Truly Scalable Grids of RISC-V Processors with Local Memories

Lars Luchterhandt[1], Vivek Govindasamy[2], Yutong Wang[2], Rainer Dömer[2], Wolfgang Mueller[1], and Christoph Scheytt[1]
[1]Heinz Nixdorf Institute, Paderborn University, Paderborn, Germany
Email: {larluc,wmueller,cscheytt}@hni.upb.de
[2]Center for Embedded and Cyber-physical Systems, University of California, Irvine, CA, USA
Email: {vbgovind,yutongw5,doemer}@uci.edu

Abstract

Massively parallel computer architectures based on identical microprocessor tiles are well known for their high scalability and performance. In this work, we introduce the design and simulation of a scalable on-chip grid of RISC-V processors with local memories in a structured EDA flow. We specify a Grid of Processing Cells (GPC) with two non-trivial software applications and successively generate and validate the hardware models at lower abstraction levels, namely TLM, ISS, RTL, and FPGA. Our comprehensive experimental evaluation of a physical implementation on an FPGA shows the true scalability of the GPC architecture. While existing works have indicated performance benefits for GPCs at high abstraction levels, our results confirm their highly reduced interconnect contention for the first time on fully constructed hardware with clock-cycle accuracy at RTL and on FPGAs.

1 Introduction

The ever-increasing demand for higher chip performance drives the growing need for highly parallel platforms that are truly scalable to meet the complexities of current and future applications. Aware of many prior projects using array-structured architectures starting with the era of Transputers [8, 15], a Grid of Processing Cells (GPC) [9] has recently been proposed that aims at eliminating the bottleneck of shared memory by using cells of processors with local memories. Here, every local on-chip memory is only accessible by its local processor and four neighbors (Fig. 1).

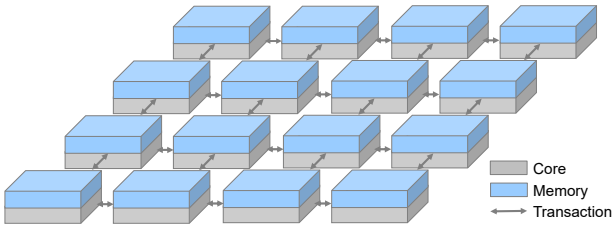


Figure 1 Grid of Processing Cells (GPC) with processor-memory pairs connected in North, East, South, and West directions

In this work, our primary focus is the evaluation of the local memory interconnect in GPCs with significant software workloads. We generate and simulate GPC models with increasing levels of accuracy to evaluate the inter-processor communication and, in particular, contention.

Problem Statement. First works at untimed TLM2 level [12, 25] report only weak indication of scalability when comparing the GPC to the traditional SMP (Symmetric Multiprocessor) architecture. Slightly stronger results [2, 3] support the scalability at loosely-timed TLM2 when taking communication contention into account. [13] analyzes the GPC with instruction-set simulation, confirming prior results at the ISS level. However, for a solid conclusion on true scalability, the GPC hardware must be implemented with sufficient detail at clock-cycle accurate levels at RTL and on FPGAs. [19] introduced an early GPC model at RTL with FPGA and chip implementations with simple handshake communication for local memories. [19] advanced the communication schemes by shared memory and CSR synchronization. Later, [18] introduced a detailed study for the refinement of different GPC variants and applications from first functional to final RTL and FPGA implementations.

In contrast, we present now in this work a fully implemented buffered communication with interrupts and scalable interconnects for non-trivial software applications, which we evaluate over various grid and cell configurations with respect to communication tradeoffs.

We choose 32-bit RISC-V Rocket cores as processors with application-specific memory sizes and FPU configurations in each cell, so that our results become comparable to other studies to some extent. We fully design the memory communication by introducing a local interrupt scheme between neighbors and equipping each cell with a 32-bit FPU to broaden the application. For RTL generation and FPGA implementation, we apply the Chipyard framework [1] and use customized generators implemented in the hardware construction language Chisel [5] for scalable grid, buffer, and memory sizes with individual processor configurations.

Our work provides two main contributions. First, we design an advanced local communication scheme with interrupt-based synchronization and observe the key metric of contention in comparison with polling. Second, we

provide an extensive performance analysis of GPC scaling using two freely configurable software applications which shows for the first time in RTL and on FPGA that communication contention remains flat when the grid is scaled in size, thus, providing true scalability of the GPC. Our analysis follows a gapless codesign flow where GPC hardware and non-trivial software workloads are successively configured, constructed, and synthesized from functional specification down to FPGA implementation along the toolflow which we introduced in [18].

2 Related Work

With the introduction of multi-core and multi-processor architectures, Flynn classified computer architectures along different instruction and data streams: single/multiple instructions and single/multiple data [11]. In contrast to traditional single processor systems with single instruction single data (SISD) streams, modern architectures operate multiple instruction and/or multiple data streams in parallel, termed SIMD, MISD, or MIMD.

The idea of configurable processor grids for massively parallel processing stems from the principles of transputers, introduced in the 80s to overcome the Von-Neuman bottleneck [20]. A transputer is a single 32-bit RISC microprocessor with local memory and four serial message-passing IOs, establishing communication links to adjacent processors [8]. They were applied as building blocks of massively parallel supercomputers with up to several hundred nodes [15]. Transputers rely on the principles of a *nothing-shared* architecture with the advantage of high scalability and identical cells. Similar, the Grid of Processing Cells (GPC) introduced in [9] follows a neighborhood shared strategy. GPC hardware studies with partial synthesis flows and configurations for FPGA and ASIC were undertaken in [19].

