Supporting efficient channel-based communication in a mesh network-on-chip

Sebastian Raase, Süleyman Savas, Zain Ul-Abdin and Tomas Nordström
Centre for Research on Embedded Systems, Halmstad University
{sebastian.raase, suleyman.savas, zain-ul-abdin, tomas.nordstrom}@hh.se

ABSTRACT

In the dataflow programming model, data flows through unidirectional communication channels. In order to maximize performance in highly parallel systems, it is necessary to provide a low-overhead communication mechanism. This proposal for dedicated hardware support implementing channel-based communication does not require changes to the core mesh architecture or the instruction set, is relatively cheap to implement in hardware and aims to reduce the overhead for channel-based communication substantially compared with a software-only solution. The estimated speedup for token submission exceeds 43x.

1. INTRODUCTION

Exploiting parallelism is becoming increasingly important for many different, extensively studied reasons. As Amdahl's law shows, even a relatively modest amount of serial execution reduces the achieveable speedup through large-scale parallelization immensely. Thus, for maximum performance, especially on highly parallel systems, loss of parallel execution needs to be avoided.

Applications developed using the dataflow programming model are inherently parallel, making them a suitable choice for parallel architectures. In this model, independent processes (actors) communicate through channels, which are implemented as FIFOs. When a process is divided into smaller processes to be executed in parallel, these processes usually need to communicate. Communication may even represent a significant share of the processing time, limiting application performance. In order to avoid or minimize the overhead of communication, it should either be kept as short as possible or overlapped with computation.

While some parallel architectures, e.g. Ambric [1], support circuit-switched channels in hardware, most architectures use a packet-switched message-passing network-on-chip with different topologies. However, these messages are often not exposed to the user and hidden behind a shared-memory or other abstraction.

The Epiphany [2] manycore architecture provides a two-dimensional packet-switched mesh network-on-chip implementing a shared address space. To support dataflow applications on this architecture, we have implemented a communications library [3], which provides an abstraction of communication channels. Even though this level of abstraction is beneficial in terms of productivity, the overhead of a software-only solution turned out to be prohibitive, especially for small messages.

Comparing our software solution to our experiences with the Ambric architecture, we believe that hardware-based support for communication channels will reduce the overhead and allow for substantial performance gains in dataflow applications. Therefore, we propose a simple hardware extension to efficiently support channel-based communication between cores on a manycore architecture.

The rest of this paper is organized as follows. Section 2 discusses related works. Section 3 discusses the challenges we considered in our design, which is described in section 4. Section 5 concludes the work.

2. RELATED WORKS

The Alewife multiprocessor [4] supported both communication channels and shared memory accesses on top of their communication network to efficiently support applications preferring either abstraction. However, they require the CPU to actively receive each message, which becomes infeasible if packets become small and frequent.

HAQu (Hardware-Accelerated Queuing) [5] provides one-to-one communication channels by adding a hardware unit to each core, but also ties in rather deeply with the memory subsystem. In this system, software and hardware queues are compatible and distributed across cores. The CAF (Core-to-core Communication Acceleration Framework) [6] centralizes queue management in a NoC-level hardware unit (QMD), providing multi-producer, multi-consumer queues. Both approaches focus on the cache hierarchy and, in contrast to our design, require changes to the instruction set architecture. The Pronto message passing system [7] is designed for manycore architectures. Using DMA, this system reduces the end-to-end latency, but, unlike our approach, requires a network round-trip per message to handle buffer management and synchronization.

In this work, we propose a simple hardware peripheral, which provides virtual point-to-point communication channels on top of an existing message-based communication infrastructure. Each of these channels uses a pre-allocated message buffer and can be used with very low overhead by the application software to transmit tokens between cores. Our design is suitable for manycore architectures providing distributed memories with limited or no cache coherence.

3. CHALLENGES

We have identified three major challenges, which must be considered in order to provide useful support for channel-based communication on top of a message-based infrastructure. These are blockwise processing of data, speed differences between producers and consumers, and network la-

tency. This section covers them in detail. We assume that the network is reliable and does neither corrupt or drop message. In addition, it must not reorder messages from a single source to a single destination.

3.1 Blockwise Processing

The majority of dataflow applications require actors to collect multiple input tokens, possibly from different sources, before producing output tokens. The number of tokens produced or consumed by an actor may also vary wildly over short time intervals, even if the average throughput is constant. These bursts of activity put transient pressure on the network, which may influence the latency of unrelated messages. More importantly, feedback loops between such actors may cause deadlocks if the maximum number of tokens in the loop is below a specific threshold. This becomes more apparent if some actors engage in vector processing, as vectors may require multiple messages to be transmitted between actors.

