Zipper: Latency-Tolerant Optimizations for High-Performance Buses

Shibo Chen
chshibo@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Hailun Zhang
hzhang664@wisc.edu
University of Wisconsin
Madison, Wisconsin, USA

Todd Austin
austin@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

ABSTRACT

As heterogeneous designs take over the world of hardware designs, the data bus plays a vital role in interconnecting hosts and accelerators. While past works have emphasized increasing communication bandwidth for data-hungry workloads extensively, this work focuses on optimizing the communication latency for latency-sensitive acceleration applications. We first study the pattern of various accelerator workloads and demonstrate that various optimization opportunities exist to reduce the overhead of communication latency. To help developers exploit these opportunities, we introduce *Zipper*—a protocol optimization layer that reduces communication costs by enabling device and request level parallelism and exploiting data locality for existing bus standards. We applied Zipper to two domains and implemented the end-to-end system on a heterogeneous hardware platform with integrated FPGA. Our physical experiments show that Zipper provides 8x speedup for one accelerator with 4.3% logic overhead and 1.5x speedup for another with 0.9% logic overhead.

ACM Reference Format:
Shibo Chen, Hailun Zhang, and Todd Austin. 2025. Zipper: Latency-Tolerant Optimizations for High-Performance Buses. In *30th Asia and South Pacific Design Automation Conference (ASPDAC '25), January 20–23, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3658617.3697546

1 INTRODUCTION

Data buses are vital in connecting heterogeneous components in today's hardware designs. While high-performance data buses have ramped up bandwidth over time, the access latency has not been scaling on par because link traversing scales poorly as the technology node shrinks [40]. A recent study [24] shows that the round-trip latency through the popular PCI Express Gen 3.0 [2] or Intel Ultra Path (UPI) Interconnect [19] languishes at the microsecond scale. Many applications cannot tolerate microsecond-level latencies, leading to most of these latencies being fully exposed [8]. As such, long-latency bus transactions hinder the broad deployment of a wider spectrum of applications and accelerators in the production environment.

Although communication latency is hard to reduce through improved physical designs, we observed two significant opportunities for latency-tolerant optimization. First, there is parallelism at both the request and the device level. We can enhance host and accelerator utilization by enabling out-of-order and parallel execution; second, many compute kernels exhibit significant temporal locality: the results of previous requests often become inputs for subsequent requests. We can exploit this locality to reduce data movement.

ASPDAC '25, January 20–23, 2025, Tokyo, Japan
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0635-6/25/01...$15.00
https://doi.org/10.1145/3658617.3697546

To capitalize on latency-tolerant optimization opportunities in a real production system, we desire a generic and reusable solution that provides the following benefits:

- **Simple parallelism model**: Data dependencies can exist between host and accelerator instructions or within accelerator instructions. Harvesting parallelism requires a dynamic scheduling mechanism that works across ISAs and device boundaries.
- **Efficient data tracking**: Since data is continually moved between host and accelerator, the system needs to precisely and efficiently track the location of the data to ensure functional correctness.
- **Reduced design complexity**: Due to the sheer size of possible accelerator designs and platforms, customizing APIs and compilers would be a heavy technology burden for ordinary system developers. Major vendors only provide function-level APIs [26, 39], which makes adding compiler support even more difficult. A portable and extensible solution is necessary to make latency optimizations accessible, scalable, and bus-standard agnostic.

To deliver these features, we propose *Zipper*. Working on top of existing data buses, Zipper is a dynamic bus protocol optimization layer that reduces bus transaction latency between heterogeneous devices connected through a high-performance data bus. It dynamically analyzes data dependencies, tracks data movement across devices, and exploits locality and parallelism as a program proceeds. Zipper uses a software-defined request scheduling approach that requires no modification to application logic, compilers, or data buses. Zipper's runtime library identifies temporal locality and parallel execution opportunities and schedules optimized accelerator requests to enable resource-constraint-aware parallelism and data reuse. The implementation details of this runtime library are hidden from developers, and the developers can use encapsulated data types as if they are host-native. For hardware, Zipper offers a small request buffer to cache data and enables out-of-order execution of requests with data reuse. In Section 4.1, two FPGA-based case studies are presented and shown to benefit significantly from Zipper. Our experiments show that Zipper provides uniformly good end-to-end application speedups, with as much as 8x speedup for one case study.

We summarize the contributions of this work as follows:

- We detail the design of Zipper, a general protocol optimization layer to optimize bus transaction latency for heterogeneous

system communication over existing high-performance buses. The optimization layer exploits locality and parallelism opportunities that are currently missed, without making demands on the underlying data bus or programmers.
- We show a real-world FPGA-based implementation of Zipper, built over Intel's QuickPath Interconnect (QPI) for CPU-integrated FPGAs. We dive into the hardware and runtime software components that were implemented for this platform.
- We present two real-world FPGA-based case studies with Zipper in a production environment and show significant application-level performance improvements (as much as 8x), with modest area overheads (of less than 5% increase in logic). In both studies, our real-world application demonstrated significant amounts of temporal locality and transaction parallelism, even for designs with only a 4-entry request scheduling window.

