Direct Inter-Process Communication (dIPC): Repurposing the CODOMs Architecture to Accelerate IPC

Lluís Vilanova[1,2] Marc Jordà[1] Nacho Navarro[1] Yoav Etsion[2] Mateo Valero[1]

[1] Universitat Politècnica de Catalunya (UPC)
& Barcelona Supercomputing Center (BSC)
{vilanova,mjorda,nacho,mateo}@ac.upc.edu

[2] Electrical Engineering and Computer Science
Technion — Israel Institute of Technology
yetsion@tce.technion.ac.il

Abstract

In current architectures, page tables are the fundamental mechanism that allows contemporary OSs to isolate user processes, binding each thread to a specific page table. A thread cannot therefore directly call another process's function or access its data; instead, the OS kernel provides data communication primitives and mediates process synchronization through inter-process communication (IPC) channels, which impede system performance.

Alternatively, the recently proposed CODOMs architecture provides memory protection across software modules. Threads can cross module protection boundaries inside the same process using simple procedure calls, while preserving memory isolation.

We present *dIPC* (for *"direct IPC"*), an OS extension that repurposes and extends the CODOMs architecture to allow threads to cross process boundaries. It maps processes into a shared address space, and eliminates the OS kernel from the critical path of inter-process communication. dIPC is 64.12× faster than local remote procedure calls (RPCs), and 8.87× faster than IPC in the L4 microkernel. We show that applying dIPC to a multi-tier OLTP web server improves performance by up to 5.12× (2.13× on average), and reaches over 94% of the ideal system efficiency.

CCS Concepts • **Software and its engineering** → **Operating systems**; *Process synchronization*

1. Introduction

Software systems comprise a complex collection of independent software components. Contemporary OSs offer two methods to construct such systems: a collection of libraries hosted in a single process, or a distributed collection of intercommunicating processes. The two present a classic performance/programmability vs. security/reliability trade-off [3, 5, 8, 11, 18, 23, 26, 33, 34, 42, 44, 47, 51]. Hosting libraries in a single process provides the best performance, by offering fast inter-component interactions using synchronous function calls and data references. But the lack of inter-library isolation allows errors to propagate across components, eventually compromising the whole system.

Alternatively, software components can be isolated in separate processes by using individual page tables managed by the privileged OS kernel. But inter-process communication (IPC) is known to impose substantial runtime overheads [40, 43, 60]. This is because processes provide a loose "virtual CPU" model, leading to an asynchronous and distributed system in which each process hosts its own private resources and threads. This, in turn, forces processes to explicitly synchronize and communicate through the OS's IPC primitives, which often require copying data across processes. Furthermore, applications require dedicated code to (de)marshal data and to (de)multiplex requests across IPC channels. In addition, the process isolation model is conservatively designed for the worst case of complete mutual isolation. Notably, local remote procedure calls (RPCs) conveniently hide all the aforementioned programming complexities of IPC behind regular synchronous function calls[1].

IPC imposes overheads on a multitude of different environments, from desktops to data centers. Our measurements show that local RPC communication is more than 3000× slower than a regular function call.

This paper presents *direct IPC* (*dIPC*), an OS extension that leverages the recent CODOMs architecture [61] to marry the isolation of processes with the performance of synchronous function calls. dIPC enables threads in one process to call a function on another process, delivering the same performance as if the two were a single composite application, but *without compromising their isolation*.

dIPC builds on top of the CODOMs architecture [61], which was designed to isolate software components inside a single process while enabling inter-component function calls at negligible overheads. dIPC repurposes CODOMs to isolate multiple processes on a shared page table to achieve low-overhead and secure inter-process function calls. Using

[1] Throughout this paper we use the term RPC to refer to efficient UNIX socket-based RPC and not to generic TCP/IP-based RPC facilities (which allow multiple hosts to transparently communicate over the network).

EuroSys'17, April 23-26, 2017, Belgrade, Serbia
DOI: http://dx.doi.org/10.1145/3064176.3064197

dIPC, threads in one process can efficiently and securely invoke predefined *entry points* in another process.

dIPC entry points translate into runtime-generated code thunks that safely *proxy* function calls across processes, freeing the OS kernel from managing IPC and data communication. Memory access isolation across processes is offloaded to CODOMs, while dIPC-managed proxies provide the minimum safety guarantees required for the inter-process calls. The bulk of isolation enforcement is instead implemented in untrusted user-level code. This allows untrusted application programmers to build their own isolation policies, from asymmetric isolation — e.g., protecting an application from a plugin — to the full mutual isolation of processes.

The main contributions of this paper are as follows:

- A detailed analysis of the performance (and, by proxy, energy) and programmability overheads imposed by current process-based isolation and IPC (§ 2).
- The design and implementation of dIPC (§§ 3, 5 and 6), an IPC facility based on the CODOMs architecture [61] (§§ 4 and 6.1.2), that enables threads to efficiently and securely invoke functions across processes without going through the OS kernel. dIPC further supports user-defined and user-implemented isolation policies.
- An evaluation of dIPC using micro- and macro-benchmarks (§ 7). The evaluation shows that dIPC is 64.12× faster than local RPC in Linux, and 8.87× faster than IPC calls in the L4 Fiasco.OC microkernel. We further show that using dIPC in a multi-tier OLTP web workload improves performance by up to 5.12× (2.13× on average), in all cases above 94% of the ideal system efficiency.

2. Overheads of Process Isolation

Process-based isolation brings overheads to many existing systems, from desktops to data centers. Throughout this paper, we examine the impact of IPC overheads using a running example of a real-world multi-tier online transaction processing (OLTP) web application stack [15] that is built from a collection of intercommunicating processes. This application is composed of a *Web* server frontend process, a backend *Database* that runs as an isolated process to enforce its own data access policies, and an intermediate *PHP interpreter* that is isolated from the rest of the system using a third process. Similar scenarios are also common in many other environments. For example, HDFS [55] uses a per-node process to survive the crashes of its client Spark processes [66]; the Chromium browser uses intercommunicating processes to isolate the UI, the page renderer, and its plugins from each other [1]; and multi-server microkernel systems isolate services like network and disk I/O into separate processes.

Figure 1 examines the runtime overheads associated with process-based isolation for our example OLTP web application (§ 7.4 details the experimental methodology). The figure depicts the runtime breakdown for the unmodified *Linux* baseline, which uses processes to isolate each tier, and compares it to an *Ideal* (but unsafe) configuration that runs all components inside a single process. The figure shows that forgoing isolation in the *Ideal* configuration allows it to run 1.92× faster than the safe *Linux* baseline.

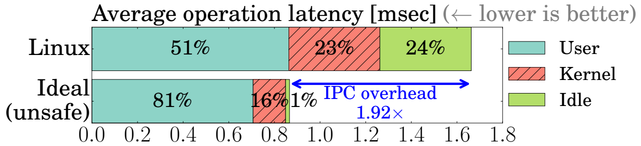


Figure 1: Time breakdown of a OLTP web application stack (see § 7.4). It uses either multiple processes for isolation (*Linux*), or an unsafe single-process system for *Ideal* performance.

2.1 Properties of Process-Based Isolation

We first define the properties of process-based isolation used throughout this paper. We also refer to isolation units using both *process* and *domain*, even when processes can contain multiple isolation domains [19, 20, 65].

Security is built on ***resource isolation***, i.e., what resources can be accessed (such as memory addresses or files); ***state isolation***, i.e., preventing processes from interfering with each other's state; and ***argument immutability***, i.e., whether a process is forbidden to modify the arguments it communicated before the other process signals its completion (e.g., between the time a server validates and uses data, to avoid a time-of-check-to-time-of-use vulnerability — TOCTTOU). In turn, reliability is built on state isolation, ***fault notification***, and application-specific code to recover from faults [29]. *Note that it is the programmer's responsibility to write code that responds to faults with a recovery.*

2.2 Sources of Overheads in IPC

Current architectures provide privilege levels and page tables for resource isolation and exceptions for fault notification, which are then used by OSs for process isolation. The co-evolution of OSs and architectures has led to a fairly efficient "virtual CPU" model for processes, but we argue that the ideal performance offset in Figure 1 is due to intrinsic inefficiencies and difficulties in that same model.

Figure 2 shows the performance of synchronous IPC with a one-byte argument using the following primitives:

***Sem.*:** POSIX semaphores (using `futex`) communicating through a shared buffer. *(=CPU)* and *(≠CPU)* indicate whether processes are pinned to the same CPU.

***L4*:** IPC in L4 Fiasco.OC passing data inlined in registers.

***Local RPC*:** RPC on UNIX sockets using *glibc*'s `rpcgen`.

The results were obtained using the methodology described in § 7.2 (we did not examine time breakdowns for L4). In all cases, traditional IPC is orders of magnitude slower than a function call, which takes under 2ns, while an empty system call in Linux takes around 34ns.