We can find many other examples of array-based many-core architectures, like the Raw Processor, which features tiles with a 32-bit MIPS-like processor with local I- and D-memory, a configurable on-chip network, and up to 64 tiles per chip [22]. Tile Processors [27] come with identical VLIW processor tiles with local L1 and L2 caches. The Tile64 was composed of 64 tiles on the chip, which were connected through configurable switches [7]. The Intel Teraflops research chip was introduced with 8x10 VLIW cores connected by a 2D mesh network-on-chip (NoC) [23]. The KiloCore processor array is based on a hybrid memory structure [6], which combines shared and distributed memories. Its 992 independent processors come with local memories. Instructions and data may come from the local memory or are fetched from 12 64 kB shared SRAM modules. Communication is performed via a switching network with small package routers.

These architectures were introduced for improved bandwidth to local memory. Unfortunately, their work neither revealed any details of the design process nor do they give information about the configurability of individual cells.

Our work revisits the principles of the GPC architecture in [9] and introduces fully implemented RISC-V based cells that can be individually configured with respect to processor type, local memory size, message type, and communication scheme during design time for the execution of non-trivial software applications. Our design flow combines classical SystemC TLM-2.0 with the Chipyard framework and complements them with additional configuration and generator scripts, which seamlessly supports multiple abstraction levels from multi-threaded C++ functional level to implementations on FPGA.

3 Design Flow

Our system design flow from functional specification down to FPGA implementation encompasses five executable models at different abstraction levels: functional multi-threaded C++, SystemC TLM-2.0, SystemC ISS, SystemVerilog RTL, and an FPGA prototype implementation.

3.1 Functional Specification in C++

The design flow starts with a pure functional model specified in multi-threaded C++. A number of tasks (i.e. `std::thread`) perform the application's functions and communicate via standard communication (e.g. `std::queue`) and synchronization methods (e.g. `std::condition_variable`). Model compilation and execution allow validation of functionality, but there is no notion of timing nor structure. These properties are added and refined step by step in derived and generated models following a top-down approach. Software and hardware functions are combined in one model and are subject to partitioning in later steps.

3.2 Modeling and Simulation in SystemC TLM-2.0

After the tasks have been mapped onto a suitably sized grid [26], we generate a first GPC model in SystemC TLM-2.0, which accurately reflects the tasks each cell performs and explicitly shows inter-cell transactions via memories shared between neighbors. This model is accurate in memory transactions[1], but only loosely timed. Interconnect contention of data transfers can be accurately monitored. Hardware and software functions can be clearly identified. Both are compiled for execution on a simulation host. As the instruction set of the host typically differs from the target architecture, the simulation arrives at a lower accuracy.

3.3 Instruction Set Simulation (ISS)

For an increased accuracy with the final instruction set architecture of the software, we next generate an instruction set simulation model in SystemC. At this step in the design flow, we move from host-compiled software applications to software that is cross-compiled for the final target architecture, 32-bit RISC-V in our case. The cross-compiled and linked ELF (Executable and Linking Format) files are loaded into the on-chip memories at the start of the simulation. Using an interpreted ISS [14], the RISC-V core

[1] TLM-2.0 transactions include blocking transport and DMI.

modules retrieve both instructions and data from their local memory through SystemC TLM-2.0 `b_transport` function calls. Every instruction is then decoded and executed with a corresponding delay.
Note that the number of instructions is accurate in this model, whereas the number of clock cycles and the delay of their execution are only an estimated approximation.

3.4 Hardware Construction, Simulation, and Synthesis

To derive a first cycle-accurate model with precise timing, we implement a GPC RTL generator by extending the Rocket chip generator [4] of the Chipyard framework [1]. This GPC generator constructs additional DTIM (Data Tightly-Integrated Memory) adapters attached to the local scratchpads of each tile and connects them via TileLink to the corresponding adjacent tiles, enabling direct inter-tile communication [21]. At this step, we replace the abstract instruction decoding and execution cycle with the constructed hardware of the individually configured processor and memory.

3.4.1 Hardware Construction

The GPC generator written in the hardware construction language Chisel [5] generates the desired grids and emits FIRRTL [17] based on the desired parameters for the individual GPC, e.g., grid size, memory size, or FPU support. The FIRRTL circuit compiler compiles the intermediate representation and generates a SystemVerilog model for the chosen Chipyard design flows, namely RTL simulation and FPGA prototyping.

3.4.2 RTL Simulation

For RTL simulation of the generated GPC models, we use the open-source SystemVerilog simulator Verilator [24] which compiles the SystemVerilog model into a multi-threaded C++ model from which the simulator can be built. The application software for the GPC hardware model is cross-compiled for the RISC-V target and loaded into the scratchpad memories at the beginning of the simulation via the Front-End Server (FESVR). We enhanced the FESVR to communicate with all processing cells during the simulation and to load ELF files into their memories. RTL simulation time can be significantly decreased when we build and run many RTL simulators of different designs in parallel on a High-Performance Computing (HPC) cluster. We thus extended our flow to automate job submission for the HPC workload manager Slurm [10].

3.4.3 FPGA Synthesis

To arrive at a first hardware implementation with higher performance, we synthesize the RTL design for the AMD VCU108 FPGA using AMD Vivado. After synthesis, the bitstream with the GPC hardware is loaded onto the FPGA. Thereafter, the cross-compiled application software is loaded into the scratchpad memories via JTAG and executes on the FPGA at the given frequency. Hence, the FPGA prototype combines both the performance from higher abstraction levels and the low-level accuracy at RTL.

