MEDEA: A Hybrid Shared-memory/Message-passing Multiprocessor NoC-based Architecture

MEDEA: a Hybrid Shared-memory/Message-passing Multiprocessor NoC-based Architecture

Sergio V. Tota, Mario R. Casu, Massimo Ruo Roch, Luca Rostagno, Maurizio Zamboni
Dipartimento di Elettronica, Politecnico di Torino, Corso Duca degli Abruzzi 24, I-10129 Torino, Italy
{sergio.tota,mario.casu,massimo.ruoroch,maurizio.zamboni}@polito.it

***Abstract*—The shared-memory model has been adopted, both for data exchange as well as synchronization using semaphores in almost every on-chip multiprocessor implementation, ranging from general purpose chip multiprocessors (CMPs) to domain specific multi-core graphics processing units (GPUs). Low-latency synchronization is desirable but is hard to achieve in practice due to the memory hierarchy. On the contrary, an explicit exchange of synchronization tokens among the processing elements through dedicated on-chip links would be beneficial for the overall system performance. In this paper we propose the *Medea* NoC-based framework, a hybrid shared-memory/message-passing approach. *Medea* has been modeled with a fast, cycle-accurate SystemC implementation enabling a fast system exploration varying several parameters like number and types of cores, cache size and policy and NoC features. In addition, every SystemC block has its RTL counterpart for physical implementation on FPGAs and ASICs. A parallel version of the Jacobi algorithm has been used as a test application to validate the metodology. Results confirm expectations about performance and effectiveness of system exploration and design.**

I. Introduction and Related Work

The increasing number of cores that can be integrated in a die is leading to a deep revolution in the microprocessor semiconductor industry [1]. General purpose architectures like chip multiprocessors (CMPs) as well as domain specific multi-core architectures like graphics processing units (GPUs) are quickly switching from few complex out-of-order processors to many smaller and simpler in-order architectures [2][3][4]. The communication infrastructure plays a key role in this environment. Current implementations use bus or ring network solutions [5][6] which do not provide enough scalability for next-generations multi-core architectures. The use of packet-switched on-chip networks (called NoC) using switches placed on-chip according to a regular topology in order to provide an efficient and scalable communication sub-system is now a well-accepted concept [7]. The scalability of the programming model is another facet of the problem. Classic shared-memory paradigm is facing the limit of the standard memory-hierarchy, which is the true performance wall [8]. Historically, the message passing approach was proposed as a solution for parallel and efficient communication of cluster-based systems. The reasons that brought to that choice now seem to be appropriate for the new environment which looks like a *cluster-on-chip*. In particular, the opportunity that is given by the message-passing paradigm is that synchronization as well as data-exchange among different cores can be done in parallel thanks to the distributed low-latency on-chip network without any need to access a shared memory resource even if shared-memory programming model is fully supported.

In this work we propose the *Medea* framework, a configurable hybrid shared-memory/message-passing architecture. Instructions fetch and load/store operations adhere to the standard shared-memory model whereas synchronization and data exchange among cores may occur, for performance or cost reasons, by means of an explicit low-latency message-passing technique using a NoC. System design exploration is performed with a high-performace cycle-accurate simulator which makes it possible to perform simulations of more than one hundred configurations in just one day and use obtained results in order to properly tune the system.

The idea of an architecture including hardware support both for shared memories and message passing dates back to the 90's. Examples are Stanford FLASH multiprocessor [9], [10], MIT Alewife machine [11] and ASCOMA [12] in which a conventional processor with cache and local memories interfaces to a special purpose device managing I/O and interconnection to other nodes in the network. Tilera [4] produces a chip with 64 microprocessors organized in an 8x8 mesh, and interconnected through a network-on-chip approach. Each microprocessor is a 3-wide VLIW machine. As this device is targeted to high data bandwidth applications, five different interconnection networks co-exist, dedicated to different tasks, and with different routing policies. Our approach differs from the preceding one because our nodes do not include any MMU with TLB. Each node includes a simple RISC-type microprocessor with a special link for NoC I/Os. Only a single interconnection network is used. The router itself is as simple as possible due to the so-called hot-potato routing strategy. The choice of architecture parameters is application-driven thus requiring a highly-efficient design-exploration technique which is provided in our work.