Improving the software IPC implementation or hardware separately is not sufficient. A bare-metal process switch involves a sequence of `syscall`, `swapgs` (to get a

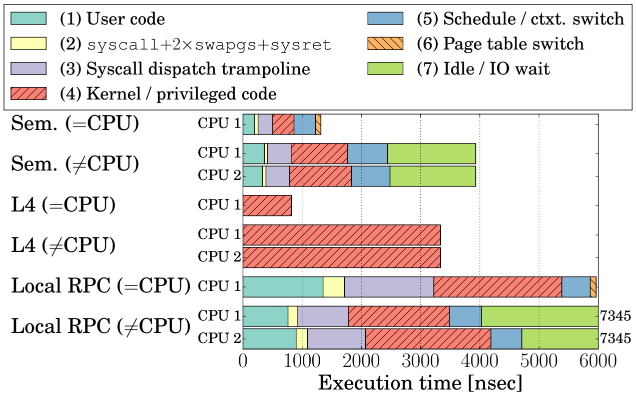


Figure 2: Time breakdown of different IPC primitives. Note that a function call takes under 2ns, while a system call in Linux takes around 34ns.

trusted `gs` segment for the kernel's per-CPU variables in *x86-64*), a page table switch, `swapgs` (to restore the user's `gs` segment) and a `sysret` (blocks 2 and 6 for *Sem. (=CPU)* in Figure 2). About 80% of the time is instead spent in software, which introduces second-order overheads by polluting caches, TLBs, and the branch predictor. Conversely, *L4 (=CPU)* successfully minimizes the kernel software overheads, but is still 474× slower than a function call.

Resource isolation has costs due to page table switches (block 6 in Figure 2) and, in Linux, switching the per-CPU `current` process descriptor (which provides access to the process's file descriptor table, part of block 5).

State isolation (i.e., context switching) also comes at a substantial cost (block 5 in Figure 2). In the asynchronous model of POSIX IPC, the kernel must save and restore all the registers of the processes it switches between. L4 instead provides synchronous IPC primitives, and uses the registers to pass the beginning of the message payload.

Going across CPUs is even more expensive and brings little benefit in synchronous communication [56], which is dominated by the costs of inter-processor interrupts (IPIs). Compare block 4 in *Sem. (=CPU)* and *Sem (≠CPU)* (sending and handling the IPI). Moreover, temporarily scheduling the idle loop adds to context switch time costs (block 5).

Argument immutability (i.e., copies) is inherently slower than using pointers in a function call, but also makes the IPC kernel code more complex (block 4 in *RPC (=CPU)*) and generates cache and TLB capacity misses. Using shared memory in *Sem. (=CPU)* avoids the copies, but requires applications to agree on pre-shared buffer sizes beforehand, or share pages on demand (which is limited in granularity and can be very expensive in multi-threaded applications [62]). Furthermore, avoiding copies to/from these shared buffers is not always possible, since the application might need to pass data that is received through a different shared buffer.

IPC generality and semantic complexity prevent optimization. The OS kernel's IPC paths must support a wealth of cases due to their complex semantics. This leads to overheads such as finding the target process, performing a context switch, and executing the syscall dispatch trampoline (block 3 in Figure 2). Overheads also trickle into applications, leading to increased user time (block 1); callers and callees must (de)marshal the arguments and results to/from the buffer used by the IPC primitive, and callees must also dispatch requests from a single IPC channel into their respective handler function. *RPC (=CPU)* hides these complexities, but at the cost of additional user code (block 1; comparing CPUs 1 and 2 in *RPC (≠CPU)* shows the difference between the caller's and callee's operations).

2.3 False Concurrency

Processes bind isolation and concurrency, forbidding threads from crossing process boundaries. Even when the application has no concurrency, IPC imposes a control transfer between threads of separate processes, and programmers must often use low-level asynchronous IPC primitives. For example, when a *PHP interpreter* thread in the OLTP example, referred to as a ***primary thread***, initiates a database query, it must transfer control to a ***service thread***, which executes the query in the *Database* process, and wait for its result. Similarly, a Spark process might block until HDFS finishes a data stream operation. Service threads are thus artifacts of the process-based isolation model.

2.4 Mechanism and Policy Separation

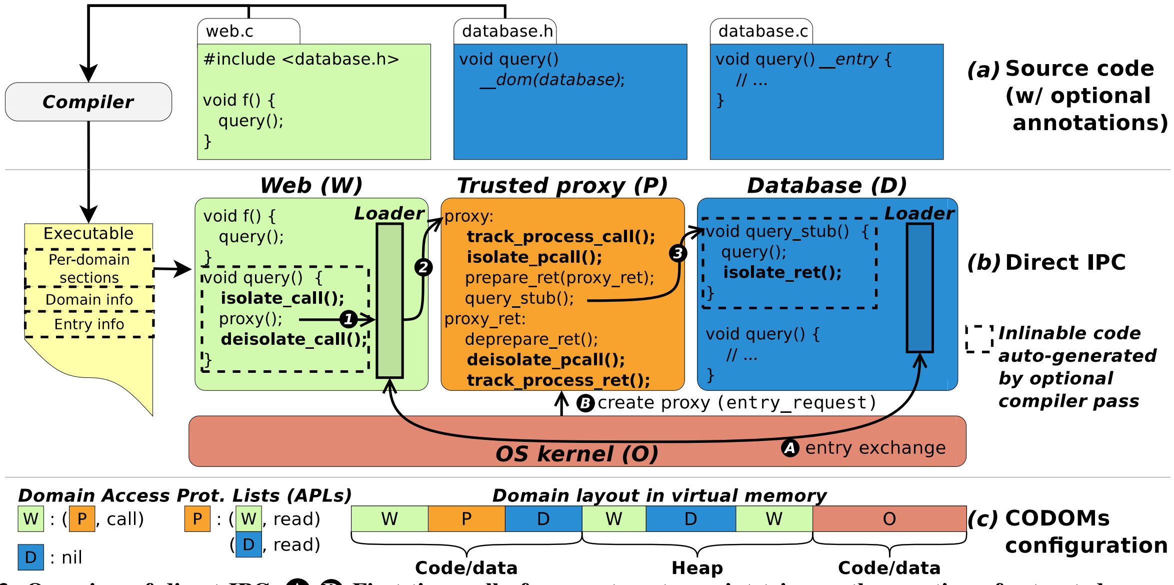
Process-based isolation often needlessly enforces *symmetric* isolation, whereas software components often have *asymmetric* relationships. The PHP interpreter in the OLTP example of Figure 1 illustrates this clearly. The interpreter need not be isolated from the web server, allowing direct accesses from the server and avoiding IPC. The server, however, is isolated from the interpreter, so that the intended isolation properties are preserved. Implementing these policies inside the IPC primitives prevents the compiler from optimizing policy implementation in the application's calling site (e.g., through register liveness information), leading to unnecessary register fiddling in the OS kernel.

State isolation can also be eliminated when faults are merely forwarded. For example, if a lazy PHP programmer has no code to recover from communication errors with the database, it will be left in an inconsistent state. Therefore it is more efficient to eliminate state isolation from the interpreter, and instead forward faults to the caller web server (loosely achieving exception semantics).

Moreover, argument immutability (i.e., copies) can also be elided when there is no TOCTTOU vulnerability. For example, a thread could modify a buffer while it is being written to disk by its driver, without affecting the driver's security properties (like in an asynchronous write).

3. dIPC Overview

Based on the observations in § 2, the goal of dIPC is to support fast IPC by enabling threads to directly invoke code in



```
#include <database.h>

void f() {
   query();
}
```

```
void query()
   __dom(database);
```

```
void query() __entry {
   // ...
}
```

```
void f() {
   query();
}
void query()  {
   isolate_call();
   proxy();
   deisolate_call();
}
```

```
proxy:
   track_process_call();
   isolate_pcall();
   prepare_ret(proxy_ret);
   query_stub();
proxy_ret:
   deprepare_ret();
   deisolate_pcall();
   track_process_ret();
```

```
void query_stub()  {
   query();
   isolate_ret();
}

void query() {
   // ...
}
```

Figure 3: Overview of direct IPC. Ⓐ-Ⓑ First-time call of a remote entry point triggers the creation of a trusted proxy for that function. ❶–❸ This proxy bridges the call across processes. Code enforcing isolation policies can be moved into the non-privileged application code (functions `query` in *Web* and `query_stub` in *Database*). Bold code is optional, and depends on the isolation policy.

other processes without going through the OS, giving the untrusted applications as much control of this process as possible. For example, Figure 3b shows a thread in process *Web* (domain *W*) directly calling function `query` in the *Database* process (domain *D*) using a regular function call. To attain this goal, we require an architecture with efficient primitives for cross-domain interaction. This can be achieved by (1) allowing regular function calls across domains, without adding overheads to existing out-of-order pipelines, and (2) by providing memory capabilities to allow safe cross-domain pointers and, therefore, avoid marshalling, copying and demarshalling of IPC arguments [16, 39].

The CODOMs architecture provides the two aforementioned primitives; it does so by isolating memory accesses and instruction privileges between multiple domains inside the same page table (domains *W* and *D* in Figure 3c; see § 4).

dIPC extends OSs that use processes as their unit of isolation, and runs on top of CODOMs to provide the higher-level OS abstractions that enable applications with faster IPC. To accomplish its goal, dIPC fulfills three key requirements:

R1 Avoid general-purpose code for request processing and (de)multiplexing in the OS and applications's fast path (§§ 2.2 and 2.3), and instead use more efficient application-specific code paths without bloating OS complexity.

R2 Allow domains (i.e., processes) to be dynamically created and destroyed and, by extension, to dynamically specify new communication channels.

R3 Expose isolation policies to the user-level code, where they can be more efficiently implemented (mechanism and policy separation; § 2.4). The challenge is in enforcing a security model where domains cannot compromise the security policies specified by other domains (§ 5).

