Paths to Fast Barrier Synchronization on the Node

Conor Hetland   Georgios Tziantzioulis   Brian Suchy   Michael Leonard
Jin Han   John Albers   Nikos Hardavellas   Peter Dinda

ABSTRACT

Synchronization primitives like barriers heavily impact the performance of parallel programs. As core counts increase and granularity decreases, the value of enabling fast barriers increases. Through the evaluation of the performance of a variety of software implementations of barriers, we found the cost of software barriers to be on the order of tens of thousands of cycles on various incarnations of x64 hardware. We argue that reducing the latency of a barrier via hardware support will dramatically improve the performance of existing applications and runtimes, and would enable new execution models, including those which currently do not perform well on multicore machines. To support our argument, we first present the design, implementation, and evaluation of a barrier on the Intel HARP, a prototype that integrates an x64 processor and FPGA in the same package. This effort gives insight into the potential speed and compactness of hardware barriers, and suggests useful improvements to the HARP platform. Next, we turn to the processor itself and describe an x64 ISA extension for barriers, and how it could be implemented in the microarchitecture with minimal collateral changes. This design allows for barriers to be securely managed jointly between the OS and the application. Finally, we speculate on how barrier synchronization might be implemented on future photonics-based hardware.

CCS CONCEPTS

• **Software and its engineering** → **Process synchronization**; **Runtime environments**; *Ultra-large-scale systems*; • **Computer systems organization** → *Multicore architectures*;

KEYWORDS

synchronization, collective communication, parallel computing, HPC

1 INTRODUCTION

The vast majority of parallel applications and execution models depend heavily on synchronization for correctness or as an intrinsic computational element. Their performance in turn depends on the latency of synchronization primitives, and this dependence becomes ever more critical as the scale of the parallel machine increases or the granularity of parallelism decreases, for example when strong scaling is needed.

Our focus is on collective communication at the **intranode** level, particularly the barrier primitive. The individual node is of course growing in scale and complexity. The node is also where the finest granularity parallelism exists now, and where even finer granularity parallelism is likely to be extracted in the future through compilers and other means. For example, task-based execution models are already seeing such small tasks that correspondingly low overhead task systems are needed. Others have already been argued that such systems require hardware support because memory system-based synchronization is simply not fast enough [4]. Indeed, entire tasking systems have been proposed to run at the hardware level [27]. Similar arguments have been made for hard real-time systems, where having low (and predictable) overheads in scheduling is necessary even for correctness. In pursuit of this goal, systems have been built that offload CPU scheduling to FPGAs [42].

In contrast to tasking models of computation, in which global synchronization can be minimized, we are interested specifically in computational models that demand global synchronization, and in particular barriers. There is a lot to be said for such models (indeed, the success of GPUs hinges on them), but they are not currently a good fit for the CPUs of modern nodes outside the limited domain of vector instructions. We seek to improve that fit.

This paper is a deep dive into hardware barriers on modern and future x64 machines. Can we make a modern or future x64 machine amenable to parallel applications and execution models that demand low-overhead global synchronization?

Our approach to this question starts in the near future, in the form of leveraging emerging reconfigurable hardware close to the processor. Reconfigurable computing has a long history, as exemplified by the IEEE's long-running Symposium on Field-Programmable Custom Computing Machines (IEEE FCCM), now in its 27th year. Throughout this history, a commonly envisioned hardware model has been that of a field programmable gate array (FPGA) as a

co-processor, operating in tandem with standard processors. This model is now on the threshold of widespread adoption, with FP-GAs becoming available in many forms, ranging from PCI-based accelerators, through processor socket-based implementations [11], to integration with processors in multichip modules, as in the Intel HARP platform we use in this paper. FPGAs are becoming more and more tightly coupled with processors, and this evolution may eventually extend to an on-die integration with processors, or even merging of FPGA resources with traditional processor logic and functional units. Can we use a closely coupled FPGA to give us a better barrier?

It may also be feasible to simply integrate a hardware barrier mechanism on a next generation x64 processor. Changing intra-chip and inter-chip communication technologies might also simplify the implementation of the primitive. Such integration may also be commercially viable. Arguably, the synchronization needs within the node for HPC-focused applications and execution models are closely related to the needs of more general parallelism models beyond the scope of HPC.

Our contributions are as follows:

- We motivate the need for a low latency hardware barrier mechanism on modern processors. We argue that current software barriers are slow, limit current application performance, limit novel parallel execution models, and can be much faster. Our analysis is partially based on the measurement of the performance of a microbenchmark, a widely used application benchmark (PARSEC StreamCluster), and an influential parallel run-time model (NESL) on four platforms, including four and eight socket x64s and an Intel Xeon Phi KNL.
- We describe the design and implementation of a barrier mechanism for the Intel Hardware Advanced Research Prototype (HARP) platform, which integrates a large FPGA and a modern x64 processor into a single multichip module. This includes an experience report of using such a platform for non-application use cases. Of particular concern is the latency of communication to and from logic in the FPGA.
- We evaluate our HARP-based solution against a suite of high performance software barriers for the microbenchmark, application benchmark, and run-time.
- We consider the integration of a hardware barrier primitive directly into an x64 processor. We present the design of a simple ISA interface to the kernel and application, and a plausible, minimalist microarchitectural implementation of it. Our design would allow an application to safely and securely use the mechanism without invoking the kernel in the critical path.
- We speculate on how hardware barrier synchronization could be readily added to a future processor design that uses photonic communication.

2 MOTIVATION

Barrier synchronization is a fundamental primitive of parallel computing in which all participating threads must synchronize ("arrive at the barrier") before any can continue ("depart from the barrier").

Our focus is on barrier synchronization among a group of threads within a single shared memory node.

If the time between barriers is large, the barrier implementation hardly matters. We are concerned with applications and parallel execution models in which barriers are frequent, and thus barrier implementation overhead is critical. This typically occurs when fine grain parallelism can be extracted and is desirable. Beyond the obvious desire for better strong scaling, others have made the case for and described approaches to fine-grain parallelism in OpenMP [3].

It is important to note that as the granularity of parallelism is reduced and the time between barriers shrinks, it is also much more likely that the common case is that participating threads arrive at a barrier with close synchrony. Various kernel-level coordinated scheduling models [17, 35] can further increase the chances of this case. Now, the barrier overhead, indeed, the minimum barrier cost, becomes increasingly critical. In the limit, granularity falls to one or close to it, giving us a SIMD machine or a GPU. There is a large gap between this point and what can be supported on a current processor and node. We aim to bridge that gap.

Current barriers are slow. Numerous algorithms for barriers exist [32] and in our target environment, we have implemented several in software on top of the hardware shared memory system, leveraging the hardware's general purpose coherence, consistency, and atomicity primitives. That is, these techniques are centered around the caches.

Figure 1 shows the performance of various software barriers, including those in typical practical use, on four distinct platforms, as a function of scale. *NUMA-8* is a Supermicro 7089P-TR4T, which sports eight 24 core, hyperthreaded 2.1 GHz Intel Xeon Platinum 8160 processors (384 hardware threads total), and 768 GB of RAM split among 8 NUMA zones. *NUMA-4* is a Dell R815, which sports four 16 core 2.1 GHz AMD Opteron 6272 processors and 128 GB of RAM split among 8 NUMA zones. *Phi* is a Colfax KNL Ninja platform. This is an Intel-recommended platform for Xeon Phi Knights Landing (KNL) development. It is essentially a Supermicro 5038ki, and includes a Intel Xeon Phi 7210 processor running at 1.3 GHz. The processor has 64 cores, each of which has 4 hardware threads (CPUs in our terminology). The processor is tightly coupled to 16 GB of MCDRAM, and more loosely to 96 GB of conventional DRAM. *HARP* is a prototype Intel platform that integrates a Broadwell Xeon processor and a large FPGA in a single socket (described in more detail in Section 3).

The barriers considered here and elsewhere in the paper include the default pthread barrier implementations (called Pthread), the default OpenMP barrier implementation in GCC with the libgomp run-time (OMP), a hand-tuned two generation counting barrier (Counting), a pool barrier (Pool), a ticket barrier (Ticket), a dissemination barrier (Dissemination), and a tournament barrier (Tournament). Testing is done here using back-to-back barriers with software threads locked to individual hardware threads. The machine is otherwise quiescent, so minimal or no context switching occurs, and we are considering minimum times, as these approach the intrinsic overhead of the barrier.

