versa. They are briefly described in the following.

5.1. Parallel Benchmarks

It can be shown that JACOBI is an iterative solver for the class of elliptic 2D partial differential equations[33]. Having chosen a spatial discretization step, the 2D domain becomes a $N \times N$ matrix of calculation points (the largest square which contains the domain). From the point of view of computational complexity, a simple serial implementation requires $7N^2$ floating point operations in each iteration. The parallel implementation we have used consists in the subdivision of the matrix in horizontal slices, of size $N \times N/P$ each where P is the number of available processors, then in the assignment to each computing node of a subset of rows to process, and in the exchange of the two boundary rows among adjacent processors at each iteration. Each node executes $7N^2/P$ floating point operations per iteration, but $2NP$ floating point numbers must be exchanged on the communication network, too, which limits the effectiveness of parallelization. Iterations are stopped when the error decays within a specified precision limit. We implemented JACOBI following both a pure shared-memory and a hybrid MPI/shared-memory approach. In both cases instructions are read from the external shared-memory at every cache miss. In the full shared-memory case data are also exchanged by memory explicit accesses assisted by a semaphore-based synchronization. One of the cores, namely processor number 0, handles synchronization and decides to stop execution when the global error is under a predefined threshold. In the MPI version, data are explicitly exchanged via the short NoC links between cores. Synchronization is then implicit and one of the cores is used for initial data dispatch and final data collection after error checking (all cores send to such core their residual errors).

Figure 8 shows the communication patterns among the cores in the shared-memory and the MPI cases, under the hypothesis of a sixteen nodes NoC with 15 cores and a single MPMMU. Core number 0 is in charge of collecting data and of synchronization. Cores 1-14 execute the JACOBI computation and the MPMMU is the last processor. It's clear that the single MPMMU, numbered 15 in figure, suffers from excessive pressure in the full shared-memory case.

We implemented the AES cypher algorithm with 256 bits key length in MPI. When cores end working, they send encrypted data back to core 0. Hence, only core 0 accesses the external shared memory with loads/stores operations. The synchronization is done internally by processor 0 which keeps track of active and inactive cores through busy/idle variables stored in its local cache and not written back to external memory during operations. Parallelization consisted in dividing

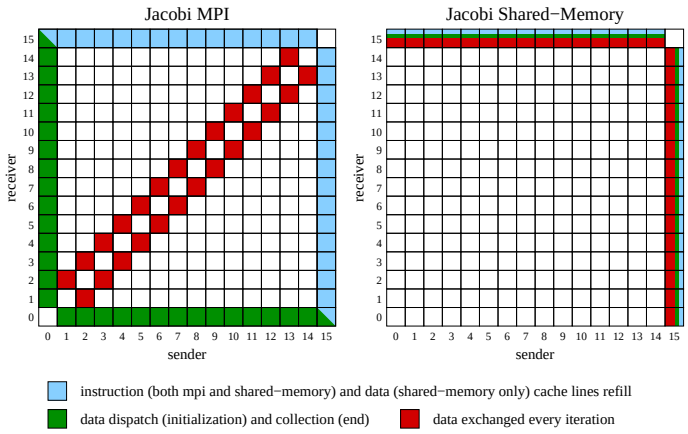

Figure 8: Jacobi communication patterns.

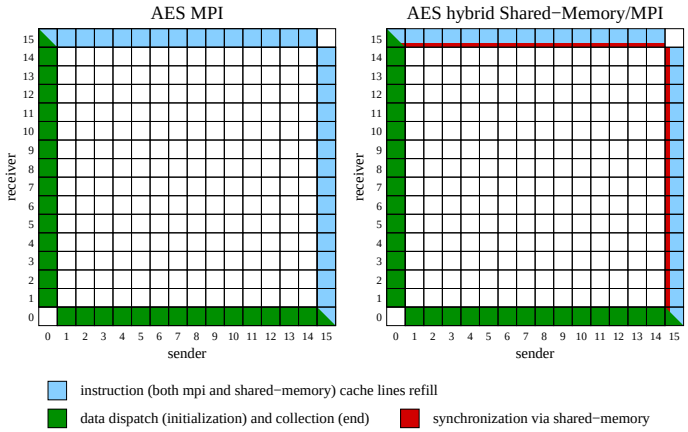

Figure 9: AES communication patterns.