3.1 Fast IPC Without OS Intervention (R1)

dIPC auto-generates a function-specific *trusted proxy* (domain *P* in Figure 3) that has access to domains *W* and *D* and bridges calls between them. A cross-domain call in dIPC looks like a regular synchronous function call; it performs an in-place domain switch, and redirects execution to the target function (steps ❶–❸; functionality across processes is most often and naturally expressed as a synchronous operation, where process parallelism yields no benefits). These calls in dIPC are 8.87×–64.12× faster than regular IPC because: **(1)** They avoid generic IPC primitives and user-level request (de)multiplexing, using instead a thin privileged code thunk that *safely proxies* calls between processes and into the target function. This proxy simply ensures that dIPC has control of where and when cross-domain calls and returns are executed; domains *W* and *D* cannot directly access each other, whereas domain *P* has access to both. **(2)** They eliminate the overheads of process synchronization (eliminate *false concurrency*), since proxies perform an in-place domain switch.

3.2 Dynamic Domains and Comm. Channels (R2)

Processes and domains in dIPC can be created and destroyed dynamically. In turn, domains can dynamically register new local entry points to export, and remote entry points to import. Imported entry points are treated like regular dynamic symbols by the application loader; the first time one is used (like *Web*'s call to `proxy` in Figure 3), the loader will trigger a request to create a new proxy for it, which will be reused in future calls (steps Ⓐ–Ⓑ end up creating the `proxy` routine).

3.3 User-Defined Isolation Policies (R3)

Untrusted application programmers can define their own isolation policies in dIPC, further increasing IPC performance speedups. An optional compiler pass takes source code annotations to identify domains and entry points (Figure 3a) and emits extended information on the output executable (left of Figure 3b). The application loader then uses this information to auto-configure the domains and public entry points of each process using dIPC's primitives.

The compiler also provides annotations to specify the desired isolation policies, independently for the caller and callee of an entry point (see § 5 for the security model). This allows programmers to build the most adequate isolation policy for their needs on a case-by-case basis (e.g., the example PHP interpreter of § 2.4 does not need resource isolation with the web server, nor state isolation with the database, eliminating unnecessary register and stack fiddling). Furthermore, the security model of dIPC allows isolation policies to be implemented in the untrusted application. The optional compiler auto-generates function *stubs*, whose contents are defined by the policy annotations (dashed boxes in functions `query` and `query_stub` for *Web* and *Database* in Figure 3b). Implementing isolation in user-level stubs lets the compiler optimize them with the rest of the application (e.g., by inlining them and taking advantage of register liveness optimization when handling register state isolation).

Therefore, the run-time generated proxies only contain the policy-enforcement code that requires privileges (e.g., `proxy` in Figure 3 switches the active file descriptor table). Note that the steps Ⓐ-Ⓑ also provide signature information of the target function, which allows dIPC to further fine-tune the performance of proxies.

3.4 Backwards Compatiblitiy

Finally, dIPC also provides compatibility and incremental adoption. Processes can selectively and incrementally use dIPC while retaining backwards compatibility. When required by the application's semantics, dIPC-enabled processes can provide concurrency for asynchronous calls using regular threads, and *argument immutability* can be implemented in non-privileged user code by simply copying argument contents. dIPC-enabled processes can coexist with other regular processes, and all processes can keep using the regular IPC primitives provided by the OS kernel.

4. The CODOMs Architecture

The CODOMs architecture is designed to subdivide the address space of a single process into multiple protection domains (e.g., to isolate sensitive libraries in an application). We highlight the two concepts that differentiate CODOMs from traditional architectures (summarized in Table 1), but a more comprehensive description of it can be found in the original paper by Vilanova et al. [61].

4.1 Code-Centric Domain Isolation

CODOMs verifies whether the *current instruction* is allowed to access a specific memory address, by using the instruction pointer as the subject of access control checks. This is in contrast to conventional systems, which check whether the *current OS process* can access the target memory address.

Architecture	*Operations*
Conventional CPU	**S:** 2×`syscall` + 4×`swapgs` + 2×`sysret` + page table switch **// D:** `memcpy`
CHERI	**S:** 2×exception **// D:** capability setup
MMP	**S:** 2×pipeline flush **// D:** copy data into pre-shared buffer, or write/invalidate entries in privileged prot. table
CODOMs	**S:** call + return **// D:** capability setup

Table 1: Brief comparison of best-case round-trip domain switch with bulk data on different architectures. *S*: domain switch, *D*: bulk data communication

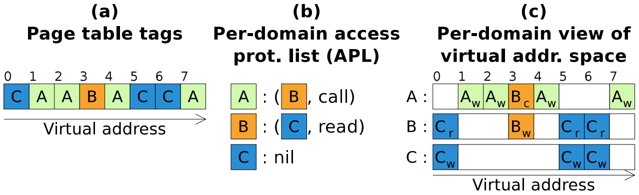


Figure 4: Domain isolation in CODOMs, showing the page table tags (left), the APL configuration (center) and the resulting access permissions for each domain (right).

Page tables in CODOMs are extended to contain multiple domains. In the spirit of architectures with memory protection keys [25, 32, 35], each domain is associated with a tag, and the page table has a per-page tag to associate each page with a domain. Additionally, CODOMs associates every tag (or domain) T with an *Access Protection List (APL)*: a list of tags in the same address space that code pages in domain T can access, along with their access permissions. Therefore, a domain will usually have its private code and data pages associated with its own tag, and the domain's APL will point to other domains to which it has shared access (e.g., a sandboxed library). Each APL entry grants the source domain one of the following permissions to the target domain:

***Call*:** Allows calling into public entry points of a domain. Any code address used with this permission is an entry point if it is aligned to a system-configurable value.

***Read*:** Allows reading from the destination domain as well as call/jump into arbitrary addresses of it.

***Write*:** Same as *read*, plus writing into the destination.

CODOMs honors the per-page protection bits in the page table; e.g., an APL with *write* access to a domain will not allow writing into a read-only page of that domain.

Figure 4 illustrates an example with three domains. It associates pages 1,2,4,7 with domain A (Figure 4a), whose APL permits calling into the entry points of domain B (page 3 in Figure 4b). This demonstrates the ability to directly invoke procedures across domains. Once the code in A calls into code in B (allowed by A's APL), CODOMs uses B's APL (the instruction pointer now originates from a page in domain B; Figure 4c). In turn, B can jump into the code of

C (pages 0,5,6), not directly accessible to *A*. To make this efficient, CODOMs has an independent software-managed *APL cache* for each hardware thread, which contains the access grant information of recently executed domains.

CODOMs extends the page table with a per-page *privileged capability* bit, which identifies code pages that are allowed to execute privileged instructions, eliminating the need for system call instructions and privilege mode switches.

Distinguishing features: The detailed micro-architecture simulations of CODOMs show that jumping into code of another domain (and effectively switching between domains) has a negligible performance impact [61]. CODOMs is able to avoid stalls in out-of-order pipelines, maintaining the processor's instruction-level parallelism. This is because APLs (specifically, the APL cache) together with the privileged capability bit allow an implicit change of the effective key set and privilege level. In comparison, traditional architectures require a system call into privileged mode to switch between page tables, key-based architectures require a system call to switch the privileged active key set, and domain switches in architectures such as Intel's call gates, CHERI [64], or MMP [63] require a processor exception or, in the best case, a pipeline flush. Furthermore, having independent APL caches makes it easy to scale to multiple cores, and being software managed allows the scheduler to swap an APL's contents during a context switch.

4.2 Transient Data Sharing Capabilities

Domains can also share arbitrary data buffers through *capabilities* [16, 39]. Capabilities are created and destroyed by user code through special hardware instructions, and CODOMs guarantees that they cannot be forged or tampered with. A new capability is always derived from the current domain's APL or from an existing capability, and is stored in one of the 8 per-thread capability registers provided by CODOMs. These registers are managed independently of regular pointers (stored in the regular CPU registers). By default, accesses are checked against all 8 capability registers.

Capabilities can be stored in memory, where they occupy 32 B. All capabilities can be spilled to a per-thread *domain capability stack (DCS)*, which is bounded by two registers that can only be modified by unprivileged code through capability push/pop instructions. Capabilities can also be stored to (and loaded from) memory pages that have the special *capability storage* bit set in the page table, and CODOMs ensures that user code cannot tamper with them.

Distinguishing features: Unlike traditional architectures, capabilities avoid cross-domain argument immutability (copies) and its associated complexity, while adding no overheads to the memory checks themselves (capability range checks can be executed in parallel with the TLB and cache lookups). Furthermore, most capability architectures fuse capabilities and memory pointers into a single entity, whereas CODOMs keeps them separate to minimize memory bandwidth requirements; the compiler can schedule capabilities into capability registers and use them for multiple memory accesses (e.g., by using a single capability to traverse an entire dynamic data structure). The use of capabilities is further reduced in CODOMs because domains can be readily accessed without them through the APL (e.g., domain *B* in Figure 4 has implicit read-write access to itself, and read-only access to *C*). When requested by the programmer, capabilities in CODOMs can be passed across threads and support immediate revocation through revocation counters (synchronous vs. asynchronous capabilities in § 4.1.5 of [61]), whereas other capability architectures rely on garbage collection or memory non-reuse, which would be very difficult to ensure across processes. Finally, the capability storage bit in CODOMs can differentiate capabilities from regular data without resorting to memory tagging.