The paper is organized as follows. Section II describes the *Medea* architecture with details about different system components. Simulation results are presented in section III together with their interpretation. Finally, section IV reports the conclusions and gives hints about future research directions.

978-3-9810801-6-2/DATE10 © 2010 EDAA

II. System Architecture

The system is composed of three basic elements: an on-chip network, Processing Elements and their interface to the NoC, and the Multiprocessor Memory Management Unit (MPMMU) (Fig.1).

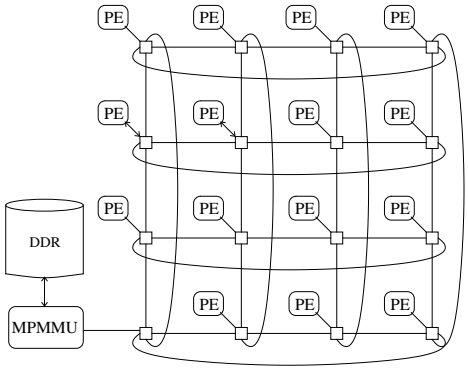

Fig. 1. An example *Medea* configuration

A. NoC Infrastructure

The Network-on-Chip infrastructure is based on a two-dimensional folded torus topology. Intrinsically 2-dimensional networks such as meshes and tori optimally suit silicon implementation [13]. Concerning network routing strategies, switches implement the deflection-routing algorithm which uses a full-blown packet-switching methodology by allowing different routing for every flit of the same packet. The basic idea is that of choosing the presently "best" route for each incoming flit, without ever keeping more than one flit per input channel (thus the alternative name of "Hot Potato" routing). This type of adaptive routing does not suffer from deadlock [14] while livelock may occur in theory. However in our previous works [15] we observed sporadic cases of single flits delivered with high latency (larger than average) that did not significantly hamper execution times. In this case too we did not observe any overhead due to significant excess of latency. Another advantage of deflection-routing is the small area of the switch. Its storage requirements are the theoretically minimal ones (as much memory as the incoming flits), no bottleneck is created by long packets as in wormhole routing, and no back-pressure mechanism is needed. The expense is the introduction of potentially out-of-order reception of flits belonging to the same packet at destination. These considerations have been taken into account during the implementation of our network interface as discussed in the next paragraph.

B. Processing Element and NoC Interfacing

The high degree of configurability of the Tensilica Xtensa-LX processor was used to implement the MPI message-passing interface as a high-speed direct link between each processor and the switch using TIE (Tensilica Instruction Extension) ports. This I/O directly connects to the processor register-file and behaves as a FIFO queue interface (Fig.2-a). When a packet of length L flits must be sent, the interface puts a sequence number into all flits. An address in the form X-Y is put as well. In order to speed-up the operation and to afford a maximum throughput of a flit per cycle, an additional counter for the sequence number and a LUT for addressing has been instantiated within the processor core and is directly supported by custom TIE instructions. The sequence number is used at the receiver to avoid any buffer for sorting out-of-order received flits. When a flit arrives, the PE first reads a flit from the NoC storing it into a register and then uses the sequence number of the given flit as an offset address for the storage into the processor data memory. Another register contains the base address. A double buffer technique enables one clock cycle read operations (Fig.2-b).