the data blocks in chunks dispatched by core 0 to the other cores which are not busy. The computation cost per core is then N/P basic cypher operations to which we must add the communication cost, which in this case is again proportional to N/P per core but that sums up to N because communication is serial (core 0 sends and receives data to and from one core at a time). Differently from JACOBI in which a communication cost proportional to P makes this cost dominate and increase for large values of P, leading to a non monotonic speed-up curve, here speed-up saturates to a maximum value. Instead of making a full shared-memory version of AES-256, we only exposed the variables used for synchronization in the external memory, letting every processor access them and change their values to signify their willingness to accept new data. Despite the minimal modification, the results reported later will show that there are significant differences in performance. Similarly to Figure 8, Figure 9 shows the communication patterns among 16 active cores numbered from 0 to 15. In this case, data communications from and to core 0 dominate whereas the MPMMU is accessed in MPI during cache misses only. In the modified version, called for simplicity "AES hybrid shared-memory/MPI" in figure, the MPMMU is accessed also for synchronization.

The marching cubes algorithm was implemented in a pure shared-memory fashion without any use of MPI API's. At every iteration step, every active core processes a pair of 2D slices (cut along the z-axis) of a 3D data representation. Given N slices and P active cores, the amount of work per core scales like N/P. However the I/O is relevant given that any slice contains $M \times M$ values of 8-bit pixels intensity read from memory at every iteration, and given that for each voxel, that is a cube formed by eight adjacent pixels, the intersections of the surface with the cube edges are calculated and stored into memory. Therefore we expect performance dominated by communication costs. Core 0 is still used to synchronize the beginning and end of computation, but does not send any data to other cores which access external memory through the MPMMU at every instruction cache or data cache read/write misses. No further synchronization is required because all cores store computed data into separate address spaces. For all of these reasons, a MPI implementation would not give any additional speed-up. The communication patterns for this case and with 16 cores and a single MPMMU are then point-to-point connections from every core to the MPMMU and vice versa, and look just like the shared-memory version of Jacobi in Figure 8. Again, the MPMMU will be under high pressure.

We ran all the simulations concerning these three parallel benchmarks on two servers equipped with dual Xeon 3.2 GHz/1MByte L2 cache processors, 8 GByte RAM and SCSI Ultra 320 10k rpm hard disks. For every benchmark we varied:

- The number of active cores from 2 to 15 with at least cores 0 and 1 always included and the 16th NoC node being the MPMMU.
- The cores' cache size (data and instructions varied together) and the MPMMU's cache size (varied separately from the core's one) in four logarithmic steps from 2 kB to 16 kB.
- The size of the problems for JACOBI (16x16, 30x30 and 60x60 matrices) and AES (data blocks of 100, 200, 400 and 800 elements to encrypt) whereas only one case was considered for MC (16 2D slices of 256 pixels each).

The possible architectural configurations are 224, that is 14 cores $\times$ 4 core cache sizes $\times$ 4 MPMMU cache sizes. The simulation time varied significantly over the various benchmarks. Simulating MC required one day for the entire set of configurations and a similar time was necessary for each AES problem (then about 4 days for the whole AES simulations). JACOBI required about twice the time, because of the iterative nature and the larger use of floating point computations. Following sections report and discuss obtained results.

5.2. *JACOBI Simulations*

The six histograms in Figure 10 report the results for the execution time of a single JACOBI iteration, MPI on the left