Note that on all of the platforms, the barrier latency is in ranges of thousands to tens of thousands of cycles at any kind of scale that approaches the scale of the machine. For NUMA-8 at 384 threads,
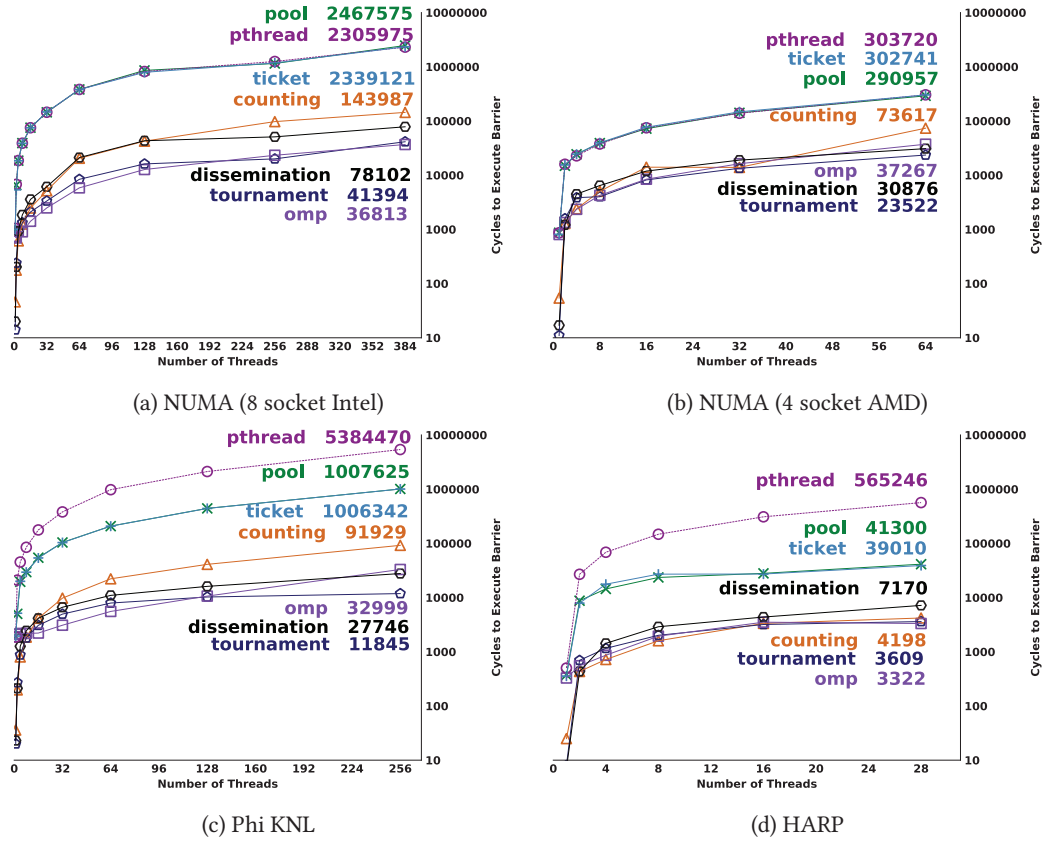
(a) NUMA (8 socket Intel)

(b) NUMA (4 socket AMD)

(c) Phi KNL

(d) HARP

Figure 1: Latency as a function of scale for various barrier implementations on different hardware.

the best software barrier (OMP) costs ~37000 cycles. For NUMA-4 at 64 threads, the best software barrier (Tournament) costs ~24000 cycles. On Phi at 256 threads, the best software barrier (Tournament) costs ~12000 cycles. On HARP, at 28 threads, the best software barrier (OMP) costs about ~3300 cycles. The tournament and counting barriers follow closely behind. These latencies place similar lower bounds on the granularity of parallelism that involves synchronization of many threads. One response is of course to adopt parallelism models that require no such synchronization. We argue that making the barrier faster is another effective response.

Current barriers limit current applications. The effects of barrier latency are felt today, particularly when strong scaling is desired. Figure 2 shows the performance of the widely used PARSEC 3.0 StreamCluster benchmark [6], given native input, on our four platforms using the different barriers previously described. The OpenMP barrier is not used, as we are using the PThread-based version of StreamCluster. The dissemination and tournament barriers are also not used here due to integration issues. In a StreamCluster iteration, one thread makes decisions and the threads must use a barrier to wait for these decisions to be made and also to synchronize work. This means that for a fixed problem size, scaling becomes critically dependent on the barrier cost.

As we can see from the figure, on all platforms the choice of barrier implementation has a profound effect on performance and

scalability, particularly as the core count grows. The fastest barrier considered here (Counting) can produce a speedup of as much as 123x over the worst barrier (PThread). This is on the NUMA-8 platform, which has the highest thread count available. Scaling simply stops after 32 threads with the pthread barrier.

Now consider the Phi platform more closely. The 7x speedup of StreamCluster going from PThread to Counting is due to the 59x speedup between these two barriers, as measured in our microbenchmarks (Figure 1). Yet the absolute cost of the best barrier on Phi (Tournament) is still ~12000 cycles, meaning there remain several orders of magnitude for improvement. The result would likely be enhance speedups of StreamCluster. Additionally, as we scale higher, we would expect the gain to increase.

Current barriers limit novel execution models. Barrier latency restricts the application of various execution models on a general purpose shared memory processor/node. For example, nested data parallelism has demonstrated considerable promise. In the classic implementation of the highly influential NESL language [8], the compiler generates code for an abstract vector stack machine (for arbitrary length segmented vectors), known as VCODE. The execution engine is an interpreter, compiler, or JIT of VCODE. The classic mapping was to a vector supercomputer, and current work maps VCODE to GPUs [5]. Non-NESL-based implementations of nested data parallelism, for example Futhark [20], also map to GPUs. A
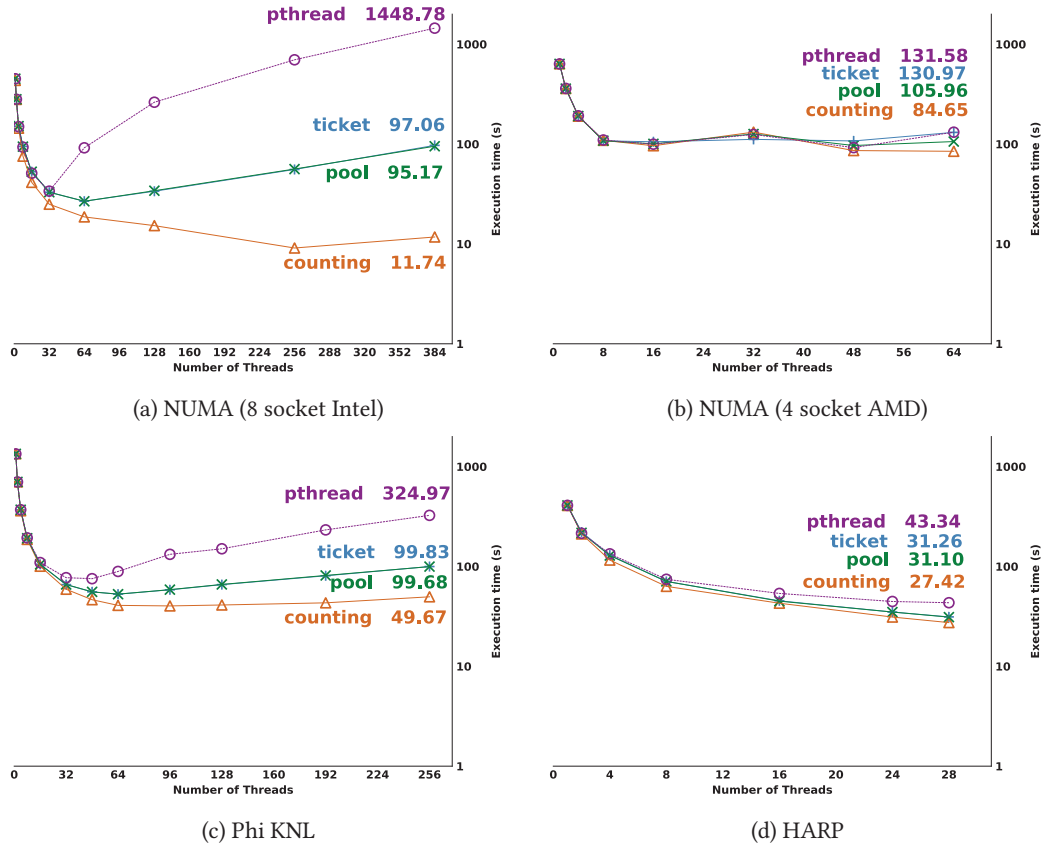
Figure 2: Performance of stream cluster given different barrier implementations on different hardware.

common reason is that GPUs provide inherent fast group synchronization.

