CAP: Communication Aware Programming

Jan Heisswolf[◇], Aurang Zaib[⋆], Andreas Zwinkau[◇], Sebastian Kobbe[◇],
Andreas Weichslgartner[†], Jürgen Teich[†], Jörg Henkel[◇],
Gregor Snelting[◇], Andreas Herkersdorf[⋆], Jürgen Becker[◇]

[◇]Karlsruhe Institute of Technology (KIT), Germany
[⋆]Technische Universität München (TUM), Germany
[†]Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

ABSTRACT

Networks on Chip (NoC) come along with increased complexity from the implementation and management perspective. This leads to higher energy consumption and programming complexity of NoC architectures.
This work introduces communication aware programming to address communication resource management and efficient programming of NoC architectures. A programming interface is introduced to express communication requirements at the language level. These requirements are evaluated by an operating system component, which configures the communication hardware accordingly. The proposed concept enables an intuitive use of NoC features like end-to-end connections and Direct Memory Access (DMA). The presented results show that communication aware programming can improve performance and energy consumption.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming, Parallel programming*

General Terms

Design, Language, Performance

Keywords

communication, network on chip, many-core, X10, invasive

1. INTRODUCTION

Physical limitations prevent designers from further increasing the performance of single cores. Consequently, parallel architectures have emerged as a major choice for increasing computational performance. From the communication perspective, such parallel architectures lead to new challenges. At the architectural level, limitations of bus-based communication need to be overcome. Networks on Chip (NoC) [2] have been presented as a scalable communication infrastructure for many-core architectures. Compared to bus-based interconnects, NoCs offer a better scalability for two major reasons: (1) Distributed communication. (2) Wire length and clock frequency are independent of the number of compute nodes. However, deploying such complex communication systems in modern architectures, comes with several drawbacks: (1) NoCs have a noticeable share in the overall power consumption, as analyzed for Intel's Single-chip Cloud Computer (SCC) [15]. (2) On-chip networks are required to be managed by the Operating System (OS) or applications because of their distributed nature. (3) In contrast to buses, the distance between communicating nodes should be taken into account. It is essential to address the above-mentioned drawbacks during application development which may otherwise lead to semi-optimal performance and higher power consumption. Therefore, efficient mechanisms to manage and utilize NoC-based communication infrastructures are required. However, such strategies may also result in an increased software development complexity. Thus, we introduce communication aware programming as an approach for developing parallel applications for NoC-based architectures. An easy to use constraint system is introduced. It enables to express the communication requirements of the application in an intuitive way. These constraints are evaluated by the operating system during run-time, taking the current utilization of the communication infrastructure into account. The high level constraints expressed by the programmer are broken down into low level control routines for the communication hardware. Afterward, the NoC hardware is configured according to the requirements of the application. A feature-rich scalable NoC is introduced to enable efficient communication. The presented architecture is realized as an FPGA prototype and used for the detailed investigations. For power analysis, ASIC synthesis results are presented.

The rest of this work is organized as follows: Section 2 summarizes related work. The general concept of Communication Aware Programming (CAP) is introduced in section 3. Section 4 introduces the language extensions and the constraints system. In section 5 Compilation, OS support and hardware management is discussed. Section 6 gives an overview on the NoC hardware extensions and their implementation. The benefits of the proposed concept are investigated in section 7 with respect to performance and power consumption. The work is concluded in section 8.