Zipper is open-sourced and ready to deploy in real production environments that connect accelerators using AXI, CCI-P, or CXL data buses. The implementation, written in C++ and SystemVerilog, will be available via a URL, which has been omitted for anonymity.

2 DISCOVERING BUS OPTIMIZATION OPPORTUNITIES

This section discusses three key opportunities that we focus on to optimize bus communication latency.

2.1 Host-Accelerator Communication Convention

After the host connects to the accelerator, the developer creates a shared memory space between the two for them to pass inputs and compute results. To kick off the kernel, the host writes inputs into the shared memory and issues instructions with metadata (*i.e.*, input starting address, result write back address, etc.) to the accelerator through, typically, Memory Mapped Input Output (MMIO). After receiving the instruction, the accelerator fetches the input data from the shared memory, writes back the result to the specified write-back address, and notifies the host. This is usually agnostic to physical implementations (*e.g.*, UPI [19], PCIe [2], Infinity [5], etc.) or data transfer protocols (*e.g.*, CCI-P [18], CXL [28], AXI [7], etc.).

2.2 Optimization Opportunities

In this section, we use a reduction algorithm over an abstract hardware-accelerated operator $\otimes$, as shown in Algorithm 1, to demonstrate existing latency-tolerant opportunities. In this algorithm, we want to calculate the product of the $2n$ inputs over a hardware accelerated operator $\otimes$ and store the result to the write-back address. The accelerator is attached to the host system, which runs the algorithm by high-performance data buses, *i.e.*, UPI, PCIe, etc.

Figure 1 shows the data-dependence graph of Algorithm 1 and the optimizations to eliminate dependencies, exploit locality, and enable parallelism. Starting with the unoptimized implementation in (a), the developer partitions the instructions based on the devices' capabilities: execute instruction 1, 2, 4, and 6-8 on the host, and offload instruction 3, 5 to the accelerator. A stock compiler cannot optimize cross-device dependencies; thus, the unoptimized system has to execute instructions sequentially in program order. As a result,

Algorithm 1: A reduction algorithm with operator $\otimes$.

```
Data: An array of 2n elements: arr[2n], A writeback address
      $addr_{wr}$
Result: y = summation of all elements in arr over special
        operator $\otimes$
$i \leftarrow 0$; $result \leftarrow 1$;
while $i < 2n - 1$ do
    1: a $\leftarrow$ load(Mem[arr+i]);
    2: b $\leftarrow$ load(Mem[arr+i+1]);
    3: c $\leftarrow$ a$\otimes$b;
    4: fetch c;
    5: result$\leftarrow$ c$\otimes$result;
    6: fetch result;
    7: i $\leftarrow$ i+2;
end
8: Mem[$addr_{wr}$]$\leftarrow$ result
```

the host always waits for the accelerator to complete computation and loads the result from the shared memory (performed by `fetch` instructions 4 and 6) before it can move on to the next instruction and/or use the result in subsequent requests.

2.2.1 *Exploitable Temporal Locality.*

We notice that not every dependency is created equal. A cross-device data dependency is much costlier to resolve than a local data dependency due to the bus communication overhead. Based on this observation, we eliminate cross-device dependencies and replace them with local ones whenever possible. That is, instruction 5's two operands are the result from instruction 3 and its result from the last iteration. Therefore as shown in Figure 1b, instruction 5 does not need to wait for instruction 4 on the host side to complete and then get its input from the host. Rather, we can remove this cross-device dependency by directly forwarding the results from the previous requests 3 and 5, as shown in (b). By relocating cross-device dependencies, we can avoid much inter-device communication and thus reduce communication overhead.

2.2.2 *Device-level Parallelism.*

Instructions 4 and 6 block instructions that fetch results from the shared memory. After the dependencies have been relocated, instruction 4 and 6 can be moved off the critical path. As shown in Figure 1c, the host can continue execution while the accelerator is working on the received requests. Being non-blocking, the host can run ahead to fetch new data for future accelerator requests. As long as there is no data dependency across devices, the two devices can run in parallel and do not need to synchronize.

2.2.3 *Request-level Parallelism.*

After relocating the data dependencies and enabling device-level parallelism, we can completely offload a sequence of requests to the accelerator. We can also extract request-level parallelism locally on the accelerator to maximize the performance gain. In our example, since instruction 3 is independent of previous accelerator instructions, shown in Figure 1d, it can bypass previous requests or interleave with other requests. The only limitations would be the number of requests the accelerator can handle simultaneously and the accelerator's compute throughput.

