Towards Disaggregation-Native Data Streaming between Devices

Nils Asmussen
nils.asmussen@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Michael Roitzsch
michael.roitzsch@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

ABSTRACT

Disaggregation is an ongoing trend to increase flexibility in datacenters. With interconnect technologies like CXL, pools of CPUs, accelerators, and memory can be connected via a datacenter fabric. Applications can then pick from those pools the resources necessary for their specific workload. However, this vision becomes less clear when we consider data movement. Workloads often require data to be streamed through chains of multiple devices, but typically, these data streams physically do not directly flow device-to-device, but are staged in memory by a CPU hosting device protocol logic. We show that augmenting devices with a disaggregation-native and device-independent data streaming facility can improve processing latencies by enabling data flows directly between arbitrary devices.

1 INTRODUCTION

Many modern applications in datacenters require accelerators to run efficiently, a major example being AI workloads requiring GPUs or NPUs. Datacenter operators however face the challenge that not every conceivable accelerator can be provisioned in every server as this will lead to dramatic under-utilization of resources. Disaggregation promises to resolve this tension by offering pools of resources connected via a fast network fabric. Applications can select a mix of devices fitting their workload and the underlying operating system will configure a tailored execution environment by establishing connections to a set of accelerators from those resource pools.

Interconnect technologies like CXL allow to disaggregate accelerators and other devices like NICs or NVMe storage in this fashion. However, the physical data flows pose an interesting challenge. Suppose an application wants to pass a stream of data from the network to multiple GPUs and NPUs for machine learning and then store the processed result on NVMe. To reduce data movement, the shortest flow would pass data directly from the NIC to the accelerators, where it travels between the needed GPUs and NPUs. The last accelerator would stream its results to the NVMe storage. In reality however, data movement will be handled by the CPU running the application logic. Data must be staged in CPU-side buffers to bridge between the different protocols of the involved devices. Therefore, data will repeatedly flow between devices and the CPU running the per-device protocol logic.

Even if CXL technically allows devices to interact directly with each other, the different device communication protocols currently prevent this feature from being practicable. While point solutions like GPU access to storage [10] or network [6] exist, it is unreasonable to expect every device to implement a driver stack for every other device to enable arbitrary device-to-device data flows. In summary, while devices are physically disaggregated by CXL, data flows remain largely centralized due to the lack of disaggregation-native data-movement facilities.

As a solution, we propose disaggregation-native devices, where today's accelerators, NICs, and storage devices are augmented with a device-independent disaggregation-compatible data-movement facility. Towards a solution, two questions need to be addressed: Protocol placement and inter-tenant isolation.

Protocol Placement. The system-wide data-movement facility must be device-independent, but must interface with concrete devices. Therefore, protocol logic that drives the system-wide device-independent facility is necessary and must run somewhere. This logic also must interface with the concrete devices and therefore implement device-specific logic. Depending on the physical placement of the protocol code, the distance and therefore the latency between application, protocol logic, and device will differ. In this paper, we present an initial discussion of placement options for this logic (Section 2), from centralized to fully distributed.

Tenant Isolation. The second challenge is the combination of direct device-to-device data movement with a strong underlying security and isolation model. In datacenters, tenants must only have access to specific accelerators out of larger pools. Tenants may even space-share the same accelerator. When such devices or device slices communicate directly with each other, the transitive closure accessible to the tenant must still be constrained. As data flows become less centralized and CPU-focused, isolation enforcement becomes harder. We present requirements and challenges for the security of device-independent data movement (Section 3) and sketch a way forward.

Related Work. Similar problems have been addressed in different contexts, such as M³ [2, 3] for systems-on-chip and FractOS [12] for SmartNIC-based networks. We are investigating this problem in the context of CXL. CXL is interesting, because it combines datacenter-scale connectivity with directly attached accelerators due to its PCIe-based nature. M³ also supports accelerators directly, but only offers on-chip connectivity. FractOS is a datacenter protocol, but relies on SmartNICs and regular servers to attach accelerators. CXL however currently lacks our postulated device-independent data movement facility. We quantify the potential gains of adding disaggregation-native devices to CXL (Section 4).

2 DATA STREAM PROTOCOL PLACEMENT

Combining multiple devices (accelerators, storage devices, network interface cards, etc.) to form pipelines or graphs of computation requires a protocol between the application and the used devices. In this work, we focus on the question of where such a protocol should be executed rather than how the protocol should be designed. We consider three different options as depicted in Figure 1.