DAC '14, June 01 - 05 2014, San Francisco, CA, USA

http://dx.doi.org/10.1145/2593069.2593103

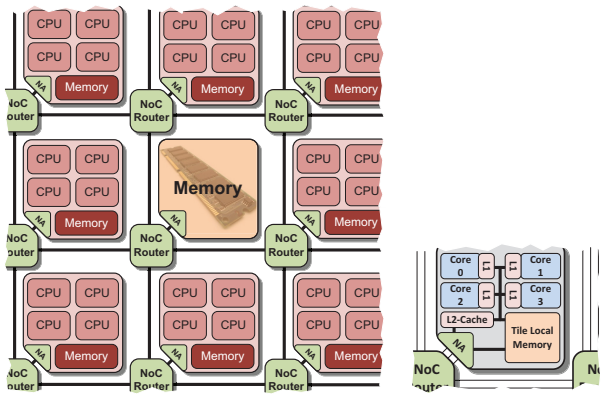
2. RELATED WORK

Automated generation of efficient parallel applications is a huge research challenge. The MAPS framework [6] aims at parallelizing C-application for MPSoCs. However, communication demands are not addressed because MAPS considers a transparent non-scalable bus-based crossbar architectures. SteamIt [23] is a language-based approach for development of streaming applications. In contrast to CAP, StreamIt only addresses streaming applications and does not take into account the characteristics of the underlying communication infrastructure. Another approach for the development of parallel streaming applications is presented in [7]. It targets the IBM cell architecture and its scratchpad memories. None of the previous discussed work addresses the demands of scalable NoCs and their features.
Resource aware programming is addressed by Lorincz et al. [17] for sensor networks. The so-called resource brokers are used to mediate between low-level physical resources and higher-level application demands. Such sensor networks

have a fundamentally different architecture and thus different communication requirements as compared to a general-purpose tiled many-core NoC architecture, addressed by our approach. Invasive computing [22] proposes a novel computing paradigm, which supports resource aware programming from application [21] as well as architecture perspective [14]. Along with these ideas this work focuses on communication resources in the context of resource aware programming and also addresses its realization for a NoC-based architecture.

3. CONCEPT

The concept of communication aware programming addresses Non-Uniform Memory Access (NUMA) architectures, such as Intel's SCC [15] or Tilera's architectures [1]. A schematic representation of a tiled NoC-based NUMA architecture is shown in Figure 1(a). It consists of processing tiles and memory tiles. The memory tiles enable access to off-chip memory (e.g. DDR memory). The internal structure of a processing tile is shown in Figure 1(b). Hardware managed uniform caches with intra-tile coherency are used to hide memory access latencies. An addressable single cycle SRAM-based tile local memory or scratchpad enables computation with a low memory latency for a limited data set size. The Network Adapter (NA) connects the tile internal bus via the L2-cache to the NoC. The NoC enables communication between tiles, access of external memories and peripherals. The memory is arranged as a Partitioned Global Address Space (PGAS). PGAS is realized through address look-ups performed by the network adapter. Each memory within the architecture can be addressed from each node. Therefore, L2-cache misses result in transparent cache-line fetching supported by the NoC. However, the concept is not restricted to the shown mesh topology.



(a) Architecture overview (b) Tile

Figure 1: (a) Tiled architecture with multi-core tiles connected via a mesh-based NoC. (b) Each compute tile has 4 cores, a local memory and caches.

3.1 Spatial data locality

For efficient programming of NoC-based NUMA architectures, the programmer has to be aware of data locality and access latencies. In general, the most frequently used program data should be located as close as possible to the processing element because of the following reasons: (1) The application performance suffers from increased latency when accessing distant data. (2) The utilization of a NoC-based communication infrastructure increases linearly with the distance. (3) The dynamic power consumption for data access is almost proportional to its distance. Caching can solve the spatial data locality problem for small data sets which are used exclusively. However, if a parallel program needs to exchange data between tiles or if the data set is to large for the cache, efficient inter-tile communication is required.
A tile local memory, as shown in Figure 1(b), can be used to explicitly store data close to the processor. However, the limited size of this memory demands its efficient utilization. It could be used in an efficient manner, if its management is handled by the application developer itself. He has the exclusive knowledge about the most recently used data. However, user driven memory handling complicates the application development. Therefore, a tradeoff between efficiency at the language level and the communication awareness is required. An efficient prefetching methodology at the language level is applied to handle this trade-off (see section 4). It enables to cache frequently used data in the tile, reducing the number of tile external memory accesses.

3.2 Hints and constraints

For communication aware programming, the application developer provides his knowledge about the communication of the application to the OS. The OS in turn manages the underlying architecture to fulfill the application requirements by taking into account the current architecture utilization. The hints provided by the application developer can be used by the OS for task mapping and for allocating communication resources. Therefore, the hardware is required to support the resource allocation at the granularity of applications. The definition, run-time evaluation and hardware configuration based on communication constraints are detailed later.