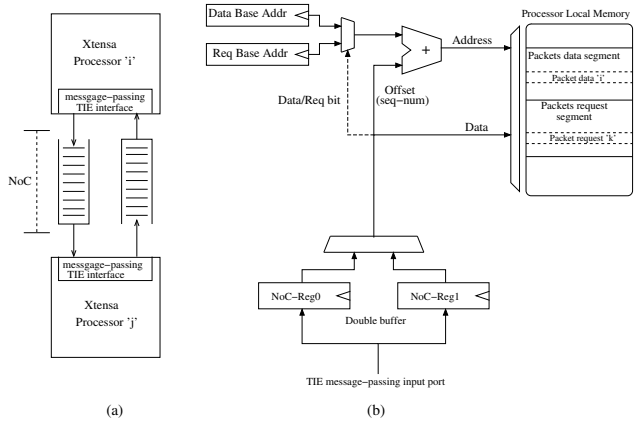

Fig. 2. FIFO-like IPC model and details of the receiving interface

The size of the sequence-number field determines the size of logic packets. Additional hardware consists of a small adder and two registers, which are seamlessly integrated by Tensilica core development tools in the processor pipeline. The choice of embedding the interface in the processor allowed the ISA and consequently the compiler to natively support all message-passing I/O thus facilitating the development of an *ad-hoc* scalable programming model. This is mandatory because scalability of the hardware infrastructure must be fully supported by software layers. Gate count overhead is around 5k gates for a 64 bit wide flit. The shared-memory interface between the processor and the NoC has been implemented through the *pif2NoC* bridge, which translates the Tensilica PIF protocol bus transactions to a sequence of NoC flits accordingly. The bridge is capable of single read/write operations as well as block transfers. The translation of a specific shared-memory address into a NoC address depends on a configuration memory inside the bridge and can be directly configured by the microprocessor. In the simplest *Medea* implementation, all the memory mapped address space is located at the unique MPMMU of the system, (even if there are no limitations in the number of MPMMUs of the system) thus the corresponding NoC address is hardwired. This solution reflects the choice of a single physical memory node. In every block-read transaction, the different flits containing words read from the MPMMU may arrive out-of-order. Block

read are common during cache misses. The current 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 *pif2NoC* bridge also contains a reordering buffer which currently has a depth of four words. Access to the NoC of the two different interfaces, message-passing and shared-memory, is guaranteed by a simple and configurable arbiter.

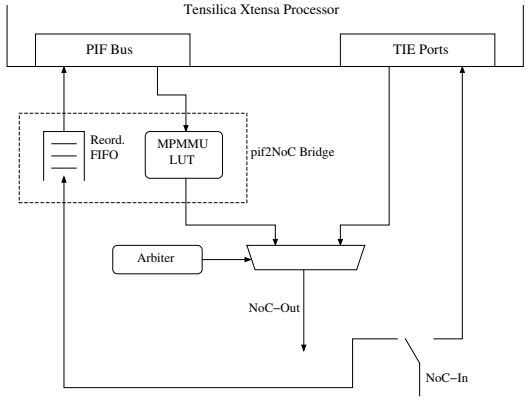

Fig. 3. Shared-memory and message-passing interface to and from the NoC

Three possible configurations are possible depending on required system performance and area availability. In the first configuration the two interfaces connect to the NoC with a simple multiplexer and no buffers (Fig. 3). In case of contention, one interface will be granted to write to the NoC while the other will wait until the release of the resource. In a second implementation a single FIFO is available thus even if the switch connected to the given processor can not accept other packets due to congestion, the two interfaces still have the possibility to store packets in the queue. In the last implementation, two FIFOs are used, one for *High-Priority* traffic 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. Since the *Medea* architecture can be used for scientific computations, a double precision floating point acceleration provided directly by Tensilica has been included [16]. With just 4k-7k more gates, an Xtensa processor can perform 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 a processor configuration that includes the "Multiply High" option.

C. Multiprocessor Memory Management Unit

The Multiprocessor Memory Management Unit (MPMMU) is a special processor which handles shared-memory transactions (reads/writes) using a protocol defined by the authors. The MPMMU has one NoC interface, - i.e. the TIE ports previously discussed - and a PIF bus connected to a DDR controller. The MPMMU can be seen as a slave, i.e. it always answers to transactions initiated by other processors. The NoC interface uses two FIFOs for incoming packets and one FIFO 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 aims to perform read/write (single/block) shared-memory transactions. The depth of this queue is as large as the number of processors. The token contains source-id of transaction, memory address and type of transaction. In case of a write request, the MPMMU issues a grant to the sender. Incoming data will be stored into the *Pif-Data* queue, read by the MPMMU and stored into memory. At the end of this operation a second acknowledge is sent to the transaction initiator (Fig.4.a). In case of a read transaction request, the MPMMU sends requested data immediately using the outgoing FIFO (Fig.4.b). Since the MPMMU has a local cache for both instructions and data, the latency of read operations strongly depends on the availability of the given word inside the cache or not. The *Request/Data* protocol has been implemented to provide an implicit flow-control scheme in order to minimize local buffers in the MPMMU.