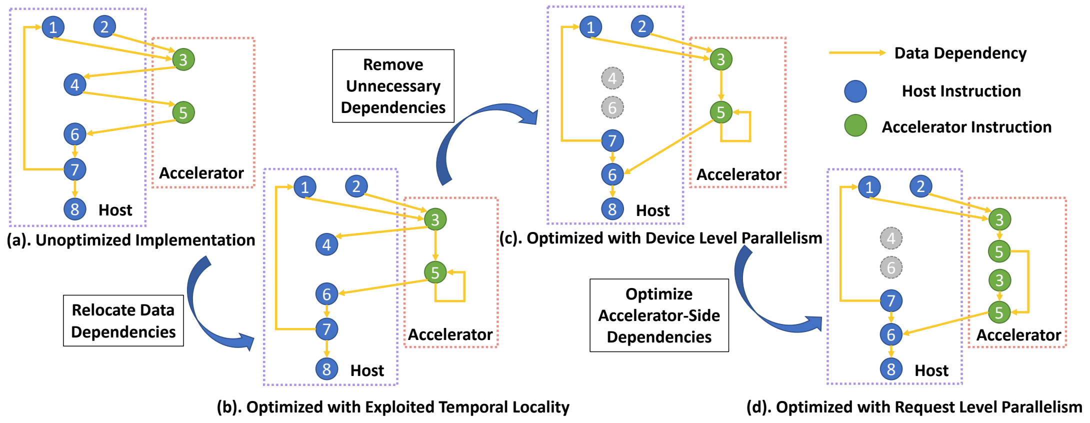

Data Dependency
Host Instruction
Accelerator Instruction
Accelerator
Host
(a). Unoptimized Implementation
Relocate Data Dependencies
(b). Optimized with Exploited Temporal Locality
Remove Unnecessary Dependencies
(c). Optimized with Device Level Parallelism
Optimize Accelerator-Side Dependencies
(d). Optimized with Request Level Parallelism

Figure 1: Latency-tolerant optimizations for the instruction sequence shown in Algorithm 1.

3 ARCHITECTING ZIPPER OPTIMIZATIONS

To enable optimizations discussed in Section 2, we propose *Zipper*. Zipper is a protocol layer that resides between the physical bus and the application logic, and thus, it does not require any changes to the compiler, compute kernel, or the underlying data bus. Zipper is a drop-in optimization that significantly reduces the exposed latency that is common for high-performance buses, essentially widening the applicability of these emerging bus technologies.

3.1 Overview of Zipper

Zipper uses a set of communication semantics that captures the locality and dependency information to connect the host and accelerator. Zipper adds a request buffer table to the accelerator that tracks the status of operands and caches recent request results. On the host side, a runtime library analyzes data dependency and catches the data reuse opportunities by observing and tracking accelerator requests. The runtime library also manages communication between the host and the accelerator and hides tedious implementation details from software developers. The rest of this section will first describe the communication protocol between Zipper host and accelerator, Zipper's hardware structure, and Zipper's runtime library.

3.2 Host-Accelerator Communication Protocol

In Zipper, the host sends requests to the accelerator through MMIO and communicates input operands and results with the accelerator through shared memory. The number of fields and the bits in each field can vary depending on the use case. As a rule of thumb, each request should include `Instruction`, `Write-back Address`, and `Operand Information`. Each request can have multiple operands. Each operand may reside in the shared memory or the Zipper hardware structure. The shared memory is the communication channel between the host-accelerator for input operands and results. We partition the shared memory into operand partition and result partition. The input operands are continuously placed in the operand partition and wrapped over to reuse the old memory when it reaches capacity. The result partition maintains the bijection with the accelerator-side buffer table entries. For input operands smaller than the size of one memory request granularity, Zipper packs multiple operands for

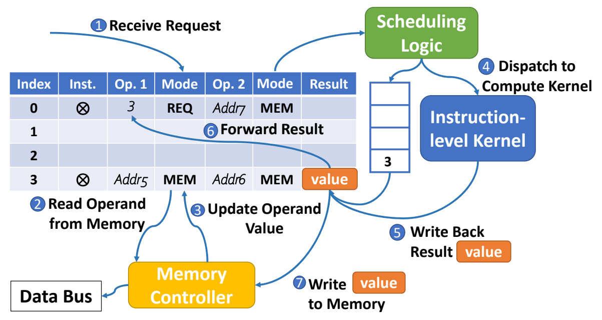


Figure 2: Zipper hardware structure and life cycle of an accelerator request.

different requests into one cache line to reduce the number of accesses to the memory. Zipper software attaches a version bit to each operand when issuing the request. The Zipper accelerator verifies the freshness of the operand by matching the version bit with the version bit it receives from the runtime library.

3.3 Zipper Hardware Structure

Zipper hardware resides on the accelerator side and handles requests it receives from the host. It consists of four parts, shown in Figure 2: a request buffer table, an execution scheduler, a memory controller, and the accelerator. The memory controller is platform-specific, and the compute kernel is user-specific. Zipper does not need to make intrusive modifications to these two components to work.

Zipper uses the request buffer table and the execution scheduler to enable request-level parallelism. In 1, when Zipper hardware receives a request from its software counterpart, typically through MMIO, it will first store the request information in the request buffer table. The request buffer table stores and tracks all the details on pending and recently completed requests: the instruction, the status of each operand, the write-back address, etc. The request buffer table can be of different sizes. We will discuss the impact of buffer size in Section 5.3. The execution scheduler decides which request is ready for execution and can dispatch instructions out-of-order. The scheduling logic prioritizes older requests when multiple requests are ready to be dispatched.