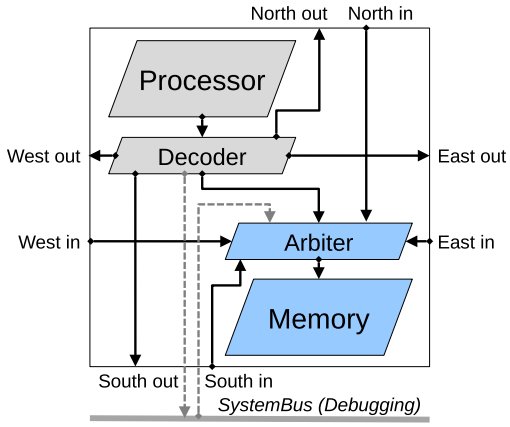


Figure 2 Cell with processor and local memory

4 GPC Architecture

The GPC is a fully scalable multiprocessing platform where processor cores are paired with local on-chip memories. Those pairs are placed as cells on the chip in a two-dimensional $N \times M$ grid. Each core has access to its own local memory and to the memories of the four neighboring cells around it. The address space of the memories is organized based on the dimensions N and M of the grid [25]. We use a linear memory map where the base address of a cell in row r column c is given as $(r \cdot M + c) \cdot S$ for the specified memory size S. Note that, while the address space in our configuration is global, individual processors can only reach addresses in their local neighborhood.

4.1 Cell with Processor and Local Memory

A GPC cell is the basic functional unit that includes every component necessary to execute a software program. As depicted in Fig. 2, each cell consists of a Processor and local Memory, plus local interconnects formed by essentially a Decoder and an Arbiter. Based on the requested memory address, the decoder directs the processor's load and store instructions to the local memory, or a neighbor's memory in the North, East, South, or West. In front of each memory, an arbiter multiplexes the access requests, granting one request at a time.
The four components are essentially the same at all abstraction levels. Their internal composition, however, is refined with more details level by level. For instance, ISS and RTL models feature detailed controllers for interrupt-based communication.
Note that we also equip our RTL models with a SystemBus that mainly serves debugging purposes. In our experiments, however, we also use this bus to compare the GPC's performance against a classic setup where cores communicate via a shared bus system (see Section 5).

4.2 Inter-cell Communication

Safe and efficient communication and synchronization between neighboring cells require a well-coordinated interac-

```
template <class T, unsigned S> void Pop(T &d)
{
  while(queue.Sent - queue.Rcvd == 0)
  {
#ifdef USE_INTERRUPT
    asm volatile ("WFI");
#else // USE_POLLING
    asm volatile ("NOP");
#endif
  }
  d = queue.data[queue.Rcvd%S];
  queue.Rcvd++;
#ifdef USE_INTERRUPT
  queue.interrupt = 1;
#endif
}
```

Listing 1 Simplified C++ implementation of the message queue Pop function

tion between hardware and software. While the hardware interconnects introduced above provide the required infrastructure for signaling and exchanging data, the software has to make efficient use of it. Here, we have implemented a software API that allows message passing through queue-based communication. Each core can host queues in its local memory, which can be configured in terms of data type T and queue size S. Neighbor cores can remotely connect to a host's queues and perform Push and Pop operations to exchange data between them.

Listing 1 illustrates our implementation of the Pop function. The core waits until data is available, retrieves the data from the shared circular buffer, and finally signals to the sender that data has been received. We support interrupt and polling-based communication. While polling repeatedly checks the queue's filling state, followed by a short wait (NOP), the interrupt scheme puts the receiving core to sleep (wait-for-interrupt, WFI) until new data is available. Here, senders and receivers synchronize by raising and awaiting dedicated interrupts implemented in hardware between neighboring cells.

5 Experimental Results

We have designed and implemented model generators for the five abstraction levels, combining the classical SystemC design flow with Chipyard. The following section gives an overview of the two different software application benchmarks before we introduce our hardware and simulation setup and present our evaluation.

5.1 Software Setup

For empirical benchmark evaluation of the different GPC configurations, we implemented two scalable software applications: TokenX and ParticleSim. TokenX is a simple proof-of-concept benchmark for communication without significant computational load with integer operations. The software starts with an initial number of tokens (payload) in each cell and continuously exchanges tokens with neighboring cells in a repeatable, pseudo-random fashion. TokenX is configurable and deterministic, and can be used for systematic validation of data transfers on a GPC.

On the other hand, ParticleSim simulates the physical movement of particles (payload) in a 2D space with configurable gravity, attracting, and repelling forces. Notably, ParticleSim has a high computational load of floating-point arithmetic. For parallel execution, the space is laid out as 2D tiles, each assigned to a corresponding cell in the grid. Each cell computes the forces and movements of the particles in its tile, also taking into account the forces of the particles in neighboring tiles. When particles cross tile boundaries, they are handed over to the neighboring cell.

In our particular configuration, ParticleSim starts with an evenly distributed payload where all cells carry the same number of particles. In other words, there is an equal computational load for all cells at the beginning. We also apply an initial velocity for all particles with a trajectory towards the bottom right. As a result, particles fall down and concentrate in the bottom right tiles, which gradually shifts the computational load to a few cells in the South-East of the grid, until the particles repel each other and also bounce off the walls, eventually resulting in a somewhat equal payload for all cells for the remainder of the simulation.

