ABSTRACT
Some of the most difficult to find bugs in multi-threaded programs are caused by unintended sharing, leading to data races. The detection of data races can be facilitated by requiring programmers to explicitly specify any intended sharing and then verifying compliance with these intentions. We present a novel dynamic checker based on this approach, called PUSH.

PUSH prevents sharing of global objects, unless the program explicitly specifies sharing policies that permit it. The policies are enforced using off-the-shelf hardware mechanisms. Specifically, PUSH uses the conventional MMU and includes a key performance optimization that exploits memory protection keys (MPKs), recently added to the x86 ISA. Objects’ sharing policies can be changed dynamically. If these changes are unordered, unordered accesses to shared objects may not be detected. PUSH uses happens-before tracking of the policy changes to verify that they are ordered.

We have implemented PUSH for C programs and evaluated it using ten benchmarks with up to 32 threads. PUSH’s memory overhead was under 2.4% for eight of the benchmarks; 127% and 260% overhead for the remaining two. PUSH’s performance overhead exceeded 18% for only three of the benchmarks, reaching 99% overhead in the worst case.

KEYWORDS
data race detection, concurrency, memory protection keys

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

1 INTRODUCTION
Data races are a major cause of concurrency bugs. Data races are caused by unintended sharing. Hence, motivated by limitations of the various alternatives, one approach to detecting data races is to require the programmer to specify the intended sharing, using special annotations, and detect accesses that violate these intentions [11, 12, 15, 22, 24, 25, 33]. Especially for languages like C and C++, this approach can simplify and reduce the overhead of data race detection since tools based on this approach do not need to autonomously differentiate between accesses that correspond to intended sharing and those that violate these intentions.

The data race detectors closest to our work are: SharC [11], Shoa [12], and a tool that we refer to as DCOP (Dynamically Checking Ownership Policies) [33]. All three tools use a combination of static analysis and software instrumentation of memory accesses. This paper presents a novel tool, called PUSH (Prevention of Unintended Sharing), that requires programs to be similarly annotated, but where the detection of unintended sharing is implemented using off-the-shelf hardware. Thus, PUSH does not rely on static analysis and does not require high-overhead software instrumentation of any normal memory accesses.

With PUSH, when an object is allocated (statically or dynamically), annotation indicates the sharing policy of the object, such as private – read/write accessible by one thread, or read-shared – potentially readable by all the threads (§2.1). Subsequently, change policy annotations can be used to change the sharing policy of objects. Unannotated objects are accessible by only a single thread. While annotating programs is an extra burden on the programmer, prior work has provided positive indications that many programmers are willing to annotate their programs for this purpose [40].

Logically, PUSH maps sharing policies to per-thread read/write access permissions (§2.3) and enforces these permissions without instrumenting any normal (read/write) memory accesses. Ideally, this enforcement would be performed by specialized hardware that efficiently supports fine-grained memory protection for variable size objects [46, 49]. A key contribution of PUSH is an efficient implementation of the mechanism with off-the-shelf hardware, utilizing page-level protection. ISOLATOR [39] also uses page-level protection for data race detection. However, ISOLATOR only addresses one particular type of data race, where one thread acquires a lock for an object and another thread accesses the object without acquiring a lock. ISOLATOR cannot detect other types of data races. Furthermore, ISOLATOR incurs high overhead if an object protected by a lock is repeatedly accessed by different threads.

If two threads perform conflicting change policy operations on the same object in unordered (concurreny) vector time frames [36] (epochs [23]), unordered conflicting accesses to the object (data races) may not be detected by simply checking sharing policy violations (§2.2). To deal with this problem, PUSH identifies conflicting unordered change policy operations utilizing the FastTrack algorithm [23] to perform happens-before tracking of these operations.
The number of change policy operations in a program is much lower than the number of normal object accesses. Hence, the performance overhead of the tracking performed by PUSH is dramatically lower than mechanisms, such as FastTrack, that track every object access. This idea has the potential to enhance any dynamic race detector that relies on explicit sharing policy changes.

PUSH could be implemented in a straightforward manner by placing each object in a separate page and using a separate page table for each thread. Such an implementation would require significant changes to the OS kernel and involve memory overhead for multiple page tables. More importantly, such an implementation would involve high performance overhead since every sharing policy change would require a system call which, at times, would require synchronously updating all the page tables. Critically, this would have to be done every time a lock is acquired or released. PUSH avoids most of these disadvantages with a novel use of memory protection keys (MPKs), recently added to the x86 ISA [26]. Multiple additional optimizations further reduce the performance and memory overheads of the straightforward implementation.