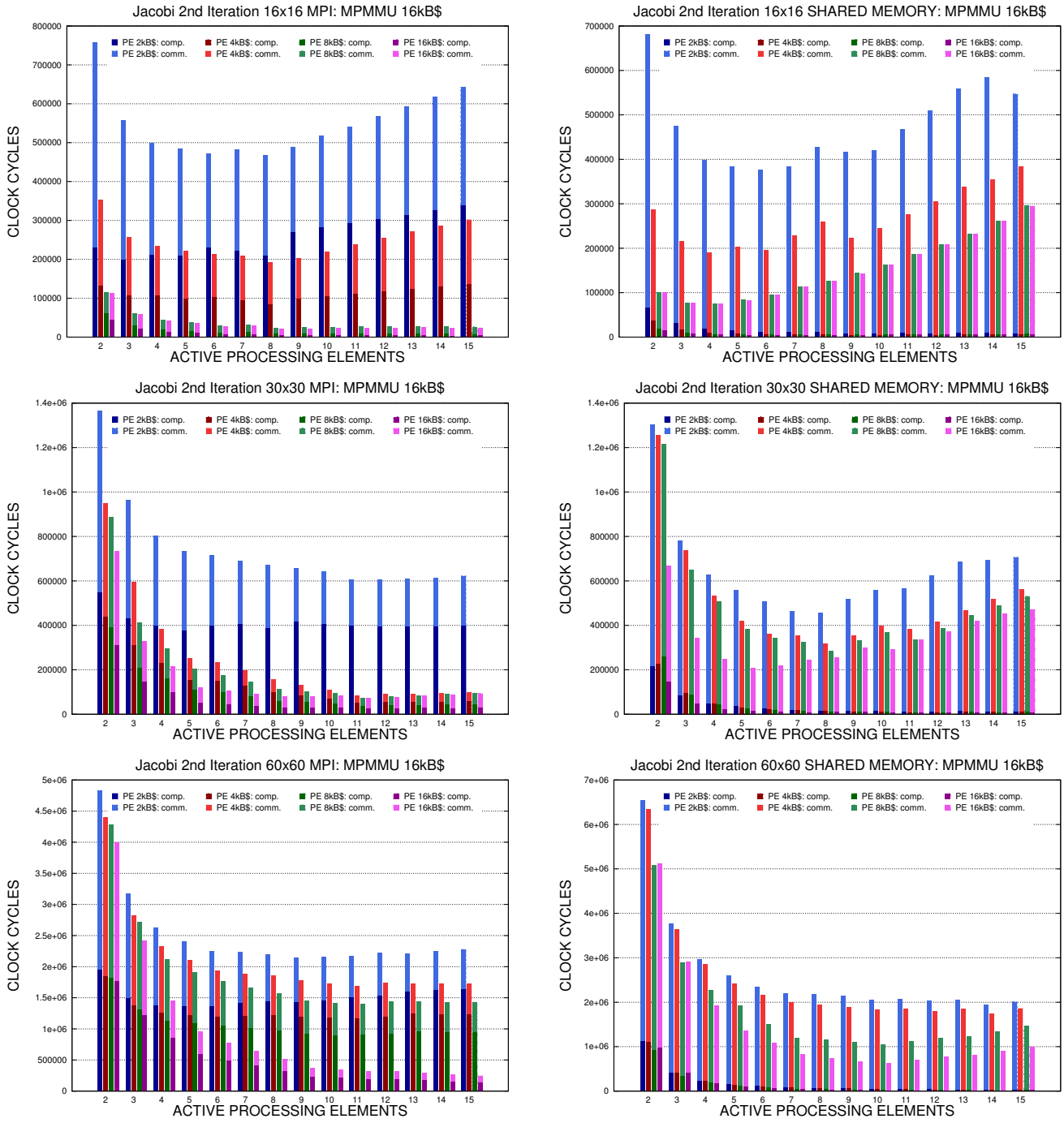

Figure 10: Execution time of Jacobi's one iteration for different problem sizes: message-passing (left) and shared-memory (right).

and shared-memory on the right, for the three different problem sizes and a 16 kB MPMMU cache size. (The graphs with smaller MPMMU cache are similar, with monotonically decreasing times with increasing MPMMU cache size.) In each graph cores cache size and number of active cores are varied. The fraction of computation and communication times are indicated by a slightly different color in each bar.

Results confirm expectations about the non monotonic behavior of execution times as a function of the number of active cores, the minimum being in the range 2-15 processors when the communication cost is very large, that is in the case with the smallest caches. For larger data size, the minimum is beyond the simulated core number. Results demonstrate the importance of choosing a cache size for the cores sufficiently large to avoid excessive cache misses and that, obviously, once the minimum cache size requirement is met increasing it further does not give any additional advantage. The comparison between MPI and shared-memory reveals that the better MPI performance is due to a smaller communication cost which instead is dominant for the shared-memory case.