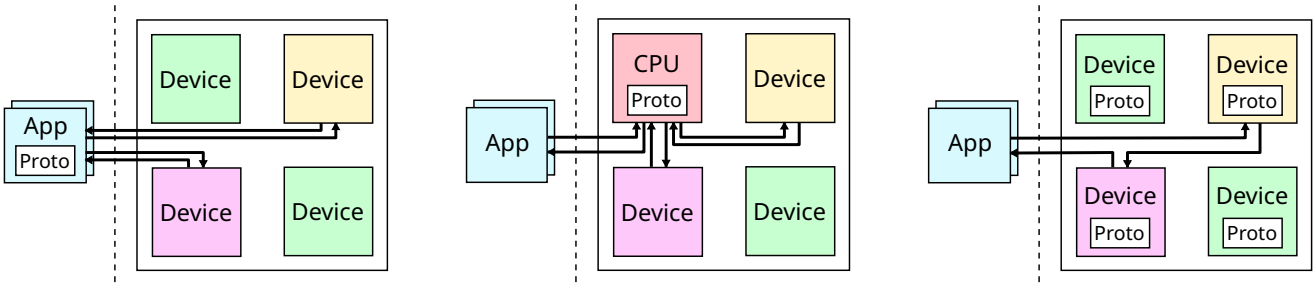

Figure 1: Different options for executing a protocol between application and devices: application-side protocol (left), central resource-side protocol (center), and distributed resource-side protocol (right). The dashed lines indicate machine boundaries.

2.1 Application-Side Protocol

The first option constitutes the established status quo in disaggregated datacenters. The application runs on its machine and has access to a device pool via the CXL fabric. For simplicity, we only show a single pool with heterogeneous devices. The devices could also be distributed among multiple pools. With this option, the devices do not need to speak a common protocol, but can each have their own device-specific protocol, which needs to be known by the application. In particular, the devices are not expected to know how to talk to one another. The application therefore communicates with each device individually using the expected protocol. Allowing applications direct access to individual devices requires means on the device side to ensure that tenants stay within the desired boundaries and are isolated from other tenants.

This variant therefore requires more hops during the communication in comparison to a direct communication between devices. As depicted in Figure 1, if the application wants to start the processing on the yellow device, it needs to copy the input data to the yellow device and trigger the start of the computation. Afterwards the application needs to first retrieve the output data from the yellow device back to its memory and can only afterwards copy this data as input data to the pink device. All of these steps cross the machine boundary between the application and the resource pool, further increasing the overall latency.

2.2 Central Resource-Side Protocol

The second option executes the protocol within the resource pool, centrally on a CPU. This CPU exclusively runs protocol code rather than application logic. This software knows the individual protocols expected by the individual devices, but the application only needs to know how to talk to the CPU. As before, the devices might be distributed among multiple pools, each with a CPU to control them. In this case the application would need to communicate with multiple CPUs and these in turn with their controlled devices.

Like for the application-side protocol, the central resource-side protocol does not require devices to use a common protocol as only the CPU needs to know how to access the devices. In contrast to the application-side protocol, applications do not access devices directly and thus the infrastructure does not require any means to securely grant such access. Access restrictions and isolation of individual tenants can simply be enforced by the CPU when performing device access on behalf of applications. However, this approach still suffers in terms of latency with increasing number of devices that collaborate. This is because the CPU still needs to access each device individually as devices cannot directly communicate with each other. Furthermore, the CPU plays the role of an intermediary within each resource pool, also leading to increased latency, in particular with multiple pools involved.

2.3 Distributed Resource-Side Protocol

The final variant executes the protocol in a distributed fashion on the devices. Programmable accelerators such as GPUs can execute the protocol as part of the application logic. Other devices such as SSDs already employ a processor next to the device, which can be used to execute communication protocols. Alternatively, a processor or a fixed-function logic block can be added next to the device to execute the protocol. If the protocol is implemented in software, it is imaginable to allow applications to deploy the desired protocol onto the devices. Custom protocols can provide more flexibility and/or efficiency at the cost of more complexity on the device side to ensure isolation between tenants.

Executing the protocol on the devices themselves can lead to latency reductions as it avoids an intermediary in the communication and can benefit from a lower latency if multiple devices are located within the same pool. For example, as depicted in Figure 1, the application directly sends input to the yellow device, which in turn sends the output directly to the pink device, which finally sends the result back to the application. This variant therefore leads to the least amount of hops and keeps the communication pool-local, if possible, but requires all devices to understand the same protocol.