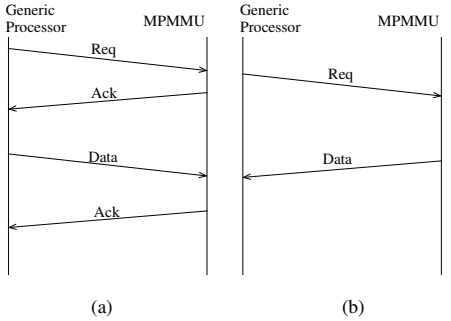

Fig. 4. Write (a) and Read (b) protocol between a processor and the MPMMU

The global shared-memory is divided into two logic segments, shared and private area. A system with N cores will thus have N private segments and one shared segment. Since the private area can be accessed only by one processor, no coherency is required between the L1 cache of that processor image of the private segment on the 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. Every 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 perform a L1 cache flush operation of the locked line in order to keep coherency. After the line flush the processor can issue an unlock. All the lock/unlock requests are stored in the *Pif-Requests/Control* queue.

D. Network Protocol

The system network protocol can be divided into three levels: *transport*, *bridge* and *application*. Transport-level is used by NoC switches to route flits through the network, requiring only the destination address expressed as X-Y coordinates and a validity bit. Destination address field depends on network size. For a 4x4 folded-torus topology two bits are required for each coordinate. A pif2NoC bridge supports memory-mapped

transactions, thus some extra fields are required: *type*, *sub-type* and *sequence-number*. 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 field, is a two bits field used in shared-memory transactions to define if the given 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 a four bits field and is used at the receiver to perform the re-ordering process of incoming packets in case of out-of-order delivery. The pif2NoC bridge has an internal re-ordering buffer which has a depth of 4 elements in current *Medea* implementation. The TIE message-passing interface instead has an internal hardware, supported at instruction-level, which uses the sequence number as an offset to properly store incoming packets in memory. All the protocol fields of the application-level are written and used by the software layer. *Source-Id* and *burst-size* are an example. In this *Medea* implementation the source-Id is a four bits while the burst-size, 2 bits wide, is used by the receiver and indicates how many flits, belonging to the same logic packet, must be expected.

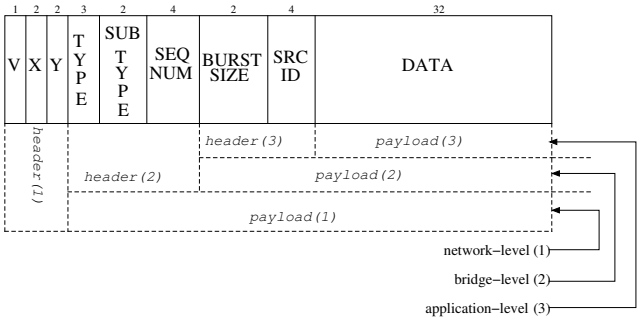

Fig. 5. Three-levels packet format description of the *Medea* architecture

E. Programming Model

At startup, the code to be executed is placed in an external DDR memory. After reset all processors start to fetch initialization routines. If a processor accesses its own private memory segment located in system-memory, no particular precautions are needed for cache-coherency, thus this segment is completely cacheable. More attention is required for shared-data. First of all shared data-structures must be placed in the shared-memory segment and declared as *volatile* to alert the compiler that this structure has potential side-effects. Second, when the producer wants to write a 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. Also the consumer of a given data in the shared segment must avoid incoherency making its address uncacheable. For small memory regions the *DII* instruction can be used, in order to invalidate a specific address of the cache thus forcing a fetch 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 implemented a subset of MPI APIs [17] called embedded-MPI (*e*MPI). With just three basic primitives, *MPI_send()*, *MPI_receive()* and *MPI_barrier()* for synchronization, a direct communication between cores is possible totally avoiding in some cases the access to the global-memory. These high-performance I/O primitives can be used for synchronization between cores as well as for data exchange. In this case the best conditions is when data to be sent completely resides in the local L1 cache.

III. Results

Cycle-accurate system-C models of architectural blocks have been developed together with their RTL versions. Tests of compliance as well as speedup compared to a HDL-ISS co-simulation have been run. 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.