The request buffer table caches recent results until new requests take the entries. Zipper hardware then reconstructs the data dependency chain based on the information embedded in the requests. For each accelerator instruction, Zipper fetches their values based on the information provided in the request. ② If the operand is in the memory, Zipper issues a read request to the memory controller and marks it as "in fetch" to avoid duplicated access. If the operand comes from a prior request, Zipper either fetches the value if it is ready in the buffer table or waits until the prior request has been completed. ④ Once all operands are resolved, Zipper marks this request as ready to be dispatched. ⑤ When the computation is done, Zipper stores the results back in the buffer table and writes the results into their corresponding write-back address in the memory, shown in ⑦. ⑥ If there are pending requests whose inputs are dependent on the newly completed request, Zipper directly forwards the value when the result is ready.

3.4 Zipper Runtime Library

In Zipper, providing a non-blocking host-side interface that tolerates multiple pending requests within the accelerator's resource limitation is essential. On the host side, Zipper conducts dependency analysis, request scheduling, and result fetching with a software runtime library. Zipper provides packaged data classes to host applications as if they were host-native types. These data classes encapsulate overridden functions and necessary metadata. This approach provides flexibility and dynamic scheduling capabilities without compiler modification. Figure 3 shows Zipper's software data structures and corresponding updates when behaving different functions.

The Zipper runtime library maintains two data structures to track data objects and communicate with the accelerator: class objects and result lists. The class objects track the status of the requests and the results if the requests are complete.

The result lists track all the software data objects associated with each hardware buffer table entry. When Zipper fetches the results back to the host or clears a table entry, it iterates through the list and updates all relevant data objects. This enables Zipper to track multiple in-flight requests.

We use a code snippet shown in the figure to demonstrate the operations of the Zipper runtime library. The code first calculates an accelerator request and its result a with input from the host and then calculates another variable b reusing a"s value. After these two accelerator requests, it re-assigns b to a. Lastly, it retrieves the a' s value from the accelerator back to the host.

3.4.1 Issuing New Requests. Figure 3a shows Zipper issuing a new accelerator request to the accelerator. ① Within the context shown in the figure, Zipper registers object a into an available slot in result lists. ② Zipper will store the input operands m and n in the shared memory and send their relative location to the accelerator. In the last step ④, Zipper updates a's validity as `false` and marks it to be inside the accelerator at location 3. After this step, the program can continue onto the next host instruction or accelerator request.

3.4.2 Enabling Accelerator-Side Caching. In Figure 3b, Zipper makes another accelerator request. Since there is no empty slot available, Zipper first clears the oldest entry as shown in Step ⓪. Zipper forces each object mapped to slot 1 to fetch its value to the host memory if they have not already and update them as not in the accelerator's buffer anymore. During the analysis stage, Zipper detects variable a is at location 3 of the accelerator buffer and its value can be reused, so Zipper will not write a to the shared memory nor need to fetch a's value back. Instead, Zipper instructs the hardware to get a's value directly from buffer table slot 3. In this way, Zipper detects the relocation opportunities on the host and utilizes the hardware buffer to exploit them. We then append b to the result lists and update its metadata similar to what we did to a in the last request.

3.4.3 Object Reassignment. When reassigning an object to track another object, as in Figure 3c, Zipper changes the data structure to reflect this reassignment. We reassign b to variable a. ① Zipper copies a's metadata to b and moves b away from its original slot in the result lists to the same slot as a. Similarly, Zipper removes the object from the result list when the object is getting deleted.

3.4.4 Lazy Fetch. Zipper never proactively retreives results until the value is needed. As the code execution progresses, the host eventually asks for the value of a to proceed, shown in Figure 3d. In this case, Zipper fetches a's result from its tracking location 3. If the result is not ready, the host will stall due to hard dependency. Once Zipper fetches the value from the shared memory, it will update a's value and its metadata. Zipper will also update all the objects that are tracking location 3. However, a's value is still in the accelerator buffer for future use as no new request evicts a yet.

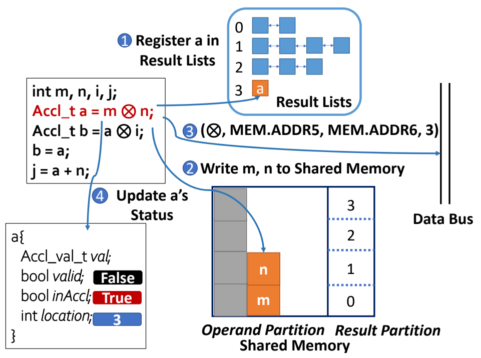
4 REAL-WORLD EXPERIMENTAL SETUP

This section discusses the case studies and hardware setup that are representative of the production environment. We conducted our experiments on Intel HARPv2 [13] with an in-package FPGA. The system contains a 64K FPGA-side coherent cache. The Zipper-augmented software runs on the Intel Xeon E5-2699v4 @ 2.2 GHz, and the Zipper-enabled hardware kernels run on the Arria10 FPGA. The host and the FPGA are connected with Intel QuickPath Interconnect using Core Cache Interface.