4. PROGRAMING LANGUAGE

X10 [19] is a programming language which brings modern features to the field of scientific computing by addressing parallelization from the start of the application development. This includes well-known advantages like type safety, system modularity, partitioned global address space, generic programming, and integrated concurrency. In addition, X10 also includes promising new features like dependent types and transactional memory (via `atomic` and `when`). Since this feature set is not available in other languages, e.g. C++, we have used X10 as a programming language to realize resource aware applications in our framework. The concurrency and parallelism semantics of X10 are defined in terms of activities, which are lightweight threads. The activities can run in parallel on different processing cores and can not be preempted by the OS.

4.1 Invade, Infect, Retreat & Claims

Invasive computing [21] addresses resource awareness with respect to computation resources. The idea of invasive programming is taken as a basis to realize CAP. A simple invasive program is shown in Figure 2. The concept of allocating, utilizing and releasing resources is known from memory allocation. Invasive computing generalizes the concept to invade, infect and retreat under specific resource constraints.
Constraints for invasion form a hierarchy [25], which are applied to realize complex applications [5]. Communication-specific constraints are an extension to the existing set of constraints. Applications can specify a need for throughput or latency with respect to global shared memory, peer activities, or parent activity, as detailed later.

4.2 Prefetching

For execution of applications like matrix multiplication or image processing, where the data is too big for the tile local memory, blocks of data should be cached instead. Those blocks should be prefetched in parallel to the execution. This can be done efficiently by using a DMA unit (see section 6.2).

```
val ilet = (id:IncarnationID) => {
    do_something(id);
};
claim = Claim.invade(constraints)
claim.infect(ilet)
claim.retreat()
```

Figure 2: The basic idea of invasive programming: The ilet function provides an action to perform in parallel on all allocated processing elements; Invade allocates resources under specific constraints keeping in view with other concurrent applications; Infect uses those resources by letting ilets (basically compute kernels) run; Retreat releases allocated resources.

First, the amount of tile local memory must be specified using the `LocalMemory` constraint. While memory in X10 is managed by a garbage collector, there are explicit `alloc` and `free` methods for tile local memory to provide the application full freedom for cache management. An example is given in Figure 3.

```
val loc = TileLocalMemory.alloc[int](cs);
val offset = id.ordinal * cs;
val future = data.fetch(offset, loc);
... // do something else, while the data is
    copied into tile local memory
val loc2 = future.force();
assert loc == loc2;
... // use the tile local data in `loc`
```

Figure 3: The code example is showing the prefetching in X10. For the programmer, the future concept is used to model the background activity of the DMA transfer. The `force` call synchronizes and waits for the transfer to finish.

4.3 Example

As an example, we have considered a Picture in Picture (PIP) task graph [3] as shown in Figure 4. The communication bandwidth requirements in this example are known. Invasion of one edge and one node of such a graph is shown in Figure 5 for C_0. In this way, arbitrary tree-form task graphs with arbitrary bandwidth requirements can be constructed.

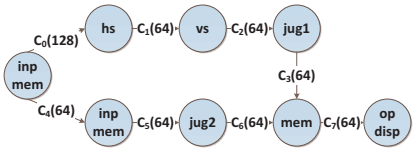


Figure 4: Picture in Picture (PIP) task graph with varying bandwidth requirements.

In order to describe all forms of task-graphs, a more flexible representation in the programming language is needed. A very intuitive way to describe such a graph is to create a `Node` object for every task with constraints concerning the processing element (e.g. `PEType`, `PEQuantity`) and define the communication connections to its successors with the corresponding bandwidth requirements. A task graph of the PIP application is shown in Figure 6 as a constrained X10 representation. We have used PIP and other more complex applications for our evaluations which are presented later in this paper.

```
val claim = Claim.invade(
  new PEQuantity(1) &&
  new Type(PEType.RISC) &&
  new ThroughputToMaster(128)
)
```