By buffering the channels, these deadlocks are avoided. Additionally, the transient network pressure from bursts of activity is reduced, affecting latency and throughput instead. In theoretical approaches, communication channels are often assumed to be unbounded, but since real systems do not support infinite memory, buffer sizes must be finite.

We believe that there is no universally optimal buffer size. Too small buffers may lead to deadlocks whereas too large buffers can cause buffer-bloat and consequently increased or fluctuating latency (jitter) and reduced throughput.

3.2 Speed Differences

In dataflow applications, producers and consumers should generally run at the same speed. System architects try to approximate this by identifying bottlenecks and distributing resources appropriately. However, real parity can usually not be achieved; some throughput difference remains, with either the producer or the consumer being faster than its counterpart. Large speed differences also occur if one of the communication partners fails or, more likely, is currently preempted. While transient speed differences can be caught adequately through buffering, permanent speed differences eventually lead to deadlock or data loss.

If a producer produces tokens faster than the consumer can consume them, all intermediate buffers will fill up. As soon as they are full, the next token that is sent will either need to overwrite a previously received token, stall, or be dropped, depending on the underlying communication and buffering policies. The first two options lead to data loss, which is generally unacceptable due to the high cost of recovering whereas stalling causes back pressure, which may interfere with unrelated communication and carries the risk of deadlock. Systems employing these communication strategies either have to provide additional means to avoid deadlocks [4] or ignore the problem by requiring the consumer to receive packets at all times [8], both of which we consider undesirable solutions.

We believe that the only real solution to this problem is preventing the producer to submit tokens if it cannot guarantee their delivery. This prevents the producer from overrunning the communication channel in the first place, but requires extra communication between consumer and producer. As an example, the Pronto message passing system [7] uses a handshaking approach, which involves a network round-trip for each message.

If the consumer is faster than the producer, it will need to spend some fraction of its time waiting, which only wastes clock cycles and energy. While low-power idle states and dynamic re-clocking strategies may be beneficial in those cases, they are outside the scope of this paper.

3.3 Network Latency

In order to prevent a producer from overrunning its communication channel, it needs to know when to stop producing tokens. This information needs to flow from the consumer to the producer through the network in the opposite direction of the data stream, introducing a feedback loop. In traditional computer architectures employing strong memory-ordering guarantees, where communication latency is not exposed to the application, this is not a correctness, but a performance problem. In distributed systems however, providing such guarantees may restrict scalability, leading to the use of weakened memory-ordering models between processing elements or abandoning the shared memory model altogether. In such systems, network latency introduces inconsistent knowledge about the state of the communication channel while it is in use.

The producer needs to stop producing as soon as the communication buffer is full, even though this information might not have reached it yet. It could be informed before the buffer gets full, so that tokens arriving late can still be stored in the buffer. However, this threshold depends on the network latency, which may depend on unpredictable factors, and misjudgment might lead to data loss. Alternatively, the producer can consider buffer spaces as reserved unless it knows for sure that they are available. Then, data loss cannot occur. However, in this case the producer needs to know the total buffer size in advance and needs to be informed whenever buffer space becomes available. This method requires additional buffer space (depending on latency) to prevent unreasonable overblocking and introduces shared state between producer and consumer. Consequently, whenever channels are established or dissolved, care must be taken to avoid race conditions.

4. DESIGN

In this chapter, we describe our proposed hardware support for communication channels. Since the core concepts stem from our Epiphany communications library, we introduce it first, and then describe the structure and behavior of source and destination ports in detail.

4.1 Communications Library for Epiphany

We have implemented a channel-based communications library for the Epiphany manycore architecture [3], which is also suitable for other shared address-space architectures. The Epiphany architecture employs a 2D mesh network-on-chip with XY routing. The mesh uses a weak memory model, so that read-after-write and write-after-write accesses to different non-local destinations happen in a non-deterministic order. While writes are fast asynchronous fire-and-forget operations, the architecture implements remote reads as a slow read request answered by a write [2], stalling the core.

Our library provides communication channels, which are unidirectional FIFOs connecting a single pair of source and destination ports and carry fixed-size messages (*tokens*) only. Each channel supports five access functions, which are listed

in Table 1, and can be operated in both blocking and non-blocking modes.

Type	Function	Blocking	Description
Source	*write*	yes	Send tokens
	space	no	Returns number of tokens writable without blocking
Dest.	*read*	yes	Read and consume tokens
	peek	no	Read tokens
	level	no	Return number of tokens readable without blocking

Table 1: Access Functions in e-commlib

Associated with each channel is a fixed-size ring buffer, which is allocated in the destination core's local memory. It is used concurrently by both the source and destination cores. Synchronization is done through read and write pointers, which are updated atomically. To avoid remote reads, both pointers are cached within both cores.