In order to highlight architectural properties, it was necessary to select a benchmark that was able to stress both computation and communication resources. Moreover, such application must be scalable with both number of processing elements as well as memory sizes in such a way to keep a constant level of pressure on system with different characteristics. The chosen algorithm is an iterative solver for 2D-partial differential equations. It can be shown that the Jacobi algorithm [18] is a solver for this class of problems. The Jacobi algorithm was selected as a good representative of the class of scientific computational kernels that may fully exploit the potential of a manycore CMP architecture using a hybrid shared-memory/message-passing approach. We have been able to run a parallel implementation of the Jacobi algorithm for three different sizes of input data on 168 different architectures in about 1 day using 5 servers equipped with dual Xeon 3.2 GHz/1MByte L2 cache processors, 8 GByte RAM and SCSI Ultra 320 10k rpm hard disks. The 168 points in the design space have been obtained varying the number of processor cores between 3 and 16 (1 of which is the MPMMU,) cache size between 2kB and 64kB (scaled according to the power of 2,) Write-Back and Write-Through cache policy. As for the data size, the Jacobi algorithm was run on arrays of 16x16, 30x30 and 60x60 double precision floating-point. The three sizes, though relatively small compared to a large Jacobi problem solved by a cluster of hundreds nodes, in this case of up to 16 on-chip cores cover three cases of small, moderate, and large amount of data per core. In particular, the smallest case will be dominated by communication costs whereas the performance of the largest one will be dictated by computation costs, at least for a properly designed system, as we will be able to demonstrate shortly.

It will also be clear that a system optimally configured for the hybrid approach (cache size and core number) is not optimal for the purely shared memory case, particularly

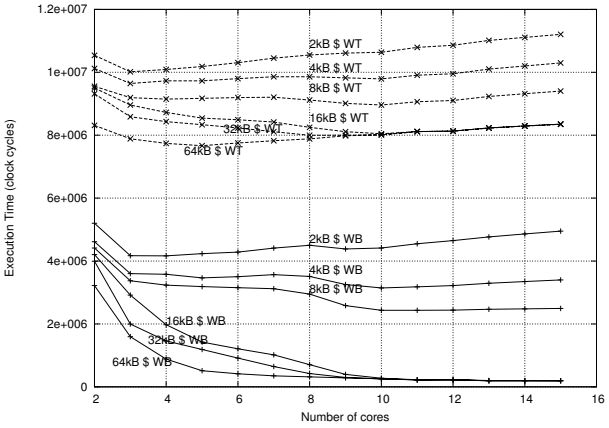

Fig. 6. Execution time for a 60x60 array varying number of cores, cache size and cache policy.

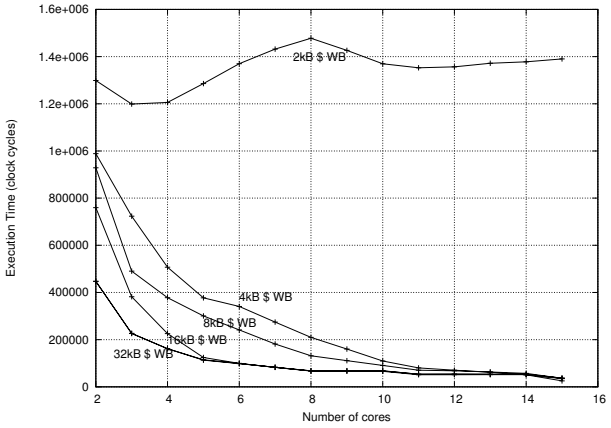

Fig. 8. Execution time for a 30x30 array varying number of cores and cache size, write-back case.

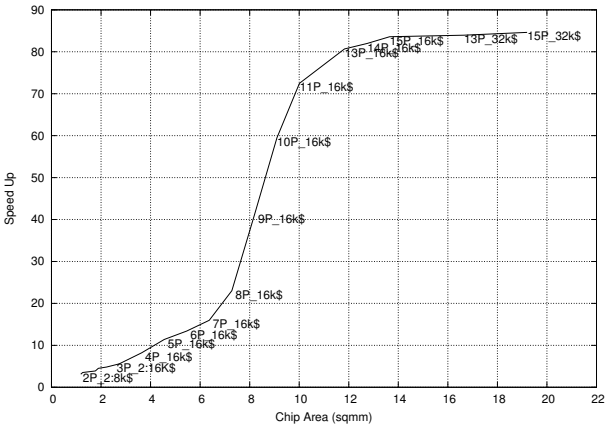

Fig. 7. Run on 60x60 array: optimal speedup and corresponding architecture configuration versus chip area.

