Proc. of International Conference on Artificial Intelligence, Computer, Data Sciences and Applications (ACDSA 2026)
5-7 February 2026, Boracay-Philippines

Inter-Process Remote Execution (IPRE): Low-latency IPC/RPC using merged address spaces

1st Alf-André Walla
University of Oslo (UiO)
Norway
fwsgonzo@hotmail.com

2nd Laurence D. Rowe
Independent Consultant
United States
laurencerowe@gmail.com

3rd Paal E. Engelstad
University of Oslo (UiO)
Norway
paal.engelstad@its.uio.no

***Abstract*—This paper introduces Inter-Process Remote Execution (IPRE), whose primary function is enabling gated persistence for per-request isolation architectures with microsecond-latency access to persistent services.**

IPRE eliminates scheduler dependency for descheduled processes by allowing a virtual machine to directly and safely call, execute functions in a remote virtual machines address space.

Unlike prior approaches requiring hardware modifications (dIPC) or kernel changes (XPC), IPRE works with standard virtualization primitives, making it immediately deployable on commodity systems.

We present two implementations: libriscv (12-14ns overhead, emulated execution at 25-33% native speed) and TinyKVM (2-4us overhead, native execution). Both eliminate data serialization through address-space merging. Under realistic scheduler contention from schbench workloads (50-100% CPU utilization), IPRE maintains stable tail latency (p99<5us), while a state-of-the-art lock-free IPC framework shows 1,463× p99 degradation (4.1us to 6ms) when all CPU cores are saturated. IPRE thus enables architectural patterns (per-request isolation, fine-grained microservices) that incur millisecond-scale tail latency in busy multi-tenant systems using traditional IPC.

***Index Terms*—rpc, tinykvm, risc-v, emulation, ipc**

I. Introduction

In fine-grained modular systems built on virtualization, the performance of Inter-Process Communication (IPC) is dominated by the substantial overhead of scheduler-mediated VM wakeups. Consider a cache-backed architecture where a cache service handles 99% of requests internally. Storage services, accessed only on cache misses, remain descheduled most of the time to conserve CPU resources. When a miss occurs, the request waits not for IPC transmission (sub-microsecond with shared memory) but for the OS scheduler to wake the storage process. On a loaded system, this introduces tail latency spikes often exceeding 10ms. Three to four orders of magnitude above the IPC mechanism's inherent cost.

To validate that IPRE eliminates scheduler dependency, we compared it against iceoryx2 [1], a lock-free shared-memory IPC framework optimized for minimal latency. Figure 2 shows latency behavior under increasing system load. At zero load, both systems achieve comparable bidirectional performance (approximately 3us median latency). However, at 100% CPU utilization, iceoryx2's p99 latency increases from 4.1us to 6ms, a 1,463x degradation, while IPRE remains stable at 4.2us.

Traditional approaches to this problem involve keeping services polling (wasting CPU) or accepting unbounded scheduler latency. Recent work on KVM message-passing workloads [2] and production optimizations [3] demonstrates that even with adaptive halt-polling, millisecond-scale tail latencies persist.

Production systems demonstrate this problem clearly. A Drogon web server with 6us baseline latency experienced p99 degradation to 25ms under the default schbench load (5k RPS, 32 threads) on the same system as other forth-coming benchmarks in this paper.

This paper presents Inter-Process Remote Execution (IPRE), which avoids the scheduler entirely for the common case where the remote process is descheduled. When a cache miss occurs, the requesting thread directly executes the storage lookup function within the storage process's own context. The storage process need not be scheduled at all, as IPRE "borrows" the caller's thread for synchronous execution.

IPRE operates in two distinct architectural modes, each making different trade-offs:

1) Software-based (libriscv): Context switching occurs entirely in userspace within a single process. Guest code executes under emulation (25-33% of native speed on average), but context switches add only 12-14ns overhead. Suitable for scenarios where emulation overhead is acceptable and extreme call frequency demands minimal per-call cost.
2) Hardware-assisted (TinyKVM): Context switching uses hardware page faults between isolated processes. Guest code executes at native speed, but context switches incur 2us overhead from page fault handling. Suitable for scenarios requiring native performance where 2us per remote call is acceptable.

This sequence: page fault, host handling and resume, avoids scheduler intervention and the associated context switch of the vCPU thread. It replaces an unbounded scheduling delay with a predictable, hardware-mediated trap better suited for VM-to-VM interactions.

979-8-3315-7191-7/26/$31.00 ©2026 IEEE

Both our prototype implementations share the same core mechanism: merged address spaces enable direct memory access during remote execution, eliminating data marshaling. However, they target different points in the isolation/performance design space.

In this paper we assume these criteria to be true in order for IPRE to be a viable choice:

- Fast shared memory access to the remote VM
- The remote VM is cooperative (non-adversarial), belonging to the same tenant or is a trusted system service
- Synchronous/serialized access to the remote VM is acceptable

It is future work to access a remote VM without serializing access.