Figure 5: The code example shows the invasion of RISC processing element with a throughput of 128 Mb/s to its master, which triggered the invasion.

```
val inp_mem = new Node("inp_mem")
val hs = new Node("vs");
val vs = new Node("vs");
val jug1 = new Node("jug1");
val inp_mem2 = new Node("inp_mem2")
val jug2 = new Node("jug2");
val mem = new Node("mem");
val op_disp = new Node("op_disp");
inp_mem.connect(hs, 128);
hs.connect(vs, 64);
vs.connect(jug1, 64);
jug1.connect(mem, 64);
inp_mem.connect(inp_mem2, 64);
inp_mem2.connect(jug2, 64);
jug2.connect(mem, 64);
mem.connect(op_disp, 64);
```

Figure 6: The code representing the task graph of Figure 4 in X10. It can be converted to a series of invade calls like in Figure 5.

5. RUN-TIME AND OPERATING SYSTEM

The adaptations of the language and the hardware, require support from the intermediate layers as well. We have added an additional backend [4] to the X10 compiler and adapted the run-time system according to language requirements and operating system capabilities. In the context of CAP, the compiler and run-time system pass on the constraints from the application to the OS. The X10 `at` construct is the standard communication primitive and is implemented via OS mechanisms which exploit NoC features.

Inside the operating system, an agent-based resource management is introduced. Each application is represented by an agent. This approach distributes the resource management overhead over the entire system to achieve the scalability required for future many-core architectures, as investigated in [16]. Basic OS functionality is provided by OctoPOS [18]. The application uses the afore-mentioned language features (i.e. constraints, local memory allocation and the invade function call) to inform its agents about its resource requirements. The agent is then responsible for allocating resources, i.e. assembling a hardware claim that fulfills the constraints specified by the application. The allocation of resources and the mapping of tasks to resources themselves are complex operations, which are performed iteratively. However, the agents consider characteristic properties of the applications while allocating resources. The agents pessimistically aim at allocating more resources than necessary for successful mapping of the task graph. This strategy avoids multiple iterations. The required communication bandwidth are considered and can be obtained from the constraints provided by the application developer at the language level. The agents use locally available resource status information to perform the resource allocation. This information consists of the available resources obtained from other agents (e.g. resources that are currently allocated by another application but not used or actually idle resources) and the suitability of resources obtained from monitoring information (e.g. NoC link monitoring) to evaluate the use-

fulness of resources for the invading application, as detailed in [13]. Once enough resources have been allocated, an actual mapping of the task graph (obtained from the constraints) to the allocated resources is performed using the heuristics presented in [20]. In this step, the guaranteed service connections of the NoC (see section 6.1) are set up for the application to be mapped and the detailed communication constraints are realized. If it is not possible to map the tasks to the allocated resources, the agent has to allocate additional resources. Once the task graph has been successfully mapped, the idle cores among the allocated resources are released and the claim is returned to the application. Figure 7 summarizes the OS flow.

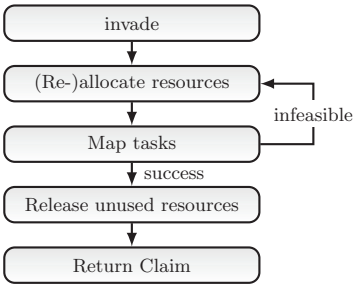


Figure 7: Flowchart representing the iterative allocation and mapping process

The agents use the system messages provided by the NoC to communicate and collect information (see section 6.3). The individual actions of the agents are executed in the interrupt handler which is triggered on arrival of such a system message by the NoC. This mechanism allows the agents to virtually execute in parallel to the application running on the computational resources and reduces the latency introduced by the distributed architecture.

6. COMMUNICATION HARDWARE

A wormhole packet switching meshed network with virtual channels (VC) [8] is used as a basis. Each physical link is shared between a predefined number of virtual channels. Distributed XY routing is used to ensure scalability. A similar NoC is realized in Intel's Single-chip Cloud Computer [15] for inter-tile and main memory communication. It is used later as a reference. However, different features have to be introduced in the NoC to support CAP as detailed now.