One way to evaluate PUSH is by comparing it to ThreadSanitizer (TSan) [4, 42]. TSan is a widely-used and well-maintained data race detector, which is included as part of gcc. A clear advantage that TSan has over PUSH is that it does not require annotation. Our comparison was based on ten C benchmarks. Running with eight threads, in terms of performance overhead (additional execution time relative to the stock applications), for TSan the range was 332% to 12,820%, while for PUSH it was 0% to 99%. Furthermore, with PUSH the overhead exceeded 13% for only one application. For four applications, the TSan slowdown exceeded 1600%, while the maximum overhead of PUSH for those applications was 13%. In terms of race detection, neither TSan nor PUSH had any false positives (identified false races). Excluding races due to standard library calls (see §5.1), PUSH detected all the data races detected by TSan. These results indicate that, in many deployment scenarios, PUSH can be used in production runs, while TSan is restricted to use in offline debugging.

We have evaluated PUSH using the ten C benchmarks mentioned above running with up to 32 threads. For two applications that include a large number of dynamically allocated objects, the maximum virtual memory overhead (additional virtual memory use relative to the stock application) with PUSH was 127% and 260% (§5.4). For the remaining eight benchmarks, the virtual memory overhead was negligible —under 2.4%. The worst performance overhead was 99% for two of the benchmarks, due to a high rate of policy changes. For seven of the benchmarks the performance overhead was under 19% for all thread counts. PUSH detected a total of ten data races. As mentioned above, these were also the races detected by TSan.

We make the following contributions: 1) a novel technique for using current MMUs with MPKs for efficient data race detection by the prevention of unintended sharing coupled with happens-before tracking of only sharing policy changes; 2) several optimizations for increasing detection accuracy as well as reducing memory and performance overheads, including enhanced annotations, kernel changes, software caching, and the use of a universal family of hash functions; 3) analysis of the sources of overhead of PUSH and the effectiveness of the different optimizations; 4) comparison of PUSH to the most closely-related annotation-based race detectors schemes [11, 12, 33] as well as TSan.

Section 2 is an overview of the sharing policy annotation framework of PUSH and the basic approach of preventing unintended sharing using access permissions of protection domains. §3 is a detailed description of the implementation, including the various optimizations and their potential impact. The experimental setup and evaluation results are presented in §4 and §5, respectively. §6 is a summary of the limitations and disadvantages of PUSH and of its current implementation. Related work is presented in §7.

2 OVERVIEW OF PUSH

For any data race detection mechanism based on detection of violations of explicitly-specified intended sharing, there are two key issues: how the intended sharing is specified and how are violations of these intentions detected. The core sharing policies of PUSH and the permitted sharing policy changes are described in §2.1. In order to avoid hiding some data races, some combinations of sharing policy change operations on the same object performed by different threads must not be concurrent. The use of happens-before tracking [23, 36] of these operations to detect these situations is described in §2.2.

PUSH relies on hardware-enforced protection domains to detect unintended sharing. To facilitate efficient implementation, PUSH supports two additions to the core sharing policies: sticky-read (§2.2) and locked (§2.3). As explained in §2.3, in some specific scenarios, the use of the locked policy can result in false negatives (undetected races). §2.3 also provides a high-level overview of PUSH’s mapping of sharing policies to read/write access permissions in “ideal” protection domains. The implementation using off-the-shelf hardware is described in §3.

For a single execution, PUSH, as all dynamic data race detectors, can only detect races in code that is actually executed. However, PUSH also has the “pseudo-completeness” property, as defined in [19]: with a sufficient number of different executions, PUSH will eventually detect all the data races. As other tools, PUSH requires synchronization operations to be explicitly identified. Only Pthreads operations are currently supported.

2.1 Core Sharing Policies and Policy Changes

The core sharing policies of PUSH are essentially identical to those of DCOP [33]. Since every global object is potentially shared, PUSH must associate sharing policies with all such objects, which we will henceforth refer to as tracked objects. There are five core sharing policies: private, read-shared, racy, inaccessible, and untouched. We use the term “private object” to refer to an object whose current sharing policy is private. Similarly for “read-shared object,” etc. Two additional sharing policies: sticky-read and locked, are discussed in §2.2 and §2.3, respectively.

A private object is read/write accessible by one thread. A read-shared object is potentially readable by all the threads. Racy objects are read/write accessible by all the threads. This sharing policy is used for objects that are intentionally racy, such as synchronization objects (e.g. lock variables). An inaccessible object is not accessible by any thread. An untouched object becomes private to the first thread that accesses it. The initial sharing policy for an object is
specified when a static object is declared or when an object is

dynamically allocated. Static objects that are not annotated are
untouched. Objects allocated dynamically with the standard API
(e.g., malloc) are private to the invoking thread.

The racy policy allows incorrect annotation to hide races. How-

ever, with the C11 Standard [27], this policy can be removed (used
only internally for synchronization objects). Without racy, incorrect
annotation cannot result in false negatives. Incorrect annotation
can cause false positives. However, those are eliminated during the
annotation process (§5.2). C11 atomics are currently not supported.