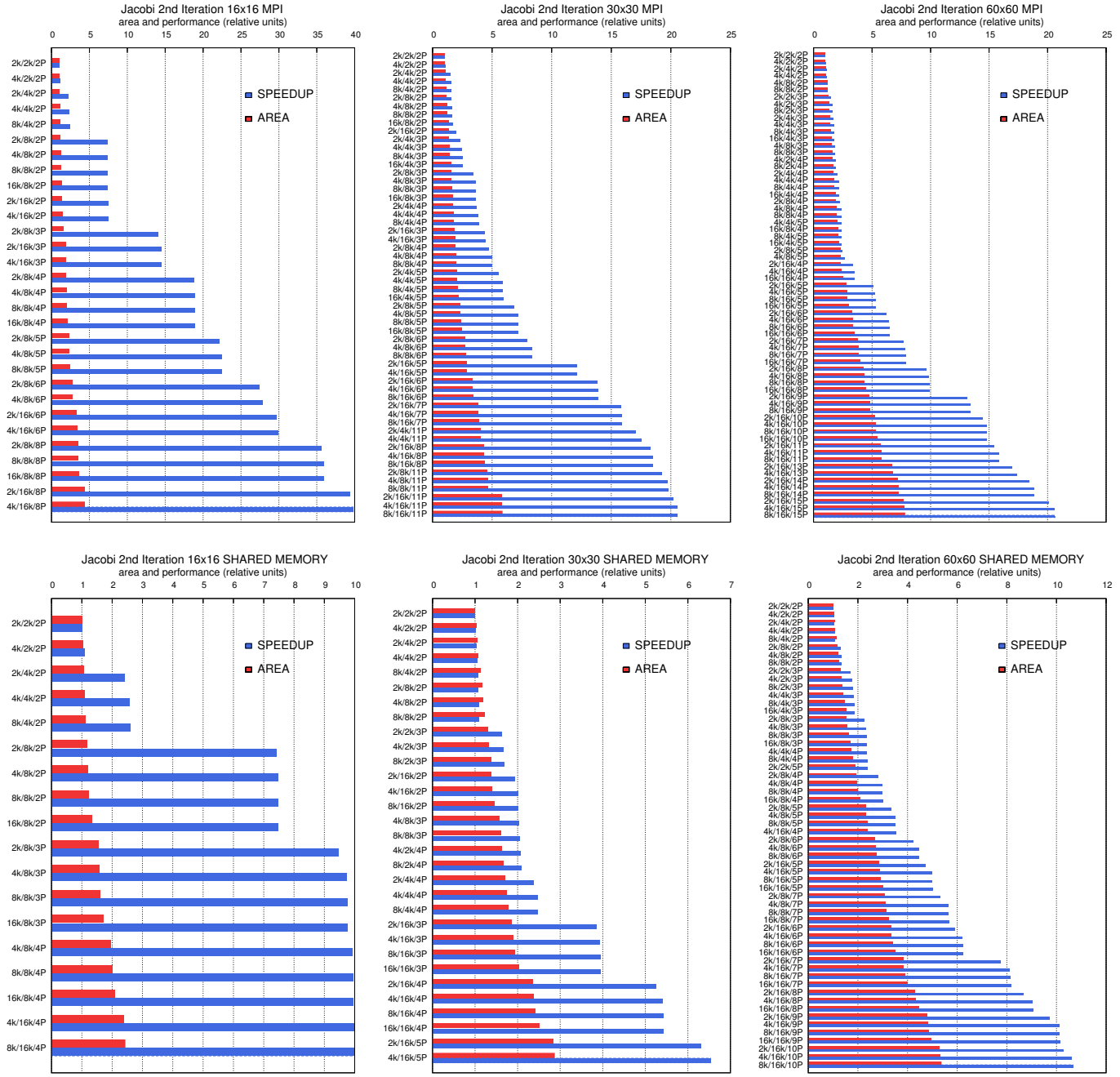

Figure 11: Jacobi optimal configurations for different problem sizes and their normalized area and speedup: message-passing (top) and shared-memory (bottom).

Graphs in Figure 10 gives an overview of the trend but do not reveal which are the optimal configurations. We thus pruned the explored solutions that were Pareto-dominated (larger area for a smaller performance) and kept only those that resulted in a performance increase at the minimum cost, starting from the architecture with the smallest area. The six histograms in Figure 11 report the speedup of optimal configurations, sorted from smaller to larger area, with respect to the smallest one. MPI results are on top whereas graphs on bottom correspond to shared-memory simulations. The area overhead is also reported so as to make clear the "cost" of a given performance speed-up. The larger the gap between area and performance bars, the better the configuration. The configurations are labeled as Xk/Yk/ZP where Z is the number of active cores, Y is the size of their private cache and X is the size of the MPMMU's cache. Under this perspective, it is possible to observe that

1. MPI exploits better the area at disposal giving a larger speedup for a given surface.
2. The larger the data size, the smaller the gap between speed-up and area overhead both in MPI and shared-memory.
3. The larger the data size, the more configurations become optimal (less configurations were pruned).

