RVipc: Low Latency Multi-core Communication Protocol for Trusted Environments

Kaustubh Khulbe, Sanjeevi Sengottuvel
University of Illinois at Urbana-Champaign
{kkhulbe2,ss152}@illinois.edu

Abstract

With the collapse of Denard scaling and Moore's Law, architects are transitioning towards multi-core systems. This paradigm shifts allows multiple cores on a single die, connected by high-bandwidth interconnects.

These interconnects are typically designed for high bandwidth but are elaborate NoCs, which often lead to nondeterministic latencies. Moreover, they are software-transparent, meaning application programmers cannot directly control these fabrics. Application programmers are left with shared memory as the primary means of communication.

However, shared memory suffers from nondeterministic latencies due to the cache hierarchy, TLB flushes, and cache pollution. This is especially problematic for latency-critical applications such as autonomous vehicles, drones, robotics, and augmented/virtual reality applications.

In this paper, we present RVipc, a low-latency inter-core communication protocol for real-time systems. RVipc is designed for trusted environments, such as embedded systems. It provides a mechanism to synchronize and communicate between cores with low latency and predictable latency metrics.

1 Introduction

Multi-core systems are becoming increasingly fundamental to modern computing, from servers down to embedded systems and edge devices. This paradigm shift is driven by the need for higher compute power, as traditional scaling methods have plateued.

Multi-core systems incorporate multiple cores on a single die, often tied together by high-bandwidth interconnects. For example, AMD utilizes Infinity Fabric [1], a proprietary interconnect technology, to connect multiple chips and cores.

There are several challenges with these forms of interconnects. First, they are usually designed for high bandwidth, and often cause nondeterministic latencies. This could be due to the routing algorithms or contention. More importantly, these interconnects are software-transparent, meaning application programmers cannot directly control how traffic flows across them. This limitation motivates the use of shared memory [2] as the primary means of communication.

Shared memory suffers from nondeterministic latencies due to the cache hierarchy, TLB flushes, and cache pollution. Modern NoCs also suffer from high variance in latencies [5].

This is problematic for latency-critical applications such as autonomous vehicles, drones, robotics, and augmented/virtual reality applications [3]. The success and safety of these applications directly depends on low-latency synchronization and communication between cores. A failure to do so significantly lowers the quality and feel of these products.

This motivates the need for a low-latency inter-core commmunication protocol. In this paper, we present RVipc, a low-latency inter-core communication protocol designed for trusted environments. Our contributions are as follows:

1. Deterministic latencies in communication between cores
2. Low-latency communication between cores
3. Software tooling to automatically communicate between cores in the most efficient manner

2 Design

In order to achieve deterministic latencies, we need to ensure both hardware and software usage is deterministic.

2.1 Hardware Design

To do so, we allow cores to sync and set up a dedicated FIFO buffer between them.

We decided to use RISC-V as our target architecture as it is open source and designed to add extensions easily [6]. RVipc is a set of RISC-V ISA extensions that enable dedicated FIFO communication.

This FIFO buffer is used to pipe data between cores. Since it is a dedicated resource, there is no contention for it and will provide highly predicatable latencies in hardware.

The design then is a pool of FIFO buffers with a hardware unit to allocate buffers to cores. This hardware unit allows dedicated FIFOs to be set. Therefore, the only source of nondeterminsm is initializing the FIFO buffers.

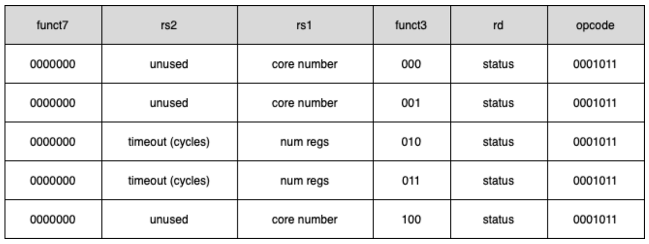
Once initialized, there is a fixed latency guarantee to send and recieve data between cores.

The ISA extensions we propose are described below.