A. IPRE Requires a VMM

IPRE's mechanism requires capabilities not possible with current userspace API in Linux. The design requires capabilities that only a VMM like KVM or a userspace emulator (with virtual paging) can provide: interception of cross-boundary jumps, context switching with guaranteed cleanup on timeout or exception. Bonus points for rapid VM resets (also used by per-request isolation architectures). Traditional process isolation cannot provide these guarantees for the same reason why pthread_cancel is unsafe and being deprecated. Process reset requires fork/exec (milliseconds). And cross-boundary call trapping needs ptrace (prohibitively slow) or signal handlers (unreliable for guaranteed cleanup).

dIPC [4] achieves lower overhead through hardware tag-based isolation but requires either CODOMs support or significant kernel changes. XPC [5] achieves 21-cycle IPC through new CPU instructions. IPRE trades higher overhead (2-4us for TinyKVM) for immediate deployability on commodity hardware.

II. Scheduler Independence: Experimental Validation

To directly measure IPRE's scheduler independence, we benchmark TinyKVM against iceoryx2 [1] under realistic scheduler contention. We use schbench to generate workloads that exercise the scheduler's coordination mechanisms, creating conditions similar to multi-tenant production systems. Load levels are controlled by varying schbench worker thread counts: 16 threads (50% of available cores), 28 threads (90%), and 32 threads (100%, saturating all available cores).

A. Experimental Setup

Both systems execute 10,000 bidirectional request-response cycles. For IPRE, the caller VM invokes a function in the remote VM that immediately returns. For iceoryx2, we use event-based notification where the sender triggers an event and waits for the receiver's response event. We report results from iceoryx2's IPC variant, which uses inter-process communication primitives. All measurements were performed on an AMD Ryzen 9 7950X (32 logical cores) with CPU frequency scaling disabled.

B. Results

Table II summarizes the results. Under zero load, both systems achieve similar performance: IPRE shows 2.52us p50 and 4.32us p99, while iceoryx2 shows 3.0us p50 and 4.1us p99. This establishes that IPRE's overhead is competitive with optimized lock-free IPC when no scheduling contention exists.

Under moderate load (50-90% CPU), both systems show modest degradation. iceoryx2's p99 increases from 4.1us to 7.6us (85% increase), while IPRE's p99 remains below 4.3us (1% variation). The behavior changes dramatically when all cores are saturated. At 100% load, iceoryx2's p90 jumps to 3ms and p99 to 6ms, while IPRE's p90 remains at 3.2us and p99 at 4.2us.

This pattern reflects scheduler queue buildup. Under partial load, the Linux scheduler can still provide reasonable wakeup latency. Processes wait briefly in run queues before receiving CPU time. When all cores saturate, threads experience unbounded queueing delays. The iceoryx2 p50 remains stable (4.7us) because the majority of operations still complete quickly once both processes are scheduled. However, the tail latency reflects cases where one or both processes wait in scheduler queues, creating millisecond-scale delays.

IPRE avoids this entirely. The calling thread executes the remote function directly without requiring the remote process to be scheduled. This produces stable latency regardless of scheduler queue depth: whether zero threads or thousands are waiting to run, IPRE's mechanism is unaffected by the scheduler.

III. Limitations

IPRE addresses scheduler-induced latency for descheduled processes in shared-memory environments. It does not replace traditional IPC in all scenarios.

A. Fundamental Constraints

- IPRE cannot work across network boundaries. For distributed systems, traditional RPC with serialization remains necessary.
- The caller cannot continue executing while the remote accesses caller memory. This is inherent to zero-copy VM-to-VM IPC. Without the pause, one VM could trample memory while the other was reading it, leading to crashes and potential exploitation.
- The current implementation serializes access to each remote VM. For high fan-in workloads, remote VMs can be scaled horizontally through forking. Applications requiring a single source of truth may add a backing store (e.g., KV-store), increasing architectural complexity.
- IPRE assumes the remote VM is non-adversarial and usually belongs to the same tenant. It could also be a system-provided service (like Binder IPC in Android). Traditional IPC with explicit boundaries is more appropriate for untrusted remotes.
- Both sides must maintain compatible ABIs for function calls. Changes to remote functions require relinking dependent programs or recompiling FFI shared objects.

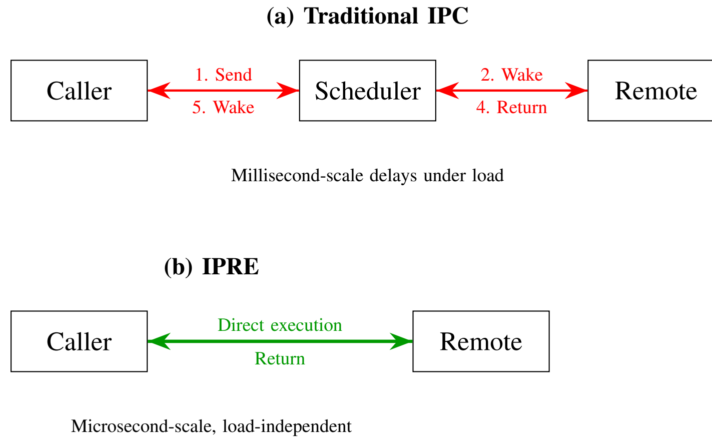

Fig. 1. Traditional IPC requires scheduler mediation with unbounded queueing delays, while IPRE enables direct execution in the remote context without scheduler involvement. IPRE avoids cross-core memory transfers and cache-line churn.