We are developing an implementation of a VCODE interpreter for x64 machines that partitions the work of each VCODE instruction hierarchically, with the top level being implemented with thread parallelism, and the second level being implemented with x64 vector instructions. Barriers are used to enforce agreement among threads as to which specific VCODE instruction is currently being handled (control flow), as well as to stage execution steps to avoid races.

Figure 3 shows the cost of a VCODE NOP instruction in our interpreter for each of the barriers across the four platforms, demonstrating the absolute limits of the fine-grain parallelism our interpreter is capable of. We also include an Ideal curve, for which we've disabled barriers, to measure all other interpreter overhead. Our implementation requires multiple barriers per VCODE instruction; it is based on top of pthreads or kernel threads, and thus no OpenMP barrier is shown. The best software barrier is tournament barrier on all four machines.

As can be seen, barrier latency dominates the overhead of VCODE instruction execution. On a modern x64 such as the HARP, the interpreter needs O(100) cycles to decode and handle a VCODE NOP, but takes O(10000) cycles with proper synchronization in place, using the best software barrier we have found. Each of our machines demonstrates this difference of two orders of magnitude. The high

barrier latency essentially limits the granularity of computation that a VCODE instruction can support, which we believe can be supported down to the O(100) cycle mark.

Barriers should and can be faster. There is nothing intrinsic to the barrier primitive that requires the kinds of latencies that are seen in practice with even the best software barrier implementations. Fundamentally, a barrier primitive is an AND gate. At the hardware level, the time costs of transporting a logic level to and from the construct across a processor chip or between processors is what should be limiting barrier latency. That is, a barrier is really limited by the latency of the wires. A within-processor barrier should be able to operate within cycles, not tens of thousands of cycles.

It is true that there are subtleties in barrier design, whether in software or hardware. For example, multiple generations are typically needed to handle races from the release from one barrier to the arrival to the next. However, these are unlikely to significantly change the costs since they also involve very minimal logic and do not change the wire lengths.

Similar reasoning has been previously applied to barriers and collective communication among distributed memory nodes, resulting in specialized hardware for these environments that had much lower latency than implementations that relied on commodity communication hardware. Examples are given in Section 1.
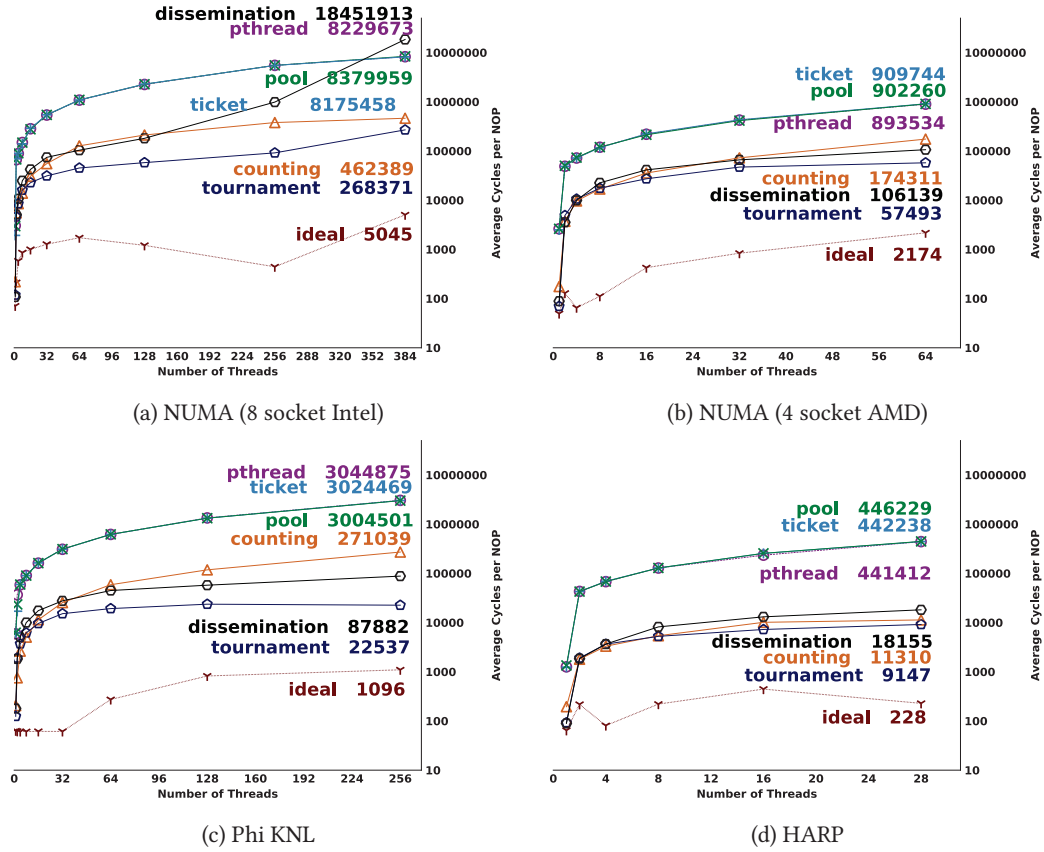
Figure 3: NESL VCODE instruction execution latency for various barrier implementations on different hardware..

3 HARP BARRIER

We designed, implemented, and evaluated a hardware barrier on the HARP, a prototype Intel platform that integrates a modern x64 processor and a large FPGA in a single package.

3.1 HARP platform

The HARP multichip module fits in one socket of a server class motherboard. The server motherboard itself has a single socket, a single NUMA zone, and 64 GB of DRAM. It is important to understand that both hardware and software barrier performance is limited by the performance characteristics of the processor (particularly caches), the FPGA, and the processor↔FPGA interconnect, all of which are within the multichip module.

Figure 4 illustrates the multichip module. The processor is from the Intel Xeon E5-2600 v4 family, which is based on the Broadwell microarchitecture [33]. The processor operates at 2.4 GHz, and has 14 cores, each with two hyperthreads, for a total of 28 hardware threads. The cache hierarchy includes split 32 KB L1 caches, 256 KB L2 caches, and a 35 MB shared last level cache. For timing, we use the cycle counter (i.e., the rdtsc instruction and friends), which on this hardware runs at a constant rate regardless of DVFS.

The FPGA in the multichip module is an Intel Altera Arria10 GX 1150 [43], which is among the largest and fastest available FPGAs. It includes 1.15 million equivalent logic elements. The logic in the
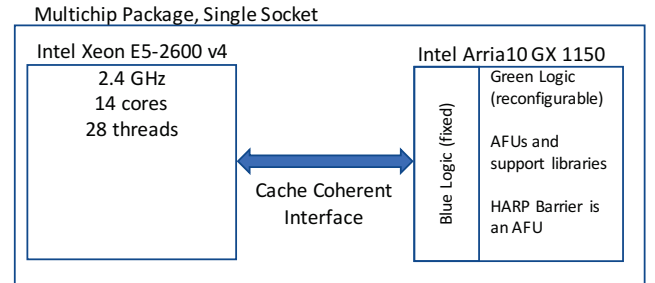
Figure 4: HARP Broadwell+Arria multichip module.

FPGA is partitioned into a "blue" component, which is proprietary and implements, among other things, the interface to the processor, and a "green" component, which is dynamically reconfigurable by the user. Our hardware barrier is implemented as an application functional unit (AFU) that resides in the green logic. Our design is based on the base clock provided to an AFU, which operates at 400 MHz. Six cycles pass on the processor for every cycle in our AFU. We implement directly in the hardware description language Verilog, using Quartus Prime Pro 16.0.0, a common commercial synthesis toolchain for this kind of FPGA. Resynthesizing our AFU takes 1-2 hours.

The FPGA, via the blue logic, is a full fledged participant in the cache coherence network of the processor. An AFU can present control/status registers (CSRs) to the processor via memory-mapped I/O, which the kernel modules can in turn map directly into a process's address space. An AFU can also execute main memory reads/writes itself, as well as be the target of them. The cache coherent interface makes the FPGA (and hence the AFU) a full peer with the processor in terms of memory. There is also a mechanism, UMsg, for low-latency messages between the FPGA and CPU which piggy-backs on the interface for maintaining cache coherence.