4.3 Extensions Specific to dIPC

We have extended CODOMs to accelerate cross-process calls in dIPC. The APL cache maps domain tags to: (1) the permissions of that domain, and (2) a *hardware domain tag*, used internally to perform memory access checks (the 32-entry cache yields a 5-bit hardware domain tag).

We have added a privileged instruction that allows software to retrieve the 5-bit *hardware domain tag* of any domain present in the cache. Since the cache is quite small, this lookup operation takes less than a L1 cache hit. The use of this feature is described in § 6.1.2.

5. dIPC Design

This section describes dIPC's interface, whereas § 6 describes the details necessary to implement it efficiently.

5.1 Security model

Processes in dIPC are mutually distrustful by default. In order to allow user-defined policies, dIPC provides a security model with the following properties:

P1 Processes can only access each other's code and data when the accessee explicitly grants that right.

P2 Inter-process calls are always invoked through the entry points exported by the callee, and guarantee the correctness of the callee's state when it starts executing.

P3 Inter-process calls always return to the expected point in the caller in a way that guarantees the caller's correctness.

P4 Caller and callee agree on the signature of an entry point, ensuring adherence to their requested isolation policies.

P5 A process's failure to implement its declared policy will not affect the isolation of any other process.

Therefore, any erroneous or malicious use of dIPC will only affect the process executing that operation, but never the OS kernel or other processes it communicates with.

5.2 OS Support

dIPC-enabled processes get access to three new OS objects (see Table 2, described in the following sections):

Object attributes	***Description***
domain.tag	A CODOMs tag.
.perm	Permission to the domain: {*owner*, *write*, *read*, *call*, *nil*} (ordered set).
grant.src	Domain granting access from.
.dst	Domain granting access to.
.perm	Permission of the grant.

Object attributes	***Description***
entry.dom	Domain with entry points.
.count	Number of entries.
.entries[].address	Entry point address.
.entries[].signature	Number of input/output registers and stack size.
.entries[].policy	Isolation policy properties (see § 5.2.3).

Operation	***Effect***
$\texttt{dom\_default()} \rightarrow \texttt{dom}_{\texttt{d}}$	Return dom_d with *owner* permission to process's default domain.
$\texttt{dom\_create()} \rightarrow \texttt{dom}_{\texttt{d}}$	Return domain dom_d with *owner* permission to a new tag.
$\texttt{dom\_copy(dom}_{\texttt{src}}\texttt{, perm}_{\texttt{p}}\texttt{)} \rightarrow \texttt{dom}_{\texttt{dst}}$	Return domain dom_{dst} with permission $perm_p$ and dom_{src}' tag iff $perm_p \leq dom_{src}.perm$.
$\texttt{dom\_mmap(dom}_{\texttt{d}}\texttt{, ...)} \rightarrow \texttt{...}$	*mmap*-like allocation on a domain iff dom_d has *owner* permission.
$\texttt{dom\_remap(dom}_{\texttt{dst}}\texttt{, dom}_{\texttt{src}}\texttt{, addr, size)}$	Reassign selected pages from dom_{src} to dom_{dst} iff pages are in dom_{src}, and both dom_{src} and dom_{dst} have *owner* permission.
$\texttt{grant\_create(dom}_{\texttt{src}}\texttt{, dom}_{\texttt{dst}}\texttt{)} \rightarrow \texttt{grant}_{\texttt{g}}$	Return domain grant with $dom_{dst}.perm$ permission to dom_{dst} in dom_{src}' APL iff $dom_{src}.perm == owner$.
$\texttt{grant\_revoke(grant}_{\texttt{g}}\texttt{)}$	Set permission to *nil* for $grant_g.dst$ in $grant_g.src$' APL.
$\texttt{entry\_register(dom}_{\texttt{d}}\texttt{, count,}$ $\texttt{entries[count])} \rightarrow \texttt{entry}_{\texttt{e}}$	Return entry $entry_e$ for the given entry descriptors iff $dom_d.perm == owner$ and all descriptors point to dom_d.
$\texttt{entry\_request(entry}_{\texttt{e}}\texttt{, count,}$ $\texttt{entries[count])} \rightarrow \texttt{dom}_{\texttt{p}}$	Return domain dom_p with *call* permission to a new domain with proxies to $entry_e$ iff $\forall i < count : entries[i].signature == entry_e.entries[i].signature$. Each descriptor is set to its proxy's entry point on return. Per-entry policy is $entries[i].policy \cup entry_e.entries[i].policy$.

Table 2: Relevant core objects and operations in dIPC.

Isolation domains represent isolated memory allocation pools (setting a CODOMs tag to their pages).

Domain grants define the direct permissions between domains (configuring the APL of a domain).

Entry points identify the public entry points of a domain, as well as their signature and isolation properties, and produce the proxies that enable cross-domain calls.

These objects are private to each process and can be communicated to other dIPC-enabled processes to delegate access. All processes get a single *default* domain, and all dIPC-enabled processes share a global virtual address space. New domains are isolated from other domains (are not added to any CODOMs APL; P1), and therefore dIPC can share a single page table between the different dIPC-enabled processes.

5.2.1 Thread Management

Each *primary thread* can flow across different dIPC-enabled processes, and uses three types of stacks:

Data stack is isolated between threads by giving each a thread-private capability to it (a synchronous capability in CODOMs's terms; see § 5.2.3).

Capability stack (DCS) is where threads spill capabilities.

Kernel Control Stack (KCS) tracks the call stack across domains. The proxy used for a call through an entry point pushes a KCS entry with information about the caller, whereas a return pops it.

dIPC charges CPU time and memory to each process as usual. Primary threads appear with different identifiers on each process, so that users can control threads only when executing on processes they own (P5). When a thread crashes, a process cannot be simply terminated, since other processes can be up in the call chain and termination could lead to a deadlock [21]. Instead, a thread crash is redirected to the OS kernel, which unwinds the KCS to the entry with the oldest calling domain still alive, flags an error to it (similar to setting an `errno` value), and resumes execution on the proxy recorded on that KCS entry. Process kills are therefore treated using the same technique.

5.2.2 Domain Management (P1)

Each domain represents an isolated memory allocation pool, and processes can manage them through the domain handles described in Table 2. Pages allocated with `dom_mmap` get that domain's tag, while regular `mmap` and `brk` calls use the process's default domain. Domain handles have an additional *owner* permission, present only in software, that allows managing that domain's APL (see below). Processes can pass each other domain handles as file descriptors, and can use `dom_copy` to "downgrade" the permissions of a domain handle before passing it (e.g., to pass read-only handle).

Domain grant handles allow memory accesses from a *Src* to a *Dst* domain by modifying the APL of *Src* according to the permissions in the *Dst* handle. For example, calling `grant_create` with an *owner* handle for domain *Src* and a *read* handle for domain *Dst* will allow *Src* to perform read-only accesses to *Dst*. If *Src* does not have the *owner* permission, the operation fails. If *Dst* has the *owner* permission, dIPC translates it into the *write* permission in CODOMs. Since a domain handle must be explicitly communicated to operate on it, P1 is preserved.

Note that grant_create can be used to directly access data or execute code from another process's domain. A useful pattern for data is to allocate a subset of memory into a separate domain (e.g., a memory allocation pool for a complex dynamic data structure), and pass a handle to it to allow direct accesses. If only temporary access to that memory is necessary (a short-lived grant), processes can instead use a CODOMs capability. Passing direct access to code prevents dIPC from interposing proxies; the callee code will therefore execute as if it were the caller's process (e.g., using the caller's POSIX user ID and file descriptor table). Note that this situation complies with dIPC's security model, since passing a handle to that domain was an intentional operation (P1).

5.2.3 Entry Point Management (P2 to P5)

Domain grants configure direct memory access, but provide no guarantees to inter-process calls. Therefore, dIPC also provides entry points (entry_* operations in Table 2).

Entry point handles represent an array of public entry points in a domain, with an isolation policy specified through *isolation properties* (see below). First, a process *Dst* creates an entry point handle with entry_register, passing the *address*, *signature* and desired *isolation properties* of each entry in the handle's array. A process *Src* can then receive the entry point handle (step Ⓐ in Figure 3). *Src* then calls entry_request on that handle with the expected *signature* and desired *isolation properties* of each entry in the handle's array. At this point, dIPC creates a domain with one *trusted proxy* for each entry point (step Ⓑ). The result is a domain handle with *read* permission, which can then be accessed by *Src* after calling grant_create.

Since dIPC knows the signature and isolation properties requested by *Src* and *Dst* for each entry point, it generates proxies that are specialized for the target function signature and avoids unnecessary policy enforcement overheads by exclusively enforcing the requested isolation properties.

Isolation Properties

dIPC defines two properties for each sensitive resource: (1) *integrity* (i.e., write, trusting a domain to follow its application binary interface (ABI)); and (2) *confidentiality* (i.e., read, trusting a domain with private information).

Isolation properties define the contents of the generated proxy. When the optional compiler pass is used (see § 5.3.1), the auto-generated stubs implement the isolation properties themselves (function query in the caller and query_stub in the callee of Figure 3), and the runtime avoids passing them to entry_* when creating the proxies. dIPC provides the following isolation properties:

Register integrity saves live registers into the stack before the call, and restores them afterwards. Implemented in user stubs (isolate_call and deisolate_call in Figure 3, respectively).

Register confidentiality zeroes non-argument registers before the call, and zeroes non-result registers afterwards. Implemented in user stubs (isolate_call and isolate_ret, respectively).