TABLE I
IPC LATENCY COMPARISON UNDER SYSTEM LOAD

Method	Idle (0%)	Saturated (100%)	Key Characteristics
IPRE (libriscv)	**12-14 ns**	**12-14 ns**	Emulated (25-33% native speed)
IPRE (TinyKVM)	**2.5 us (p50)**	**2.9 us (p50)**	Native execution
	4.3 us (p99)	**4.2 us (p99)**	Load-independent latency
iceoryx2	3.0 us (p50)	4.7 us (p50)	Requires both processes scheduled
	4.1 us (p99)	6000 us (p99)	1,463x p99 degradation
Unix Domain Sockets	5-6 us	6-12 ms	Scheduler-mediated
VM-to-VM (KVM)	25-40 us	10-50 ms	VMEXIT + scheduler + VMRESUME

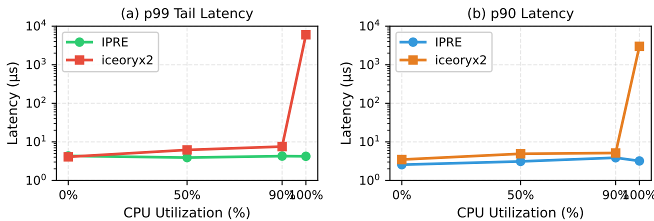

Fig. 2. IPRE maintains consistent p99 latency across system loads while iceoryx2, a state-of-the-art lock-free IPC framework, had 1,463x latency degradation when all CPU cores are saturated. The logarithmic scale is necessary to display both systems on the same graph.

TABLE II
LATENCY UNDER SCHBENCH LOAD (ALL VALUES IN MICROSECONDS)

System	0%	50%	90%	100%
p50 (median) latency:				
IPRE	2.52	2.21	3.80	2.87
iceoryx2	3.00	4.64	4.71	4.71
p90 latency:				
IPRE	2.56	3.11	3.88	3.20
iceoryx2	3.48	4.91	5.15	2998.84
p99 (tail) latency:				
IPRE	4.32	3.91	4.27	4.22
iceoryx2	4.09	6.18	7.55	5998.75

Modern build systems (e.g., CMake) automate this, but unlike schema-based systems, there's no built-in version negotiation.

- Language-level exceptions cannot cross remote call boundaries. A C++ exception reaching the boundary calls *std::terminate*, same as with shared libraries.
- If a remote can be accessed concurrently, programmers must use traditional atomic operations, mutexes, or lock-free data structures, just as with shared memory threading.

These constraints make IPRE complementary to, not a replacement for, traditional RPC in distributed systems.

IV. RELATED WORK

This paper focuses specifically on local IPC where both processes share physical memory. We do not address distributed systems or network communication, where serialization remains necessary. Our target scenario is fine-grained isolation within a single host, such as per-request isolated sandboxes communicating with persistent storage services, or modular applications decomposed into isolated components.

A. Marshaling-Based Approaches

Traditional RPC systems like gRPC [6], Protocol Buffers, MessagePack, and Cap'n Proto [7] serialize data into a

portable format that can traverse process boundaries, network connections, or storage media.

IPRE cannot replace marshaling-based RPC for distributed systems, but offers better performance for local inter-sandbox communication where marshaling or copying overhead would dominate. For example even near-zero-overhead deserialization (like Serde) still has to serialize the data, which is in the best case a full copy.

Benchmarks of traditional IPC mechanisms [8] [9] confirm that serialization overhead dominates for complex data structures, making zero-copy approaches advantageous when applicable.

B. IPC with shared memory

IPC is high performance and very low latency with async lock-free queues, when both sides are scheduled. This approach works well when the data is marshaled or semi-marshaled for this purpose. If the amount of data overshadows the need for fast locking, one can even use robust IPC locks that gracefully handle crashes (on one side, unlocking the other).

In Android the most common IPC is using Binder, where serialized Parcel objects are passed to system services. This method is a common source of UI response lag in applications.

Our approach requires direct memory access to the shared memory because two separate programs are temporarily connected with no typical queue-like communication mechanism. We currently serialize access to remote VMs, although there are plans to extend it to concurrent access in the future. For now, we scale remote VMs horizontally through forking if contention is an issue, or if there is a need to process for longer durations. Alternative serialization formats like FlatBuffers [10] and Protocol Buffers [11] optimize for different trade-offs but still require data transformation.

C. Historical Context: LRPC

Lightweight RPC [12] pioneered many concepts IPRE builds upon. LRPC reduced cross-domain call overhead through optimized kernel paths and argument passing in registers. However, LRPC required kernel involvement for protection domain switching and could not eliminate scheduler dependency for descheduled processes.