During execution, the intended accessibility of an object may
change. For example, a private object may later become read-shared
by multiple threads. Hence, PUSH supports runtime change policy
operations. Acquire/release write operations by a thread make an
object private to the thread or release the association between the
object and the thread. Similarly, acquire/release read operations
allow or disallow the thread from reading and object. Since one
thread cannot force another to relinquish its access, a thread can
acquire write only if the object is untouched, inaccessible, locked, or
is the only thread for which the object is read-shared. Similarly, a

thread can acquire read only if the object is untouched, inaccessible,
locked, private to this thread, or read-shared by other threads. An
object that is released by all the threads becomes inaccessible. An
object that is untouched or inaccessible can be changed to racy.

2.2 Ensuring Ordering of Policy Changes

Simply enforcing the sharing policies presented in §2.1 does not
guarantee that all data races will be detected. For example, as shown
in Fig. 1, a thread may write to a private object, release the write
permission, and later, without an intervening synchronization op-
eration, another thread can acquire write and write to the object.
Allowing this kind of scenario is a limitation of prior data race detec-
tors based on the detection of sharing policy violations [11, 12, 33].

PUSH detects conflicting concurrent policy changes on the same

object using the FastTrack algorithm [23] for happens-before track-
ing. Specifically, since a thread must have exclusive access in or-

der to write to an object, in PUSH’s deployment of FastTrack, ac-
quire write and release write are processed as writes, while acquire
read and release read are processed as reads from the object. Thus,
the happens-before tracking is performed only for sharing policy
changes. By combining this idea with hardware enforcement of
sharing policies, it is possible to significantly reduce the perform-
ance overhead compared to conventional mechanisms based on
happens-before tracking, without incurring false negatives.

With the happens-before tracking of policy changes, an ideal
implementation of PUSH would be sound and precise (no false neg-
atives or positives), without requiring instrumentation of normal
read/write accesses.

A drawback of all happens-before tracking is the memory over-
head for each tracked object. This overhead is particularly large
for read-shared objects [23]. PUSH mitigates this overhead by in-

troucting the sticky-read sharing policy. All the threads can read
sticky-read objects. Once an object is changed to sticky-read, its
sharing policy can never be changed. Hence, following the change,
there is no need to perform any tracking of the object and this
does not compromise soundness. To avoid false negatives, a policy
change to sticky-read is allowed only when happens-before track-
guarantees that there is only one thread in the process. PUSH
enforces this constraint and flags violations.

2.3 Enforced Protection Domains

PUSH prevents unintended sharing by associating each global object
with a Logical Protection Domain (LPD). “LPD” denotes a set of
addresses for which each thread has the same read/write access
permissions, but the access permissions for different threads may
be different. Sharing policies are enforced by assigning the proper
access permissions (§2.1) to each of the LPDs. While LPDs could be
implemented using software instrumentation of normal memory
accesses, this would result in excessive performance overhead. Thus,
ideally, the LPDs should be implemented using hardware-enforced
fine granularity, variable size protection domains [46, 49].

In general, the overhead associated with protection domains in-
creases as the number of domains increases. This motivates PUSH to
map the LPDs to a smaller number of Enforced Protection Domains
(EPDs). This requires including multiple objects in the same EPD,
regardless of their physical location. For each thread, PUSH maps all
the LPDs that are private to that thread to the same EPD. For each of
the sharing policies untouched, inaccessible, racy, and sticky-read,
all the LPDs containing objects with that policy are mapped to a
single EPD. The above consolidations of LPDs into fewer EPDs do
not impact the soundness of PUSH.

PUSH places all the read-shared objects in a single EPD. This
consolidation does impact soundness since it is not possible to
strictly enforce the requirement for a thread to acquire read before
it can read the object. Once one thread performs an acquire read, all
the threads are able to read. Thus, PUSH would not detect a data race
where a thread writes to a private object, later it, or another thread,
changes the object to read-shared, and finally a third thread reads
the object but there is no intervening synchronization between
the original write and this read. Fortunately, in many cases, all the
reading threads are executing the same code. In such cases, either
none of the threads or all of the threads perform the acquire read.
With either possibility, the data race will be detected.

With DCOP [33], and thus also with the core annotations in §2.1,
every critical section requires changing the sharing policy of all
the accessed objects to private and then back to inaccessible. If
policy changes to/from private are slow (e.g., require a system call
§3.2), this may incur prohibitive overhead. Hence, PUSH adds the
locked sharing policy [11], that associates an object with a specific
lock. Multiple objects can be associated with the same lock.
locked object is read/write accessible by a thread that holds the lock associated with that object. All the LPDs containing locked objects protected by the same lock are mapped to the same EPD. Policy changes to/from locked are processed as writes (§2.2).