Data stack integrity creates a capability for the in-stack arguments and one for the unused stack area before the call, restoring them afterwards. Implemented in user stubs (isolate_call and deisolate_call, respectively).

Data stack confidentiality + integrity splits data stacks between domains (arguments and results are copied according to the signature). Implemented in the trusted proxy (isolate_pcall and deisolate_pcall switch the stack pointers).

DCS integrity adjusts the DCS base register in CODOMs to prevent access to non-argument entries, and restores it afterwards. Implemented in the trusted proxy (isolate_pcall and deisolate_pcall, respectively).

DCS confidentiality + integrity uses a separate capability stack for each domain (entries are copied according to the signature). Implemented in the trusted proxy (isolate_pcall and deisolate_pcall to set the new stack and restore the old one, respectively). Only applies to callees.

dIPC must track the current process across entry points to account for resource usage (e.g., CPU time) and provide resource isolation (e.g., switch the file descriptor table). When entry_register and entry_request are called on different processes, dIPC generates track_process_call and track_process_ret in the proxy (Figure 3).

Register and stack integrity allow implementing cross-domain exception recovery. Data stack confidentiality is implemented in proxies to avoid intermediate copies, and so is DCS management since its bounds registers are privileged in CODOMs. Data stack and DCS confidentiality are activated when any side requests it, and integrity properties are only activated when requested by callers (i.e., when passed to entry_request). Note that the DCS and data stack are thread-private, so integrity is enforced both ways.

Security of Entry Point Calls and Returns

The following points clarify how call and return operations maintain dIPC's security model (§ 5.1):

P2: grant_create returns domains with *call* permission, and generates properly aligned proxies to ensure that CODOMs alignment restrictions (§ 4) force calls to go to the first instruction of the proxy. In turn, proxies redirect control into the target functions. Proxies also check the validity of the current stack pointer, ensuring that a callee uses its assigned per-thread stack when there is no stack switch.

P3: Proxies ensure that callers can correctly resume execution when a callee returns. The proxy saves the current process, return address, and stack pointers into the KCS and replaces the return address with one of its own (prepare_ret in Figure 3). The new return address points to proxy_ret, ensuring that the callee will return into the proxy. Since the callee's APL (*Database*) does not allow a return into proxy_ret, the proxy also creates a capability to it. On the return path, the proxy restores the saved pointers (deprepare_ret). This provides a minimal execution environment for the caller to restore the rest of its state (e.g., by restoring other registers saved to the stack).

Data stacks must be thread-private in x86 because proxies use the return addresses stored in them; a proxy copies a return address into the safe KCS after a call, and later uses that copy to return. Therefore, thread-private data stacks would not be necessary in other architectures where return addresses are stored in a register (e.g., ARM and MIPS).

P4: dIPC checks that the entry point signatures passed to entry_register and entry_request match. This ensures that caller and callee agree on a common ABI (e.g., register confidentiality on the callee will not expose a temporary register that is declared as a return register by the caller).

P5: Isolation properties are split between caller and callee stubs and proxy code in a way that ensures that the untrusted user stubs cannot impact the isolation of their peers. For example, an incorrect isolate_call or deisolate_call will only impact the caller's isolation guarantees, but never the guarantees of the proxy or the callee.

5.3 Compiler and Runtime Support

The operations in Table 2 are sufficient to write dIPC-enabled applications, but the optional dIPC-aware compiler pass and application loader and runtime simplify their use.

5.3.1 Compiler Annotations

The compiler provides four types of source code annotations: (1) __dom assigns code and data to domains; (2) __entry identifies entry points; (3) __perm specifies direct cross-domain permissions inside a process; and (4) __iso_caller and __iso_callee specify the isolation properties for callers and callees described above (respectively).

The compiler auto-generates caller stubs whenever it calls functions from another domain (function query in database.h of Figure 3 is assigned to a domain and used by web.c). It also auto-generates a callee stub whenever a function is identified as an entry point (function query in database.c). The generated stubs (central row of Figure 3) contain the code for all isolation properties that can be implemented outside a proxy. The stubs can be inlined into and co-optimized with the user application, allowing the compiler to exploit knowledge about function signature and register liveness to decide which registers to safeguard and/or zero at the point of a call. To highlight the importance of co-optimization, we have performed a simple experiment that compares exception recovery by saving registers (setjmp) vs. using a C++ try statement before calling a simple function. C++ try clauses produce code around 2.5× faster, since the compiler can reconstruct the state from constants and stack data in case of an error, instead of always saving register values.

5.3.2 Application Loader and Runtime

The compiler also uses the annotations to auto-generate additional sections in the output binary [38], which the program loader uses to load code and data into their respective domains, configure domain grants inside a process, and manage the dynamic resolution of domain entry points and proxies. This information also tells the runtime to trigger the generation proxies without the isolation properties that were already implemented in the auto-generated stubs.

5.4 Asynchronous Calls and Time-Outs

One-sided communication (or multiple returns) is a form of asynchronicity that leaks into the application interface semantics. Therefore, it can be supported in the same way as other asynchronous calls by creating additional threads, or even by using conventional IPC primitives if desired.

dIPC supports cross-process call time-outs by "splitting" a thread at the site that timed-out. The "split" operation will duplicate the kernel thread structure and KCS. The kernel will unroll the caller's KCS to the timing-out proxy, flag the error, and resume the caller's execution at that proxy. The callee thread can resume normal operation, and will be deleted when it returns into the proxy that produced the split (recorded in the KCS). This is similar to how thread kills and crashes are handled in § 5.2.1, but will only work if the timed-out caller uses a stack separate from the callee's (i.e., stack confidentiality + integrity was enabled).

6. dIPC Implementation

We have implemented dIPC assuming the CODOMs architecture, but the design and a large part of the implementation also applies to other architectures that support multiple memory protection domains on the same address space, such as CHERI [64] or MMP [63]. We believe the costs of the stronger isolation policies will be dominated by software overheads, making the finer micro-architectural performance trade-offs between these architectures less relevant.

Although cross-process timeouts are supported by the design, we have not implemented them (§ 5.4), since they are not used by the applications we evaluated.

6.1 OS Support

We have prototyped dIPC with over 9 K lines of new code for Linux (version 3.9.10) and 2 K for the runtime.

6.1.1 Run-Time Optimized Proxy Generation

dIPC has a set of "proxy templates" for different combinations of entry point signature and isolation properties. When entry_request creates a proxy, it chooses the template that matches the signature and requested isolation properties. It then copies the template into the proxy location, and adjusts the template's values via symbol relocation [38] (e.g., adjusting instruction immediates with control flow addresses and the assigned domain tag). We wrote a single parametrized "master template" in assembly that produces around 12 K templates at build time (averaging at 600 B each), reminiscent of the code specialization of Synthesis [46].

6.1.2 Fast Process and Stack Switching in Proxies

Cross-process proxies must switch Linux's pointer to the current process (the `current` variable) to maintain proper resource accounting and isolation (in `track_process_*`; see § 5.2.3), and must switch between stacks when requested (in `(de)isolate_pcall`).

The *hot path* of `track_process_call` looks up the target process's domain tag in the APL cache (the tag is known when generating a proxy). The resulting *hardware domain tag* is used as an index to a small per-thread cache array (32 entries), which points to the target process/thread identifier pair information (recall from § 5.2.1 that each *primary thread* has a per-process thread identifier). On a cache array miss, the *warm path* looks it up in a per-thread tree, indexed by the domain tag, and adds its information into the cache array. On a tree miss, the *cold path* upcalls into a management thread in the target process; this thread executes a system call to create the needed OS structures, and restarts the lookup.

Finally, `track_process_call` stores the `current` variable in the KCS and replaces it with the target's value. On a return, `track_process_ret` simply restores `current` from the KCS. Inter-process calls are treated as a time slice donation, until a system call or interrupt is executed. Note that stack switching uses the same mechanisms to locate and lazily allocate stacks when needed.

Proxies must also switch to the thread-local storage (TLS) [17] of the target process as part of a process switch. For simplicity, we use a costly `wrfsbase` in x86-64 to maintain the current TLS model, but it would be more efficient to add a new TLS mode where dIPC-enabled processes appear as dynamically loaded modules on the same TLS segment (provided each process defines its entries in one or more pages set with its domain tag).

6.1.3 Global Virtual Address Space

dIPC-enabled processes are loaded into a global virtual address space (which in turn allows using a shared page table), while other processes are loaded normally. Multiple shared virtual address spaces can co-exist in dIPC, but we describe it as a single-address-space OS (SASOS) for simplicity [10, 28, 31, 52, 59]. The OS memory allocator has two phases: first, a process globally allocates a block of virtual memory space (currently 1 GB), and then it sub-allocates actual memory from such blocks.

dIPC-enabled programs must be compiled as position-independent code (PIC) [38], which is common for server applications and dynamic libraries. Each process needs a "virtual copy" of its libraries, but code and read-only data of all virtual copies of the same library point to the same physical memory and cache lines. The POSIX `fork` operation, with its traditional copy-on-write semantics, temporarily disables dIPC in new processes to maintain backwards compatibility (marking only the process's pages as copy-on-write). When the POSIX `exec` operation is invoked with a PIC executable, dIPC is re-enabled in the process, which is loaded into a unique virtual address.

Board	Dell PowerEdge R210 II
Processor	Intel E3–1220 V2 @ 3.10GHz, 4 cores
Memory	16 GB
Ethernet	Broadcom BCM5716 (1Gig)
Infiniband	Mellanox MT26428 (10GigE)

Table 3: Evaluation machine configuration.