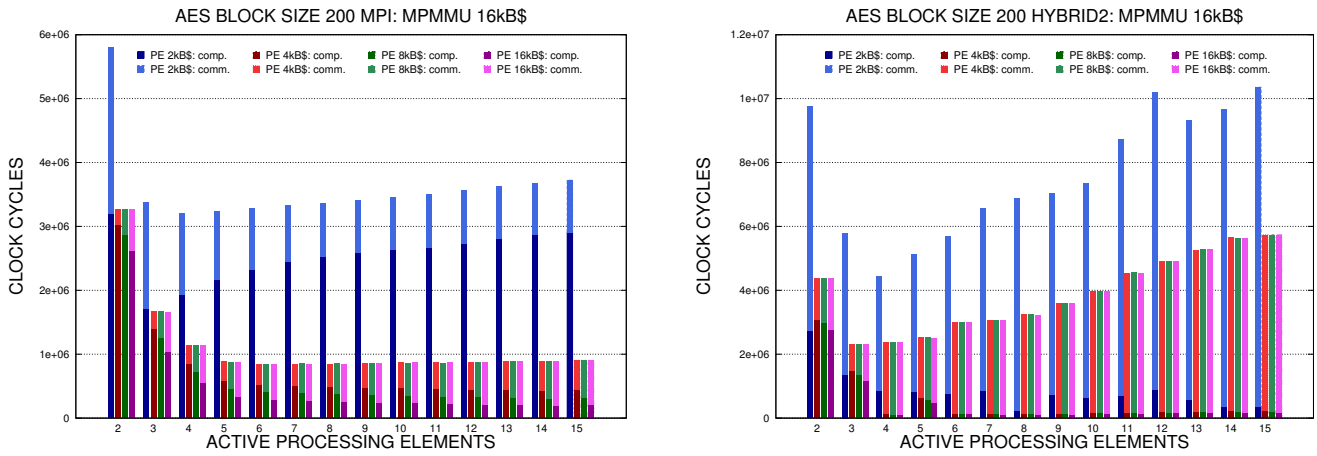

Figure 12: AES execution time for 200 data block size: message-passing (left) and shared-memory (right).

Observation 1 is straightforwardly explained by the smaller communication cost in MPI, something that we already noticed. The second observation simply derives from the fact that a given architecture, and so a given area, elaborates more data in more time. The third one is less intuitive and is explained by two facts. First, the larger the data matrix, the bigger is the processor count at which execution time reaches its minimum (we already noticed that it is not monotonic). Hence the configurations with the highest processor count, pruned when the data size is smaller, are instead not pruned when the data size is bigger. Second, if on the one hand configurations with large caches are useless for small size data, on the other hand progressively larger data require increasingly larger caches, and so the corresponding configurations are not pruned.

As far as the comparison between MPI and shared-memory goes, the advantage of the first is clear for the JACOBI implementation.

5.3. AES Simulations

Differently from the previous case, varying the size of the problem in the range (100,200,400,800) of data blocks size did not cause changes in trend lines. Therefore we report results in graphic form only for the 200 data block case, chosen arbitrarily as a representative of the class. In Figure 12 the MPI version and the hybrid one with shared-memory synchronization are compared in terms of execution time. Only the cases with 16 kB MPMMU's cache have been reported because other results scale uniformly as a function of this parameter. The MPI implementation confirms expectations about saturation of speed-up, whereas the hybrid version exhibits an initially decreasing execution time as the number of cores increases, followed by a reverse behavior. The reason is that the time spent for external synchronization depends on the number of active cores – the higher the worse – and so becomes a limiting factor. This is also corroborated by the much larger communication fraction of execution time in the shared-memory case, the computation time being more or less the same in the two cases.

Figure 13 reports the set of optimal configurations listed in the same spirit of previously reported JACOBI data in Figure 11. Results represented in Figure 12 show that hybrid implementation is at loss for architecture configurations more complex than just three cores (last and biggest good configuration is 2k/4k/3P – Figure 13-right). The MPI versions, too, stop giving a useful speed-up beyond 6 cores (last configuration is 8k/4k/6P– Figure 13-left), even though the advantage of keeping synchronization off shared-memory is evident.