An AFU also implements a set of well-known CSRs, analogous to the configuration space on PCI devices, that allows introspection and rendezvous with software components. We build software components using Intel's Accelerator Abstraction Layer (AAL), a C++ framework designed to facilitate software use of available AFUs via this mechanism.
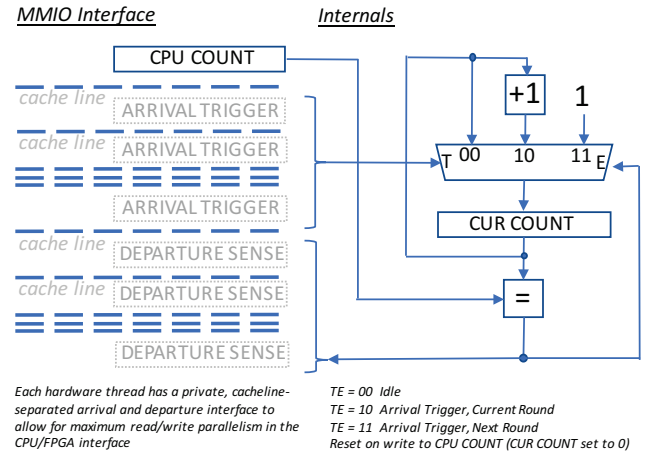
The server runs Ubuntu 14.04 with the Linux kernel being version 3.13-0-123. Kernel support for the FPGA components takes the form of Linux kernel modules. It is important to understand that while software and hardware *rendezvous* via the kernel modules, in actual *operation*, user space software and AFUs interact without kernel involvement. Operation of our hardware barriers involves complete OS bypass—interactions with our AFU are done using single `mov` instructions within the application code.

3.2 Design and implementation

The design of our barrier is split between a performance critical component, which is implemented in FPGA hardware as an AFU, and a software interface that interacts with the hardware. The design is based around a counting barrier model with atomic arrival counting and termination count notification pushed into hardware. The software side also handles a common race condition encountered in counting (and other) barrier designs by making use of two copies of the hardware. While other hardware barrier designs are certainly possible, and indeed, more in keeping with the parallelism provided by an FPGA, Amdahl's Law applied to this specific platform suggested to us that the critical path would not be in the FPGA hardware, but rather the interaction with it. As such, we explored very nearly the full range of possible means to communicate between the CPU and FPGA, while leaving the barrier mechanism relatively untouched.

Shared hardware. Figure 5 illustrates our hardware design (on the right) and its interface to software (on the left). Two copies of this design are instantiated in the FPGA, and the software alternates between them. One is handling the threads departing from the "current" barrier, while the other is handling the threads arriving at the "next" barrier. We now consider just one copy.

At its core, the hardware increments an internal counter register, `CUR COUNT`, each time an arrival trigger CSR is written by the software. When `CUR COUNT` equals the size of the barrier, the hardware resets `CUR COUNT` and signals that a barrier departure is in progress to the software. There is some slight subtlety in handling the first arrival after a hard reset, versus the next arrival after a previous departure, and this is captured in the three input multiplexer shown in the figure. As a whole, however, the hardware is very simple.

Each hardware thread has a private, cacheline-separated arrival and departure interface to allow for maximum read/write parallelism in the CPU/FPGA interface

TE = 00 Idle
TE = 10 Arrival Trigger, Current Round
TE = 11 Arrival Trigger, Next Round
Reset on write to CPU COUNT (CUR COUNT set to 0)

Figure 5: HARP barrier hardware and its MMIO interface. Two copies are instantiated. Software alternates between the two copies, similar to a classic counting barrier. A wide range of alternative interfaces to the internals were also implemented and evaluated.

Not shown is set of CSRs that allow configuration of how the AFU will be informed of barrier arrivals and how it will inform the CPU of barrier departures.

Read and write parallelism. To allow the hardware to process barrier arrivals and signal barrier departures efficiently, we need to leverage parallelism within the memory system, specifically the cache coherent interconnect that integrates the processor and FPGA. Although both the processor and the blue logic within the FPGA are black boxes to us, we were able to infer that such parallelism does exist. Because the interconnect operates at the level of cache lines, we can access this parallelism by segregating the targets of design-independent read/write operations into distinct cache lines. It is vitally important that we do this. Empirically, if we do not, performance suffers dramatically.

Consider a simple interface in which a barrier arrival occurs by writing a `ARRIVAL TRIGGER` CSR, and barrier departure is sensed by reading a `DEPARTURE SENSE` CSR. In the MMIO interface, hardware handles a write to this CSR by advancing the current count as described above. Our design synthesizes to operate within a single cycle of the 400 MHz clock. The trick is to keep it fed with arrivals at this rate. If `ARRIVAL TRIGGER` is mapped to a single MMIO address, then a write to this address from one hardware thread may have to wait on a write to the address from another hardware thread due to the nature of the coherence protocol. This will occur even if the blue logic is capable of feeding the writes into our AFU at the full rate. In effect, by having every hardware thread use the same address for `ARRIVAL TRIGGER` we create false sharing for the coherence protocol, artificially limiting the rate at which we can push arrivals into our hardware. To ameliorate this, we replicate `ARRIVAL TRIGGER` in the MMIO space, separating the replicas by a cache line, the unit of operation of the coherence network. Each hardware thread then writes to its own, private replica of `ARRIVAL TRIGGER`, avoiding creating any potential false

sharing within the memory system logic, and giving the potential to feed our hardware with arrivals at the maximum rate it can support. Similar reasoning applies to the design of the MMIO interface for the DEPARTURE SENSE, hence its replication across cachelines.

Arrival triggering. There are multiple means available to communicate an arrival to the AFU, of which we considered CSR writes, UMsg and UMsgH writes, and FPGA-based memory polling. Each of these methods have their own subtleties and implications when being used to receive low-latency messages in parallel.

In CSR write triggering, a hardware thread simply writes to its own private cache line to access the ARRIVAL TRIGGER CSR as in Figure 5.

In UMsg and UMsgH triggering, a hardware thread sends data to the FPGA by writing specially reserved addresses in a special UMAS region that in turn is visible to AFU via a specialized interface. This is intended by Intel to be a low latency path, and is administered via QPI or UPI as the interconnect, directly leveraging the same hardware used to maintain cache-coherence across the CPU and the FPGA. Our understanding is that the UMsgH is the equivalent of a cache-line snoop/invalidation, whereas a UMsg is the equivalent of the transfer of a new cache-line upon invalidation.

Unfortunately, the number of simultaneous addresses currently supported for UMsg/UMsgH is only eight, which is a severe limitation as a serviceable barrier needs to scale up to a larger number of hardware threads. Even worse, concurrent writes to the same UMsg/UMsgH address can result in a single UMsg/UMsgH or a lost write. Without resorting to an atomic write to the UMAS region address (invalidating the point of a hardware barrier in the first place), arrival triggering in this way is likely to be incorrect if the eight UMsg/UMsgH addresses are shared. Even when limited to four hardware threads, reserving one address for each generation, UMsg/UMsgH triggering did not produce significant performance gains over CSR write triggering.

In FPGA-based memory polling, the AFU directly polls target addresses in main memory, waiting for the CPU to write to them to signal arrival. This has no restrictions in terms of the number of addresses that can be monitored, and is supposed to have the same latency as a UMsg. Of course, polling main memory across multiple locations is likely to be problematic.

Departure sensing. We also experimented with different means of AFU→CPU messaging, namely CSR polling, and AFU-based memory writes across different interconnects.

In CSR polling, each hardware thread polls, via its own private cache line, the DEPARTURE SENSE output of Figure 5.

In AFU-based memory write polling, each hardware thread simply polls a main memory location (per-thread or shared) that is written by the AFU when the barrier is complete. The write can be executed across several interconnects. We found no significant difference between using "VA" (the automatically scheduled interconnect), "VH0" and "VH1" (the PCI-e lanes), or "VL0" (QPI/UPI), or using any intentional mixture/parallelism of the channels.

Software. The software interface provided to the programmer is simple and its implementation is dominated by AFU discovery, loading, and configuration concerns. The barrier wait function itself simply triggers the barrier and waits for a barrier departure using

the configured mechanisms. It also alternates between two copies of the hardware. The code is carefully engineered so that these operations involve, at heart, single, user-level mov instructions.

3.3 Evaluation

Our AFU is a single cycle design that operates at 400 MHz, while the processor operates at 2.4 GHz. One cycle of our AFU takes six CPU cycles. Consider the case where the interconnect and the blue logic can feed our hardware at the maximum rate. Assume that all CPUs arrive at the barrier simultaneously, which is what we are trying to optimize for. The best possible case for any barrier hardware then occurs when every CPU does its write to its ARRIVAL TRIGGER before any do their read from their DEPARTURE SENSE For 28 hardware threads, we would then expect this to take $2 \times 28 = 56$ FPGA cycles, or $56 \times 6 = 336$ CPU cycles. This is an order of magnitude faster than the best software barrier we considered in Section 2.