6.2 Compiler and Runtime Support

We have implemented a dIPC-aware source-to-source compiler pass using the Python interface of CLang [37], resulting in around 3.5K lines of Python and 300 lines of base system includes in C. The resulting file contains the auto-generated stubs and attributes to generate the additional binary information from § 5.3.1, and calls to entry points are substituted by calls to the auto-generated caller stubs.

6.2.1 Entry Resolution

Calls to proxies (in the caller stubs) are resolved at runtime by the program loader through an application-provided hook. The dIPC runtime provides a default implementation that uses UNIX named sockets to exchange entry point handles (step Ⓐ in Figure 3). Similar to other RPC systems, programmers only specify the named socket paths used by each process they communicate with. Programmers can use file permissions to control socket access, or provide their own entry resolution hooks (e.g., using a central service).

6.2.2 ABI Information

For simplicity, we did not change the compiler backend, so cross-domain calls do not have access to detailed ABI information. As of now, the function signatures used in Table 2 must be specified manually. Not integrating with the compiler backend also misses some opportunities to optimize-out code for register integrity and confidentiality according to the compiler's register liveness analysis information.

7. Evaluation

We evaluated dIPC on two sets of benchmarks. The micro-benchmarks provide a quantitative comparison of different IPC mechanisms and dIPC, whereas the macro-benchmarks explore the improvements on applications using dIPC.

7.1 Methodology

To avoid being limited by the long full-system simulation times of a multi-core, we evaluated dIPC natively using the

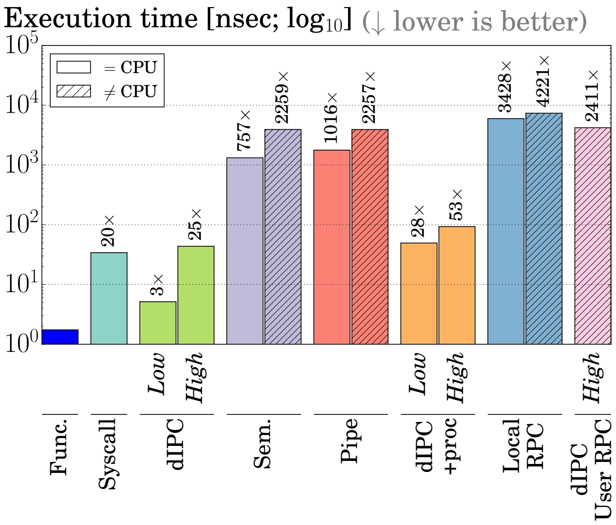


Figure 5: Performance of synchronous calls in dIPC and other primitives. A function is under 2ns.

platform described in Table 3. This allowed us to evaluate the broader effects of dIPC by executing the applications and OS in their entirety. We modified our code to emulate the CODOMs architecture in a way that we believe provides a reasonable approximation of the underlying hardware.

Per-CPU hardware resources: Every resource in CODOMs was emulated using per-CPU variables (`gs` segment in Linux x86-64), which are handled as part of the CPU state on context switches (using memory variables is slower than a hardware implementation of CODOMs).

Memory and privilege protection: Processes share a single page table, but APL checks are not enforced. To emulate the *privileged capability* bit in CODOMs, we ran all kernel and user code in privileged mode using KML [45]; it replaces `sysret` with an indirect jump in system calls, but that instruction takes only a fraction of the already small cost of block 2 in Figure 1. Furthermore, all system requests are performed through Linux's regular `syscall` path, instead of accelerating them with dIPC.

APL cache: We omitted APL cache lookups (§ 4.3) in the proxy routines, and instead directly used domain tags as hardware domain tags (the 5-bit index for the fast-path in process switching; § 6.1.2). This is reasonable because the cache is a small associative memory (32 entries), so a lookup should take 1–2 cycles. Furthermore, none of our benchmarks induces an APL cache miss: the APL cache is a per-hardware-context resource (i.e., its size only affects the number of domains frequently running with that cache, not on the entire system), and even the largest benchmark (the web application) uses only 7 domains, well below the 32 available.

Capabilities: We used regular loads and stores to move capabilities between memory and CODOMs's capability registers (also emulated in memory, and therefore not checked during memory accesses; each capability is 32 B).

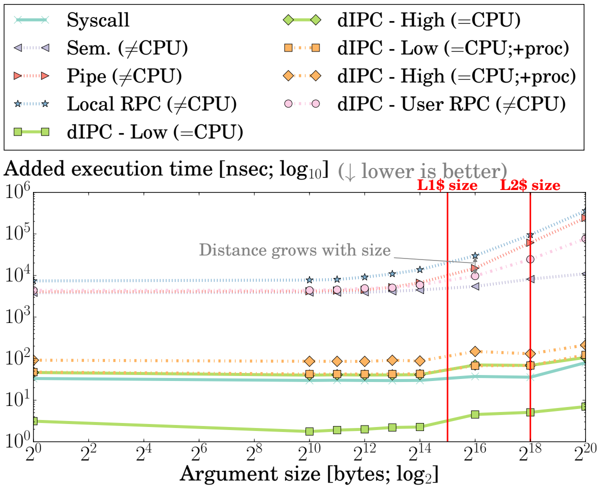


Figure 6: Comparison of dIPC and existing primitives for a consumer-producer synchronous call passing an argument of increasing size. Shaded areas show standard deviation.

The simulations of the CODOMs architecture show that memory, privilege, capability register and domain switch checks take place in parallel with the regular processor operation, adding no performance overheads [61]. Given that, the performance of dIPC is largely driven by the code executed in the user-level applications and the proxy routines.

7.2 Case Study: The Costs of Isolation

As observed in § 2.2 and Figure 2, process-based isolation and IPC add very large overheads. We thus quantify the effectiveness of eliminating them with dIPC.

Figures 2 and 5 show the mean execution time of various synchronous IPC primitives using a one-byte argument, whereas Figure 6 shows the performance effect of increasing the argument size; because the caller writes and the callee reads the argument, we show the execution time added by each primitive compared to the baseline function call. All experiments have standard deviation and timing overheads below 1% and 0.1% of the mean, respectively (with a 2σ confidence). Frequency scaling is disabled on all CPUs to maximize performance; otherwise, cross-CPU IPC takes longer (not shown for brevity).

Figure 5 shows two isolation policies for *dIPC* between domains of the same process: *Low* shows the effects of a minimal non-trivial policy, and *High* is equivalent to process isolation, highlighting the importance of asymmetric policies. The figure shows that dIPC can be much faster than even a system call (*dIPC - Low*), or have comparable performance when offering greater isolation (*dIPC - High* has mutual isolation, while user/kernel isolation is asymmetric by definition). In all, different asymmetric policies in dIPC can have up to a 8.47× performance difference.

The difference is even greater for cross-process calls (*dIPC +proc*, which has the same *Low*/*High* policies). dIPC maintains synchronous function call semantics and arguments are passed by reference, while achieving speedups between 14.16× (*dIPC +proc - High* vs. *Sem.*) and 120.67× (*dIPC +proc - Low* vs. *RPC*). The TLS segment switch in dIPC takes a large part of the time, so optimizing it (§ 6.1.2) would substantially improve performance (1.54×–3.22×).

Pipe and *RPC* have to copy data across processes (argument immutability), while dIPC allows passing arguments by reference, leaving copies up to the programmer. This is manifested in the increasing overheads for *Pipe* and *RPC* in Figure 6. While *Sem.* avoids cross-process copies, the programmer still has to populate the shared buffer. In comparison, dIPC can use CODOMs's capabilities to pass references to existing structures, and only to those structures.

Finally, the *dIPC - User RPC ($\neq$CPU)* experiment provides the same semantics as a cross-CPU RPC, but is largely implemented at user level. The server process makes a copy of its arguments and waits for a thread on another CPU to process them. This is almost twice as fast as *RPC*, and only uses the OS to synchronize threads of the same process. Interestingly, the copies in dIPC are more efficient than those in *Pipe* or *RPC*. This is because kernel-level transfers must ensure that pages are mapped in its address space before performing process-to-process copies. dIPC allows critical synchronization code in the OS kernel to be greatly simplified, since complex cross-process memory transfers can now be implemented at user level.

7.3 Case Study: Device Driver Isolation

SR-IOV allows directly assigning modern NICs to applications or virtual machines, avoiding the OS kernel or hypervisor on their fast path. Infiniband provides similar features. In both cases, processing must be offloaded into additional NIC and I/O infrastructure hardware. But more importantly, direct device assignment makes it harder to migrate computations [30, 67] and impairs the OS's ability to apply its policies on the device (e.g., packet scheduling is implemented in the NIC and cannot be easily extended).

We use Infiniband as our upper-bound performance scenario, and explore how dIPC can allow the OS to participate in the management of data flows without impacting performance. Figure 7 shows the latency and bandwidth overheads of isolating Infiniband's user-level driver into a separate domain/process. We used the *netpipe* benchmark (`NPtcp`), using Infiniband through the `rsocket` library. We also interposed the driver's operations to make synchronous requests to an "isolated driver" domain, without additional copies between the application, the driver and the NIC (just as is done in the original driver). dIPC uses an asymmetric policy between the application and the driver.