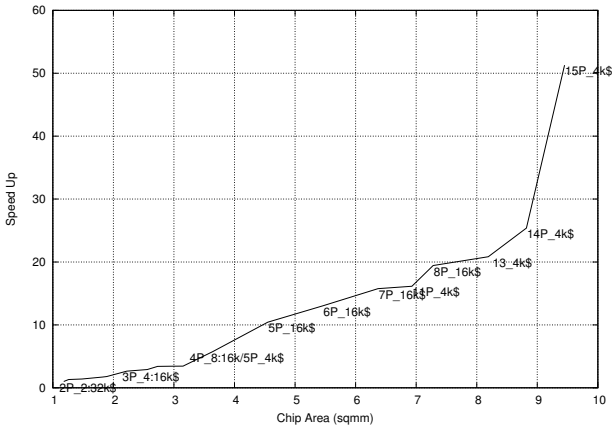

Fig. 9. Run on 30x30 array: optimal speedup and corresponding architecture configuration versus chip area.

in terms of number of cores. The curves in figure 6 are representative of the type of system data the simulator can present to the user. In this case, plots represent execution time in clock cycles for an iteration of the Jacobi algorithm after cache warm-up as a function of the number of active processors (2 to 15) and varying the cache size. The data size is 60x60. As expected, the Write-Through policy, if on the one hand makes it easier to keep caches coherent, shows poor performance due to the excessive amount of traffic. As for the Write-Back case, communication cost due to high miss rate almost dominates for cache size less than 8kB leading to small or no speedup at all. When the amount of data per core fits in the cache size, computation costs emerge that clearly scale with core number. Figure 8 reports execution time for the 30x30 case, write-back only. Scalability is hampered if caches are not properly sized. In the 30x30 case cache must be at least 4kB large, a value 4x less than the larger 60x60 case because the array is 4x smaller here.

The last remark concerning area (and as a related variable also power consumption) is important for cost-effective or power-constrained multicore implementations. In general, a good rule-of-thumb allows an increase of a resource size only if for every 1% increase in core area there is at least a 1% increase in core performance (the "Kill" rule, t.i. "kill if less than linear" [19]). 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 resulting "optimal Speedup" is plotted in figure 7 for the three cases of data size as a function of the chip area. The latter was estimated from core/cache data given by the processor vendor for a TSMC 65nm CMOS technology and including an overhead for NoC switches, bridges and routing area of about 100% of the total core area (excluding caches) [20]. Labels along the curves describe the processor/cache configuration that corresponds to the optimal (area, speedup) point of the curve to which the label is attached. The first reported case of figure 7 concerns the bigger data size case. It can be easily correlated with the curves in figure 6 which show that execution time decreases in 4-10 cores range faster than in 11-15 range. This corresponds to the upper knee of the optimum speedup curve. The 11 processor pool equipped

with 16 kB L1 cache each is the limit beyond which increasing area any further does not produce a proportional performance increase (kill rule). As for the lower knee, the speedup abruptly increases when the amount of data for each processor fits in a 16kB cache. In the range 7-10 mm^2 area increase is worth as the speedup increase is (70-20)/20=2.5 for an area increase of (10-7)/7=0.43. 8-11 is then the range of processor cores that will lead to an "optimal" design in the sense that perfectly exploits the available area.

As for the second case of figure 9 which describes the optimal speedup for the 30x30 data size, a similar remark can be done concerning the lower knee of the curve which occurs, as expected, for a 4x lower cache and, somewhat unexpectedly, for a larger number of cores. The upper knee and so the limit of the kill rule would probably occur for a number of cores larger than 15. Below the lower knee, the optimal cache size is bigger than 4kB, a symptom that miss rate dominates in that area range.