This analysis ignores aspects of the latency through the interconnect; however, it is doubtful that the hardware provides perfect parallelism in the interconnect. Our analysis suggests it provides 8-way parallelism at best. Even if we were to improve our core barrier, for example, via a tree-based model, Amdahl's law tells us the speedup would likely be minimal compared to our simple hardware since the critical path is not likely to be the barrier hardware.

Our measurements bear this out. In Figure 6(a) we show microbenchmark results for each interesting configuration of the HARP barrier. While there are differences, none of the HARP barrier configurations are able to beat the counting barrier and OpenMP barrier significantly. The intrinsic barrier cost of 336 cycles is swamped by the 3000+ cycle cost of communicating with the AFU, leading to lower performance than our analysis would suggest. These results are reflected in the StreamCluster benchmark (Figure 6(b)) and NESL execution rate (Figure 6(c)).

In Figure 7 we show the microbenchmarks for all of the barriers considered in the paper on the HARP platform, except Dissemination and Tournament, which are no better here. We also include a projection of HARP barrier performance that assumes the above analysis and sets the latency of communication with the barrier hardware to be on par with last-level cache latency. A hypothetical HARP system that could provide this latency to the FPGA would be able to perform barriers at full scale over three times faster than the best software barrier on the platform.

Improving HARP. In our opinion, there are two primary issues HARP presents in allowing for the effective implementation of barrier synchronization and other similar primitives that are latency sensitive. First, there is no CPU↔AFU communication path that has the latency that might be expected given the close physical proximity of the two. Second, there is limited parallelism for traffic going into and out of the FPGA itself, thus causing queuing that compounds the latency limitation. If a highly parallel, low latency path for signaling small amounts of data to/from the AFU from every hardware thread were made available, implementing a far-faster-than-software barrier on the FPGA side would be extremely plausible. Even a single, low-latency pin from each hardware thread would make a huge difference.
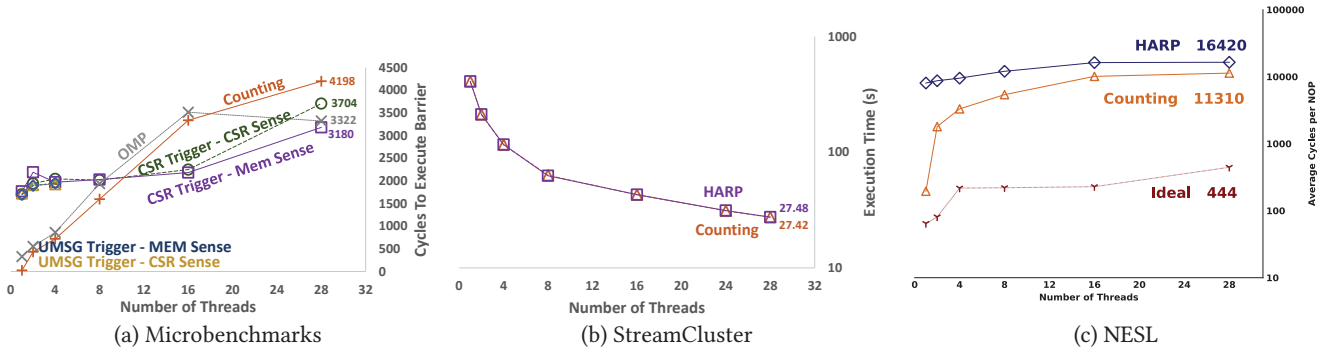
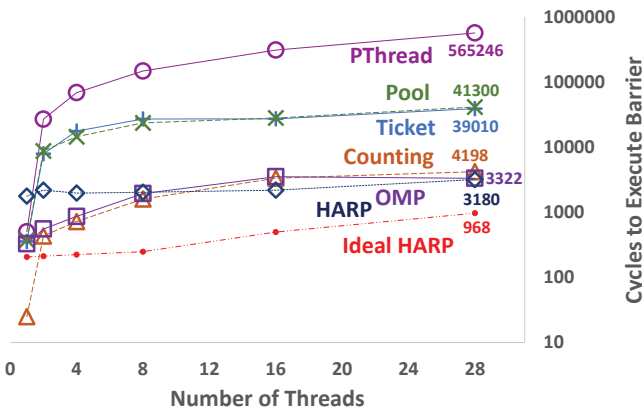| (a) Microbenchmarks | (b) StreamCluster | (c) NESL |

Figure 6: HARP barrier performance results.

Figure 7: Barrier microbenchmarks on HARP for all software barriers, HARP barrier implementation, and HARP barrier implementation assuming ideal communication latencies between the CPU and FPGA.

4 ON-PROCESSOR IMPLEMENTATION

We now consider how fast hardware barriers could be integrated into the processor chip and processor complex itself, focusing on adding subset barriers to x64. This model is likely to result in far lower latencies than would be possible with reconfigurable hardware, even after latency optimizations such as we describe in Section 3.3 are applied. On the other hand, the model requires careful consideration of security and protection because the instruction set architecture (ISA) is changed, and an implementation should require minimal changes in the existing microarchitectural design.

4.1 ISA and kernel integration

Our proposed x64 ISA extension pushes as much responsibility for barriers as possible to the kernel. Only the critical fast path of executing a quickly-completing barrier wait is invoked directly by the application. The design is influenced by Intel's Cache Allocation Technology (CAT) ISA extension [22, Volume 3, Chapter 17], Intel's TSC Deadline Mode feature for the Advanced Programmable Interrupt Controller [22, Volume 3, Chapter 10], and Intel's Hardware Lock Elision (HLE) feature [22, Volume 1, Chapter 14]. The

software-visible structures have been selected to minimize the degree of coordination among hardware threads of a single processor and across the entire processor complex, with the goal that the only coordination is that needed to do the actual barrier synchronization.

Drawing on CAT's interface as a model, the processor will report, via a cpuid leaf, the maximum number of barrier participants (n) and the maximum number of simultaneous subset barrier groups it supports (m). n is defined to be the number of hardware threads the processor provides plus the number of peer processors it supports. A processor with 8 cores, hyperthreading, and quad-socket support would report $n = 8 \times 2 + 3 = 19$. m is implementation-specific.

A region of m model-specific registers (MSRs) will be added to the MSR address space at a given base. Each MSR will represent a subset barrier group, and will consist of a vector of n bits and a timeout value t, in cycles. The ith subset barrier group will be represented by the MSR at offset i from the base.[1]

MSRs are conceptually per-hardware thread constructs. The kernel is responsible for managing barriers across hardware threads and processors, which essentially means maintaining the MSR contents across these. To create and initialize a barrier, the application makes a system call:

```
int bar = create_barrier(thread_list, timeout);
```

In response, the kernel will (a) determine if the application's desired timeout is acceptable given the kernel's global constraints, and (b) attempt to find an unused barrier group MSR to allocate.[2] If both are successful, it will use privileged wrmsr instructions to copy the barrier group and timeout to the MSR. The barrier is now known to the hardware. The kernel will then interrupt all other hardware threads, forcing them into the kernel to install the new MSR.

At this point, all threads will also invoke a *privileged* barrier init instruction, barinit %rax, with the index of the new barrier being the argument. This will reset the barrier for first use. Finally,

---

[1]As MSRs are only 64 bits wide, and we can only expect n to grows with time, the obvious clustering approach will be used as needed. For example, on a Phi KNL there are 256 hardware threads, so groups of five 64 bit MSRs would be used to represent the 256 bits and the 64 bit timeout value. i would then be scaled by five. The actual cluster size would be provided by a CPU ID leaf.

[2]The bit vector in the MSR represents the processor-local hardware threads, plus the remote sockets, hence on a multi-socket machine, the kernel also needs to translate the application's flat request into this two level hierarchy in the obvious way.

the kernel will return the MSR index it selected to the calling application, which it shares with its other threads through existing mechanisms.

At this point, the barrier is set and ready to use by the application. Invoking the barrier is done by an *unprivileged* instruction,

```
barwait %rax
```

where the argument is the previously returned index. In the common case, all threads participating in the barrier are currently scheduled and invoke their `barwait` instructions with near simultaneity. The result is that these threads execute a barrier with only minimal overhead.