2.2 Software Design

2.2.1 Software Stack Overview

To avoid nondeterministic latencies in the software stack, we need to minimize the use of the kernel [4]. This means we cannot invoke system calls and kernel resources, as they utilize the trap, I/O, and scheduling that is out of the programmer's control.



funct7	rs2	rs1	funct3	rd	opcode
0000000	unused	core number	000	status	0001011
0000000	unused	core number	001	status	0001011
0000000	timeout (cycles)	num regs	010	status	0001011
0000000	timeout (cycles)	num regs	011	status	0001011
0000000	unused	core number	100	status	0001011

Figure 1. RISC-V ISA Extensions

We can do this by providing a set of user-space instructions that are exposed to the application programmer. These instructions can be used to configure FIFO buffers, send and receive data, and tear down FIFO buffers.

We chose to use a polling mechanism to configure and tear down FIFO buffers. This is because polling, as opposed to interrupt-based, does not need the kernel and the programmer has full control over the frequency of polling.

We modified the RISC-V toolchain and ISA [6] to add the following instructions. The exact semantics of these instructions can be seen in Fig. 2

1. `fconn`: Connect to a FIFO buffer
2. `fcreate`: Create a FIFO buffer
3. `fsend`: Write to a FIFO buffer
4. `frecv`: Read from a FIFO buffer
5. `fclose`: Close a FIFO buffer

We provide a modified toolchain and compiler to be able to support these instructions natively.

2.2.2 Handshake Protocol

The singular point of nondeterminism in RVipc is the setup of the FIFOs. This is due to limited hardware resources and the need for an arbitrator. Since each core polls to acquire a buffer, the FIFO `fconn` and `fcreate` instructions have non-deterministic latencies.

The following handshake protocol is used to correctly configure the buffers. Let Hart 0 be the producer and Hart 1 be the consumer.

1. Hart 0 issues `fcreate`, which sends a request to the pool to initialize a FIFO queue.
   a. If the request completes, `status` will show 0.
   b. If there are no available FIFOs, `status` will show $1 \ll$ `ERR_NO_FIFO_AVAIL`.
   c. If a FIFO already exists with the issued core, `status` will show $1 \ll$ `ERR_FIFO_EXISTS`.
2. There is no dependence on Hart 1 when setting up the FIFO queue.
3. The FIFO pool automatically closes the FIFO if inactive for `INACTIVE_THRESH` cycles.
4. Hart 1 issues `fconn`, which sends a request to the pool to connect to a FIFO queue.
   a. If the request completes, `status` will show 0. The FIFO is now ready to use.
   b. If there is no FIFO set up for the core, `status` will show $1 \ll$ `ERR_NO_FIFO_CREATED`.
   c. If the core has already set up a prior connection with the same core, `status` will show $1 \ll$ `ERR_ALREADY_INITIALIZED`.
5. Hart 0 polls the pool with `fstatus`, which sends back `status`.
   a. `status` will include information if the handshake was successful, FIFO is ready, etc.
6. Hart 1 polls the pool with `fstatus`, which sends back `status`.
   a. `status` will include information if the handshake was successful, FIFO is ready, etc.
7. Hart 0 is now able to send with `fsend`. Hart 1 is now able to receive with `frecv`.

Note that Hart 0 and Hart 1 need to poll to set up the FIFO with `fstatus`. In a many-core system this can cause back-pressure on the pool to service these instructions. However, the aim of polling is to allow `fstatus` to resolve by the end of the WriteBack stage, so as to not cause any pipeline stalls or increase critical path. This mechanism also avoids any kernel traps which further improves latency.

2.3 Implementation

We utilized Gem5 to implement the hardware FIFO pool. There is a global class `HwIpc`, which contains the necessary data mappings for the FIFO pool and appropriate locking mechanisms to ensure proper simulation.

We configured the simulated machine to have a dual core system. This is sufficient as we are testing latencies in sending and receiving data, not FIFO setup contention.

Each core has an instruction and data cache. Each cache is connected to the memory controller. The exact Gem5 configuration we utilized is attached in Fig. 3.