6.1 Resource allocation

One major principle of CAP is the exclusive communication resource allocation for an application. This resource allocation can be enabled by end-to-end connections which allocate virtual channels exclusively. These connections enable hard guarantees and are thus called Guaranteed Service (GS) connections. In the past, GS connections have been mainly used to enable hard guarantees for real-time or safety critical applications. In the context of CAP, such end-to-end connections are used to improve performance and reduce communication overhead on behalf of the application programmer. The proposed NoC realizes a fully decentralized and scalable resource allocation scheme as detailed in [11]. All virtual channels can be either used by Best Effort (BE) communication or GS end-to-end connections. Best effort communication is done in form of packets, each containing a header, (several) body and a tail flit. To establish end-to-end connections, a header flit is injected in the start which allocates a virtual channel at each link. After connection setup, body flits are used for communication. A tail flit is used to release the resources allocated by the end-to-end connections. OS triggers an end-to-end connection setup by configuring the memory mapped registers inside the network adapter. If a connection between two tiles exist, it is used transparently by the NA (e.g. access of tile-external memories in case of cache misses). More details about this communication concept are given in [10] and [11].

Figure 8 illustrates a data transmission example. Transmission 1 shows an end-to-end connection setup using a header flit. An established connection between two tiles is shown by transmission 2. Due to the flexible and distributed virtual channel allocation scheme, different VCs are allocated at different links for transmission 2. Transmission 3 represents a BE packet. BE transmissions allocate communication resources (virtual channels) only for a short time duration.

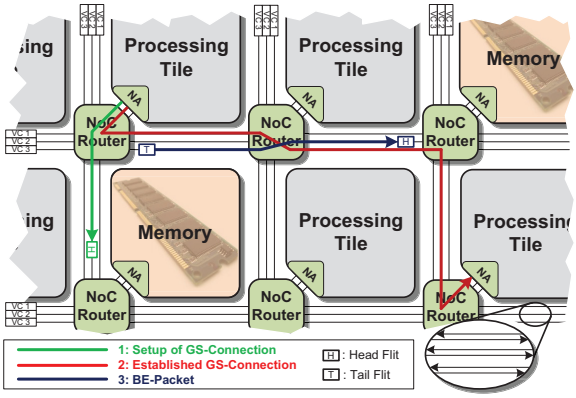


Figure 8: Example of BE- and GS data transmissions within the NoC.

6.2 Data prefetching

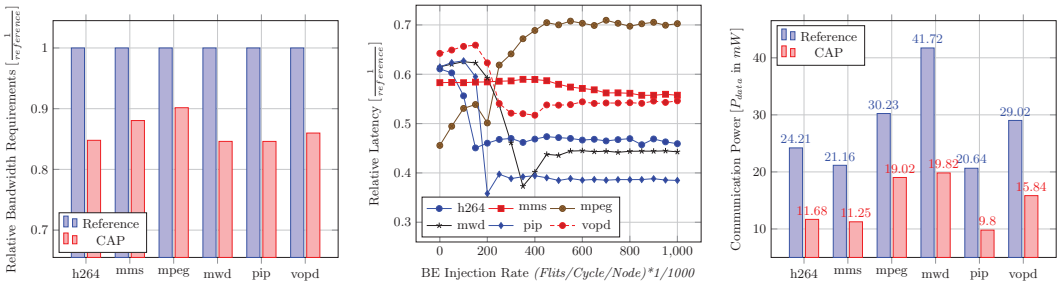
Besides communication resource allocation, the CAP concept exploits spatial data locality controlled at the language level by an advanced data prefetching methodology. It enables explicit control of the fast tile local memory by applications.

To move data between different memories of the architecture, hardware support is incorporated. Therefore, a DMA unit is realized in the network adapter of each tile. It can be configured via memory mapped registers. The DMA support improves the performance by offloading the processing elements from the data transfer overhead. A DMA unit is restricted to only push data from the local memory of the tile to a remote memory location. Pull DMAs are emulated by the OS using push DMA. The benefits of hardware supported DMA transfers are investigated in section 7.2.