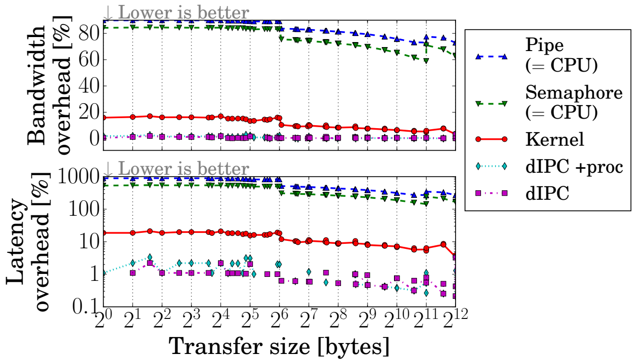


Figure 7: Bandwidth and latency overheads when isolating the Infiniband driver using dIPC, processes (IPC) or a kernel driver (user/kernel isolation).

Only dIPC sustains Infiniband's low latency, with a ∼1% overhead. In comparison, system calls incur a 10% overhead, and IPC incurs more than 100% latency overheads. Bandwidth is less affected, but we still see overheads above 60% for a 4 KB transfer in the IPC scenarios. Finally, the difference between the pipe and semaphore results show that unnecessary IPC semantics produce further slowdowns, since copies are not needed in this case.

Unlike other mechanisms, dIPC's latency is low enough to to be used in the future to regain OS control of I/O transfer policies while maintaining close-to-bare-metal performance.

7.4 Case Study: Dynamic Web Serving

We used a multi-tier web server to study the system-wide speedups of dIPC. The server uses Apache (2.4.10) to generate dynamic pages with PHP (5.6.7), which in turn retrieves data from a MariaDB database (10.0.22). The database uses either a regular hard disk or an in-memory file-system (using `tmpfs`) to approximate current in-memory services (using an SSD would provide a third intermediate point).

We ran the OLTP DVDStore macro-benchmark [15] (ver. 2.1) with 500 MB, 1 GB and 10 GB inputs, running for 3 min after a 2 min warmup period. All components were executed with 4 to 512 threads each, in order to isolate the impact of server concurrency. Starting at 1024 threads, baseline performance drops dramatically due to over-subscription. Performance was compared with the following configurations:

Linux as the baseline, where all components ran as isolated processes. All experiments were tuned to provide maximum throughput in this configuration: Apache used the multi-threaded *mpm-worker*, PHP ran using FastCGI[2] [48] and a bytecode cache, MariaDB used a threaded process, and all processes communicated using UNIX sockets (faster than TCP/IP due to header processing and additional intermediate data copies).

Ideal (unsafe), intended to show the ideal performance if all inter-process communication costs were eliminated. This configuration runs on the baseline *Linux* system, but embeds all components in a single process. PHP is used as an Apache plugin, and MariaDB is embedded into PHP

[2] CGI spawns a new process for each request, but FastCGI (commonly used for performance) dispatches requests to multiple long-lived processes.

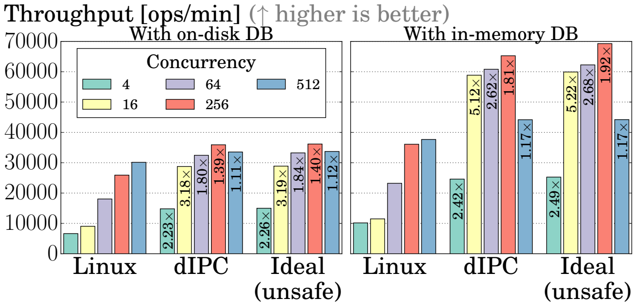


Figure 8: Performance of different dynamic web server configurations using vanilla Linux and dIPC. To isolate communication and performance factors, the benchmark was run with 4 to 512 threads (1 to 128 threads/CPU).

using the *libmariadbd* library. The core implementation is thus identical to the baseline, but it is stripped from unnecessary concurrency across processes, IPC calls and the glue code needed to manage IPC.

***dIPC*,** which places Apache, PHP and libmariadbd in separate domains with inter-process proxies. They use asymmetric policies where only PHP trusts all other components.

Figure 8 shows the throughput of each configuration on the 1 GB database; the other sizes show similar results. Since register and stack usage is unknown without low-level compiler backend support, the caller and callee stubs are folded into the proxies while assuming the worst case (all non-volatile registers are considered live).

Ideal (unsafe) shows that removing the baseline's IPC provides speedups of up to 3.19× and 5.22× for the on-disk and in-memory configurations, respectively. Interestingly, *Ideal* is able to achieve much higher throughput with much less concurrency, pointing to a much more efficient use of the system (see 512 threads in *Linux* vs. 16 threads in *Ideal*). This is due to large reductions in time spent in user and kernel code for the sole sake of managing process isolation and IPC. For the 256-thread results (largest throughput), the on-disk version eliminates 5.45% and 46.65% of user and kernel time, respectively. The in-memory version is able to reduce them by 13.72% and 60.51%, respectively. Another important speedup factor is a reduction of idle time, which goes from from 24% to 1% in the in-memory version. The asynchronous communication model of processes and the large number of threads necessary to fill the system lead the scheduler to temporarily imbalance the CPUs, at which point synchronous IPC must wait to contact a remote process. Instead, *Ideal* immediately starts executing the code of the target component.

The *dIPC* configuration has performance similar to *Ideal* for the same reasons, with speedups of up to 3.18× and 5.12× for the on-disk and in-memory configurations, respectively. The performance loss is split between inefficiencies in the OS and executing the caller/callee stubs and proxy routines. First, there is contention in global virtual memory block allocation, but using per-CPU allocation pools would easily improve scalability [7]. Second, we use a simplistic algorithm to resolve page faults across processes; the algorithm iterates over all processes in the current global virtual address space, which could be sped up by instead locating a faulted process by the address of its faulted virtual memory space block. Third, context switches are more expensive due to the additional structures in CODOMs. All other existing system operations have no measurable performance impact.

These results clearly demonstrate the benefits of the inter-process calls in dIPC. dIPC outperforms a standard Linux setup by 3.18× when using an I/O bound setup, and by up to 5.12× when using a faster storage medium. In all cases, it achieves more than 94% of the ideal system efficiency.

7.5 Evaluation Limitations

Emulating a CODOMs processor might underestimate three aspects of the actual hardware overheads.

First, in the case of an APL cache miss, CODOMs will trigger an exception and dIPC would have to update the software-managed APL cache. We emphasize that this event never happens on the presented benchmarks. Also note that the APL cache can be switched lazily to accelerate context switches (akin to the FPU or vector registers).

Second, detailed simulation of CODOMs shows negligible hardware overheads [61], but we nevertheless measured how hardware-domain crossing overheads would influence dIPC's performance. To that end, we counted the average number of cross-domain calls per operation. For example, the 256-thread in-memory configuration of *Ideal* in § 7.4 has 211 calls per operation. This corresponds to an average call overhead of 252 nsec for *dIPC*, higher than in the micro-benchmarks (Figure 5) due to additional cache pressure between the application and the proxies. Therefore, cross-domain calls could be up to 14× slower before voiding any performance benefit in dIPC.

Finally, capability loads/stores are present in the proxies and stubs, but are missing from the application code of the macro-benchmarks, since we do not have a full compiler backend to automatically manage capabilities (CHERI demonstrated that this is possible even for the C language [12]). We thus modeled the worst–case overheads by measuring the number of cross-domain memory accesses and assuming that they always require loading an additional capability from memory. For example, around 2% of the memory accesses in the 256-thread in-memory configuration of *Ideal* are across domains. This corresponds to a modeled 12% throughput overhead for the *dIPC* configuration (if we account for its average cache hit ratios and latencies), still leaving a 1.59× speedup over the *Linux* baseline. Note that this is a worst-case overhead estimate; capabilities and pointers are managed independently in CODOMs (there are separate capability registers and general-purpose registers). This allows a compiler to use standard register scheduling techniques to reuse the same capability register for accessing memory through multiple pointers, thereby minimizing the number of capability loads and stores in memory.

8. Related Work

The reduction of isolation overheads on existing systems has been the focus of many studies: directly accessing privileged hardware and I/O devices [3, 4, 50], using safe languages or virtual machines to isolate applications [34, 44, 51], or isolating application and kernel extensions [6, 9, 19, 20, 24, 31, 54, 57, 58, 65]. Instead, dIPC provides fast inter-process communication while maintaining backwards compatibility, and can also be used to isolate components inside a process.

The Burroughs systems provided support for calling functions on protected libraries (i.e., a form of dynamic linking) [49]; nevertheless, its stack architecture presented very different implementation considerations, it was unable to isolate errors across cross-domain calls (callers were terminated on a callee error), and mutual isolation required the system to create new stacks on every call. Multics [53] and Plessey System 250 [39] also had hardware support to accelerate cross-domain communication (e.g., by implementing processes and RPCs in hardware). All these systems hardwired isolation policies into the architecture, making them either insufficient or too heavyweight, while dIPC lets users build the most appropriate policy.

CrossOver [41] takes the concept of more efficient cross-domain control transfers into virtual machines. The architectural design is only concerned with eliminating intermediate calls to the hypervisor (plus the two guest kernels when communicating between guest user applications on different VMs), and bulk data communication must take place through shared memory pages. It provides a security model based on protected tokens that are part of the cross-domain control transfers arguments, and callees then manually check the token identifying a caller to decide whether to authorize the requested operation. It is not clear how this model works in the face of concurrent cross-domain transfers, since a single entry point in a domain must distinguish between being called into and being returned to.