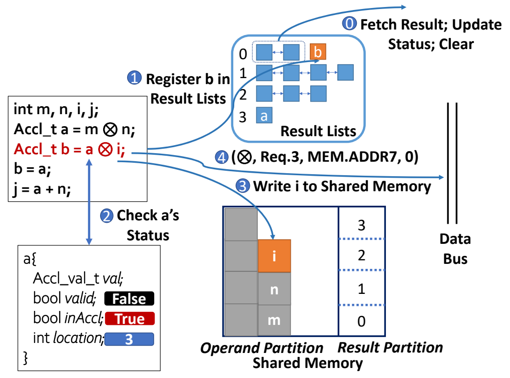
4.1 Real-World Case Studies

We evaluated Zipper on two applications that rely on CPU and accelerator to compute and are highly sensitive to the communication latency between the two devices: (1) we replaced the floating point representation in the NASA Parallel Benchmark (*NPB*) [11] with a Posit32 number representation. Posit is a 32-bit number format that achieves better precision than floating points, but currently lacks native hardware support. All posit computations are computed with a hardware kernel; (2) we implemented hardware isolation support for the integer subset of VIP-Bench [10]. VIP-Bench is a set of algorithms implemented in a data-oblivious manner where only the SE hardware enclave can see the plaintext values of the secrets[9]. We prototyped the SE enclave on an FPGA, and all privacy-enhanced operators are offloaded to the SE enclave.

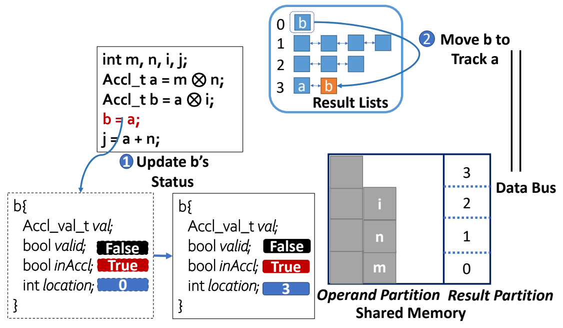
These two latency-sensitive applications represent interesting privacy and HPC acceleration opportunities that are gaining traction and benefit greatly from fine-grained offloading. We used an 8-entry buffer table design for the Posit32 accelerator and a 2-entry design for the SE enclave design for optimal performance-area trade-offs. In the baseline design, each request is issued and executed sequentially.



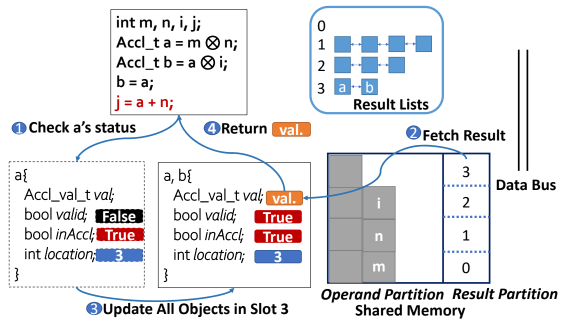
(a) Issuing new request to accelerator



(b) Issuing new request with operand reuse



(c) Object reassignment



(d) Fetching results from accelerator

Figure 3: Zipper software data structure and scheduling.

5 EXPERIMENTAL EVALUATION

This section provides an analysis of the performance speedup, the area overhead, the impact of various optimizations, request buffer table sizes, workload profiles, and other relevant design aspects.

5.1 Performance Speedup and Logic Overhead

Figure 4 shows the relative performance of Zipper over the baseline design. The figure also provides insights into each feature's contribution to the overall performance. On average, Zipper provides 8x speedup for NPB with Posit32 and 1.5x for VIP-Bench with the SE.

We synthesized our design with Intel Quartus Pro 16.0.0.211 onto the targeted FPGA platform. We compute the logic overhead by taking the added Zipper logic over the accelerator and the existing bus control logic. The logic overhead of Zipper is only 4.3% for the 8-entry Zipper Posit32 design over the baseline design and 0.9% for the 2-entry Zipper SE enclave design over the baseline design.

5.2 Accelerator Memory Access

Zipper's performance benefits greatly from reducing accelerator memory demands by exploiting temporal locality and memory coalescing. Figure 5 shows the percentage of the bus transactions Zipper and other de-featured design options make over the baseline design.

For NPB with Posit32, Zipper reduces the accelerator's bus transactions by 77% from the baseline. Request-level parallelism enables out-of-order execution but does not reduce any memory access. Since Zipper can pack 8 input operands into one cache line, memory coalescing reduces 63% of bus transactions over the baseline.