Like LRPC, IPRE also uses the calling thread to execute remote work. Where LRPC optimized the kernel path, IPRE eliminates it entirely for the VM-to-VM case.

D. Hardware-Assisted IPC Primitives

Recent work has proposed hardware extensions for fast IPC. XPC (Cross Process Call) [5] enables direct switching between caller and callee without kernel traps, reducing IPC latency from 664 to 21 cycles on ARM. SkyBridge [13] uses hardware virtualization to safely switch page tables, achieving significant speedups for microkernel IPC.

These systems demonstrate the value of eliminating kernel involvement, but require either new CPU instructions (XPC) or are tightly coupled to specific microkernel architectures (SkyBridge).

E. Direct Inter-Process Communication

The dIPC system [4] addresses similar performance problems as IPRE but through a different architectural approach. dIPC repurposes the CODOMs tagged memory architecture to enable threads to cross process boundaries using simple procedure calls. By mapping processes into a shared address space with hardware-enforced tag-based isolation, dIPC eliminates the OS kernel from the IPC fast path, achieving 64.12x speedup over local RPCs. Like IPRE, dIPC enables zero-copy communication through direct memory access. Threads can pass pointers to existing data structures rather than marshaling, and specialized proxies avoid unnecessary policy enforcement overhead. Both systems recognize that eliminating kernel mediation is critical for low-latency synchronous IPC.

- dIPC uses hardware memory tagging within a shared address space. IPRE uses separate VM address spaces with distinct page tables, providing stronger isolation boundaries.
- dIPC requires CODOMs tagged memory support or significant OS kernel modifications. IPRE works with standard KVM or userspace emulation, requiring no hardware changes or kernel patches.
- dIPC processes must trust the OS kernel to correctly configure memory tags. IPRE's VM-based isolation does not require kernel trust for the isolation boundary itself.
- dIPC is designed for decomposing monolithic applications into isolated components within a single system. IPRE targets sandboxes communicating with other (trusted) services, where complete process-level isolation is maintained except during explicit remote calls.

Both approaches demonstrate that removing scheduler dependency and kernel mediation from synchronous IPC enables dramatic performance improvements, though they occupy different points in the hardware-support and isolation-strength design space.

F. Partitioned Global Address Space (PGAS) Models

UPC++ [14] provides distributed computing through a Partitioned Global Address Space (PGAS) model. Like IPRE, UPC++ enables direct memory access across process boundaries, though with different design goals and trade-offs.

UPC++ allows remote memory access through `rget` and `rput` operations, remote procedure calls via `rpc`, and manages asynchronous operations using futures and promises. Programs can reference objects across process boundaries using global pointers, making it easier to write distributed applications than traditional message-passing approaches.

The key difference lies in transparency and coordination. UPC++ requires explicit programming for remote operations. Developers must distinguish between local and remote data, manage futures, and handle serialization for complex types. It provides coordination primitives like futures and promises to manage asynchronous operations. IPRE takes the opposite approach: remote calls look identical to local ones with no built-in coordination.

The performance characteristics reflect different targets. UPC++ is designed for distributed systems where operations take microseconds to milliseconds, using asynchronous operations to hide network latency. IPRE operates directly on shared memory, making immediate execution practical and eliminating the need for futures.

UPC++ provides explicit futures and RMA operations for distributed coordination. IPRE provides transparent synchronous calls for shared-memory scenarios. UPC++'s coordination primitives suit workloads where managing multiple concurrent remote operations is valuable. IPRE suits workloads where call transparency and minimal per-call overhead are paramount. These are complementary approaches for different architectural needs.

G. RDMA

Both IPRE and RDMA aim for ultra-low latency by avoiding kernel overhead and data copies.

- RDMA is fundamentally about data transfer (one-sided reads/writes). IPRE is about remote execution. With RDMA, you move data, then signal the remote CPU to process it. With IPRE, the local thread itself crosses the boundary to execute the code, which is a different and more tightly coupled model.
- RDMA is designed for machine-to-machine communication over a network. IPRE is designed for sandbox-to-sandbox communication on a single host.
- IPRE makes the remote call look identical to a local call. RDMA requires an explicit, asynchronous programming model (e.g., managing completion queues).
- RDMA requires a custom API to program against, while IPRE makes remote access transparent to the user. Both have mental load in that IPRE is an invisible crossing, and users need to understand that a heap allocation on one side needs to be copied into a new heap allocation on the other side in order to not be dangling data. RDMA is even more complex and requires a deep understand of the underlying hardware that enables it and the custom API and libraries that come with it.

Both IPRE and RDMA require understanding non-obvious boundaries and careful resource management. RDMA's explicit async model makes coordination visible; IPRE's transparent calls hide the boundary, requiring attention to ownership semantics. The appropriate choice depends on whether network distribution (RDMA) or shared-memory transparency (IPRE) better fits the architecture.

V. IMPLEMENTATION

A. Triggering Remote Execution: Software vs. Hardware Traps

The core mechanism of IPRE involves trapping a jump to a remote address and initiating the address-space merge. We have implemented two distinct methods for this trigger, each with unique performance characteristics.