3 DISAGGREGATION-NATIVE DEVICES

Considering the different options in the previous section, we believe that the distributed resource-side protocol is the most promising candidate as it offers the lowest possible latency. We call devices that communicate in a peer-to-peer fashion and without CPU involvement *disaggregation-native devices*. We observe the following requirements for such devices:

(1) **Direct communication:** To minimize communication overhead, accelerators and other devices should communicate directly and avoid intermediaries such as the CPU in interactions. This is particularly important for longer chains of accelerators, where a star-shaped communication can lead

to large overheads. In any case, removing intermediaries from interactions can lead to significant energy savings [2].

(2) **Access restrictions:** With a CPU-centric approach, mutually distrusting tenants can be isolated from each other via traditional means on the CPU (e.g, different address spaces and memory mappings). However, direct communication between accelerators demands that individual accelerators can be restricted. These restrictions should be enforced externally instead of by the accelerator itself (e.g., via IOMMUs) to avoid that all tenants need to trust all accelerators, which are typically provided by third-party vendors.

(3) **Generic or custom protocols:** Combining different kinds of accelerators and devices requires a common protocol instead of the per-device-type protocols used today. This can be either achieved by designing a generic protocol for data movement that suites all desired use cases sufficiently well and can be implemented by all accelerators. Alternatively, it is imaginable that applications can bring their own protocol and load it onto the participating accelerators.

(4) **Protocol deployment:** The protocols should be implementable by different kinds of devices. For example, programmable accelerators such as GPUs might not need additional hardware, but can simply implement the protocol in software as part of the application logic. Other accelerators might already use a co-processor for internal management tasks and could implement the protocol on the co-processor. Fixed-function accelerators might want to introduce a co-processor for that purpose or implement the protocol as a fixed-function logic block.

(5) **Cross-machine communication:** Ideally, accelerators and devices should be able to interact with each other regardless of their physical location. This requires that the same protocol can be used even if one communication partner resides in a different machine. However, due to the potentially different performance characteristics, optimizing the placement or selection of communication partners should be feasible.

3.1 M³ System Architecture

Given these requirements, we believe that the M³ system architecture [3] is a good starting point. Previously focused on system-on-chips (SoCs), M³ builds upon a tiled hardware architecture [13] and runs a custom tailored operating system on top, as shown in Figure 2. Most importantly, each tile is equipped with a new hardware component called *data transfer unit* (DTU), which is used for cross-tile messaging and memory accesses instead of relying on coherent shared memory.

3.1.1 Heterogeneous Devices. M³ was designed for heterogeneous systems from the beginning and thus tried to minimize the assumptions on the individual devices. For that reason, the M³ kernel (red) runs on a dedicated *kernel tile* and leaves the remaining *user tiles* free for applications. User tiles therefore do not need OS support (such as virtual memory or different privilege levels), simplifying the integration of accelerators and other devices into user tiles. Nevertheless, OS services such as file systems and network stacks are

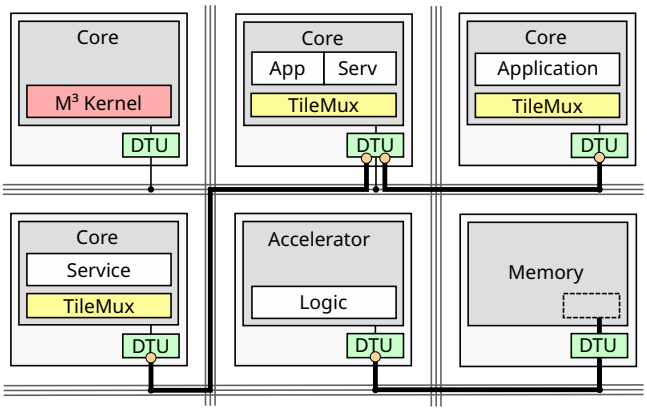

Figure 2: System architecture of M³: one DTU per tile isolates tiles from each other and selectively allows communication as configured by the M³ kernel. TileMux multiplexes its tile among the applications on this tile.

available on all user tiles, because these are offered via DTU-based communication protocols.

The M³ kernel manages the applications and OS services on user tiles as *activities*, comparable to processes. An activity on a general-purpose tile executes code, whereas an activity on an accelerator tile uses the accelerator's logic.

3.1.2 Access Restrictions. Supporting DTU-based communication between user tiles requires the enforcement of access restrictions by the DTU. For that reason, message passing and memory accesses via DTU require an established *communication channel* (thick black lines in the figure). Communication channels are represented as *endpoints* in the DTU (orange dots). At runtime, each endpoint can be configured to different endpoint types: A *receive endpoint* allows to receive messages, a *send endpoint* allows to send messages to a specific receive endpoint, and a *memory endpoint* allows to issue DMA requests to tile-external memory.