Window Size / Application	0	2	4	8
	Exploitable parallelism			
NPB w/ Posit	1	1.89	3.53	6.17
VIP-Bench w/ SE	1	1.87	2.69	3.94
	Percentage of results to be fetched back			
NPB w/ Posit	100%	45%	26%	22%
VIP-Bench w/ SE	100%	55%	21%	10%
	Distance between issue and use			
NPB w/ Posit	251.86	543.59	840.71	821.11
VIP-Bench w/ SE	65	136.68	1656.9	2243.7

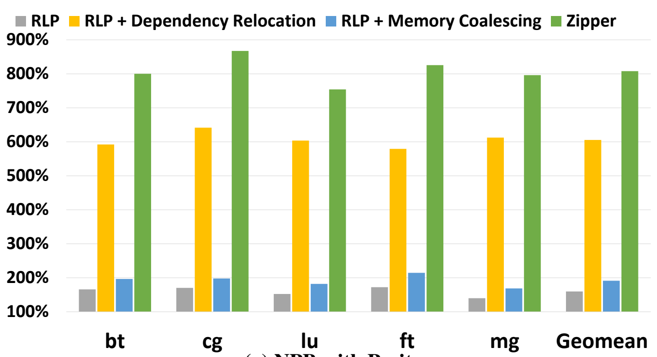
Table 1: Zipper characteristics under different instruction window sizes for two applications on average.

Dependency relocation exploits temporal locality and data reuse, reducing 34% of bus transactions over the baseline.

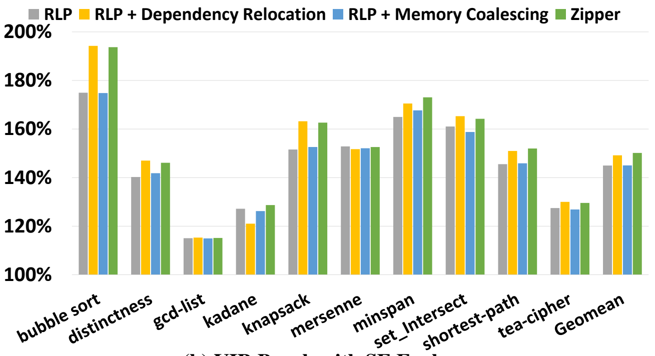
Since operands are larger in the VIP-Bench with SE enclave design, Zipper cannot pack the operands as tight as with Posit32 numbers. Therefore, it is more likely that the operands for the same request span over two cache lines, which leads to more memory access and fewer opportunities for memory coalescing. Zipper reduces 46% of the bus transactions while memory coalescing and dependency relocation reduce 37% and 27% of the bus transactions over the baseline SE enclave design, respectively.

5.3 Impact of Hardware Buffer Size

To study the optimal number of buffers for different workloads, we analyzed the distance of the data dependency chain in Zipper requests or the number of entries we need to provide for efficient

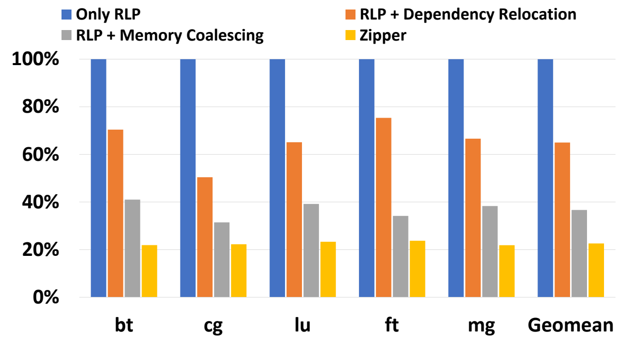


(a) NPB with Posit

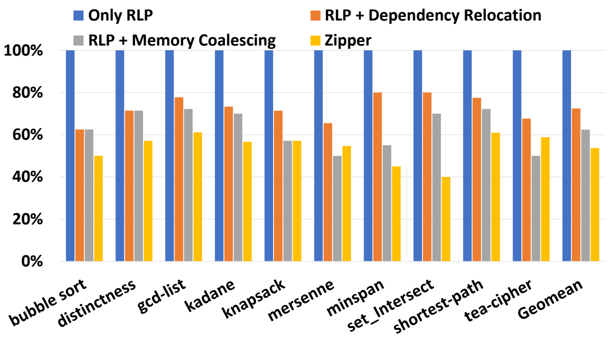


(b) VIP-Bench with SE Enclave

Figure 4: Relative performance of Zipper and various de-featured Zipper over the baseline. RLP = Request-level Parallelism.



(a) NPB with Posit32



(b) VIP-Bench with SE Enclave

Figure 5: Comparison of the number of bus transactions by accelerator between Zipper, de-featured Zipper, and the baseline.

dependency relocation. Our experiment results show that 91% and 92% of the temporal locality can be captured with only four buffer entries for NPB and VIP-Bench, respectively. Table 1 shows the number of requests that can be processed in parallel, the percentage of results required to be fetched back into the host memory, and the average time distance (*in microseconds*) between the host issuing request and the hosting using the request result. As we increase the number of buffer entries, Zipper can exploit more parallelism while facing diminishing returns. As Zipper harvests more operand reuse with larger buffers, the percentage of results that need to be fetched decreases as more request dependencies get relocated. The average time distance also increases as fewer results are fetched back to the host, giving the host more time to execute host-side codes in parallel. Note that this analysis assumes the system has perfect knowledge of the instruction dependencies during runtime. In practice, Zipper always fetches results back when the buffer entry gets recycled to ensure correctness. A larger buffer gives more time to continue execution until it needs to recycle a buffer entry.