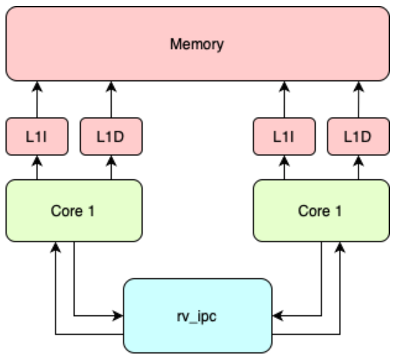


Figure 2. Architectural overview of Gem5 simulation, including the FIFO pool, CPU cores, caches, and memory controller.

The following is an example of the command used to run the simulation:

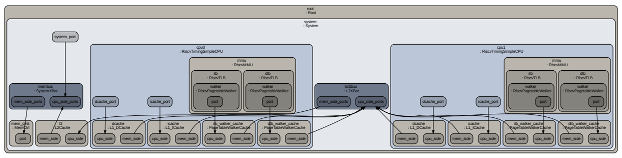


Figure 3. Gem5 configuration setup.

```
 build/RISCV/gem5.fast --outdir=m5out/ \
 configs/deprecated/example/se.py \
 --cpu-type=RiscvTimingSimpleCPU --num-cpus=2 \
--redirects /lib=/home/fluxbyt/riscv-compiled/lib \
 --cpu-clock=4GHz \
 --cacheline_size=64 \
 --caches \
 --l1i_size=8kB \
 --l1d_size=8kB \
 --l2cache \
 --l2_size=4MB \
 --l2_assoc=16 \
 --cmd="ipc_bin/test_send;ipc_bin/test_recv"
```

2.4 Parameter Sweep

After designing an appropriate hardware and software architecture for the FIFO pool, we ran an extensive design space exploration to realize under which conditions our system performs best.

2.4.1 First Message Latency

The first message latency is defined as the time from the first packet of data sent to the first packet of data received.

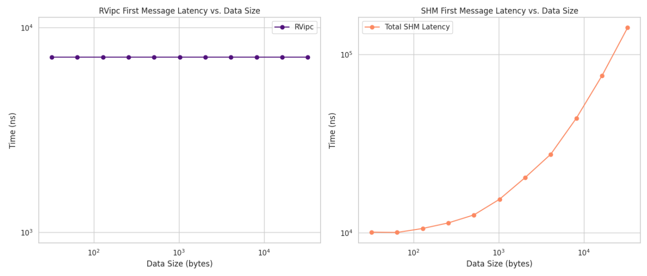


Figure 4. First message latency of RVipc and SHM.

The first message latency of the shared memory mechanism scales as the size of the data increases. However, RVipc maintains constant latency. This is key in ensuring deterministic latency behavior for the set of applications we are targeting.

2.4.2 Overall Latency

RVipc has a constant first message latency, and linear latency for data size as indicated in Fig. 4. Let NUM_CYCLES denote the number of cycles for a single packet of data to be sent. By definition, this is constant as we have a dedicated FIFO link between the communicating cores. Let NUM_SETUP denote the number of cycles required to create and set up the FIFO. Additionally, let N denote the number of packets we are sending.

The overall latency is then defined as:

$$\texttt{latency} = \texttt{NUM\_SETUP} + \texttt{NUM\_CYCLES} \cdot \texttt{N} \tag{1}$$

This further ensures deterministic latencies.

Contrastingly, we note that SHM has unpredictable latencies, even when broken into the read and write latency separately. Additionally, there is a large benefit to using RVipc for either low-latency first messages, or relatively small amounts of data.

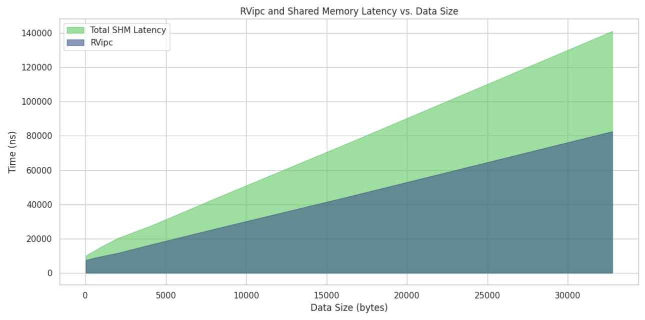


Figure 5. Overall latency comparison of RVipc and SHM.