6.3 Additional hardware features

Two additional extensions of the NoC hardware are realized to support CAP. System messages are implemented for fast OS-internal communication. A system message is initiated by writing to memory mapped registers of the network adapter. At the receiving tile an interrupt is triggered on arrival of a system message payload. This enables low latency OS communication.

The second extension of the NoC for system software are hardware communication monitors. These monitors observe the communication for a given time period. The current implementation comprises monitors for the link utilization and the virtual channel utilization, as detailed in [13]. The data can be accessed and collected efficiently using NoC hardware support [12]. The OS takes these monitoring information into account during resource allocation and mapping phase.



(a) Communication bandwidth (b) Communication latency (c) Power consumption

Figure 9: Communication bandwidth, latency and power consumption for six parallel multimedia applications with CAP and without (*Reference*).

7. RESULTS

The goal of communication aware programming is to increase the performance and efficiency of parallel applications by improving their communication. Therefore, the impact of communication resource allocation and data prefetching is investigated in the following.

7.1 Simulation

A cycle accurate SystemC model of the NoC, presented in section 6 is used. The instantiated meshed NoC has a size of 10x10 nodes and uses distributed XY routing. The routers have a four stage pipeline. Header flits traverse all of the stages, whereas body and tail flits take only two of the stages. This behavior reflects the real hardware implementation used for our FPGA prototype. Abstracted behavioral models of processing cores and applications are used for traffic generation. The communication constraints, discussed in section 4, and the constraint evaluation and mapping performed by the OS (see section 5) are modeled abstract in the SystemC simulation framework. The used task graphs are constrained according to their communication behavior. The mapping decisions and resource allocation are based on the constraints using nearest neighbor mapping [20]. For the following investigations communication graphs of different multimedia applications are used: Video Object Plan Decoding (VOPD, 12 cores), MPEG4 video decoding (14 cores), Picture-In-Picture (PIP, 8 cores), and Multi-Windows Display (MWD, 14 cores), presented in [3], as well as a H.264 CAVLC encoder (H.264, 16 cores) and a Multimedia System (MMS, 25 cores), provided by NoCTweak [24]. The *reference* used in the following is a BE packet switching NoC with VCs, very similar to the one realized in the Intel Single-chip Cloud Computer [15].

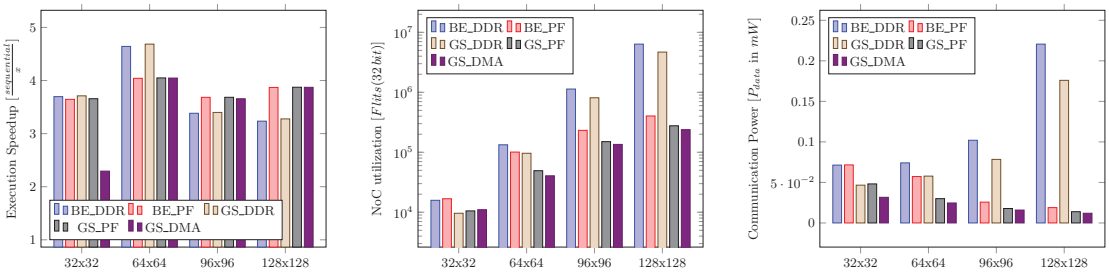
Figure 9(a) shows the bandwidth required for communication while executing the different applications. All results are relative to the *reference*. Compared to this reference, the communication bandwidth can be reduced between 9.8% for the MPEG decoder and 15.4% for the H.264 encoder. This reduction is due to a reduced gross data rate resulting from end-to-end connections. Figure 9(b) shows the communication latency of the applications under different load situations. To generate additional load, uniform random traffic is injected by the nodes of the architecture that are not used by the investigated application. The results show that CAP can reduce the data transmission latency by 29% to 64%. On average delay can be reduced by 47% compared to the reference. The reason for this significant improvement is the pre-allocation of communication resources. This pre-allocation enables low latency communication since routing and VC allocation is performed in advance to data transmission (two router pipeline stages are skipped). The SystemVerilog hardware implementation used for the FPGA prototype is now taken to estimate the power consumption. An ASIC synthesis of the NoC was performed using a *TSMC* 45 nm general purpose standard cell library (*tcbn45gsbwpwc*) with worst case operating conditions. For power estimation, the toggle rates were derived from netlist simulation under application specific NoC load. The power consumption of an idle router (P_{idle}) is 7.94 mW. $P_{data} = P_{total} - P_{idle}$ is used in the following. Figure 9(c) shows the estimated power consumption for data transmission over the NoC while executing the investigated applications. The results show that communication aware programming can reduce the power consumption for communication significantly. For the H.264 encoder, PIP and MWD more than 50% of the power which is directly related to data transmissions (P_{data}) can be saved. This significant reduction of the power consumption has two reasons: (1) Due to the previous resource allocation, enabled by end-to-end connections, data transmission is simplified because routing and virtual channel allocation are only performed once while connection setup. (2) The protocol overhead of end-to-end connections is reduced and thus the gross data rate is decreased.