We then analyzed the performance and logic overhead of various sizes. We construct our experiments around the buffer size of 4. The results are shown in Figure 6. The logic overhead increases exponentially as the size of the buffer increases because we need more logic for scheduling and more space to store results and operands. For NPB with Posit32, the speedup increases logarithmically as we put more entries in the buffer table. However, VIP-Bench with SE Enclave's performance only increases slightly with more buffer entries.

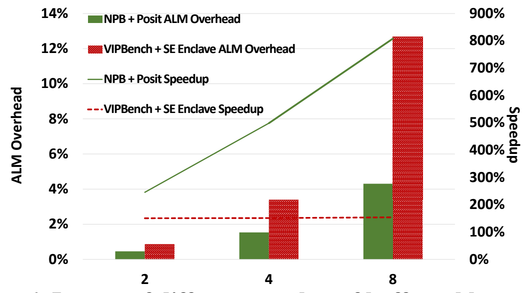


Figure 6: Impact of different number of buffer table entries on performance and area for Zipper.

The difference is attributed to the latency of each compute kernel. The Posit32 kernel takes two cycles to complete an instruction, while the SE Enclave kernel takes 24 cycles for each instruction. The SE Enclave is more compute-bound to the kernel itself.

6 LIMITATIONS AND FUTURE WORK

While Zipper demonstrates tremendous performance improvement over the baseline, there are additional improvements we can explore as the continuation of this line of work: (1) **Request Reordering**: Zipper leverages the optimization opportunities that applications present. However, there would be more temporal locality by reordering the requests and exploiting operator commutativity and associativity. To achieve this, Zipper can issue requests in batches after requests within a scheduling window have been optimized. (2)

Multi-Agent Cooperation: We considered the scenario with only one accelerator in this work. Multiple accelerators can cooperate to complete the computation in a more complex system. Zipper poises well to enable such extensions as the developer can use optimized scheduling algorithms to dispatch requests to different accelerators.

7 RELATED WORKS

With the emergence of heterogeneous and large-scale systems, communication latency between nodes has become a key focus.

There are four major approaches to tolerate latency: prefetching [1, 4, 6, 20, 21, 27, 29, 36], caching [12, 16, 23, 30, 34, 37], multithreading [3, 14, 32, 38], and relocating [15, 17, 22, 25, 31, 33, 35]. **Prefetching** predicts the memory access pattern and issues memory accesses before the data is used. This technique does not apply to the challenge tackled in this paper because accelerator requests often rely on host-side data-based control flow, making it hard to issue in advance. **Caching** keeps data closer to the compute by exploiting spatial and temporal locality. Being tailored specifically to CPU-accelerator interactions, Zipper is more flexible and area-efficient than cache-based designs. **Multithreading** hides access latency by allocating the hardware resources to another thread while waiting for the long-latency operation to complete. However, its benefits diminish when the operation is at or below the microsecond level due to context switch overhead. Moreover, multithreading relies on having enough threads to schedule and focuses on the throughput. In comparison, Zipper does not rely on switching to other work to occupy the host and significantly speeds up the end-to-end latency. **Relocating** (*i.e.*, in-memory/near-memory computing) is a design philosophy that moves compute closer to the data. However, even if placed near the memory, the system still needs to tolerate the latency between the host and the accelerator. As a result, this challenge is not directly addressed by relocation.

8 CONCLUSIONS

This paper details *Zipper*, a bus latency optimization framework for latency-sensitive accelerated applications. By carefully tracking CPU-accelerator dependencies, Zipper can exploit device- and request-level parallelism and temporal locality to significantly reduce exposed bus transaction latency. Zipper is implemented as a protocol optimization layer over an existing bus interface. Minimal system or programmer support is required, as Zipper uses runtime library support for dynamic scheduling and an additional hardware structure for executing the requests from the host. Zipper is deployed on Intel's HARPv2 platform, where two real-world accelerated applications are examined with and without Zipper optimizations. Zipper achieves a 1.5x-8x speedup with low logic overheads for the two case studies presented. This work demonstrates that protocol latency optimizations have significant promise to reduce the exposed latency of high-performance buses and widen their applicability to future application-acceleration opportunities.

*References

[1] Sam Ainsworth and Timothy M Jones. Software prefetching for indirect memory accesses. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 305–317. IEEE, 2017.

[2] Jasmin Ajanovic. Pci express 3.0 overview. In *Hot Chips Symposium*, pages 1–61, 2009.

[3] Haitham Akkary and Michael A Driscoll. A dynamic multithreading processor. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–236. IEEE, 1998.

[4] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857, 2020.

[5] AMD. Amd infinity architecture: The foundation of the modern datacenter. https://www.amd.com/system/files/documents/LE-70001-SB-InfinityArchitecture.pdf, Aug 2019.

[6] DW Anderson, FJ Sparacio, and Robert M Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967.

[7] ARM. Amba axi and ace protocol specification. version h.c. https://developer.arm.com/documentation/ihi0022/hc/?lang=en, Jan 2021.

[8] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, mar 2017.