1) Software-based Trapping in libriscv: In a userspace emulator like libriscv, the entire execution flow is managed by an interpreter loop. This provides a natural, low-cost intervention point for detecting remote calls.

The mechanism works as follows:

1) **Address Space Partitioning**: Local functions reside at low addresses (e.g., 0x400000+), remote functions at high addresses (e.g., 0x50000000+). This is established at link-time for static executables or load-time for relocatable executables.
2) **Jump Detection**: Before executing a JALR (jump and link register) instruction, the interpreter checks if the target address exceeds a threshold. This is a single integer comparison in the hot path.
3) **Context Switch**: For remote addresses, the interpreter:
   - Saves caller's register state
   - Switches thread-local pointer to remote VM
   - Begins executing remote VM's code at target address
   - On return, restores caller's context

This software-only approach achieves 12-14ns overhead because it requires only a branch misprediction and userspace state manipulation. No kernel involvement, no system calls, no MMU operations. Remote pages are loaned lazily.

The trade-off is that all guest code, both local and remote, executes under emulation (using the same emulator). On our test system (AMD Ryzen 9 7950X), libriscv with ahead-of-time optimization achieves approximately 25-33% of native speed. Applications must determine whether 12-14ns remote call overhead with 3-4x execution slowdown is preferable to 2us overhead with native speed. For configuration-type work where there is a rich API that has to be used (for example as a replacement for VCL, a domain-specific language for configuring HTTP caches), the libriscv approach with just a few nanoseconds overhead to call out to the host API is a clear choice.

2) Hardware-assisted Trapping in TinyKVM: For guests requiring native execution speed, we implement IPRE using KVM with hardware-triggered page faults. TinyKVM [15] provides the lightweight VM infrastructure necessary for this approach, with minimal memory overhead per instance.

The mechanism operates as follows:

1) The host VMM configures the guest's page tables to mark remote address ranges as not-present, while keeping them mapped in the VMM's address space.
2) When the guest attempts to jump to a remote function, the CPU generates a page fault, triggering a VMEXIT trap that switches from guest to host mode and delivers control to the VMM in userspace.
3) Address Space Merge:
   - Temporarily maps remote VM's memory into faulting guest's address space
   - Modifies instruction pointer to target function
   - Resumes guest execution with KVM_RUN

4) Return Handling: On function return, TLBs are cleared and context restored with return values preserved.

The 2us overhead (p50=1.94us, p99=2.78us) comes primarily from two VMEXIT/VMRESUME cycles, page table manipulation and TLB flushing at the end. This is approximately 1000x slower than libriscv's software trap, but guest code executes at native speed. This is still an order of magnitude faster than scheduler-mediated IPC (around 12us for futex wakeup, 10-50ms under load), and the latency remains consistent regardless of system load.

B. When to Use IPRE

Table III provides guidance on when IPRE is appropriate compared to alternatives.

TABLE III
IPC MECHANISM SELECTION GUIDE

Scenario	Recommended	Reason
Fast ops (<10us), descheduled remote	IPRE (TinyKVM)	Avoids scheduler latency
Fast ops (<10us), always-scheduled remote	Lockless + shared mem	Lower overhead when no wakeup needed
Medium ops (10-100us), low fan-in	IPRE (TinyKVM)	Synchronous execution acceptable
Medium ops, high fan-in (100+)	Thread pool + futex	Parallelizes remote work
Long ops (100us+)	Async queue	Don't block caller
Complex data, cache-sensitive	IPRE	Zero-copy + cache locality
Distributed (no shared memory)	gRPC / network RPC	IPRE requires shared memory
Untrusted or adversarial remote	Traditional IPC	IPRE requires trusted remote

C. Cache Coherence Advantages

Traditional IPC between processes on different cores incurs cache coherence overhead invisible to most performance measurements. When Process A on Core 1 sends data to Process B on Core 2, the data must traverse the cache hierarchy. Cache lines are evicted from Core 1's L1/L2 caches, written back through L3, and then loaded into Core 2's cache hierarchy. For complex data structures spanning dozens of cache lines, this may create hundreds of nanoseconds of additional latency beyond the IPC mechanism itself.

IPRE avoids this overhead when the calling thread executes the remote function. The caller's thread performs all memory accesses using its own L1/L2 cache state. Even when reading data "belonging" to the remote VM, those cache lines are now local to the executing core.

This advantage compounds with IPRE's zero-copy design. Traditional IPC copies or produces data from sender's cache into a shared buffer (evicting sender's cache lines), then receiver loads from shared buffer into its cache. IPRE accesses the data structures directly where they reside, maintaining cache warmth throughout the operation.

D. Emulator Architecture

IPRE requires emulators with virtual paging and the ability to execute code in multiple areas of the address space (which also means that the emulator supports JIT-compilation in the guest). In libriscv, the interpreter marks executable segment boundaries during ELF loading. When the CPU jumps to an address outside the local range, a callback triggers remote context establishment. This "execution fault" mechanism enables two merging strategies: (1) join address spaces during remote calls, or (2) join address spaces during remote calls, but also loan remote pages for data access at any time.