7.2 Prototype

In addition to the simulation results, an FPGA-prototype of the architecture presented in section 3 was realized. The prototype has four processing tiles with one Leon3 RISC core [9] per tile due to the limited amount of resources available on the used ML605 FPGA board. The tiles memory hierarchy is same as shown in Figure 1(b). Each tile has a tile local memory of 256 kB, 512 byte L1 data- and instruction cache and a 4 kB L2 cache. One of the four tiles has a DDR3 memory attached to its internal bus. The DDR memory is accessible from the other tiles via the NoC.

A parallel version of an integer matrix multiplication was used to investigate the impact of CAP mechanisms on the hardware prototype with respect to execution time of the application, NoC-utilization and NoC power consumption. The results are given in Figure 10. Five different variants of the matrix multiplication were investigated: Two versions use *BE* communication, the other three versions use communication resource allocation by *GS* end-to-end connections. Each communication variant is investigated as *DDR*, where the source matrices are located in the main memory and *PF*, where the required parts of the source matrices are prefetched from the DDR to the tile local memory according to CAP principles. The *GS_DMA* variant performs prefetching by the use of the hardware DMA unit located inside the NA. The use of DMA is only investigated in combination with end-to-end connections due to hardware requirements. The *BE_DDR* variant is used as a reference. It does not use any CAP mechanism and could be realized on other architectures, such as the Intel SCC [15].

Figure 10(a) compares the speedup of the different variants relative to a single core variant. The results show that



(a) Application speedup (b) NoC utilization (c) NoC power consumption

Figure 10: Parallel matrix multiplication executed with different settings on a 4 tile architecture prototype. CAP mechanisms (prefetching and end-to-end connections) are compared against a reference (*BE_DDR*).

prefetching has no benefit with respect to execution time for small matrix sizes due to the fact that all data fit into the L2-cache. A matrix with *64x64* elements even reaches speedups higher than four. The reason is the increased overall cache size if four cores are used instead of one. If the matrix sizes become bigger, prefetching improves performance significantly. For a matrix of *128x128* elements, prefetching introduced by CAP improves performance by 26% compared to the reference (*BE_DDR*). Figure 10(b) shows the NoC utilization caused by executing the five variants of the matrix multiplication. To obtain these numbers, the NoC link monitors (see section 6.3) available on the prototype have been used. As expected, the amount of communication increases with the matrix size. For larger matrix sizes, the amount of flits can be reduced by 26% if resource allocation is used (*GS_DDR*). DMA prefetching (*GS_DMA*) can reduce the amount of communication by up to 96% compared to the reference. The reason is the reduced main memory communication resulting from the use of the tile local memory. Finally, the power consumption which is directly related to data transmissions is analyzed for the ASIC implementation of the NoC, as detailed in section 7.1. Figure 10(c) summarizes the results. The benefit of prefetching with respect to the power consumption increases with the size of the matrix. The *GS_DMA* variant, which applies all of the proposed CAP mechanisms, can reduce P_{data} by up to 95%. If no prefetching is used the allocation of communication resources (GS) can help to reduce the power consumption for communication by up to 20% depending on the matrix size.