In order to understand and quantify the advantage of the hybrid shared-memory/message passing approach, we redesigned the Jacobi code in two ways: a pure shared memory and a hybrid solution where only synchronization uses message passing primitives while data exchange occurs through the shared memory. We expect that the amount of traffic generated toward the memory as well as the serialization of accesses will degrade the performance in both cases, but it is important to understand how much of the speedup will be due to synchronization and how much to data exchange. Given a 60x60 array, we compared the optimal speedup of Medea reported in figure 7 to the pure shared memory case and observed a 2x improvement of Medea below the lower knee and an increasing gap beyond the knee ranging from 2x at 6 processors 16 kB to more than 5x at 10 processors, same cache size. This behavior confirms expectations. When the traffic generated by the miss rate is relevant in the Medea case, results are still better than the shared memory case in which, even in the absence of miss rate, traffic is always present. The difference, however, is not as dramatic as where memory accesses beyond the L1 cache are nullified in the hybrid case. In the same conditions, we evaluated the speedup when both data and synchronization exchanges occur through message-passing with respect to the case in which messages are sent for synchronization only. In the same range in which the speedup was 2x compared to the pure shared memory case, the speedup is similar (only 2-20% smaller). When miss-rate is negligible, the speedup ranges between 2x and 2.8x instead of 2x-5x. We can thus state that much of the improvement of the full-blown message-passing approach is due to better synchronization which accounts for at least 100·2.8/5=56% of the record 5x improvement that we mentioned above and up to 100% of the 2x cases. As a partial conclusion of this analysis, the hybrid approach - in both its variants, synchronization only as well data plus synchronization case - seems to scale better and to utilize silicon area for additional core instances in a more efficient way compared to a standard shared memory approach. Cache size is a critical parameter: When miss rate becomes relevant, the advantage fades out.

IV. Conclusion and Future work

In this paper, *Medea*, a hybrid shared memory/message passing architecture has been proposed. Measurements performed on a specific algorithm, the Jacobi iterative solver, confirm the work hypotheses concerning the ability of a hybrid approach in reducing synchronization overheads as well as allowing fast on chip exchange of data among the cores. The *Medea* framework has been subsequently used to find optimal solutions for area-constrained designs. Result show that, using the Jacobi algorithm for a given data structure size, it is possible to obtain the best trade-off between number of processors and cache size. Future work will be based on the porting and execution of standard parallel benchmarks, the MPMMU optimization, simulation base enlargement.

References

[1] International Technology Roadmap for Semiconductors Web Site, http://www.itrs.net

[2] Umesh Gajanan Nawathe *et al.*, "An 8-core, 64-thread, 64-bit, power efficient SPARC SoC (Niagara2)", ISSCC 2007, February 2007

[3] M. Tremblay *et al.*, "A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor", ISSCC 2008, February 2008

[4] S. Bell *et al.*, "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect", ISSCC 2008

[5] B. Flachs *et al.*, "A streaming processing unit for a CELL processor", ISSCC 2005, February 2005

[6] L. Seiler *et al.*, "Larrabee: A Many-Core x86 Architecture for Visual Computing", ACM Transactions on Graphics, Vol. 27, No. 3, Article 18, August 2008

[7] L. Benini *et al.*, "Networks on Chips: A new SoC paradigm", IEEE Computer, vol. 35, no. 1, pp 70-78, January 2002

[8] 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

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

[10] J. Heinlein *et al.*, "Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor," Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) 1994, October 1994.

[11] A. Agarwal *et al.*, "The MIT Alewife Machine," Proceedings of the IEEE, March 1999.

[12] C.-C. Kuo *et al.*, "ASCOMA: an adaptive hybrid shared memory architecture," In proceedings of International Conference on Parallel Processing, August 1998

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

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

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

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

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

[18] G.E. Karniadakis *et al.*, "Parallel Scientific Computing in C++ and MPI," Cambridge University Press, 2003.

[19] A. Agarwal *et al.*, "The KILL Rule for Multicore," Design Automation Conference (DAC), 2007. 4-8 June 2007, pp. 750-753.

[20] S. Tota *et al.*, "A Case Study for NoC Based Homogeneous MPSoC Architectures," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, March 2009.