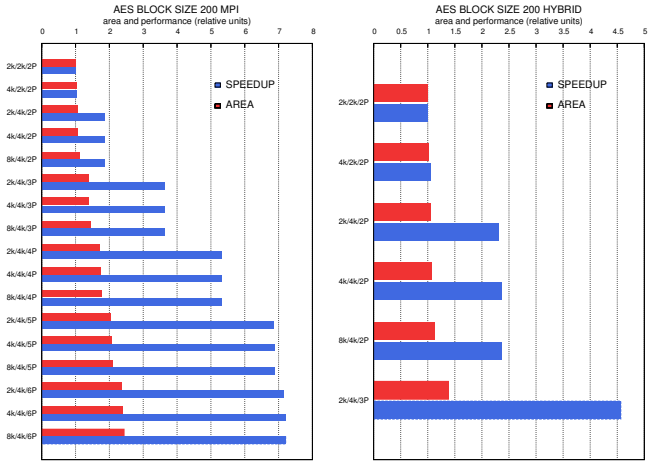

Figure 13: AES optimal configurations for 200 data block size and their normalized area and speedup: message-passing (left) and shared-memory (right).

5.4. MC Simulations

In the case here considered, MC execution time first decreases, then increases, as the histogram in Figure 14 demonstrates. Communication time dominates, as expected, and increases slightly as the core count grows beyond a certain amount, leading to optimal configurations with up to ten cores, as shown in Figure 15. The trend is similar to what happened in shared-memory implementations of JACOBI and AES, albeit with a slower pace, but the reason is not immediately apparent. In JACOBI that increase was justified by the increasing communication time with core count and in AES by the frequent access to external memory for synchronization. Here synchronization

cost is negligible and the amount of communication scales, theoretically, with the core count. However, slight deviations from theory are possible and communication I/O latencies that do not perfectly scale with processor count occur and result in an increase of execution time beyond ten cores. Such reverse scaling is more evident for smaller cache size where it is likely that not just data but instruction misses too sum up to communication costs.

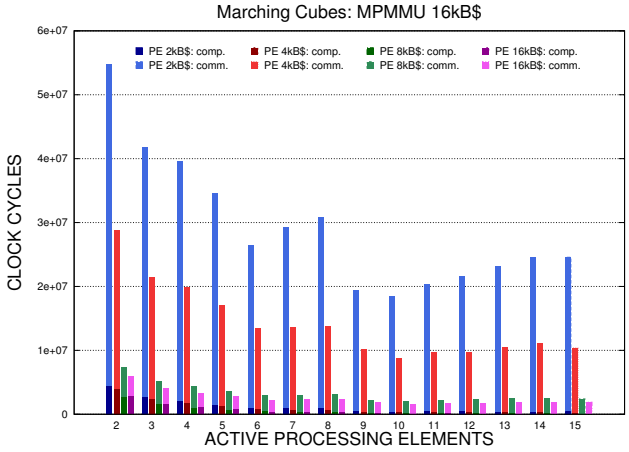

Figure 14: MC execution time.

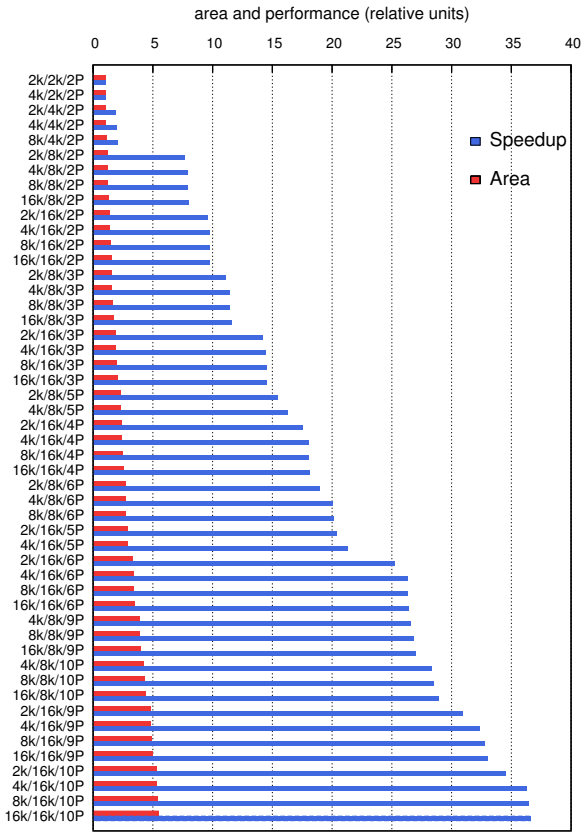

Figure 15: MC optimal configurations and their normalized area and speedup.