Both software applications are structured in a configurable main loop that repeats (1) computation and (2) communication with neighbors. Higher software workloads can be easily configured with more iterations, which results in longer simulation times. As such, the simulator runtime can be adjusted by the assigned workload. We use this in our experiments at lower abstraction levels to keep the runtimes acceptable.

Both benchmarks are also configurable in their communication scheme. Here, the synchronization with neighbor cells is based on either interrupts or polling.

5.2 Simulation & Hardware Setup

For experiments on the functional, TLM-2.0, and ISS model, we use C++ and SystemC version 2.3.3 with a g++ compiler. The simulation host for those levels is a Linux workstation with an Intel Xeon E-2388G CPU (16 cores at 3.2 GHz). The ISS model is based on the SystemC-based RISC-V VP from [14], which is also compiled and executed on the workstation, where the application software is cross-compiled with g++rv32 for the RISC-V platform. Our experiments at RTL and on FPGA are fully au-

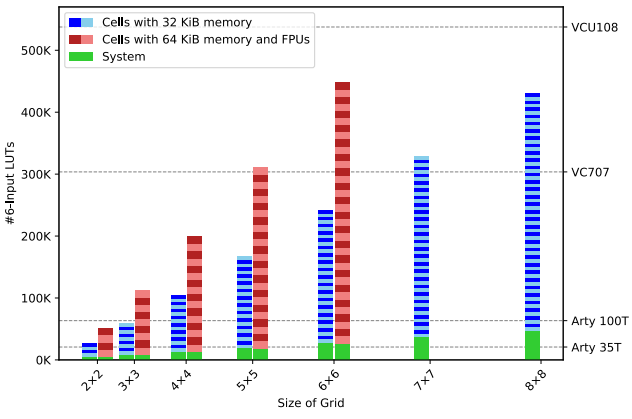


Figure 3 #LUTs required for GPCs synthesized for the VCU108 FPGA Evaluation Board

Appl.	Model	Software Workload	6×6 GPC				8×8 GPC			
			Runtime	Instructions	Clock Cycles	CPI	Runtime	Instructions	Clock Cycles	CPI
TokenX	Functional	100%	12.62s	-	-	-	22.46s	-	-	-
	TLM-2.0	100%	1:23m	-	-	-	2:51m	-	-	-
	ISS	1%	55.86s	2.68M	-	-	1:56m	2.71M	-	-
	RTL	0.01%	10:34m	25.0K	74.6K	2.98	33:59m	25.1K	74.7K	2.97
	FPGA	100%	7.43s	252.3M	743.2M	2.95	7.73s	258.2M	746.3M	2.89
ParticleSim	Functional	100%	28.10s	-	-	-	30.00s	-	-	-
	TLM-2.0	100%	3:46m	-	-	-	4:00m	-	-	-
	ISS	0.5%	23:40h	3.73B	-	-	24:37h	1.86B	-	-
	RTL	0.1%	69:21h	470.9M	1.31B	2.78	49:35h	217.9M	668.2M	3.07
	FPGA	100%	1:23h	134.2B	499.4B	3.72	Design exceeds #LUTs on VCU108 FPGA 761.5K LUTs required but only 537.6K available			

Table 1 Experimental results for 6×6 and 8×8 GPC targets

tomated with Chipyard 1.9.1 and our extensions in Python on an HPC (High-Performance Computing) cluster. As such, we use Verilator for RTL simulation, which was installed and executed on our HPC cluster equipped with 1126 AMD EPYC CPU nodes. Each normal node comes with 256 GiB RAM and two AMD Milan 7763 CPUs (64 cores at 2.45 GHz). More complex designs are executed on large nodes with 2 TiB RAM and AMD Milan 7713 CPUs. Our architectural studies with different grid configurations from 2×2 to 16×16 on different FPGAs are shown in Fig. 3. The figure compares the utilization of different grid configurations on different FPGA boards: VCU108, VC707, and Arty7 35T/100T. The comparison shows RISC-V cell configurations with (i) 32 KiB of memory without FPU for the TokenX application and (ii) 64 KiB of memory with FPU, which are required for the computation-intensive ParticleSim application. The figure indicates that the VCU108 board limits the grid size to 8×8 for cells with 32 KiB local memory and to 6×6 when cells are synthesized with FPU and a larger memory.

5.3 Simulation Results

We simulated our GPC hardware with grid sizes 6×6 and 8×8 at five different abstraction levels: functional multi-threaded C++, SystemC TLM-2.0, SystemC ISS, SystemVerilog RTL, and FPGA. Table 1 shows the results for the different configurations and abstraction levels with two software applications TokenX and ParticleSim. The table quantifies the cost for conducting benchmarks in terms of runtime, clock cycles, and CPI (Cycles Per Instruction) ratios. The latter numbers give details of executed instructions and clock cycles for the applied software workloads.

In general, the table shows the increasing runtimes along the different abstraction levels in comparison with its FPGA implementation. Although TLM-2.0 has some overhead for the explicit modeling of memory transactions, the runtimes are within the same order of magnitude as the functional model. As expected, the table shows a significant increase in runtime at ISS and RTL. This is basically due to the move from host-compiled to cross-compiled software for the RISC-V ISA, with an increase by two orders of magnitude for TokenX and by four to five orders of magnitude for ParticleSim. Additional simulation costs are introduced at RTL when the software is executed on the fully constructed RTL hardware with an increase of two to three orders of magnitude.

To avoid simulation times over more than 2 days, we took a pragmatic approach and configured both software applications at ISS and RTL with a reduced number of iterations. We decreased the software workload for the RTL simulation by a factor of 100 for TokenX, and by 200 for ParticleSim, respectively. At RTL, we can see a big difference between the communication-intensive TokenX and the computation-intensive ParticleSim application.