Strategy (1) provides better isolation by limiting remote access to explicit function calls, enabling an allow-list of remotely-callable functions. The remote function validates inputs like any other function, but the call surface area remains minimal. Strategy (2) permits access to e.g. C++ member functions requiring object access, but removes the safe isolation aspect. In the case where safety is not a concern, this method makes the most sense with read-only access before the remote call, but care should still be taken regarding concurrency.

TinyKVM achieves similar functionality through page table manipulation: remote ranges marked not-present trigger page faults that invoke the VMM's address space merge logic. Both approaches maintain the same security properties while operating at different abstraction levels.

E. Security

The general architecture is that both programs are made by the same tenant, but we also consider the persistent part (the remote/storage VM) to be higher privilege due to having persistence. For remote calls we have these assurances:

- Page protections still apply, and it is not possible to change page protections in the remote emulator.
- Gating remote access to functions specifically allowed by the remote emulator means the remote call surface area is exactly minimal.
- Limiting access to remote data until after a remote call is active solves potential issues around memory safety, as the foreign function is in charge.
- The remote machine has read and write access to input arguments the caller machine passes. More on this later.
- Execution timeout still applies, limiting the time a remote call can happen.
- Memory limits apply like normal on both emulators.
- Failure in the remote VM will automatically close the remote call and then re-throw the (TinyKVM internal) exception again to propagate further up. Remote calls act like a scope.

- KVM context switches flush speculation buffers twice per IPRE call (entry and exit), providing hardware-level defense against speculative execution side-channels. The caller is paused during remote execution, preventing timing-based information leakage through concurrent observation.

When both VMs are written by the same developer, callback functions enable useful patterns (custom allocators, visitor patterns). When the remote is a third-party service, callbacks should be avoided. Rich types passed by value already enable comprehensive data exchange without requiring bidirectional trust.

Emulators without execution timeouts typically need canceling operations (eg. Roblox's Luau coroutine canceling). This incurs overhead requiring a scheduler.

When an execution timeout is present, the design depends on the type. Some execution timeouts are based on a signaling timer (eg. KVM) because those implementations typically execute a blocking call (eg. KVMs ioctl KVM_RUN). Interrupting a blocking call is a simple and robust mechanism for timeout. Emulators without a blocking call, such as a regular sandbox that executes in the same process as the host, will have to use either threads or instruction counting. Sometimes referred to as fuel (WebAssembly) due to the relative cost of certain instructions compared to others. Fuel makes it possible for the emulator execution timeout to be cheap, but incurs some overhead during execution, typically listed as 15%. For small functions, it is a winning trade-off.

F. The Illusion of Local Execution

From the programmer's perspective, IPRE creates a powerful illusion: calling a remote function appears identical to calling a local function. Consider this example:

```
// In the caller's code
int result = remote_calculator::compute(42, 7);
// Execution never appears to leave this context
printf("Result: %d\n", result);
```

Behind this simple interface, several sophisticated mechanisms collaborate:

1. The linker places remote function symbols at high addresses (e.g., 0x50000000+) while local functions reside at low addresses (e.g., 0x400000+).

2. When the CPU jumps to a high address, the emulator traps this as a remote call request.

3. The caller's registers are preserved while switching to the remote emulator's context. The call can use the remotes stack for the call while still passing arguments using the callers stack.

4. During execution, both emulators' memory becomes accessible through direct addressing.

5. After the remote function completes, control returns seamlessly to the caller.

This creates a zero-copy, zero-marshaling communication channel where complex objects (including C++ STL containers, vtables, and even lambda closures) can be passed without modification. For allocating on the heap on behalf of the caller one can use allocators.

G. ABI Compatibility and Language Interoperability

IPRE operates at the ABI level, not the language level. Any two languages that can agree on:

- Calling conventions (register usage, stack layout)
- Structure padding and alignment
- Pointer representation
- Fundamental type sizes

can interoperate through IPRE. We have successfully tested:

- **C++ to C++**: Full STL container passing, virtual functions, templates
- **C to C++**: Struct passing with manual vtable construction
- **Rust to Rust**: Complex types and containers
- **Rust to C**: FFI-compatible structs and slices (with explicit ownership management)

However, several limitations prevent true language-agnostic operation:

1) Language-level exceptions cannot cross the remote boundary. A C++ exception reaching the remote call boundary will call std::terminate. This is fundamental to how exceptions work across dynamic library boundaries.
2) Languages with moving GCs (Go, Java, C#) require pinning objects passed to IPRE calls, which may degrade GC performance.
3) Languages with complex runtime systems may have thread-local state that doesn't transfer during context switches.

For C, C++, Rust and Zig interoperation, IPRE is seamless. For other language pairs, careful interface design is required, similar to FFI boundaries. We do not claim IPRE is universally language-agnostic, only that it works for languages with compatible ABIs (or where a common ABI can be specified) and well-understood runtime requirements.