Activities can use existing communication channels, but only the M³ kernel is allowed to establish such channels. This is done via capabilities like in other microkernel-based systems [7, 8, 11]. By default, no communication channels exist and thus tiles are isolated from each other. Additionally, applications are placed on different tiles by default, but as shown by M³v [1], tiles with general-purpose cores can also be shared efficiently and securely among multiple applications. For that reason, every core-based user tile runs a multiplexer called *TileMux* (yellow), which is responsible for isolating and scheduling the applications on its own tile, similar to a traditional kernel. However, in contrast to a kernel, each TileMux instance has no permissions beyond its own tile. Instead, only the M³ kernel can make system-wide decisions, hence its name.

3.1.3 Direct Communication. The DTU is designed to support direct communication between user tiles. For flexibility and efficiency reasons, the DTU's interface is therefore split into a control plane and data plane. The control plane is used during the setup phase by the M³ kernel to establish the required communication channels. The data plane is used by applications to use the previously established communication channels. During the setup phase capabilities

are used to manage and distributed permissions in the system and also to constrain the communication channels as desired. The constraints are enforced by the DTU's data plane.

3.1.4 Protocol Implementation. The DTU provides message passing and DMA-like memory accesses to implement arbitrary protocols between activities. These mechanisms have been used in the past to implement, for example, system calls, network access, and file access. M³x has also shown with the so called *File Protocol* that such protocols can be implemented both in software and as a fixed-function logic block for simple accelerators. This design allows to combine regular applications and accelerators in pipelines.

3.2 Challenges

In summary, we believe that the building blocks provided by M³ are well suited for disaggregation-native devices. However, multiple challenges and open questions remain.

(1) M³ is currently designed for SoCs and therefore optimized for a low communication latency between tiles. Extending the system to off-chip communication with PCIe or cross-machine communication with CXL will increase the latency and thus require adaptions in hardware and software. For example, polling until a message is delivered successfully might no longer be desirable.
(2) The M³ kernel currently manages the user tiles within the same SoC from a dedicated kernel tile within that SoC. Moving to disaggregation-native devices that do neither need nor desire CPU cores raises the question whether the devices can be managed externally (e.g., from a server with CPUs) by an equivalent of the M³ kernel.
(3) How a protocol has to be designed to be efficient and generic is an open question. Similarly, it is unclear whether the system should rather support the deployment of custom application-specified protocols instead of implementing a single generic protocol.

4 EVALUATION

We intend to demonstrate the benefits of a distributed resource-side protocol and study the suitability of M³ as a foundation for disaggregation-native devices. We therefore perform experiments based on the current state of M³, which is available as open source [1].

4.1 Measurement Setup

We use the gem5 platform [4] and take advantage of its customizability to configure the system similar to future disaggregated CXL-based systems. We configure gem5 to simulate two machines connected over an interconnect and equip each machine with multiple locally connected devices. As the exact performance characteristics of CXL are still unknown, we optimistically assume rather low latencies, because higher latencies further increase the latency benefit of the distributed resource-side protocol. Concretely, we configure 1μs round-trip latency across the two machines and 500ns round-trip latency within the machine. The latter is half of a typical round-trip latency for PCIe gen 3 [5, 9].

[1] https://github.com/Barkhausen-Institut/M3

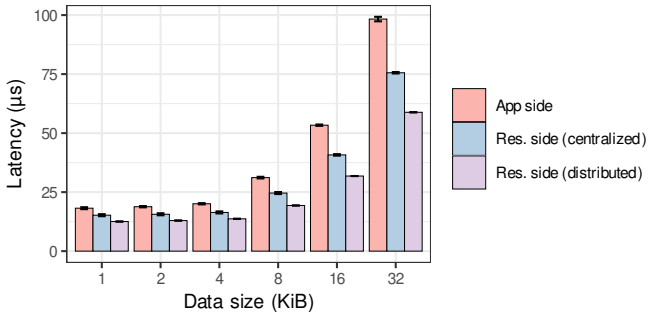

Figure 3: Performance comparison of different protocol placements using different data sizes.

The application runs on a CPU in the first machine and all devices (and potential protocol CPUs) are located in the second machine, acting as the resource pool. Since we focus on the protocol rather than the accelerators or devices, we do not simulate actual accelerators, but only their control cores. We conservatively use the in-order RISC-V CPU model clocked at 1GHz for per-device control cores and the out-of-order RISC-V CPU model with 4GHz for all other CPUs. Higher-performance control cores are of course possible and would benefit the distributed protocol.