Once an object’s sharing policy is changed to locked, any thread holding the associated lock can access the object even if that access is unordered with respect to a prior access to the object before its sharing policy was changed to locked. Thus, when a thread performs a change policy operation to locked, PUSH actually maps the object to the inaccessible EPD. Hence, the first access to the object is trapped. PUSH then verifies that the access is performed while the thread holds the lock and the access is ordered (happens-after) with respect to the sharing policy change. If this verification succeeds, the object is moved to the EPD of the associated lock. Subsequent accesses to the object under the lock are data race free, as such accesses are ordered by the lock.

Since a locked object may be accessed by any thread holding the associated lock, a change policy from locked should be ordered with respect to accesses under the lock. Verifying this would involve happens-before tracking of every access to an object under a lock. For performance reasons, PUSH avoids tracking normal read/write accesses to objects. Hence, in some executions, the current implementation of PUSH can fail to detect accesses under lock which are unordered with respect to a change policy from locked, leading to the possibility of failing to detect a data race.

3 IMPLEMENTATION

PUSH is usable with current widely-available off-the-shelf hardware. In fact, the development of PUSH was motivated by the introduction of memory protection keys (MPKs) to the Intel x86 ISA [26]. This section describes the implementation of PUSH, including motivating and evaluating key optimizations.

Subsection 3.1 describes a straightforward implementation based on off-the-shelf hardware with only one modification to the OS: a separate page table for each thread. This subsection also lists the deficiencies of the straightforward implementation. PUSH’s optimizations that mitigate the impact of these deficiencies are discussed in the remaining subsections.

3.1 Basic Implementation

PUSH implements EPDs using page-level protections. Specifically, each EP is implemented as a set of pages with the same access permissions. These access permissions are set as described in §2.3. A straightforward implementation of this idea requires a separate page table for each thread. Since with standard OS kernels all the threads of a process share the same page table, this implementation requires a change in the kernel’s memory subsystem.

With the above implementation, access permissions are enforced at the granularity of pages. Hence, tracked objects that have different sharing policies must be in separate virtual pages [18]. If several tracked objects are in the same virtual page, a change policy for one of them would require a memory copy. Thus, tracked objects that may have different sharing policies in the future should also be in separate virtual pages.

PUSH places each tracked object in separate pages upon creation. For statically-allocated objects, this is done by a script that modifies the alignment of global objects in the assembly file. For dynamically-allocated objects, this is done by modified functionality of the calls to functions such as malloc. Due to the way the glibc heap allocator handles its metadata, it is not suitable for use with PUSH. Instead, PUSH links to the application the tcmalloc [9] heap allocator, which uses a dedicated memory region for its metadata. The current PUSH implementation does not track local objects allocated on the stack.

Lock acquire and release operations change the access permissions of pages containing the protected object. This is necessary so that locked objects are only accessible to threads that acquire the appropriate locks. PUSH utilizes the --wrap option in the Linux linker to intercept Pthreads synchronization operations and add the required functionality. The alignment requirement for dynamic memory allocation is enforced in the same way.

PUSH incurs storage overhead for metadata for each tracked object, each synchronization object, and each executing thread. For each tracked object, this metadata consists of at least 48 bytes, out of which 24 bytes are for the happens-before tracking. The storage overhead for synchronization objects and executing threads is mostly for vector clocks. Since the number of synchronization objects and executing threads is typically small, the related storage overhead is insignificant.

The rest of this section addresses the following deficiencies with the straightforward implementation:

(1) Each thread needs a separate page table, which requires significant changes to the OS kernel’s memory subsystem. Keeping the page tables properly synchronized incurs significant overhead.

(2) Several operations performed by PUSH require system calls to change page table permissions. The resulting performance overhead is exacerbated by item (1) above. The relevant operations are: (2.1) acquiring and releasing locks, (2.2) allocating and freeing dynamically-allocated objects, and (2.3) sharing policy changes.

(3) Allocating each tracked object on separate pages incurs substantial memory overhead due to internal fragmentation: the remaining space in the page is wasted.

3.2 Permission Management Using MPKs

Starting with the straightforward implementation of PUSH (§3.1), deficiencies (1) and (2.1) can be largely alleviated using memory protection keys (MPKs), recently added to the x86 ISA [26]. This optimization is the focus of this subsection.

With MPKs, each virtual page is tagged, in its page table entry, with a single protection domain number. There are 16 domains. A per-thread user-accessible register, PKRU, controls the access permissions to each protection domain for the current thread. The possible access permissions are: no access, read-only access, and read-write access. An access to the memory only succeeds when both the page table entry permission bit and the control bits in PKRU for the corresponding protection domain permit the access.

MPKs enable a user-level program to modify the memory access permissions of its threads without the overhead of a system call (mprotect on Linux). Specifically, such changes are performed by simply changing the contents of the PKRU register. With our experimental platform (§4), changing the PKRU register takes approximately 13ns, while the latency of an mprotect call is between 913ns and 12µs, for 1 and 32 threads, respectively.
For PUSH, with a small number of EPDs, MPKs eliminate the need for separate page tables. Specifically, the access permissions of each thread to the different EPDs are determined, in part, based on the contents of its PKRU register.