The RTL models were finally synthesized for an AMD VCU108 FPGA with a frequency of 100 MHz. As the ParticleSim exceeds the 32 KiB of memory and requires floating-point operations, we add FPUs and double the size of the local memories. Therefore, the 8×8 grid exceeds the limits of our FPGA. The numbers show that the FPGA achieves a similar order of runtime as our initial functional implementation for TokenX. However, this only holds up for the light computational load of TokenX. With the heavy floating-point workload of ParticleSim, the FPGA runtime is two to three orders of magnitude slower than the functional model running on a workstation CPU.

5.4 Contention Analysis

To demonstrate the performance benefits of GPCs over many-core system-bus architectures, we generate and compare three different hardware configurations at RTL and on FPGA:

(1) instructions and data of a cell are fetched from the cell's local memory with local inter-cell access to the memories of its four surrounding neighbors (local instructions/local data) as introduced in Section 4.

(2) instructions and data of a cell are fetched via a shared system bus (global instructions/global data).

(3) instructions are fetched from the cell's local memory, data are fetched via the system bus (local instructions/-global data).

For cycle-accurate performance evaluation of the three

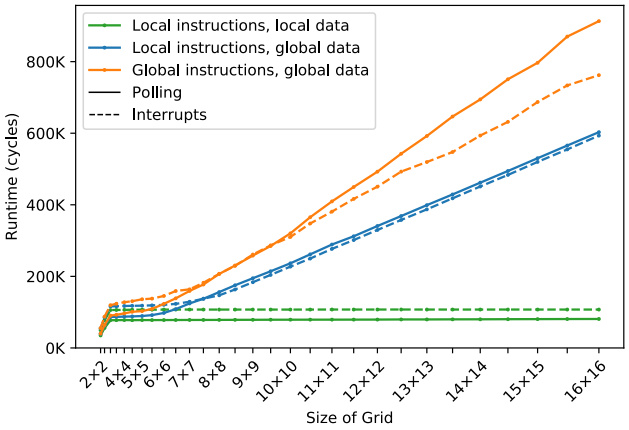


Figure 4 TokenX runtime of RTL simulation

variants, we start with RTL simulation of TokenX for grid sizes from 2×2 to 16×16 as summarized in Fig. 4. The figure compares the runtime for the three hardware configurations with local vs. global instruction and data access. The dashed and solid lines additionally show the difference between the two message-passing schemes (interrupt and polling) for each of the three configurations. While configurations 2 and 3 with full and partial system-bus access are clearly affected by contention, we can observe lower constant values for the GPC architecture (configuration 1) for both synchronization schemes. On the shared-bus variant, the overhead in runtime caused by contention increases linearly with the number of cells in the design. As expected, the global instructions variant has more contention than the local instructions variant. We find polling is more efficient than interrupts for the TokenX benchmark running on a GPC, as all cells are running the same payload per iteration, synchronized by the token exchanges. Note here, that interrupts impose an instruction overhead for proper raising and masking of interrupts, which explains the additional cycles on the GPC configuration. When dealing with increasing contention on a shared bus, interrupts avoid additional bus traffic for polling, and thus, the management overhead for interrupts pays off again.

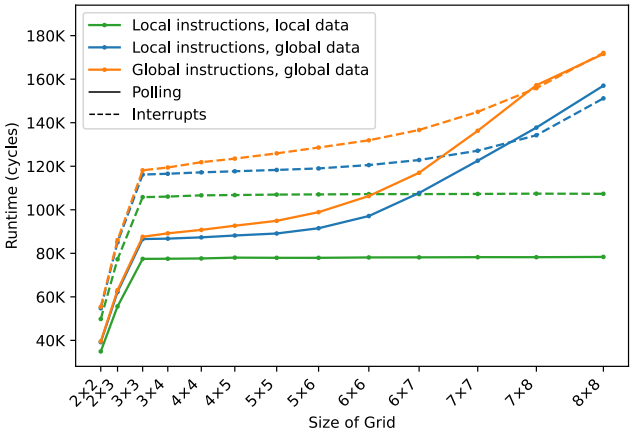


Figure 5 TokenX runtime on FPGA

Fig. 5 shows corresponding results for TokenX on FPGA. The grid size is limited to 8×8 by our FPGA. Compared with the RTL simulation, we can see that the results are accurate apart from small deviations caused by minor differences in the design (hardware UART) and ELF files (no syscalls on FPGA). Once there are cells surrounded by four neighbors in the grid starting from size 3×3, the GPC remains running with a constant number of cycles, whereas the two global data variants run into linear contention. Under contention, interrupts outperform polling starting at a grid size of 7×8. With an increasing number of cells, more bus traffic is caused, which further increases the contention. Once the bus exceeds a certain load, interrupts are beneficial again, as the management overhead of interrupt handling becomes less expensive than causing more bus traffic by polling.