4.2 Protocol Implementation

For simplicity and comparability, we use the same protocol implementation called *data channel* for all three protocol placements. Data channels are employed between a sender and receiver and are instantiated multiple times to form larger pipelines or graphs. The sender pushes the data into the memory of the receiver and sends a notification message at completion. The receiver waits for this notification and sends a response to the sender when the data has been processed. We use the data channel to build the three placements as depicted in Figure 1. For example, with the client-side placement, the client has two data channels to the yellow device (outbound and inbound) and two data channels to the pink device.

4.3 Results

The benchmark is run with different data sizes and uses 50 repetitions after 4 warm-up runs. As shown in Figure 3, the distributed version achieves the best results and is between 45% and 67% faster than the application-side version and 21% to 28% faster than the centralized version. These results are of course preliminary due to several unknowns about future CXL-based systems. However, due to our conservative settings we believe we can still conclude that speedups can be achieved by executing data movement protocols in a distributed fashion on the devices themselves.

The measurements also revealed a shortcoming of the current M³ platform: data transfers are performed in at most 4 KiB packets, dictated by the page size. This approach works fine on SoCs with fast on-chip networks, but is not well suited for interconnects with higher latencies as used in this benchmark. For this reason, the interconnect latency is paid multiple times for the data sizes above 4 KiB leading to a faster latency increase than strictly necessary.

5 CONCLUSION

Our experiments are based on simulation, but all parameters were configured to not unduly benefit the distributed protocol implementation. Still, we see a significant latency benefit, leading us to conclude that disaggregation-native devices are a logical next step in datacenter disaggregation. M³ has demonstrated suitable security primitives on the system-on-chip level. It remains open, how these primitives can be mapped to CXL fabrics and which features of CXL must be augmented to integrate disaggregation-native devices with strong yet decentralized inter-tenant isolation mechanisms.

6 ACKNOWLEDGEMENTS

We thank Mark Silberstein for the food of thought in this direction. This research is funded by the European Union's Horizon Europe research and innovation program under grant agreement No. 101092598 (COREnext).

REFERENCES

[1] Nils Asmussen, Sebastian Haas, Carsten Weinhold, Till Miemietz, and Michael Roitzsch. 2022. Efficient and Scalable Core Multiplexing with M³v. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Lausanne, Switzerland). ACM, 452–466. https://doi.org/10.1145/3503222.3507741

[2] Nils Asmussen, Michael Roitzsch, and Hermann Härtig. 2019. M³x: Autonomous Accelerators via Context-Enabled Fast-Path Communication. In *USENIX Annual Technical Conference (ATC)* (Renton, WA, USA). USENIX, 617–632.

[3] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Atlanta, GA, USA). ACM, 189–203. https://doi.org/10.1145/2872362.2872371

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Computer Architecture News* 39, 2 (u 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[5] Keith G. Erickson, M. Dan Boyer, and D. Higgins. 2018. NSTX-U advances in real-time deterministic PCIe-based internode communication. *Fusion Engineering and Design* 133 (2018), 104–109.

[6] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 201–216. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kim

[7] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP'09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[8] Adam Lackorzynski and Alexander Warg. 2009. Taming Subsystems: Capabilities As Universal Resource Access Control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems* (Nuremburg, Germany) *(IIES'09)*. ACM, New York, NY, USA, 25–30. https://doi.org/10.1145/1519130.1519135

[9] L. Rota, M. Vogelgesang, LE. Ardila Perez, M. Caselle, S. Chilingaryan, T. Dritschler, N. Zilio, A. Kopmann, M. Balzer, and M. Weber. 2016. A high-throughput readout architecture based on PCI-Express Gen3 and DirectGMA technology. *Journal of Instrumentation* 11, 02 (2016), P02007.

[10] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: integrating a file system with GPUs. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, 485–498. https://doi.org/10.1145/2451116.2451169

[11] Udo Steinberg and Bernhard Kauer. 2010. NOVA: a microhypervisor-based secure virtualization architecture. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, Christine Morin and Gilles Muller (Eds.). ACM, 209–222. https://doi.org/10.1145/1755913.1755935

[12] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. 2022. Slashing the Disaggregation Tax in Heterogeneous Data Centers with FractOS. In *European Conference on Computer Systems (EuroSys)*. Rennes, France, 352–367. https://doi.org/10.1145/3492321.3519569

[13] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. 2007. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro* 27 (10 2007), 15–31. https://doi.org/10.1109/MM.2007.89