MPKs can also reduce the overhead for acquiring and releasing locks. Specifically, if an object protected by the lock is in a separate MPK domain, only a thread with the appropriate value in its PKRU register can access the object. By default, the PKRU registers of all the threads are set to prevent any access to the object. The lock acquire operation is augmented to modify the PKRU register to allow the thread to access the protected object after the thread acquires the lock. Before actually releasing the lock, the lock release operation changes the PKRU register to restore the accessibility of the object to what it was prior to the lock acquire. Thus lock acquire and release operations can be performed without the slow mprotect system call.

We use the ctrace benchmark (§4) to demonstrate the value of MPKs for PUSH. In ctrace a lock protects a hash table, which is accessed frequently by multiple threads. In a simple setup, this lock protects objects that are stored in six pages. Without MPKs, a PUSH implementation would have to invoke mprotect a total of twelve times during critical section entry and exit. We approximate the execution time of ctrace with an implementation of PUSH without the MPK optimization by delaying the lock acquire or releasing a lock by the measured latency of the mprotect call multiplied by the required number of invocations. For one execution thread, with PUSH the execution time compared to the stock version increased by only 18%, while for the version without the MPK optimization, the execution time increased by a factor of 322.

PUSH maps the four EPDs for read-shared, racy, inaccessible, and untouched objects to a single domain: domain 1. Normal page table permissions are used to provide the access permissions appropriate for each of these sharing policies. For example, read-only access for the read-shared EPD. All the objects private to a particular thread are in a single EPD. All the locked objects protected by the same lock are in a single EPD. Each EPD is mapped to a different MPK domain, chosen from domain 2 to domain 15. MPK domain 0 is used for memory that is not tracked. For private and locked EPDs, PUSH maps each thread IDs and each lock addresses, respectively, to an MPK domain number between 2 and 15.

3.3 Dealing with the Limited Number of MPKs

The sum of the number of threads and number of locks often exceeds 14, while only 14 MPK domains are available for the private and locked EPDs. PUSH uses hashing to map thread IDs and lock addresses to domains. Obviously, multiple EPDs may map to the same MPK domain. These hashing collisions can hide data races. For example, a lock address may map to the same domain as a particular thread ID, allowing that thread to access the protected object without acquiring the lock. Similar problems can occur if the addresses of two different locks map to the same MPK domain or the thread IDs of two threads map to the same domain. PUSH’s mechanism for mitigating this problem is presented in this subsection.

If the sum of the number of threads and number of locks is much larger than 14, when a single instance of a race occurs, there is a probability of $1/14$ that the race will not be detected. If the hash function is changed and the same race occurs again, there is a probability of $1/14$ that the second instance of the race will not be detected. However, the probability that both instances of the race will be missed is $1/14^2$. This is the basis for PUSH’s mechanism for mitigating the problem of hashing collisions.

PUSH periodically changes the hash function. For this, it uses a universal family of hash functions, based on multiply-mod-prime [44]. Based on linearity of expectation, it can be shown that, with $n$ inputs (thread IDs and lock addresses), $b$ MPK domains, and $k$ different hash functions, for any pair of inputs, an upper bound on the probability of a collision in all $k$ hash functions is $n(n−1))/(2^k)$. This bound is useful only when it is small. For example, if $n = 40$, $b = 14$ (as it is with PUSH), and $k = 6$, the upper bound is 0.01%. Thus, if, due to a hashing collision, a race is missed the first time it is encountered, as long as it occurs multiple times in the program, it will be detected later, when another hash function is in use. Since a different starting hash function is chosen every time the program executes, executing the program multiple times will ensure detection even if the race occurs only once in the program.

To implement the above, a timer periodically interrupts the main thread, causing it to generate a new hash function and send signals to all the other threads, causing them to stop. The main thread then iterates over the metadata of all the tracked objects. The MPK domain of every private or locked object, is reset, based on the new hash function. The main thread then signals all the other threads to resume execution. Based on the new hash function, each one of the application’s threads re-initialize its PKRU register to “open” the protection domain for its current private domain and the domain whose corresponding locks are currently held by the thread. To facilitate this operation, the PUSH runtime maintains a list of all the locks currently held by threads. Finally, all threads return back to the user application. This entire rehashing procedure is effectively atomic since all the threads are blocked for its duration.

The effectiveness of the rehashing mechanism in practice is demonstrated in §5.3. The results presented in §5.5 show that, if the hash function is changed every few seconds, the performance overhead of rehashing is negligible.