Because `barwait` is both unprivileged and blocks the hardware thread, the hardware and kernel must protect against attacks, bugs, or simply surprising delays that would cause starvation. The purpose of the timeout is to handle these uncommon cases correctly. Recall that the kernel vets any choice of timeout the application asks for, hence it can easily have a policy about the maximum allowed timeout. If during `barwait` any hardware thread's cycle counter reaches the timeout point, the hardware will raise a local exception for the `barwait` instruction, invoking the kernel (an interrupt at this time will also invoke the kernel). At this point, the kernel interrupts all the other participating hardware threads. Every participating hardware thread is now in the kernel either due to a timeout of its `barwait` or due to an interrupt from some other hardware thread that has timed out.[3] The kernel then coordinates its action across the threads. The coordination model would depend on the scheduling model, but, for example, the kernel can simply do a rescheduling pass and then resume by restarting all the interrupted instructions. Another alternative might be to raise a signal to the application. The application could use such signals to dynamically choose among barrier implementations—if the hardware barrier is frequently timing out, it's the wrong implementation to use or the workload has become unbalanced.

To encode the `barinit` and `barwait` instructions, we use an approach similar to HLE: we overload an existing single operand instruction by using an existing instruction prefix that currently has no meaning when used with it.

4.2 Microarchitectural model and analysis

The proposed electronic barrier implementation requires only minimal hardware modifications. The hardware can implement the m barriers using a $2n \times m$-bit register file that provides on-chip hardware storage for all m barrier MSRs (Figure 8). Although MSRs are per-hardware thread constructs at the ISA level, in our *implementation*, we think of the barrier MSRs (or alternatively the internal state they front) as being shared by the hardware threads. At the hardware layer, each barrier MSR utilizes n bits to store the participation bitmask P, and an additional n bits to store the current state of the barrier S (i.e., which participating hardware threads and sockets have arrived at the barrier).
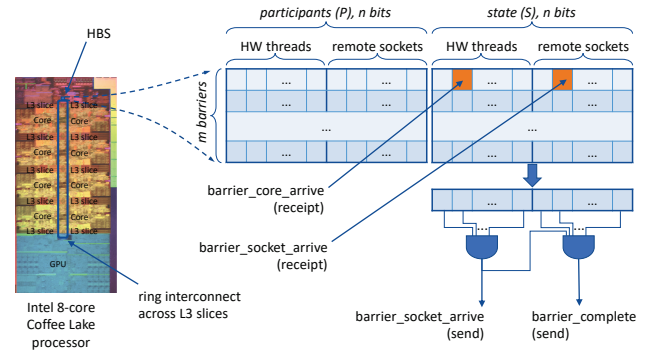
Figure 8: Hardware Barrier Structure (HBS). The example shows an HBS placed at the system agent along the L3 ring on an Intel Coffee Lake processor.

Only one such hardware barrier structure (HBS) needs to exist in a processor, as barriers are a resource shared among all cores/hardware threads and processors in a system. This achieves two significant benefits: (a) the HBS can act as a serialization structure for the entire processor, which simplifies the handling of race conditions, and (b) the HBS has a negligible silicon area overhead. For example, a 6T SRAM bitcell at 14nm finFET technology can be implemented at a mere $0.05\mu m^2$ [24]. At that density, a 4-socket system with 28 cores per socket with hyperthreading (similar to an Intel Xeon Platinum 8180) could implement support for 128 simultaneous subset barriers using only 944 bytes of storage, with an area occupancy of just $378\mu m^2$. To put this in perspective, one could fit more than $22K$ barriers in only 0.01% of the Xeon Platinum 8180's die. Due to its very small size, the HBS would also have negligible energy and power requirements.

The HBS can be placed at a location that all cores can reach using the existing interconnect, thereby obviating the need for expensive dedicated wires. In processors with a shared L3 cache, for example, the HBS could be placed at the L3 cache controller or along the L3 interconnect, making barriers roughly as expensive as an L3 cache access (around 42 cycles for Intel's Skylake and Coffee Lake processors at 14nm [45]). For the architectures in Figure 1, in particular, placing the HBS at the L3 cache controller would result in hardware barrier latency within a single socket of about 20 cycles for the AMD Opteron 6272 processor [12], and 38 cycles for the Intel Xeon E5-2600 v4 that is in the HARP [9]. A hardware barrier across multiple sockets would exhibit higher cost, dominated by the inter-socket latency. For the 4-socket NUMA AMD system in Figure 1, a roundtrip on a hyper-transport interconnect [13] in ring topology would take $4 \times 44ns \times 2.1cycles/ns = 370$ cycles. Thus, a hardware barrier across all four sockets would take an estimated 390 cycles, including the L3 access. In the case of Intel-based multi-socket systems, we estimate the latency of the QPI interconnect at around $30-110$ cycles. For a 4-socket ring, we estimate the latency of a hardware barrier to be $38 + 4 \times 110 = 480$ cycles. In either case, the expected latency for a hardware barrier is two orders of magnitude lower than the latency we measure on the 4-socket NUMA system in Figure 1.

---

[3]Depending on how closely clocks can be kept synchronized, most interrupts could likely be avoided by simply waiting. However, if a hardware thread has not yet advanced into the barrier wait instruction an interrupt is needed.

On a machine such as the Intel Xeon Phi 7210 processor [39], where there is no shared last-level cache, the HBS could be co-located with the coherence directory controller or the memory controller, or even placed somewhere along the chip's interconnect, achieving similar on-chip latency of a few tens of cycles. Placing the HBS near the center of the 36-tile mesh interconnect of the Phi results in a barrier latency of about 6 interconnect hops, which we estimate at 48 cycles (assuming a 1-cycle wire traversal and 3-cycle router/switch delay per hop). Placing it at the distributed cache coherence directory or the memory controller would result in no more than 64-88 cycles for a hardware barrier versus the ~33000 for a software barrier.

The hardware can utilize the existing interconnect to communicate with the HBS by introducing five new message types (create, initialize, core arrive, socket arrive, and complete). Upon calling create_barrier(), the core executes a wrmsr instruction to copy the barrier group and timeout to the allocated MSR. This causes the hardware to send a *barrier-create* message to the HBS along with the index to the allocated MSR and the bitmask of the participating hardware threads and remote sockets. The HBS then fills the P participation bitmask of the hardware MSR by storing 0s to all bit positions that correspond to participating hardware threads and remote sockets, and 1s to all other positions. In addition, the HBS copies the P participation bitmask to the S barrier state.

A barwait instruction causes the executing core to send a *barrier-core-arrive* message to the HBS, which sets the corresponding bit in the S state bitmask to 1, indicating that this core arrived at the barrier. All bits in the MSR's current state S are then fed into an AND gate, which calculates whether all participating cores arrived at the barrier. If they did, the HBS sends a *barrier-socket-arrive* message to the remote sockets that participate in the barrier (known by inspecting P) indicating that all participating cores in this socket arrived at the barrier. This result is further ANDed with the S bits that correspond to remote sockets. A *true* result signifies that all participating cores and sockets have arrived at the barrier. In that case, the HBS sends a *barrier-complete* message to all cores within its socket, signaling that the barrier is complete, and copies the P participation bitmask back to the S state bitmask to prepare for a new round.

Upon receiving a *barrier-socket-arrive* message from a remote socket, the HBS sets to 1 the corresponding socket bit in S to record that all cores in the remote socket arrived at the barrier, and also checks whether the barrier has completed. The execution of a barinit instruction sends a *barrier-init* message that causes HBS to store 0 at the corresponding S location, resetting the barrier state for the executing core.

5 SPECULATION: SILICON-PHOTONIC ON-CHIP BARRIERS

A highly efficient hardware barrier mechanism could be readily implemented on future processors that use photonics to transfer data. In effect, the transfer mechanism can provide a natural "wired AND" behavior that the barrier is based on. Such a barrier could operate in 10s of cycles or fewer.

Silicon photonic interconnects utilize light to transmit information. In a typical silicon photonic interconnect shown in Figure 9(a),

(a) Core 3 has not arrived at the barrier

(b) All cores have arrived at the barrier
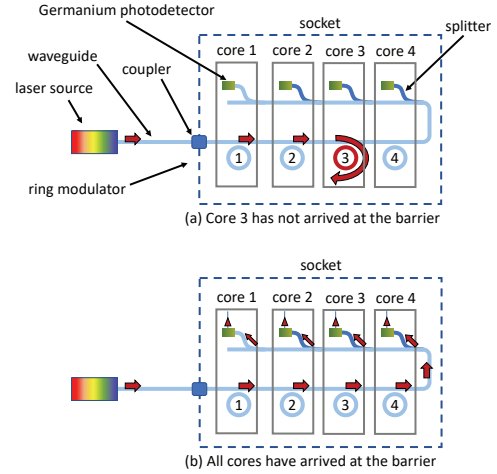
Figure 9: On-processor hardware barrier implementation with silicon photonics.

an off-chip laser source produces coherent light at a certain wavelength (typically in the C-band), which is coupled onto an on-chip silicon waveguide in which light is transmitted. Small rings made from the same material as the waveguide are placed close to the waveguide. These rings act as light modulators. They are tuned to resonate at the exact wavelength that the laser produces. When a ring is in the *ON* state, i.e. a voltage is applied to it, it couples light at its resonant wavelength and does not allow it to proceed on the waveguide. When the ring is in the *OFF* state, light is transmitted on the waveguide past the ring. The example in Figure 9(a) shows rings 1, 2, and 4 in the *OFF* state, and ring 3 in the *ON* state. Ring 3 stops light from being transmitted on the remainder of the waveguide.