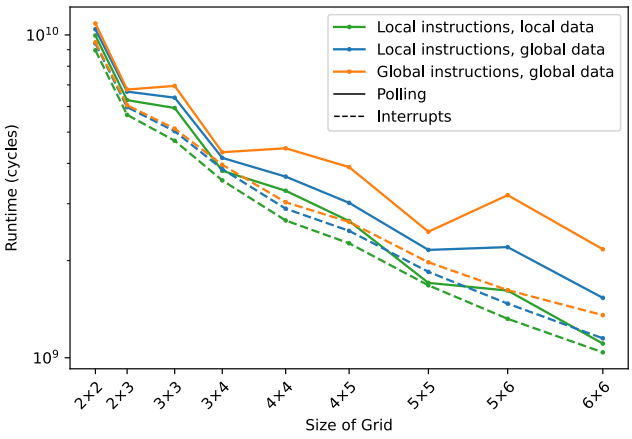


Figure 6 ParticleSim runtime on FPGA

Fig. 6 shows the runtime of the ParticleSim on FPGA. As the computational load per cell decreases inversely with the square of the total number of cells, we use a logarithmic scale to represent the runtime. Here, interrupts perform significantly better than polling, which can be explained by the different payloads among the cells. Unlike TokenX with a constant computational load per cell, the computational load of ParticleSim varies per cell. A cell managing fewer particles than its neighboring cells has to wait for the neighbors to finish computing. While waiting, continuous polling causes significant traffic and hurts the overall performance. This effect can even be observed on the GPC with local memories and interconnects.

For all three configurations, the interrupt communication scheme outperforms polling noticeably, resulting in a shorter runtime. Across all grid sizes, the GPC achieves the lowest runtime compared to the two global data configurations. This demonstrates the efficiency of local inter-cell communication over a shared bus. We can also see a trade-off between computation and communication costs. While the computational load per cell decreases on larger grids, there is an increase in communication required between the cells. This results in a divergence in runtime between the three variants when increasing the grid size.

Finally, we directly evaluate the contention by measuring the average memory access time for data requests using our additional hardware performance counters. Integrated in each cell, they count the number of data requests and the number of cycles the cell has to wait for a response. For easy access, the two counter registers are memory-mapped, making them readable by their cells. The results

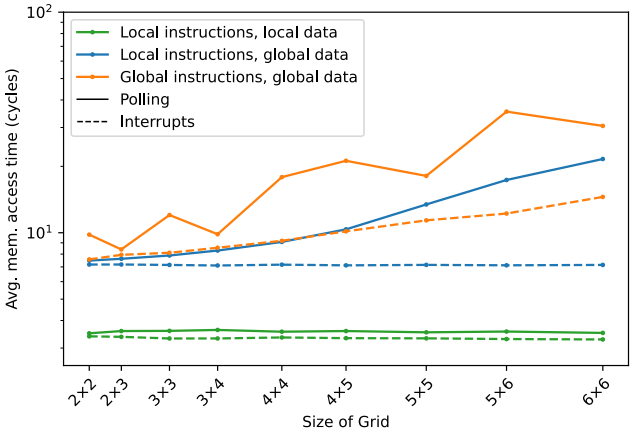


Figure 7 Average access time to neighbor memories for ParticleSim on FPGA

in Fig. 7 again confirm no contention on the GPC with interrupts. On average, the memory access time on the GPC constantly remains between three and four clock cycles for all grid sizes. Both shared bus configurations start with a minimum average memory access time of seven clock cycles with increasing grid size. The full system bus configuration with polling reaches an average of more than 30 clock cycles on the 6×6 grid. When comparing to Fig. 6, we observe a correlation between runtime and contention, especially on the shared bus models. It is worth mentioning here that the blue dashed line in Fig. 7 appears to be constant too. This indicates good contention handling with interrupts and matches the almost parallel lines in Fig. 6. However, the memory access time on the local instructions global data model with interrupts is still three to four cycles higher on average for a data request compared to the GPC.

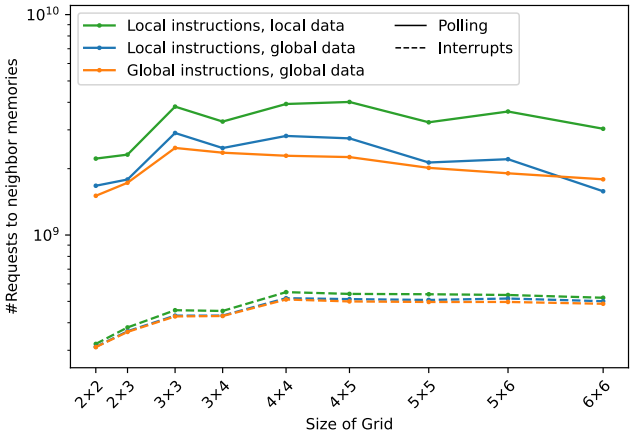


Figure 8 Total #accesses to neighbor memories for ParticleSim on FPGA

To further investigate the negative impact of polling, we take a closer look at the total number of memory accesses for each configuration, shown in Fig. 8. While the number of requests is almost identical among all configurations with interrupts enabled, continuous polling significantly increases the number of requests. The GPC handles between 5.8 and 8.3 times the number of requests when using polling. On the shared bus configurations, the global instructions variant only achieves 3.7 to 5.8 times the number of requests, with slightly more requests on the local instructions variant (3.2 to 6.7). Even given this high amount of requests handled by the GPC, there is almost no increase in the average response time per request in Fig. 7. In contrast to the GPC, the contention on the shared bus configurations causes an additional delay of at least 4 cycles up to 32 cycles on average per request.