[9] Lauren Biernacki, Meron Zerihun Demissie, Kidus Birkayehu Workneh, Fitsum Assamnew Andargie, and Todd Austin. Sequestered encryption: A hardware technique for comprehensive data privacy. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 73–84, 2022.

[10] Lauren Biernacki, Meron Zerihun Demissie, Kidus Birkayehu Workneh, Galane Basha Namomsa, Plato Gebremedhin, Fitsum Assamnew Andargie, Brandon Reagen, and Todd Austin. Vip-bench: A benchmark suite for evaluating privacy-enhanced computation frameworks. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 139–149. IEEE, 2021.

[11] Steven W. D. Chien, Ivy B. Peng, and Stefano Markidis. Posit npb: Assessing the precision improvement in hpc scientific applications. In Roman Wyrzykowski, Ewa Deelman, Jack Dongarra, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 301–310, Cham, 2020. Springer International Publishing.

[12] Jongsok Choi, Kevin Nam, Andrew Canis, Jason Anderson, Stephen Brown, and Tomasz Czajkowski. Impact of cache architecture and interface on performance and area of fpga-based processor/parallel-accelerator systems. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 17–24. IEEE, 2012.

[13] Ian Cutress. Intel shows xeon scalable gold 6138p with integrated fpga, shipping to vendors. 2018.

[14] Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, Rebecca L Stamm, and Dean M Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE MICRO*, 17(5):12–19, 1997.

[15] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The terasys massively parallel pim array. *Computer*, 28(4):23–31, 1995.

[16] James R Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th annual international symposium on Computer architecture*, pages 124–131, 1983.

[17] Daniele Ielmini and H-S Philip Wong. In-memory computing with resistive switching devices. *Nature electronics*, 1(6):333–343, 2018.

[18] Intel. Intel acceleration stack for intel® xeon® cpu with fpgas core cache interface (cci-p) reference manual. https://www.intel.com/content/www/us/en/docs/programmable/683193/current/acceleration-stack-for-cpu-with-fpgas.html, Nov 2019.

[19] Intel. Intel® xeon® processor scalable family technical overview. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html, Oct 2019.

[20] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 747–764, 2022.

[21] Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. Orchestrated scheduling and prefetching for gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 332–343, 2013.

[22] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 790–803, 2020.

[23] PAH Knoben. Software caching for tree-based algorithms on accelerator cards. Master's thesis, University of Twente, 2021.

[24] Yanqiang Liu, Jiacheng Ma, Zhengjun Zhang, Linsheng Li, Zhengwei Qi, and Haibing Guan. Megatron: Software-managed device tlb for shared-memory fpga virtualization. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1213–1218, 2021.

[25] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. Concurrent data structures for near-memory computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–245, 2017.

[26] Enno Luebbers, Song Liu, and Michael Chu. Simplify software integration for fpga accelerators with opae.

[27] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)*, 49(2):1–35, 2016.

[28] Patrick Patrick Kennedy. Compute express link cxl latency how much is added at hc34. https://www.servethehome.com/compute-express-link-cxl-latency-how-much-is-added-at-hc34/#:~:text=The%20CXL%20Consortium%20is%20using,170%2D250ns%20for%20CXL%20memory.&amp;text=If%20CXL%20seems%20to%20be,with%20Q2%202022%20Wind%2DDown., Aug 2022.

[29] R Hugo Patterson, Garth A Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 79–95, 1995.

[30] Christian Pinto, Yiannis Gkoufas, Andrea Reale, Seetharami Seelam, and Steven Eliuk. Hoard: A distributed data caching system to accelerate deep learning training on the cloud. *arXiv preprint arXiv:1812.00669*, 2018.

[31] Carlos Ríos, Nathan Youngblood, Zengguang Cheng, Manuel Le Gallo, Wolfram HP Pernice, C David Wright, Abu Sebastian, and Harish Bhaskaran. In-memory computing on a photonic platform. *Science Advances*, 5(2):eaau5759, 2019.

[32] Amir Roth and Gurindar S Sohi. Speculative data-driven multithreading. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 37–48. IEEE, 2001.

[33] Fabian Schuiki, Michael Schaffner, Frank K Gürkaynak, and Luca Benini. A scalable near-memory architecture for training deep neural networks on large in-memory datasets. *IEEE Transactions on Computers*, 68(4):484–497, 2018.

[34] Yakun Sophia Shao, Sam Xi, Viji Srinivasan, Gu-Yeon Wei, and David Brooks. Toward cache-friendly hardware accelerators. In *HPCA Sensors and Cloud Architectures Workshop (SCAW)*, pages 1–6, 2015.

[35] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. A review of near-memory computing architectures: Opportunities and challenges. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 608–617. IEEE, 2018.

[36] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, 1978.

[37] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[38] Lawrence Spracklen and Santosh G Abraham. Chip multithreading: Opportunities and challenges. In *11th International Symposium on High-Performance Computer Architecture*, pages 248–252. IEEE, 2005.

[39] Xilinx. Xilinx runtime library (xrt).

[40] Greg Yeric. Moore's law at 50: Are we planning for retirement? In *2015 IEEE International Electron Devices Meeting (IEDM)*, pages 1.1.1–1.1.8, 2015.