We have found that especially for small token sizes, the overhead of maintaining the pointers becomes prohibitive. On the other hand, supporting per-channel buffer sizes and keeping the buffers in fast local RAM allows balancing application memory, network latency and deadlock avoidance depending on the application's needs.

4.2 Overview

We propose dedicated hardware units, which handle buffer and rate management. These hardware units (source and destination *ports*) form the endpoints of a virtual, unidirectional communications channel. The channel's buffer is not located in dedicated hardware registers, but uses the destination core's memory. Buffer management is done exclusively by the destination port and tokens cannot be larger than a single network transaction; larger messages must be transmitted as multiple tokens.

Figure 1 provides an overview of the structure of source and destination ports, their connections and the information exchanged between them. When a channel is established, both the source and destination ports need to be informed about the other end and the buffer size. Then, for each transmitted token, an acknowledgement is sent back to the sender as soon as the corresponding token has been removed from the destination buffer.

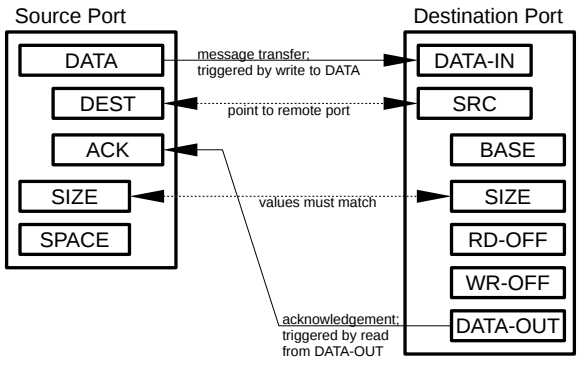

Figure 1: Overview of ports.

Both token and acknowledgement transfers are atomic by design, avoiding the need for additional synchronization. By counting the number of transmitted, but not yet acknowledged tokens, the source port implements rate-limiting to prevent buffer overruns.

Ports are a limited resource in each core and can not be reasonably shared between channels. Our research suggests that many dataflow actors only require few high-speed channels and that a software implementation for additional channels is sufficient.

Applications are free to use both the port infrastructure and any other means of communication offered by the system simultaneously to ensure full software compatibility and highest performance.

4.3 Detailed Description

Each source port is a simple peripheral unit consisting of at least the five registers `DATA`, `DEST`, `ACK`, `SIZE` and `SPACE`, as shown in figure 1. Additional registers may be added for additional functionality, such as interrupt-driven operation.

Writing to the `DATA` register triggers a transmission of a token through the communication channel. As long as the `SPACE` register contains a zero-value, a write should stall. Alternatively, write attempts may fail immediately and raise an exception. The source port is configured by writing to the `DEST` and `SIZE` registers. The `SPACE` register contains an approximation of the available space in the channel buffer and is never written to explicitly. Writing the `SIZE` register also sets the `SPACE` register to the same value. Additionally, it is decremented by the data width whenever the `DATA` register is written, and incremented whenever the `ACK` register is written. When this register contains a zero-value, the channel buffer is full and writing to the channel is currently not allowed. The `ACK` register is never accessed by software; this register is remotely written to by the destination port to update the local `SPACE` register.

Each destination port is a dedicated peripheral unit, managing accesses to the channel buffer from both the local core (reading) and the source port (writing). Figure 2 shows how four of the at least seven registers are used to manage the channel buffer, which is located in local memory.

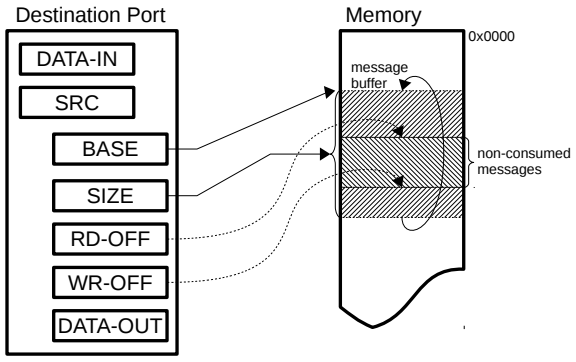

Figure 2: Destination Port and Channel Buffer

The channel buffer is designed as a ring buffer and the `BASE` and `SIZE` registers contain its location in local memory. The two offset registers `RD-OFF` and `WR-OFF` contain the read and write offset from the base address, and are updated automatically when the buffer is accessed by the port. The `SRC` register references the source port for this channel.

A token arrives as a remote write request to the `DATA-IN` register, where it is written to the local memory address

(BASE + WR-OFF), after which the WR-OFF register is updated by the hardware. To consume a token from the buffer, the local core reads from the DATA-OUT register, which results in a memory read from the address (BASE + RD-OFF), an update to the RD-OFF register, and the transmission of an acknowledgement message to the source port's ACK register. If the channel buffer is empty, reading the DATA-OUT register should stall. Alternatively, the read attempt may fail immediately and raise an exception. The DATA-IN and DATA-OUT registers may be aliased.