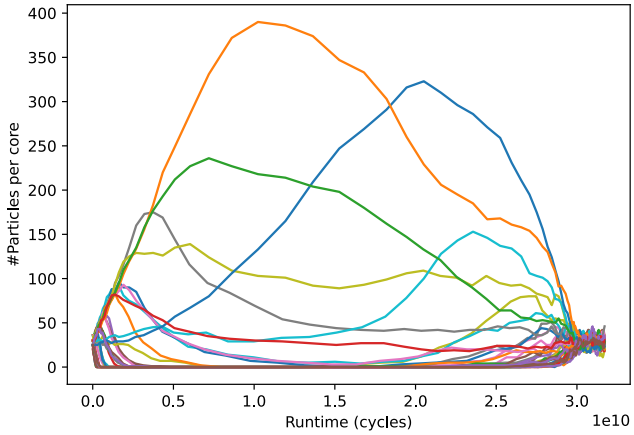


Figure 9 Particles per core over runtime

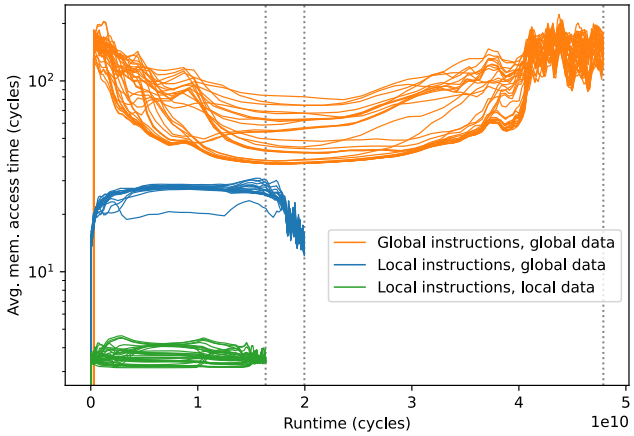


Figure 10 Average access time to neighbor memories over runtime

5.5 Payload Analysis

As ParticleSim has varying computational load per cell, we conduct a detailed cell-wise payload and memory access timing analysis on a 6×6 GPC on FPGA for polling. The payload per cell, measured as the number of particles contained in a cell, is plotted over the runtime in Fig. 9. In each sampled time frame, a total of 1K particles is distributed among the 36 cells based on the physical position of the particles. The plot nicely shows the shift in payload among the cells during runtime. While all cells start with equal payload, almost 40% of the particles are handed over to a single cell after 10B clock cycles. After 30B clock cycles, the particles end up similarly distributed again with a comparable payload per cell. The cost of communication can be derived from the gradient of the payload. A decline in the payload of a cell is caused by handing over particles to another cell. As a result, the payload of the receiving cell increases. The more particles are transferred, the greater the communication required. To measure this impact on

ISBN: 978-3-8007-6515-7

@ VDE VERLAG GMBH · Berlin · Offenbach

the contention, we run the same experiment on the two shared bus configurations for the 36 cells and compare the average memory access time over the runtime in Fig. 10. As expected, the GPC provides the shortest runtime with 16B clock cycles. The shared bus configuration with local instructions takes 22% more runtime, followed by an increase of 140% in runtime for the shared instruction variant. While the GPC achieves a constant memory access time of three to five clock cycles, both shared bus configurations show a significant increase. The pattern of handing most of the payload over to a few cells, followed by a distribution afterwards, can also be seen in the memory access time. With a growing number of particle handovers, the memory access time increases to almost 31 cycles on the local instruction variant. Once the payload is distributed again, the access time decreases, remaining between 12 and 25 cycles. An inverse behavior can be observed on the global instruction variant. Here, the access time is between 36 and 220 cycles while the payload is handled by a few cells. With a distributed payload, the access time rises, ranging from 90 to 220 cycles. This can be explained by considering the number of instructions fetched by the cells. A cell with a smaller payload spends more time waiting for its neighbors and consequently fetches fewer instructions. A busy cell, on the other hand, fetches more instructions during computation, resulting in an increased memory access time.

In summary, our gapless design flow has resulted in a fully constructed GPC model on FPGA that is both fast and accurate, and our comprehensive experimental evaluation with configurable software benchmarks has confirmed the true scalability of the GPC platform and the entire design flow.

In contrast to existing work [25, 13] which had found only high-level indications of reduced memory access contention and improved scalability of the GPC compared to traditional SMP (Symmetric Multiprocessor) architectures, our fully implemented and synthesized GPC models on FPGA enable precise performance analysis with clock-cycle accuracy. We have shown for the first time that the GPC is truly scalable at RTL, as clearly seen in the flat green lines in Fig. 4 and Fig. 5.

6 Conclusion

We presented and evaluated a truly scalable Grid of Processing Cells (GPC) architecture based on fully implemented and configurable 32-bit RISC-V based cells. We applied a highly automated design flow that seamlessly combines SystemC [16] with the Chipyard framework [1]. Our evaluation quantifies the execution speeds at five different abstraction levels over different simulation platforms by two non-trivial software applications and confirms the memory bottleneck on shared memory architectures. Future work will further investigate software applications and complementary Chipyard-supported flow automation, including FPGA-accelerated RTL simulation (FireSim) and chip design (Hammer).

Acknowledgments

This work is partially funded by the German Bundesministerium für Forschung, Technologie und Raumfahrt (BMFTR) through the Scale4Edge project (16ME0133). We gratefully acknowledge the computing time and resources provided on the Noctua 2 HPC Cluster at the NHR Center PC2.

References

[1] Alon Amid et al. “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs”. In: *IEEE Micro* 40.4 (2020), pp. 10–21. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996616.

[2] Emad M. Arasteh and Rainer Dömer. “Fast Loosely-Timed Deep Neural Network Models with Accurate Memory Contention”. In: *ACM Transactions on Embedded Computing Systems* 23.5 (Aug. 2024). ISSN: 1539-9087. DOI: 10.1145/3607548. URL: https://doi.org/10.1145/3607548.