6. Discussion and Conclusions

The results demonstrate that when communications are moved from off to on-chip with MPI, execution time reduces and silicon area is better utilized: An area increase due to more active cores or larger caches most of the times corresponds to a better performance. On the contrary, the serialization of accesses to shared memory resources wastes the potential speedup of a parallel architecture.

A possible question concerns the deflection routing strategy and the fact that in case of congestion its latency increases, compared to other techniques. This may adversely impact especially shared-memory accesses. However, in our architecture communication with MPMMU's is based on a handshake (see Figure 5) and no multiple outstanding transactions can be in place at a time. This avoids NoC congestion for memory traffic and all involved packets experience no additional latency.

It can be argued that simulations with shared-memory option suffer from the choice of using a single MPMMU which creates an evident bottleneck and that having more parallel accesses will change the picture and the conclusions. This is only partly true. First, for a given area, if we increase the number of MPMMU's we will have less space for computing cores, something that might adversely impact performance, and then it is not granted that the more MPMMU's the better (it depends on the ratio between communication and computation time.) Second, if the application requires data to be shared among all the cores, it is likely that serialization of accesses wipes out the advantage of having many parallel MPMMU's. Third, if we move from 16 to 64 cores or 128, as we expect for future chip multiprocessors, we would need 4 or 8 times more accesses to external memories, and the question is how will package technologies help to support this need. We believe that as we scale towards the hundreds core regime, the number of memory accesses will not keep up with core number, even with 3D stacking technologies. Therefore we, and not only we [5], advocate the adoption of NoC-based MPI for future on-chip messaging in massively parallel CMP's. It's clear that this has a cost, not very a hardware cost but rather a software one because programmers must learn how to use MPI primitives. But it's true that if they want to efficiently utilize hundreds or even thousands of cores that forthcoming CMP's will made available, they will certainly have to change the way they write programs in any case. Hence, passing to the MPI paradigm will just come at a small marginal cost the returns will surely pay off.

A very important point that we didn't touch concerns energy efficiency, and is the subject of our future investigations. In particular we would like to extend our constrained approach that helped us select the optimal set of architectural parameters for a given area budget to the case of a fixed power budget.

7. Acknowledgments

The authors wish to thank Luca Rostagno for his help with SystemC simulations, and Simone Bonsignore who helped develop the RTL version of the NoC interface.

References

[1] ARM Cortex A9 MPCore™ processors, White Paper, http://www.arm.com/pdfs/ARMCortexA-9Processors.pdf

[2] J.L. Shin *et al.,* "A 40nm 16-Core 128-Thread CMT SPARC SoC Processor," Proc. 2010 IEEE International Solid-State Circuits Conference, San Jose (CA), Feb. 2010, pp. 98-99.

[3] J.G. Davis *et al.,* "Maximizing CMP Throughput with Mediocre Cores," Proc. 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05), Saint Louis, Missouri, Sep. 19, 2005, pp. 51-62.

[4] S. Vangal *et al.*, "An 80-tile 1.2 TFLOPS Network-on-Chip in 65nm CMOS", Proc. 2007 IEEE International Solid-State Circuits Conference, San Jose (CA), Feb. 2007, pp. 5-6.

[5] J. Howard *et al.,* "A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS," Proc. 2010 IEEE International Solid-State Circuits Conference, San Jose (CA), Feb. 2010, pp. 108-109.

[6] Xu Wang *et al.*, "A Quantitative Study of the On-Chip Network and Memory Hierarchy Design for Many-Core Processor", 14th IEEE International Conference on Parallel and Distributed Systems, 2008, pp. 689-696.

[7] J. Kuskin *et al.*, "The Stanford FLASH multiprocessor," Proceedings of the 21st Annual International Symposium on Computer Architecture, April 1994, pp. 302-313.

[8] J. Heinlein *et al.*, "Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor," Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) 1994, October 1994, pp. 38-50.

[9] A. Agarwal *et al.*, "The MIT Alewife Machine," Proceedings of the IEEE, vol. 87, no. 3, March 1999, pp. 430-444.

[10] C.-C. Kuo *et al.*, "ASCOMA: an adaptive hybrid shared memory architecture," Proceedings of the 1998 International Conference on Parallel Processing, August 1998, pp. 207-216.