To implement an on-chip optical barrier, the silicon waveguide passes through all the cores in a socket, and each core controls one ring modulator. Upon initializing the barrier, all participating cores set their rings to the *ON* state, thereby *preventing* the propagation of light. When a core arrives at the barrier, it switches off its ring modulator, *allowing* light to pass. Only when all cores arrive at the barrier will all the rings be in the *OFF* state, thus allowing light to propagate through all the cores. In effect, one traversal through all the cores acts as a large AND gate over the *barrier-core-arrive* messages. Light propagating beyond the last core is equivalent to sending a *barrier-complete* signal. All cores need to receive that signal. To achieve this, the waveguide curves around and traverses all the cores a second time. On this second traversal, each core taps the waveguide by splitting the light with an optical splitter and redirecting some of the optical power to a Ge-doped photodetector. When photons hit the Germanium the photodetector produces an electrical charge, which can then drive transistor logic (Figure 9(b)). Detecting light at this stage is equivalent to receiving a *barrier-complete* message.

The description above discusses only one barrier and uses only one wavelength to implement it. Multiple barriers map to different wavelengths. These wavelengths can be produced by a wideband multi-wavelength laser source, and they can be multiplexed

within the same waveguide with Dense Wave Division Multiplexing (DWDM). Up to 40 discrete wavelengths can be multiplexed within a single waveguide today, with projections of 64-way DWDM coming in the near future [34]. The optical devices are also relatively small, fast, and energy efficient. On-chip waveguide arrays can be implemented at a $20\mu m$ pitch [28]. Silicon nitride waveguides (Si_3N_4) have high light confinement, offer low intrinsic optical loss in the C-band ($0.4dB/cm$), and can achieve superior reproducibility in a CMOS-compatible platform [18].

The low-latency of optics allows light to traverse even a long $43cm$ waveguide at only $5ns$ latency at energies approximating $1pJ/bit$ [15]. Rings with diameter of a few tens of μm and Ge-doped photodetectors have been manufactured and demonstrated to handle energy-efficient modulation/demodulation at $317fJ/bit$ at speeds higher than $10GHz$ [40]. The optical barrier could be easily extended to support multiple sockets by simply running the waveguide across sockets using a board-level waveguide. The costs of going off-chip optically are considerably lower than doing so electronically.

All these photonic components are compatible with today's CMOS processes, and have been integrated alongside CMOS devices on the same die [40]. Thus, we argue that an optical barrier implementation is feasible even with today's technology, and it is likely to exhibit performance that surpasses even a processor-integrated all-electronic implementation (Section 4).

6 RELATED WORK

Collective communication (c.f. [29]) including barriers, scans, and reductions, is an important concept in parallel computing and has seen significant work over decades.

Classic and modern distributed memory parallel machines such as the Cray T3E [37], Thinking Machines CM5 [30], Ultracomputer [19], iWarp [10], and Blue Gene/L [14] either specifically included physically separate low latency networks, geared for collective communication instead of data transfer, or made it possible to execute carefully planned collective communication on a shared data network that could be configured for low latency use. Clever, inexpensive hardware for commodity clusters, such as the Purdue PAPERS network [16], made possible a separate, cheap, low-latency collective communication network alongside commodity Ethernet. Infiniband's low latency enabled innovation in collective communication without adding hardware to clusters that are already built around this network technology for data transfer [21, 23], including for barriers [25]. The idea of specialized hardware support for internode collective communication is still being investigated, most recently in the context of network interfaces that provide a malleable hardware substrate, for example an FPGA [2]. Recent work provides collective communication that is adaptive to the environment [31]. In contrast, the focus of this paper is on collective communication, specifically barriers, on a shared memory machines such as chip multiprocessors and NUMA machines.

Collective communication, and barrier synchronization more specifically, is also a well-studied problem in the shared memory context as well. For software-based barriers on multiprocessors, Mellor-Crummey and Scott [32] is the classic work, both considering lock-based models and inventing lock-free/wait-free data

structures for barrier and other synchronization. The focus of this paper is on hardware-based barriers for such machines.

The desirability of specialized hardware for fast collective communication, including barriers, for shared memory machines, particularly chip multiprocessors, NUMA machines, and NoC-based processors has been pointed out before [26]. Most recently, Tang et al showed how to use the programmer-exposed NoC primitives on the SW26010 processor of the Sunway TaihuLight supercomputer to accelerate the processing of contended locks by an order of magnitude [41]. We have added a new motivation for such hardware, namely fine-grain parallel languages such as nested data parallel languages like NESL. Several works have explored the nature of such hardware [36, 38] of which the work of Abellan et al [1] is probably the closest. The present paper contrasts with such prior work in three ways. First, we study the design and implementation of a hardware barrier mechanism on a specific, compelling new prototype hardware, the Intel HARP, that has the potential to become a mainstream product. Second, we analyze and report on the capabilities of the HARP prototype platform with respect to latency-constrained functions such as barrier. This has utility beyond the specifics of barrier implementation itself. Finally, both our HARP work, and the design of our on-processor implementation (Section 4) focuses specifically on x64 platforms.

Our speculative design for silicon-photonic on-chip barriers is influenced and informed by the work of Binkert et al [7], which introduces nanophotonic barriers, and the broader context of Vantrease's thesis [44].

7 CONCLUSIONS

We have described our deep dive into hardware barriers on modern and future x64 machines. It appears to be feasible to provide hardware barriers with significantly lower latency than the best current software barriers. One approach would be via reconfigurable logic, such as the integrated FPGA of the Intel HARP, for which we estimate an order of magnitude possible improvement of latency, provided the latency of communication from application software to application hardware (AFU) were optimized.

More generally, closely coupled FPGA accelerators have the opportunity to provide much lower CPU↔FPGA latency. They should. Approaches to using FPGA accelerators have traditionally been application-centric, namely by hoisting performance-critical data paths out of software and reimplementing them as functional units on the FPGA, providing high bandwidth but not low latency. In contrast, parallel run-time and OS-centric approaches to using FPGA's generally focus on latency.

We have also presented a second approach in which hardware barriers would be integrated into the ISA and microarchitecture of the processor. Such an integration would almost certainly reduce barrier latency by several orders of magnitude compared to software barriers. It is possible to make a modern or near future x64 machine amenable to parallel applications and execution models that demand low-overhead global synchronization. Finally, we speculated about barrier support using a far future photonic processor, where we believe the barriers could be readily added, and provide overheads potentially as low as 10s of cycles and lower.

REFERENCES

[1] Abellan, J. L., Fernandez, J., and Acacio, M. E. A g-line-based network for fast and efficient barrier synchronization in many-core cmps. In *Proceedings of the 39th International Conference on Parallel Processing (ICPP 2010)* (September 2010).

[2] Arap, O., Brasilino, L. R. B., Kissel, E., Shroyer, A., and Swany, M. Offloading collective operations to programmable logic. *IEEE Micro 37*, 5 (September 2017), 52–60.

[3] Arif, M., and Vandierendonck, H. Reducing the burden of parallel loop schedulers for many-core processors. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (February 2018).

[4] Barroso, L., Marty, M., Patterson, D., and Ranganathan, P. Attack of the killer microseconds. *Communications of the ACM 60*, 4 (April 2017), 48–54.

[5] Bergstrom, L., Fluet, M., Rainey, M., Reppy, J., Rosen, S., and Shaw, A. Data-only flattening for nested data parallelism. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2013)* (Feb. 2013).

[6] Bienia, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[7] Binkert, N., Davis, A., Lipasti, M., Schreiber, R. S., and Vantrease, D. Nanophotonic barriers. In *Workshop on Photonic Interconnects and Computer Architecture (PICA 2009)* (December 2009).

[8] Blelloch, G. E., Chatterjee, S., Hardwick, J., Sipelstein, J., and Zagha, M. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing 21*, 1 (Apr. 1994), 4–14.