[3] Emad M. Arasteh, Vivek Govindasamy, and Rainer Dömer. “BusyMap, an Efficient Data Structure to Observe Interconnect Contention in SystemC TLM-2.0”. In: *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*. 2024, pp. 1–6. DOI: 10.23919/DATE58400.2024.10546686.

[4] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[5] Jonathan Bachrach et al. “Chisel: Constructing hardware in a Scala embedded language”. In: *DAC Design Automation Conference 2012*. 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[6] Brent Bohnenstiehl et al. “KiloCore: A 32-nm 1000-Processor Computational Array”. In: *IEEE Journal of Solid-State Circuits* 52.4 (2017), pp. 891–902. DOI: 10.1109/JSSC.2016.2638459.

[7] Tilera Corporation. “The Tile Processor™ architecture: Embedded multicore for networking and digital multimedia”. In: *2007 IEEE Hot Chips 19 Symposium (HCS)*. 2007, pp. 1–12. DOI: 10.1109/HOTCHIPS.2007.7482495.

[8] Roger Dettmer. “Occam and the transputer”. In: *Electronics and Power* 31.4 (1985), pp. 283–287. DOI: 10.1049/ep.1985.0185.

[9] Rainer Dömer. *A Grid of Processing Cells (GPC) with Local Memories*. Tech. rep. CECS-TR-22-01. UCI: Center for Embedded and Cyber-physical Systems, Apr. 2022.

[10] Dror G. Feitelson. “Job scheduling strategies for parallel processing proceedings”. eng. In: Lecture notes in computer science 1459. Berlin: Springer, 1998. ISBN: 3540648259.

[11] Michael J. Flynn. “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[12] Vivek Govindasamy, Emad Arasteh, and Rainer Dömer. “Minimizing Memory Contention in an APNG Encoder using a Grid of Processing Cells”. In: *Proceedings of the International Embedded Systems Symposium (IESS)*. Lippstadt, Germany: Springer, Nov. 2022.

[13] Vivek Govindasamy and Rainer Dömer. “Instruction-Level Modeling and Evaluation of a Cache-Less Grid of Processing Cells”. In: *Forum on Specification & Design Languages, FDL 2023, Turin, Italy, September 13-15, 2023*. IEEE, 2023, pp. 1–8. DOI: 10.1109/FDL59689.2023.10272195. URL: https://doi.org/10.1109/FDL59689.2023.10272195.

[14] Vladimir Herdt, Daniel Große, and Rolf Drechsler. “Fast and Accurate Performance Evaluation for RISC-V using Virtual Prototypes”. In: *2020 Design, Automation and Test in Europe Conference (DATE)*. 2020, pp. 618–621. DOI: 10.23919/DATE48585.2020.9116522.

[15] Anthony J. G. Hey. “Supercomputing with transputers—past, present and future”. In: *Proceedings of the 4th International Conference on Supercomputing*. ICS ’90. Amsterdam, The Netherlands: Association for Computing Machinery, 1990, pp. 479–489. ISBN: 0897913698. DOI: 10.1145/77726.255192. URL: https://doi.org/10.1145/77726.255192.

[16] IEEE Computer Society. *IEEE Standard 1666-2011 for Standard SystemC Language Reference Manual*. IEEE, New York, USA, 2011.

[17] A. Izraelevitz et al. “Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2017, pp. 209–216.

[18] Lars Luchterhandt et al. “A Quantitative Guide to Navigate Speed/Accuracy Tradeoffs in System Level Design of RISC-V Processor Grids”. In: *2025 Forum on Specification and Design Languages (FDL)*. 2025, pp. 1–9. DOI: 10.1109/FDL68117.2025.11165408.

[19] Lars Luchterhandt et al. “Implementation of Different Communication Structures for a Rocket Chip Based RISC-V Grid of Processing Cells”. In: *MBMV 2024; 27. Workshop*. 2024, pp. 79–89.

[20] John von Neumann. *First Draft of a Report on the EDVAC*. Tech. rep. University of Pennsylvania, June 1945.

[21] *SiFive TileLink Specification*. Version 1.8.1. SiFive Inc. Jan. 2020. URL: https://sifive.cdn.prismic.io/sifive/7bef6f5c-ed3a-4712-866a-1a2e0c6b7b13_tilelink_spec_1.8.1.pdf.

[22] Michael Bedford Taylor et al. “The Raw Processor: A Composeable 32-Bit Fabric for Embedded and General Purpose Computing”. In: 2001.

[23] Sriram Vangal et al. “An 80-Tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS”. In: *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. 2007, pp. 98–589. DOI: 10.1109/ISSCC.2007.373606.

[24] Wilson Snyder et al. *Verilator*. URL: https://verilator.org.

[25] Yutong Wang, Arya Daroui, and Rainer Dömer. “Demonstrating Scalability of the Checkerboard GPC with SystemC TLM-2.0”. In: *Proceedings of the International Embedded Systems Symposium (IESS)*. Lippstadt, Germany: Springer, Nov. 2022.

[26] Yutong Wang, Yanda Li, and Rainer Dömer. “GPCC: Grid of Processing Cells Compiler with Pareto-Optimal Design Space Exploration”. In: *MBMV 2026; 29. Workshop*. 2026.

[27] David Wentzlaff et al. “On-Chip Interconnection Architecture of the Tile Processor”. In: *IEEE Micro* 27.5 (2007), pp. 15–31. DOI: 10.1109/MM.2007.4378780.