3.4 Reducing the Memory Overhead for Arrays

PUSH’s memory overhead is directly related to the number of tracked objects, due to fragmentation (deficiency (3)) and the required metadata (§3.1). If different array elements have different sharing policies, they must all be tracked objects allocated in separate pages. With the implementation of PUSH discussed so far, this would incur prohibitive memory overhead. This section is focused on how the memory overhead of PUSH in such scenarios can be reduced. Three ideas are presented: (1) reducing the overhead for metadata by incrementally increasing the size of the metadata on demand; (2) reducing the impact of fragmentation using memory mapping; and (3) using annotation to specify array slices, where all the elements within a slice have the same sharing policy.

An object’s metadata includes its starting address and size. With optimization (1), all the elements of the array initially share one metadata structure. If the sharing policy of a slice of an array is modified to be different from its neighbors, a new metadata structure is allocated for this slice.
Optimization (2) reduces the memory overhead for arrays and structs, where each element may have a different sharing policy and must thus be placed in a separate EPD. The optimization is based on mapping multiple contiguous virtual pages to a single physical page [10, 18, 30]. Each element is allocated in a different virtual page and at a different offset from the beginning of the page. No physical memory is wasted – consecutive elements are mapped to consecutive addresses in physical memory. We refer to an array or struct to which this optimization is applied as a split array or split struct, respectively.

Unfortunately, with the current Linux memory subsystem implementation, a large number of mappings incurs prohibitive performance and memory overheads. With our platform (§4) we empirically determined 10,000 mappings to be a practical limit. Hence, this optimization is not suitable for very large arrays.

Optimization (3) facilitates efficient handling of some arrays where the number of elements exceeds the above mapping limit. It is suitable for applications where groups of array elements (array slices) have the same sharing policy. A runtime function allows the programmer to specify the number and sizes of the slices when the array is allocated. Each slice is handled as a separate tracked object. An array allocated and managed in this way is a sliced array.

For split struct, split array and sliced array, elements/members must be placed at different virtual addresses from those generated by the standard compiler. In our prototype implementation, this is done by modifying application source code. For split struct, padding is manually inserted in the struct definition. For split array and sliced array we use a script to replace the reference to the array with a macro that computes the correct memory address.

The metadata for each sliced array includes an array that stores the starting and ending index of each slice. In order to access a sliced array element, the code that maps an index to the correct memory address needs to walk through a metadata array. To reduce the performance overhead for such accesses, a very simple per-thread software cache is maintained for each sliced array. This cache stores the single most-recently-accessed slice.

To demonstrate the potential benefit of the software cache, we used a micro-benchmark that iterates over the one million integer elements of an array that consists of 32 slices. The software cache reduces the execution time overhead of Push from 4.7x to 2.2x.

3.5 Reducing Permission Changes Overhead

This section is focused on optimizations that mitigate the performance overhead of deficiencies (2.2) and (2.3) discussed in §3.1: (1) recycling recently-freed pages tagged with the same domain number, and (2) eliminating unnecessary remote TLB shootdowns.

Upon every allocation of a dynamically-allocated object, the object needs to be placed in an EPD, requiring setting the protection domain tag in the corresponding page table entries (PTEs). When the object is freed, domain tags in the corresponding PTEs must be restored to 0 (§3.2), so that the freed pages may be reused by application components, such as standard libraries, that have not been annotated and instrumented for Push.

Optimization (1) reduces the number of PTE changes associated with allocate and free operations. For each protection domain number, Push maintains a simple software cache of up to 64 pages that are in that domain. The cache holds contiguous blocks of pages consisting of up to 32 pages. When possible, objects are allocated using pages from the cache and pages of freed objects are placed in the cache. Allocating objects using pages from the cache or freeing pages to the cache are done without requiring system calls. We call this cache a domain-tagged page cache (DTPC).

The DTPC is effective in reducing the overall overhead of swappings (See §4). Without the DTPC, Push increases the execution time a factor of 2, 147 with 2, 32 threads respectively. With the optimization, the overhead caused by Push is less than 2%

Optimization (2) reduces the performance overhead for TLB shootdowns. Following every change to PTEs in the page table, those PTEs must be flushed from any TLB that may cache them. In multithreaded programs, PTEs of pages containing shared objects may be cached in TLBs of multiple CPUs. TLB consistency is ensured by broadcasting IPIs (inter-processor interrupts) to all cores running the same process, causing each of them to flush the stale PTEs. The IPIs incur a significant overhead as the number of threads increase and thus limits the scalability of Push.

Optimization (2) utilizes the information provided by the Push annotations to identify whether a stale PTE can possibly be cached in remote TLBs. Specifically, private objects and inaccessible objects cannot be accessed by any remote CPUs and thus the corresponding PTEs cannot be cached in remote TLBs. Thus, when a change policy operation is applied on these objects, no IPI broadcasting is needed.