8. CONCLUSION

This paper presented a hardware architecture that enables efficient communication for future scalable NoC architectures. The NoC enables end-to-end connections and DMA transfers. These hardware mechanisms are used by the system software and at the language level to facilitate communication aware programming. CAP enables intuitive and efficient parallel programming of NoC architectures.

Cycle accurate NoC simulations using multimedia application models show reductions of up to 15% for the required communication bandwidth, latency reductions of 47% on average, and saving of more than 50% of the data transmission power. An FPGA-prototype executing a parallel matrix multiplication, shows up to 96% less communication and 95% less transmission power due to CAP. A speedup of up to 26% for the matrix multiplication benchmark is another strong motivation for communication aware programming.

Acknowledgment

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center “Invasive Computing” (SFB/TR 89).

9. REFERENCES

[1] S. Bell, B. Edwards, J. Amann, et al. Tile64 - processor: A 64-core soc with mesh interconnect. In *ISSCC*, 2008.
[2] L. Benini and G. D. Micheli. Networks on chips: a new SoC paradigm. *Computer*, 2002.
[3] D. Bertozzi, A. Jalabert, et al. Noc synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE TPDS*, 2005.
[4] M. Braun, S. Buchwald, M. Mohr, et al. An x10 compiler for invasive architectures. Technical Report 9, KIT, 2012.
[5] H.-J. Bungartz, C. Riesinger, et al. Invasive computing in hpc with x10. In *ACM SIGPLAN X10 Workshop*, 2013.
[6] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, et al. MAPS: an integrated framework for MPSoC application parallelization. In *DAC*, 2008.
[7] W. Che, A. Panda, and K. S. Chatha. Compilation of stream programs for multicore processors that incorporate scratchpad memories. In *DATE*, 2010.
[8] W. Dally. Virtual-channel flow control. *Parallel and Distributed Systems, IEEE Transactions on*, 1992.
[9] J. Gaiesler. The leon processor user’s manual, 2001.
[10] J. Heisswolf, R. Konig, and J. Becker. A scalable noc router design providing qos support using weighted round robin scheduling. In *ISPA*, 2012.
[11] J. Heisswolf, R. König, et al. Providing multiple hard latency and throughput guarantees for packet switching networks on chip. *Computers & Electrical Engineering*, 2013.
[12] J. Heisswolf, A. Weichslgartner, A. Zaib, R. Konig, T. Wild, A. Herkersdorf, et al. Hardware supported adaptive data collection for networks on chip. In *IPDPSW*, 2013.
[13] J. Heisswolf, A. Zaib, Weichslgartner, et al. The invasive network on chip - a multi-objective many-core communication infrastructure. In *ARCS*, 2014.
[14] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, et al. Invasive manycore architectures. In *ASP-DAC*, 2012.
[15] J. Howard, S. Dighe, et al. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *Solid-State Circuits, IEEE*, 2011.
[16] S. Kobbe, L. Bauer, D. Lohmann, et al. Distrm: distributed resource management for on-chip many-core systems. In *CODES+ISSS*, 2011.
[17] K. Lorincz, B.-r. Chen, J. Waterman, et al. Resource aware programming in the pixie os. In *SenSys*, 2008.
[18] B. Oechslein et al. OctoPOS: A Parallel Operating System for Invasive Computing. In *SFMA*, 2011.
[19] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. *X10 Language Specification v2.3*, 2012.
[20] A. K. Singh, T. Srikanthan, et al. Communication-aware heuristics for run-time task mapping on noc-based MPSoC platforms. *JSA*, 2010.
[21] J. Teich. Invasive Algorithms and Architectures. *it - Information Technology*, 2008.
[22] J. Teich, J. Henkel, A. Herkersdorf, et al. Invasive Computing: An Overview. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*. Springer, 2011.
[23] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: a language for streaming applications. In *Compiler Construction*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002.
[24] A. T. Tran and B. Baas. Noctweak: a highly parameterizable simulator for early exploration of performance and energy of networks on-chip. Technical report, VCL, University of California, 2012.
[25] A. Zwinkau, S. Buchwald, and G. Snelting. Invadex10 documentation v0.5. Technical Report 7, KIT, 2013.