H. Memory Management and Ownership

IPRE is zero-copy for remote read access: the remote function can traverse arbitrarily large caller data structures without copying. Persisting caller data in the remote requires duplication to maintain sandbox isolation. A shared writable region would allow the resumed caller to trample memory the remote relies on. Any safe sandbox-to-sandbox communication requires copying data that outlives the call boundary.

It is possible to allocate data on behalf of the caller by passing an allocator as argument to the call. We do this when we want to avoid a step and make the remote fill out parts of response data directly. The alternative is to return some data from the call and then use that to produce some parts of the response to a request. We also experimented with allowing the remote to directly deliver a response, avoiding the need to return back to the caller.

I. Concurrency and Deadlocks

IPRE transforms inter-process communication into a problem analogous to multi-threaded programming, inheriting its concurrency challenges, but with unique considerations. As mentioned before, it is the local emulator that performs the

actual remote call, with all the information of the remote. This means that the stack used is the local VMs stack (although it could be a per-VM stack in the remote, if that is more secure). The remote must be treated as any other multi-threaded program, where each of the callee VMs are threads accessing shared data. As an example, Rust enforces thread safety by default, and so a typical program of any complexity level requires no changes at all.

As an example, if we wanted to keep statistics in the remote, we could do this in the usual ways, such as:

- Atomically increment a variable in the remote
- Lock and increment a counter in a structure
- Spread out locks across an array of structures by using an id from the caller VM, reducing contention
- Collect stats locally and only flush to the remote VM after a certain amount of time or requests have been handled

Deadlocks are not possible as an execution timeout is always active in the caller VM, and will also act on remote calls (not extending the timeout). If the timeout is triggered the remote call is immediately closed and the internal timeout exception is re-thrown, propagating up to the original call-site in the caller VM. Ordinarily, only the caller (request VM) is reset as part of per-request isolation. Since both VMs entered a possibly erroneous state in this scenario, they will both be reset, which is a very fast procedure in TinyKVM.

J. Why Halt-Polling Cannot Replace IPRE

Adaptive halt-polling [2] [3] appears to solve the scheduling problem by having the host poll instead of deschedule. However, this approach has limitations:

1) Idle VMs consume CPU cycles polling, even when no work arrives. This is acceptable for dedicated instances but problematic in multi-tenant environments.
2) When the poll duration is exhausted, the VM must still be scheduled out (leading to unbounded tail latency).
3) Optimal poll duration varies by workload. Too short wastes the optimization; too long wastes CPU and negatively affects other processes on the system.
4) Even exitless IPIs and timers reduce but don't eliminate context switches. IPRE requires no exits for the remote call itself.

IPRE is orthogonal: it makes the remote VM's scheduling state irrelevant. The calling thread executes the remote function directly.

Production cloud providers deploy halt-polling primarily for dedicated instances [3], where wasting CPU on idle polling is acceptable because the customer owns the entire physical core. For multi-tenant scenarios or ephemeral VMs (which may exist for <100ms), halt-polling's CPU overhead is prohibitive. IPRE provides stable microsecond-level communication without any polling overhead, making it suitable for both dedicated and shared deployments.

VI. Benchmarks

All measurements were performed on an AMD Ryzen 9 7950X (16 cores, 32 threads) running Ubuntu Linux, kernel 6.8 with CPU frequency scaling disabled (performance governor). Each benchmark executed 10,000 iterations with the first 100 discarded as warmup. Section II presented the primary result: IPRE's scheduler-independent latency. This section characterizes per-call overhead and implementation trade-offs.

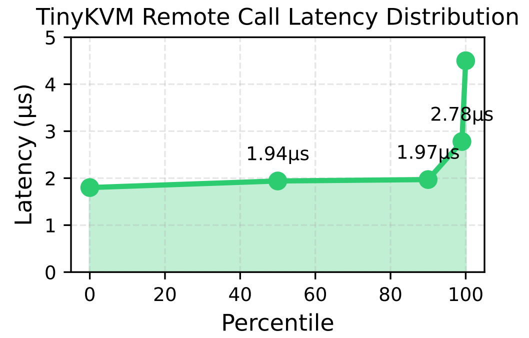

Fig. 3. TinyKVM remote call latency distribution shows consistent performance: p50=1.94s, p90=1.97s, p99=2.78s. The benchmark verified correct execution by having the remote function modify a vector, confirmed post-benchmark.

A. libriscv: Software-based Implementation

The software-based implementation achieves 12-14ns remote call overhead. This is measured as the difference between local function calls (9ns baseline) and remote calls (21ns total), attributable entirely to IPRE's context switch mechanism. This overhead remains constant across call patterns: passing function pointers adds 22ns and std::function objects add 38ns. These costs are from the objects themselves rather than the remote call mechanism.

Guest code executes under emulation at 25-33% of native speed. The 12-14ns overhead makes this implementation suitable for workloads where call frequency is extremely high (millions of calls per second) and the emulation penalty is acceptable for the guest code's computational characteristics.

B. TinyKVM: Hardware-assisted Implementation

Figure 3 shows TinyKVM's latency distribution. The p50 of 1.94s and p99 of 2.78s come primarily from two VMEXIT/VMRESUME cycles plus page table manipulation. Guest code executes at native speed between calls, making this implementation suitable for workloads where native execution performance is required and 2s per remote call is acceptable.