2.4.3 Bandwidth

While the latency of the system is the primary focus of this work, the bandwidth, or throughput, of the FIFO pool is just as crucial. We note that, as indicated in Fig. 6, that RVipc successfully surpasses the shared memory bandwidths for sufficiently large data sizes.

Though this does not take into account contention on the FIFO pool for sending data, it is a reasonable approximation that RVipc will not harm bandwidth, if not improve it.

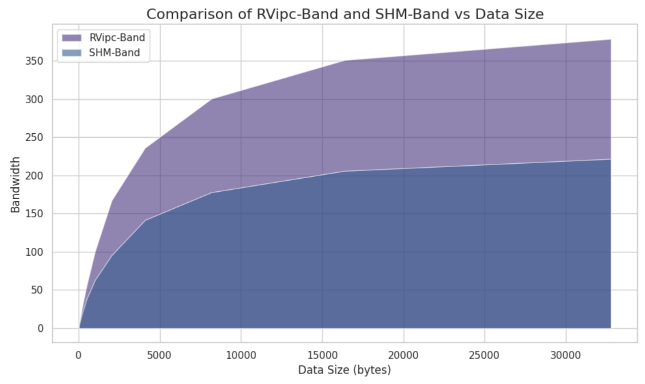


Figure 6. Bandwidth comparison of RVipc and SHM.

2.4.4 Cache Line

The last parameter we sweeped is the cache line size. The reason for exploring this parameter is because being able to send data in larger chunks has direct implications on cache pollution and message latencies. As indicated in Fig. 7, RVipc yields lower latencies and higher throughptus for larger cache lines. This provides valuable insight into what kinds of cache hierarchies are appropriate for RVipc.

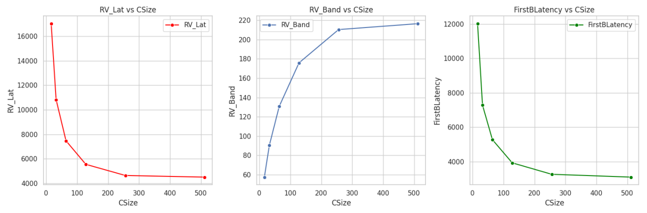


Figure 7. Bandwidth vs. Cache Line Size Comparison of RVipc and SHM.

2.5 Cache Pollution and Latency Analysis

2.5.1 Cache Pollution

We define cache pollution as the amount of cache evictions that are caused by a shared memory read-write operation. Concretely, let M_{L_2} denote the amount of lines evicted from the L2 cache when reading M. Let M_{L_1D} denote the lines evicted when bringing data into the L1 data cache. Let M'_{L_1D} and M'_{L_2} denote the amount of lines that we want to share between the cores from L1 data and L2, respectively.

We conduct our theoretical analysis on a non-includsive eviction policy, as modern systems like Intel and Apple silicon utilize this policy.

$$\texttt{pollution}_{SHM} = M_{L_2} + M_{L_1D} + M'_{L_1D_0} + M'_{L_2} + M'_{L_1D_1} \quad (2)$$

However, RVipc does not need to put M' into cache and can directly utilize the FIFOs. Therefore,

$$\texttt{pollution}_{RVipc} = M_{L_2} + M_{L_1D} + M'_{L_1D_1} \quad (3)$$

Note that RVipc always yields less cache pollution that SHM, specifically by $M'_{L_1D_0} + M'_{L_2}$. This is important to minimize latency variation in the system. The larger the amount of data being transferred over shared memory is, the more pollution the mechanism causes and the better RVipc performs.

2.5.2 Latency Analysis

Let θ_{L_2} be the time required to bring M lines into the L2 cache. Let θ_{L_1} be the time required to bring M lines into the L1 data cache from the L2 cache. Let α_{L_1} be the time required to put M lines from L1 data into L2. Let β denote the time required to send a single message via the FIFO buffers.

$$\texttt{latency}_{SHM} = \theta_{L_2} + \theta_{L_1} + \alpha_{L_1} + \theta_{L_1} \quad (4)$$

$$\texttt{latency}_{RVipc} = \theta_{L_2} + \theta_{L_1} + M \cdot \beta \quad (5)$$

Notice that as long as $M \cdot \beta < \alpha_{L_1} + \theta_{L_1}$, RVipc will yield lower latencies than SHM. The performance implications of SHM over RVipc is hardware depedendent.