[11] S .Kumar *et al.*, "A Network on Chip Architecture and Design Methodology," Proceedings of the IEEE Annual Symposium on VLSI, 2002, pp. 105-112.

[12] A. Radulescu *et al.*, "An Efficient On-Chip NI Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Configuration," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 24, no. 1, Jan. 2005, pp. 4-17.

[13] M. Forsell, "A Scalable High-Performance Computing Solution for Networks on Chips," IEEE Micro, vol. 22, no. 5, Sep.-Oct. 2002, pp. 46-55.

[14] P.G. Paulin *et al.*, "Parallel Programming Models for a Multiprocessor SoC Platform Applied to Networking and Multimedia," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 14, no. 7, July 2006, pp. 667-680.

[15] J.D. Owens *et al.*, "Research Challenges for On-Chip Interconnection Networks," IEEE Micro, vol. 27, no. 5, Sep.-Oct. 2007, pp. 96-108.

[16] W.J. Dally and B. Towles, "Route Packets, not Wires: On-Chip Interconnection Networks," Proceedings of the Design Automation Conference (DAC), 2001, pp. 684-689.

[17] C. Grecu *et al*, "Timing Analysis of Network on Chip Architectures for MP-SoC Platforms," *Microelectronics J.*, vol. 36, no. 9, pp. 833-845, Sep. 2005.

[18] J. Balfour and W.J. Dally, "Design Tradeoffs for Tiled CMP On-Chip Networks", ACM International Conference on Supercomputing, 2006, pp. 187-198.

[19] F. Gilabert *et al.*, "Exploring High-Dimensional Topologies for NoC Design Through an Integrated Analysis and Synthesis Framework ," Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip (NOCS), 2008, pp. 107-116.

[20] W.J. Dally and C.L. Seitz, "The Torus Routing Chip," Technical Report 5208: TR: 86; Computer Science Department, California Institute of Technology, 1986, pp. 1-19.

[21] M. Mirza-Aghatabar *et al.*, "An Empirical Investigation of Mesh and Torus NoC Topologies Under Different Routing Algorithms and Traffic Models," Euromicro Conference on Digital System Design Architectures, Methods and Tools, 2007, pp. 19-26.

[22] S.V. Tota *et al.*, "A Case Study for NoC Based Homogeneous MPSoC Architectures," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 17, no. 3, March 2009, pp. 384-388.

[23] Z. Lu *et al.*, "Evaluation of On-chip Networks Using Deflection Routing," Proceedings of the 2006 ACM Great Likes Symposium on VLSI, Philadelphia, April 30-May 2, pp. 296-301.

[24] S.V. Tota, M.R. Casu, L. Macchiarulo, "Implementation Analysis of NoC: A MPSoC Trace-Driven Approach", Proceedings of the 2006 ACM Great Likes Symposium on VLSI, Philadelphia, April 30-May 2, pp. 204-209.

[25] T. Moscibroda and O. Mutlu, "A Case for Bufferless Routing in On-Chip Networks," Proceedings of ISCA'09, June 20-24, 2009, Austin, Texas, USA, pp. 196-207.

[26] M. Steenstrup, ed., *Routing in Communication Networks*, pp. 263-305, Prentice Hall, 1995.

[27] A. Solomatnikov *et al.*, "Using a Configurable Processor Generator for Computer Architecture Prototyping," Proc. MICRO'09, December 12-16, 2009, New York, NY, USA, pp. 358-369.

[28] Tensilica White Papers, http://tensilica.com/pdf/DoublePrecision_FPEmulationAcceleration.pdf

[29] Marc Snir *et al.*, "MPI: The Complete Reference", MIT Press, 1998.

[30] A. Agarwal and M. Levy, "The KILL Rule for Multicore," Design Automation Conference (DAC), 2007. 4-8 June 2007, pp. 750-753.

[31] Mentor Graphics Seamless Product Page, http://www.mentor.com/products/fv/seamless

[32] SystemC OSCI Web Site, http://www.systemc.org

[33] G.E. Karniadakis and R.M. Kirby II, "Parallel Scientific Computing in C++ and MPI," Cambridge University Press, 2003.

[34] J. Daemen and V. Rijmen, *The Design of Rijndael, AES - The Advanced Encryption Standard*, Springer-Verlag 2002 (238 pp.)

[35] W.E. Lorensen and H.E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," ACM Computer Graphics, Vol. 21, No. 4, July 1987, pp. 163-169.