The authors of Alpha OS [13], Mach [21] and Spin [27] observed that dissociating threads from processes made it possible to improve the performance of inter-process control transfers. Tornado [22] (and K42 [36]) also provided similar semantics (through *protected procedure calls (PPCs)*) in order to maximize the concurrency of request servicing, and further emphasized the need to minimize the use of shared state and locking in multi-processor systems during IPC. Tornado made heavy use of (un)mapping processor-local memory at PPC boundaries to achieve locality (e.g., for sharing stacks, at the expense of costly TLB shootdowns [62]), whereas dIPC is less strict on memory allocation and leaves memory placement to Linux's allocator (note that NUMA locality is less important in modern systems [14]). Furthermore, the kernel still had to mediate IPC calls in all these systems, whereas dIPC eliminates copies and kernel code entirely from the fast path, and allows programmers to build efficient isolation policies at user level.

In-place process switching in dIPC is in the spirit of migrating threads [21], as opposed to the asynchronous cross-core communication offered by FlexSC [56] (for system calls) and Barrelfish [2] (for IPC). Bulk asynchronous communication improves instruction cache locality by dedicating cores to specific tasks, at the expense of cross-core data cache transfers. Instead, dIPC's synchronous in-place control transfers efficiently trade off an increased instruction cache footprint for better data cache locality, as shown by the results of the end-to-end processing model in IX [4].

Previous systems have proposed using a unified address space [10, 28, 52, 59], and relied on different mechanisms to enforce isolation: (1) safe languages, which limit programmer choices; (2) separate per-domain page tables, which require entering into the OS kernel to switch them; or (3) address randomization, which makes addresses hard to guess, but not absolutely protected. A unified address space is necessary to facilitate data sharing across processes, but is not sufficient to provide the features and performance of dIPC.

9. Conclusions

This paper presents *direct IPC* (*dIPC*), an extension to current systems that provides safe and efficient inter-process communication while removing the OS kernel from the critical path. dIPC-enabled threads perform regular function calls across processes in a safe manner, eliminating the overheads of unnecessary synchronization, concurrency, (de)marshaling and worst-case symmetric isolation of processes. dIPC is implemented by repurposing and extending the CODOMs architecture [61].

Experimental micro-benchmarks show that dIPC is 64.12× faster than RPCs in Linux. Moreover, a multi-tier OLTP web server benchmark, where Apache, PHP and MariaDB communicate using dIPC, achieves up to 5.12× speedup over Linux, and 2.13× on average; in all cases, results are above 94% of the ideal system efficiency where all process and IPC overheads are eliminated.

With the efficiency offered by dIPC, isolation becomes more affordable, and programmers no longer need to choose between performance and security or reliability when using multiple applications and processes.

Acknowledgements

We thank Diego Marrón for helping with MariaDB, the anonymous reviewers for their feedback and, especially, Andrew Baumann for helping us improve the paper. This research was partially funded by HiPEAC through a collaboration grant for Lluís Vilanova (agreement number 687698 for the EU's Horizon2020 research and innovation programme), the Israel Science Fundation (ISF grant 769/12) and the Israeli Ministry of Science, Technology and Space.

References

[1] A. Barth, C. Jackson, and C. Reis. The security architecture of the Chromium browser. Technical Report with unspecified number, Stanford, 2008.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *ACM Symp. on Operating Systems Principles (SOSP)*, Oct 2009.

[3] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Symp. on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.

[4] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Symp. on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.

[5] J. Bernabeu-Auban, P. Hutto, and Y. Khalidi. The architecture of the Ra kernel. Technical Report GIT-ICS-87135, Georgia Institute of Technology, 1988.

[6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *ACM Symp. on Operating Systems Principles (SOSP)*, 1995.

[7] J. Bonwick and J. Adams. Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources. In *USENIX Annual Technical Conf.*, June 2001.

[8] R. H. Campbell, N. Islam, D. Raila, and P. Madany. Designing and implementing Choices: An object-oriented system in C++. *Comm. ACM*, Sept. 1993.

[9] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *ACM Symp. on Operating Systems Principles (SOSP)*, 2009.

[10] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single address space operating system. In *IEEE Trans. on Computers*, May 1994.

[11] D. R. Cheriton, G. R. Whitehead, and E. W. Sznyter. Binary emulation of UNIX using the V kernel. In *USENIX Summer Conf.*, June 1990.

[12] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, Mar 2015.

[13] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the Alpha real-time distributed kernel. In *USENIX Workshop on Micro-kernels and Other Kernel Architectures*, Apr 1992.

[14] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quéma, and M. Roth. Traffic management: A holistic approach to memory placement on NUMA systems. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, Mar 2013.

[15] Dell. Dell DVD store database test suite. http://linux.dell.com/dvdstore.

[16] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *Comm. ACM*, Mar 1966.

[17] U. Drepper. *ELF Handling For Thread-Local Storage*. Red Hat Inc., Feb 2003.

[18] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel, an operating system architecture for application-level resource management. In *ACM Symp. on Operating Systems Principles (SOSP)*, 1995.

[19] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Symp. on Operating Systems Design and Implementation (OSDI)*, 2006.

[20] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conf.*, 2008.

[21] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *USENIX Annual Technical Conf.*, 1994.

[22] B. Gamsa, O. Krieger, and M. Stumm. Optimizing IPC performance for shared-memory multiprocessors. In *Intl. Conf. on Parallel Processing (ICPP)*, Aug 1994.

[23] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *USENIX Summer Conf.*, June 1990.

[24] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[25] C. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, and G. Heiser. Itanium — a system implementor's tale. In *USENIX Annual Technical Conf.*, Apr 2005.

[26] M. Guillemont. The Chorus distributed operating system: design and implementation. In *ACM Intl. Symp. on Local Computer Networks*, 1982.

[27] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *USENIX Summer Conf.*, Jun 1993.

[28] G. Heiser, K. Elphinstone, S. Russell, and J. Vochteloo. Mungi: A distributed single-address-space operating system. In *Australasian Computer Science Conf. (ACSC)*, Jan 1994.

[29] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. In *Operating Systems Review*, Jul 2006.

[30] W. Huang, J. Liu, M. Koop, B. Abali, and D. Panda. Nomad: Migrating OS-bypass networks in virtual machines. In *Intl. Conf. on Virtual execution environment (VEE)*, June 2007.

[31] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *Operating Systems Review*, Apr. 2007.

[32] *Power ISA* ^{TM}. IBM, version 2.06 revision B edition, Jul 2010.

[33] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *ACM Symp. on Operating Systems Principles (SOSP)*, 1997.

[34] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv — optimizing the operating system for virtual machines. In *USENIX Annual Technical Conf.*, Jun 2014.

[35] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural support for single address space operating systems. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, 1992.

[36] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *EuroSys*, Apr 2006.

[37] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Intl. Symp. on Code Generation and Optimization (CGO)*, Mar. 2004.

[38] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 1999.

[39] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[40] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *ACM Workshop on Experimental Computer Science (ExpCS)*, Jun 2007.

[41] W. Li, Y. Xia, H. Chen, B. Zang, and H. Guan. Reducing world switches in virtualized environment with flexible cross-world calls. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2015.

[42] J. Liedtke. A persistent system in real use — experiences of the first 13 years. In *Intl. Workshop on Object Orientation in Operating Systems (IWOOOS)*, 1993.

[43] J. Liedtke. On microkernel construction. In *ACM Symp. on Operating Systems Principles (SOSP)*, 1995.

[44] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, Apr 2013.

[45] T. Maeda. Kernel mode linux. *Linux Journal*, May 2003.

[46] A. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[47] S. J. Mullender. The Amoeba distributed operating system: Selected papers, 1984-1987. Technical report, Centrum voor Wiskunde en Informatica, 1987.

[48] Open Market. FastCGI. `http://www.fastcgi.com`.

[49] E. I. Organick. The B5700 / B6700 series. *Computer System Organization*, 1973.

[50] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Symp. on Operating Systems Design and Implementation (OSDI)*, Oct 2014.

[51] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. Hunt. Rethinking the library OS from the top down. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, Mar 2011.

[52] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. Pilot: An operating system for a personal computer. In *Comm. ACM*, Feb 1980.

[53] J. H. Saltzer. Protection and the control of information sharing in Multics. In *Comm. ACM*, July 1974.

[54] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *USENIX Security*, Aug 2010.

[55] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Mass Storage Systems and Technologies*, MSST, May 2010.

[56] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Symp. on Operating Systems Design and Implementation (OSDI)*, Oct 2010.

[57] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *ACM Conf. on Computer & Communications Security (CCS)*, Oct 2012.

[58] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *ACM Symp. on Operating Systems Principles (SOSP)*, 2003.

[59] D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann. A structural view of the Cedar programming environment. In *ACM Trans. on Programming Languages and Systems*, Oct 1986.

[60] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *ACM Workshop on Experimental Computer Science (ExpCS)*, Jun 2007.

[61] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting software with code-centric memory domains. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2014.

[62] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *Intl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*, Oct 2011.

[63] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, Oct 2002.

[64] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2014.

[65] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. *Comm. ACM*, Jan 2010.

[66] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Hot topics in cloud computing (HotCloud)*, June 2010.

[67] E. Zhai, G. D. Cummings, and Y. Dong. Live migration with pass-through device for Linux VM. In *Ottawa Linux Symposium*, July 2008.