3 Related Work

The paper by Zhael et. al [7] discusses various IoT communication protocols as well as their hardware implications. While RVipc focusses on the same end use cases, our paper focusses on core to core synchronization, while Zhao et. al focus on IoT devices being able to efficiently communicate with each other. This causes differing design constraints.

Another important technology is the AMD Infinity Fabric [1]. AMD's Infinity Fabric is a NoC interconnect technology for heterogeneous systems. It allows efficient communication between numerous different components on an SoC, such as CPUs and GPUs. This is a hardware communication protocol and infrastructure, similar to RVipc, but differ in the end goals. RVipc aims to provide a dedicated synchronization and communication channel between two cores, while Infinity Fabric aims to provide a general communication method from any component to any other.

Both of these technologies focus on improving low-latency communication, but RVipc aims dedicated core to core communication, and focusses especially on deterministic latencies. This causes different design constraints to be considered.

4 Conclusion

In this paper, we presented RVipc, a synchronization and communication mechanism on multicore systems for low and deterministic latencies. We achieved this by designing a hardware pool of FIFOs and an accompanying communication protocol that minimizes kernel resources and trap entries.

We explored various design parameters, such as cache line sizes and L1 data cache sizes, to see how the system performs under various workloads. We also measured the first message latency, overall latency, and bandwidth of RVipc verus the current standard that is exposed to application programmers. That is, shared memory.

We observed drastic improvements in first message latencies and significantly more deterministic latencies, along with expected improvements to bandwidth and overall latencies.

5 Future Work

The current implementation of RVipc allows configurable dedicated lniks for point-to-point communication between cores. A future exploration could involve customizeable communication for many-to-many cores, allowing for broadcasted communication that entirely avoids the cache hierarchy.

Another future exploration would be to realize the design and it's area and power implications, as that is a major consideration in hardware design.

Finally, we would also like to explore better software integration with current synchronization techniques and better integration on multi-core and distributed environments. Our evaluations were primarily done on a dual core system, and analytics on larger systems would be beneficial.

6 Metadata

The presentation of the project can be found at:

https://zoom/cloud/link/

The code/data of the project can be found at:

https://github.com/you/repo

References

[1] AMD. AMD CDNA 3 Architecture White Paper. In *AMD White Papers* (n.d.). Available at https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf.

[2] Aspnes, J. InterProcessCommunication. In *Yale University CS Lecture Notes* (n.d.). Available at https://www.cs.yale.edu/homes/aspnes/pinewiki/InterProcessCommunication.html.

[3] Elbamby, M. S., Perfecto, C., Bennis, M., and Doppler, K. Towards Low-Latency and Ultra-Reliable Virtual Reality. In *IEEE Network* (Mar. 2018), vol. 32, pp. 78–84. Available at https://arxiv.org/abs/1801.07587.

[4] Jia, J., Le, M. V., Ahmed, S., Williams, D., Jamjoom, H., and Xu, T. Fast (trapless) kernel probes everywhere. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC '24)* (2024), USENIX Association, pp. 379–386.

[5] not specified, A. A Low Latency Variance NoC Router. In *Embedded and Multimedia Computing Technology and Service* (2012), pp. 89–97. Available at https://link.springer.com/chapter/10.1007/978-94-007-5076-0_10.

[6] Waterman, A., and Asanović, K. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft. In *RISC-V Foundation* (2019). Available at https://courses.grainger.illinois.edu/ece391/sp2025/docs/unpriv-isa-20240411.pdf.

[7] Zhao, Z., Yu, W., Wei, Z., and Wei, Z. Survey of communication protocols for internet-of-things and related challenges of fog and cloud computing integration. *arXiv preprint arXiv:1804.01747* (2018).