While 2-4s exceeds libriscv's 14ns, this overhead remains an order of magnitude faster than scheduler-mediated wakeup (10-15s minimum, milliseconds under load). The consistent latency (40% variance from p50 to p99) contrasts sharply with traditional IPC's unbounded tail latency under system load.

C. Engineering Trade-offs

Beyond raw performance, IPRE eliminates serialization engineering effort. Systems using Protocol Buffers or similar frameworks must maintain schemas, handle versioning, and perform copies even in "zero-copy" implementations. For

complex data structures with frequent schema evolution, this development overhead often exceeds IPRE's runtime cost. Data insertion proceeds naturally in both VMs. The remote duplicates caller data for persistence rather than storing by reference, while caller-allocated responses can use custom allocators passed as arguments.

VII. Further Work

A. Remove only remote TLB entries using INVPCID

By enabling *CR4.PCIDE* when available and using *INVPCID* instead of a full pagetable reload, we can avoid all of the return overhead created by having to re-walk the page-tables directly after the closing of the remote. This should result in an immediate speedup, and a lowering of the lower bound for IPRE in TinyKVM. Since the overhead is quite low already it hasn't been prioritized.

B. Concurrent access to remote VMs

By pre-registering threads in the remote guest available for grabs by the calling mechanism, it can use a guest-created thread-pointer (FSBASE) by switching to it before the call, which will allow concurrent access to a remote VM. This removes the need for serializing access but also greatly increases the complexity of the overall system. It's not a feature that can be put half-baked into production. In contrast, serialized access to the remote is easy to reason about, and can be immediately put into production as-is.

VIII. Conclusion

This paper presented Inter-Process Remote Execution (IPRE), which eliminates serialization overhead and scheduler dependency for shared-memory IPC through direct execution in merged address spaces, specifically for VM-to-VM IPC. Normally VM-to-VM communication is achieved through pipes or perhaps dedicated middle-man paravirtual devices and requires at least one full data copy and remote wake-ups. IPRE was specifically designed for multi-tenant systems that employs per-request isolation, providing safe shared-memory access. The remote VM can safely process zero-copy inputs as the caller is paused. IPRE always exits and re-enters the guest, leading to a double context-switch in each direction for maximum safety, and flushes TLBs on the way out.

We demonstrated two implementations: libriscv achieves 12-14ns overhead with emulated execution, while TinyKVM provides 2-4us overhead with native performance. Both avoid data marshaling entirely. In-memory structures pass as-is without transformation.

Long-running operations scale through horizontal VM forking. Different languages interoperate through standard FFI mechanisms. By demonstrating IPRE in both software emulation and hardware-assisted VMMs, we've shown it's an architectural pattern rather than a point solution, potentially establishing a new design space for VM-to-VM IPC.

References

[1] Eclipse Foundation. iceoryx2: Service-oriented publish-subscribe middleware. https://github.com/eclipse-iceoryx/iceoryx2, 2024. Accessed: 2025-01-15.

[2] David Matlack. Message passing workloads in kvm. 2015. KVM Forum.

[3] Wanpeng Li. Boosting dedicated instance via kvm tax cut. In *KVM Forum*, 2019.

[4] Lluís Vilanova, Marc Jordà, Nacho Navarro, Yoav Etsion, and Mateo Valero. Direct inter-process communication (dipc): Repurposing the codoms architecture to accelerate ipc. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17. ACM, 2017.

[5] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. Boosting inter-process communication with architectural support. In *ACM Transactions on Computer Systems*, 2020.

[6] Using grpc for (local) inter-process communication. https://www.mpi-hd.mpg.de/personalhomes/fwerner/research/2021/09/grpc-for-ipc/. Accessed: 2024-02-13.

[7] Cap'n proto, an insanely fast data interchange format and capability-based rpc system. https://capnproto.org/. Accessed: 2024-02-13.

[8] Benchmarks for inter-process-communication techniques. https://github.com/goldsborough/ipc-bench. Accessed: 2024-02-13.

[9] Aditya Venkataraman and Kishore Kumar Jagadeesha. Evaluation of inter-process communication mechanisms. 2015.

[10] Flatbuffers, an efficient cross platform serialization library. https://flatbuffers.dev/. Accessed: 2024-02-13.

[11] Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. https://protobuf.dev/overview/. Accessed: 2025-08-11.

[12] Brian N Bershad, Thomas E Anderson, Edward D Lazowska, and Henry M Levy. Lightweight remote procedure call. In *ACM TOCS*, 1991.

[13] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. Skybridge: Fast and secure inter-process communication for microkernels. In *EuroSys*, 2019.

[14] Upc++: Partitioned global address space (pgas) models. https://upcxx.lbl.gov/docs/html/guide.htmlremote-procedure-calls. Accessed: 2025-08-11.

[15] Alf-André Walla and Paal Engelstad. Introducing tinykvm for high-performance cdn edge services. pages 18–25, 12 2023.