The RD-OFF and WR-OFF registers are never written explicitly. Instead, they are set automatically whenever the SIZE register is written. However, the RD-OFF register should be readable in order to implement a non-consuming read (*peek*) operation, if application software requires such functionality.

A communications channel is initialized by writing the source and destination port addresses to the DEST and SRC registers, respectively. At the destination, a contiguous memory buffer must be allocated and its base address written to the BASE register. The configuration is then finalized by writing the channel's buffer size in bytes to the SIZE register, which will update additional registers as well. Care must be taken to avoid using the channel before both ends are fully configured, since doing so may lead to inconsistent shared state and data loss or loss of buffer space.

Since accesses to the DATA, DATA-IN and DATA-OUT are atomic, the data width of these registers decides the maximum token size. However, shorter accesses may be possible if the underlying communication network and memory subsystem support them. While it should also be possible to use different access widths even for a single message (such as submitting a message as a word and receive it as two halfwords), this is subject to endianness issues and may lead to unaligned memory accesses. Thus, implementations are free to put additional restrictions to such accesses.

Since the DATA and DATA-OUT registers are memory-mapped, the communication channel can be used for DMA transfers if the DMA engine supports a constant source and/or destination address for the whole transfer and can be stalled.

5. DISCUSSION AND CONCLUSION

Compared to shared-memory approaches often taken by manycore architectures, a channel-based communication interface maps well to the actor model and seems beneficial for dataflow applications. We have implemented a generic software library on the Epiphany platform providing such an interface on top of the existing network-on-chip.

The time to transmit a token between two cores can be broken down into a token submission time t_S, a token transmission time t_T and a token retrieval time t_R, where t_T depends on the underlying communication hardware and is not affected by our proposal. Table 2 shows the per-token fraction of t_S in cycles for 64-bit tokens. For the software implementations, this time is further divided into a constant overhead for buffer management (163 cycles) and the actual data submission, which uses the *memcpy* library function to be generic [3]. Replacing this function call with a fixed-size direct copy provides a 1.9x speedup, but the overall overhead is still significant, considering the token size.

Our proposed hardware implementation reduces the token submission time for small token sizes substantially. We have measured the real overhead on the Epiphany. In the ideal case, submission of a 64-bit token requires a single store instruction, which can take as little as a single clock cycle. In a more realistic setting, where a message may consist of multiple tokens and imperfect scheduling incurs pipeline stalls, per-token submission times of three clock cycles are feasible when using loop unrolling, or eight clock cycles without, resulting in substantial speedups compared to the software solution. We also measured the initial startup overhead to be approximately 15 clock cycles.

case	time	speedup
SW (memcpy)	163 + 183 cycles	(base) 1.0x
SW (direct)	163 + 20 cycles	1.9x
HW (looped)	8 cycles	43.3x
HW (unrolled)	3 cycles	115.3x
HW (ideal)	1 cycle	346.0x

Table 2: Per-token submission times (64-bit tokens)

We have proposed a hardware peripheral for channel-based communication suitable for distributed manycore architectures. In contrast to other works, our approach does not require changes to the instruction set architecture or the CPU core itself. The expected performance gain is expected to be substantial, with preliminary measurements showing speedups in excess of 43x using realistic assumptions.

6. REFERENCES

[1] M. Butts, B. Budlong, P. Wasson, and E. White, "Reconfigurable work farms on a massively parallel processor array," in *16th International Symposium on Field-Programmable Custom Computing Machines*, 2008.

[2] A. Olofsson, T. Nordström, and Z. Ul-Abdin, "Kickstarting High-performance Energy-efficient Manycore Architectures with Epiphany," in *48th Asilomar Conference on Signals, Systems and Computers*, 2014.

[3] S. Raase, "A Dataflow Communications Library for Adapteva's Epiphany," Tech. Rep. diva2:895406, Halmstad University, 2015.

[4] J. Kubiatowicz and A. Agarwal, "Anatomy of a Message in the Alewife Multiprocessor," in *7th international conference on Supercomputing*, 1993.

[5] S. Lee, D. Tiwari, Y. Solihin, and J. Tuck, "HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor," in *17th International Symposium on High Performance Computer Architecture*, 2011.

[6] Y. Wang, R. Wang, A. Herdrich, J. Tsai, and Y. Solihin, "CAF: Core to Core Communication Acceleration Framework," in *25th International Conference on Parallel Architectures and Compilation*, 2016.

[7] S. S. Kumar, M. T. A. Djie, and R. Van Leuken, "Low overhead message passing for high performance many-core processors," in *First International Symposium on Computing and Networking*, 2013.

[8] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *ACM SIGARCH Computer Architecture News*, 1992.