[9] Boner, J. 7-zip lzma benchmark—intel broadwell. https://gist.github.com/understeer/4d8ea07c18752989f6989deeb769b778. Accessed: 2018-10-14.

[10] Borkar, S., Cohn, R., Cox, G., Gross, T., Kung, H. T., Lam, M., Levine, M., Moore, B., Moore, W., Peterson, C., Susman, J., Sutton, J., Urbanski, J., and Webb, J. Supporting systolic and memory communication in iWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, WA, May 1990), pp. 70–81.

[11] Choi, Y.-k., Cong, J., Fang, Z., Hao, Y., Reinman, G., and Wei, P. A quantitative analysis on microarchitectures of modern cpu-fpga platforms. In *Proceedings of the 53rd Annual Design Automation Conference (DAC 2016)* (2016).

[12] Cicotti, P., Oral, S., Kestor, G., Gioiosa, R., Strande, S., Taufer, M., Rogers, J., Abbasi, H., Hill, J., and Carrington, L. *Data Movement in Data-Intensive High Performance Computing*. September 2016.

[13] Conway, P. Amd's direct connect architecture. In *Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems* (December 2006).

[14] Coteus, P., Randall Bickford, H., M. Cipolla, T., Crumley, P., Gara, A., Hall, S., V. Kopcsay, G., P. Lanzetta, S. Mok, L., A. Rand, R., A. Swetz, R., Takken, T., La Rocca, P., Marroquin, C., R. Germann, P., and Jeanson, M. Packaging the blue gene/l supercomputer. *IBM Journal of Research and Development 49*, 2-3 (2005), 213–248.

[15] Demir, Y., Pan, Y., Song, S., Hardavellas, N., Kim, J., and Memik, G. Galaxy: A high-performance energy-efficient multi-chip architecture using photonic interconnects. In *Proceedings of the 28th ACM International Conference on Supercomputing* (June 2014).

[16] Dietz, H., Hoare, R., and Mattox, T. A fine-grain parallel architecture based on barrier synchronization. In *Proceedings of the 1996 International Conference on Parallel Processing* (August 1996), pp. 247–250.

[17] Dinda, P., Wang, X., Wang, J., Beauchene, C., and Hetland, C. Hard real-time scheduling for parallel run-time systems. In *Proceedings of the 27th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC)* (June 2018).

[18] Epping, J. P., Hoekman, M., Mateman, R., Leinse, A., Heideman, R. G., van Rees, A., van der Slot, P. J., Lee, C. J., and Boller, K.-J. High confinement, high yield Si_3N_4 waveguides for nonlinear optical applications. *Optics Express 23*, 2 (January 2015), 642–648.

[19] Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., and Snir, M. The nyu ultracomputer – designing a mimd shared-memory parallel machine. pp. 239–254.

[20] Henriksen, T., Serup, N., Elsman, M., Henglein, F., and Oancea, C. Futhark: Purely functional gpu programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2017).

[21] Hoefler, T., and Lumsdaine, A. Optimizing non-blocking collective operations for infiniband. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)* (April 2008).

[22] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Intel, September 2016.

[23] Kandalla, K., Subramoni, H., Vishnu, A., and Panda, D. K. Designing topology-aware collective communication algorithms for large scale infiniband clusters: Case studies with scatter and gather. In *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)* (April 2010).

[24] Karl, E., Guo, Z., Conary, J., Miller, J., Ng, Y., Nalam, S., Kim, D., Keane, J., Wang, X., Bhattacharya, U., and Zhang, K. A 0.6V, 1.5GHz 84Mb SRAM in 14nm finFET CMOS technology with capacitive charge-sharing write assist circuitry. *IEEE Journal of Solid-State Circuits 51*, 1 (January 2016), 222–229.

[25] Kini, S. P., Liu, J., Wu, J., Wyckoff, P., and Panda, D. K. Fast and scalable barrier using rdma and multicast mechanisms for infiniband-based clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2003).

[26] Krishnan, V., and Torrellas, J. The need for fast communication in hardware-based speculative chip multiprocessors. *International Journal of Parallel Programming 29*, 1 (February 2001), 3–33.

[27] Kumar, S., Hughes, C. J., and Nguyen, A. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)* (June 2007).

[28] Lee, B. G., Doany, F. E., Assefa, S., Green, W. M., Yang, M., Schow, C. L., Jahnes, C. V., Zhang, S., Singer, J., Kopp, V. I., Kash, J. A., and Vlasov, Y. A. 20-μm-pitch eight-channel monolithic fiber array coupling 160 gb/s/channel to silicon nanophotonic chip. In *Optical Fiber Communication Conference* (2010), Optical Society of America, p. PDPA4.

[29] Leighton, T. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufman, 1991.

[30] Leiserson, C. E., Abuhamdeh, Z. S., Douglas, D. C., Feynman, C. R., Ganmukhi, M. N., Hill, J. V., Hillis, D., Kuszmaul, B. C., Pierre, M. A. S., Wells, D. S., Wong, M. C., Yang, S.-W., and Zak, R. The network architecture of the connection machine cm-5. pp. 272–285.

[31] Luo, X., Wu, W., Bosilca, G., Patinyasakdikul, T., Wang, L., and Dongarra, J. Adapt: An event-based adaptive collective communication framework. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (June 2018).

[32] Mellor-Crummey, J., and Scott, M. Algorithms for scalable synchronization on shared memory multiprocessors. *Journal of Parallel and Distributed Computing (JPDC) 9*, 1 (February 1991).

[33] Nalamalpu, A., Kurd, N., Deval, A., Mozak, C., Douglas, J., Khanna, A., Paillet, F., Schrom, G., and Phelps, B. Broadwell: A family of ia 14nm processors. In *2015 Symposium on VLSI Circuits (VLSI Circuits)* (June 2015).

[34] Pan, Y., Kumar, P., Kim, J., Memik, G., Zhang, Y., and Choudhary, A. Firefly: Illuminating future network-on-chip with nanophotonics. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture* (Austin, TX, June 2009).

[35] Peter, S. *Resource Management in a Multicore Operating System.* PhD thesis, ETH Zurich, 2012. DISS.ETH NO. 20664.

[36] Sampson, J., Gonzalez, R., Collard, J.-F., Jouppi, N. P., Schlansker, M., and Calder, B. Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (December 2006).

[37] Scott, S. L., and Thorson, G. M. The cray t3e network: Adaptive routing in a high performance 3d torus. In *Proceedings of the 4th Annual Symposium on High-performance Interconnects (HOTI)* (August 1996).

[38] Shang, S., and Hwang, K. Distributed hardwired barrier synchronization for scalable multiprocessor clusters. *IEEE Transactions on Parallel and Distributed Systems 6*, 6 (June 1995), 591–605.

[39] Sodani, A. Knights landing (KNL): 2nd generation intel xeon phi processor. In *Proceedings of the 2015 IEEE Hot Chips 27 Symposium (HCS)* (August 2015), pp. 1–24.

[40] Sun, C., Wade, M. T., Lee, Y., Orcutt, J. S., Alloatti, L., Georgas, M. S., Waterman, A. S., Shainline, J. M., Avizienis, R. R., Lin, S., Moss, B. R., Kumar, R., Pavanello, F., Atabaki, A. H., Cook, H. M., Ou, A. J., Leu, J. C., Chen, Y.-H., Asanovic, K., Ram, R. J., Popovic, M. A., and Stojanovic, V. M. Single-chip microprocessor that communicates directly using light. *Nature 528*, 7583 (December 2015), 534–538.

[41] Tang, X., Zhai, J., Quan, X., and Chen, W. plock: A fast lock for architectures with explicit inter-core message passing. In *Proceedings of the 24th Annual ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (April 2019).

[42] Tang, Y., and Bergmann, N. A hardware scheduler based on task queues for fpga-based embedded real-time systems. *IEEE Transactions on Computers 64*, 5 (May 2015), 1254–1267.

[43] Tyhach, J., Hutton, M., Atsatt, S., Rahman, A., Vest, B., Lewis, D., Langhammer, M., Shumarayev, S., Hoang, T., Chan, A., Choi, D., Oh, D., Lee, H., Chui, J., Sia, K. C., Kok, E., Koay, W., and Ang, B. Arria 10 device architecture. In *2015 IEEE Custom Integrated Circuits Conference (CICC)* (September 2015).

[44] Vantrease, D. *Optical Tokens in Many-core Processors.* PhD thesis, Department of Computer Sciences, University of Wisconsin–Madison, 2010.

[45] WikiChip. Coffee lake - microarchitectures - intel. https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake. Accessed: 2018-10-14.