For Optimization (2), we implemented a small kernel modification, adding a local counterpart (pkey_mprotect_l) of the system call that changes the page table entry (pkey_mprotect). The local version does not broadcast IPIs for TLB flush. When a change policy operation is invoked on private or inaccessible objects, pkey_mprotect_l is invoked. It is critical to note that this works correctly only if threads are not allowed to migrate among cores. Hence, in all our experiments each application thread is pinned to a specific core.

We use pfscan (§4) to illustrate the benefit of Optimization (2). Originally, for 8 and 32 threads, the execution time increases by 31% and 80%, respectively, due to the 570 thousand calls to pkey_mprotect, where, on average, each call takes 15µs and 51µs, respectively. With the optimization, all but one pkey_mprotect is replaced by pkey_mprotect_l. The average latency of pkey_mprotect_l for 8 and 32 threads is 6µs and 34µs, respectively, resulting in application execution time increases of only 6.5% and 33%, respectively. The dependency of pkey_mprotect_l latency on the number of threads is due to contention for the kernel’s memory subsystem lock.

4 EXPERIMENTAL SETUP

The experiments are performed on a machine running Fedora 27 with Linux Kernel version 4.15.13. The machine is equipped with 192GB memory and two 2.3GHz Intel Xeon Gold 6140 processor chips. Each chip contains 18 cores and a 24.75MB L3 cache.

Ten benchmarks are used. Ctrace-1.2 [3] is a multithreading debugging library, evaluated printing 32,000,000 debug records. Pfscan-1.0 [8] is a parallel file scanner, evaluated finding 'hello' in 500 copies of DSN proceedings. Pbzzip-0.9.4 [7, 48] is the parallel version of bzip2 file compressor, evaluated compressing a 2GB file with random content. Nullhttpd-0.5.1 [6] is a simple multithreaded webserver. We evaluate it using multiple ab [1] clients to retrieve a 10KB
file 10K times. Memcached-1.2.2 [5] is an in-memory caching system. We evaluate it using Workload A and B of YCSB [17], with 100K 1KB records and 2M requests. We also use streamcluster, blackscholes, swaptions, and ferret from the PAESEC 3.0 [13, 50] benchmark suite, and fft from SPLASH2 [47], all with their native inputs.

Ctrace, pbzip2, pfsan, and nullhttpd are selected to facilitate direct comparisons with SharC [11] and DCOP [33]. For all benchmarks, inputs are set so that the execution time is at least 10 seconds with the maximum number of threads. For all benchmarks, to minimize the performance skew caused by disk I/O, all the input/output files are placed in ramdisk. All the threads are pinned to dedicate cores to eliminate the intrusion caused by thread migration. To eliminate variable network latency effects, for nullhttpd and memcached, the client programs run on the same machine as the server program on a disjoint set of cores.

Since PUSh currently only supports C, C++ benchmarks: pbzip2, streamcluster, swaptions and fft were ported to C. We have verified that performance was not affected. To remove operations that can distort performance measurements, for ctrace we removed all printf’s (which create extensive I/O activity) and replaced localtime (which serializes the threads) with a customized scalable version. We removed unnecessary sleeps in nullhttpd.

5 EVALUATION
This section presents the discovered data races (§5.1), information regarding the annotations and changes to the source code (§5.2), a validation of the rehash mechanism (§5.3), the memory overhead (§5.4), and the performance overhead (§5.5). The section includes comparisons of PUSh with SharC [11], DCOP [33], and TSan [4, 42], in terms of the required annotations and code changes as well as memory and performance overheads.

5.1 Discovered Data Races
PUSh detected ten data races: eight were violations of the intended sharing policies and two due to detection of conflicting concurrent sharing policy changes (§2.2). The data races were: one each in ferret and nullhttpd, two each in streamcluster and memcached, and four in pbzip2. Out of the eight sharing policy violations, in four one thread was attempting to read an object private to another thread. In the remaining cases, the relevant object was locked and a thread attempted to access the object without acquiring the lock.

The two races, one in nullhttpd and the other in pbzip2, identified due to detection of conflicting concurrent sharing policy changes were related to customized synchronization code, where concurrent memory accesses are used. Since the C11 C standard prohibits such accesses to normal variables, these are data races. In these cases, inserting into the code the change policy operations necessary to avoid PUSh detecting violations of intended sharing policies, resulted in PUSh detecting conflicting concurrent policy changes.

To validate PUSh’s results, we ran our benchmarks with TSan [4, 42]. Initially, TSan detected three races that PUSh did not. However, these were all related to calls to standard library functions, for which TSan uses versions which are instrumented for data race detection. As a test, we also instrumented these functions. For example, to handle a call to free() of an object as an acquire_write. Once this was done, PUSh also detected all three races.

5.2 Annotation and Code Changes
PUSh and related data race detectors [11, 12, 33] require the programmer to annotate their code. This subsection quantifies this burden, based on our benchmarks (§4). Table 1 shows the annotations overhead of PUSh. The LOC column shows the count of the lines of code in the stock benchmarks, without blank lines or comments, as measured by CLOC [2].

The Changes column shows the source code modifications, which were all due to adding the padding space for split struct (§3.4). In addition to the changes reported in Table 1, a script was used to replace all the element references for split array and sliced array with macros (§3.4). The number of lines changes by the script were 15 for fft, 72 for streamcluster, 5 for blackscholes and 30 for swaptions.

Similarly to [11, 12, 33], annotations are added using a trial-and-error approach. Initially, we run the benchmark with PUSh, without any annotations, and the annotation aborts on the first shared access. We then analyze the code and add the proper annotations. With this methodology, annotations were added to most of the benchmarks within a few hours. The one exception was streamcluster, which took around 35 hours, due to complicated object sharing patterns. It should be noted that almost all of the time spent on code annotation was on understanding the code, with which we were not familiar. Hence, the task would have been much easier for the developers or maintainers of the code.

Annotation Burden Comparison. We compare the annotation burden of PUSh to DCOP [33] and SharC [11] based on the sum of annotations and other code changes, referred to as mods. For pfsan, pbzip2, ctrace, and nullhttpd, the numbers of mods required for PUSh are 27, 31, 20, 3. DCOP [33] requires 62 (2.3x), 103 (3.3x) and 41 (2.1x), 13 (4.3x). DCOP’s burden is higher since: (1) DCOP lacks the locked policy and thus requires insertion of two annotations for every object accessed in every critical sections, and (2) with DCOP, for statically-allocated objects, the default sharing policy is inaccessible, as opposed to untouched with PUSh (§3).

For pfsan and pbzip2 the numbers of mods required for PUSh are 27 and 31. The SharC [11] mods counts are comparable: 19 and 46. In general, compared to PUSh, the SharC (and Shoaib [12]) type system requires additional annotations on, for example, function arguments, local variables, and function return values. On the other hand, unlike PUSh, SharC does not require annotations before
and after accessing read-shared objects. The barrier annotations supported in Shoal (§7), can significantly reduce the annotation overhead for programs with many barriers, such as streamcluster.

## 5.3 Validation and Effectiveness of Rehashing

This section demonstrates the effectiveness of the mechanism that periodically changes the hash function that maps thread IDs and lock addresses to MPK domains (§3.3). This is done using the eight data races reported in §5.1 detected as violations of intended sharing policies. These races were detected in ferret, memcached, pbzip2, and streamcluster.

In one test, the rehashing interval was set to five seconds and the thread count to 7-8. Each one of the four benchmarks was executed 50 times. The execution times were in the range of 15 to 222 seconds. For four of the benchmarks, all the races reported in §5.1 were detected in every execution. For pbzip2, out of the 50 executions, 43 detected all three data races. In each of the remaining seven runs, one race was missed. The missed race was always one of the two races that occurs only once, when the program exits. As discussed in §3.3, for such races, rehashing during a single execution obviously does not help.

In a second test, to increase the potential impact of domain collisions, thread IDs and lock addresses were mapped to only two MPK domains. Furthermore, the applications were run with the minimal number of threads that can trigger the race. With a fixed hash function, six out of the eight data races have a 50% probability to escape detection. The two remaining races are in Streamcluster, that performs multiple iterations on the input data. Each iteration creates new threads, with new thread IDs, that are terminated at the end of the iteration. Consequently, even with this setup, the probability of these races escaping detection is much smaller than 50%. Thus, in our experiments, they were always detected.

Changing the hash function once per second resulted in the detection of four out of the six problem data races in every execution. In those cases, the relevant code executed multiple times during a single execution of the program. As in the first test above, periodically changing the hash function did not help with two of the data races in pbzip2 that only occur when pbzip2 finishes processing.

## 5.4 Memory Overhead

This section presents PUSH’s memory overhead, including an evaluation of the effectiveness of the optimization mechanisms in §3.4.

We measured the memory usage of PUSH by measuring the maximum virtual memory size (vsize) and resident set size (rss) during the execution of each of the benchmarks with the maximum number of threads. Each of the benchmarks was executed twenty times, and the average results are reported. The measurement variations were less than 2%.

We compare the results for the stock benchmarks, linked with tcmalloc [9] (§3.1) to the results with the full overhead of PUSH, which includes additional libraries, memory fragmentation, the metadata. These measurement are not completely accurate due to, for example, the granularity at which tcmalloc allocates virtual memory to the application.

As shown in Table 2, seven of the benchmarks do not have a measurable overhead in terms of both vsize and rss. With Swaptions, both Memcached workloads, and Ferret, the increases are 7MB, 452MB, and 553MB, respectively, in both vsize and rss. Since the memory overhead increases negligibly with the number of threads, we report the results with just the maximum number of threads (31 or 32).

As explained in §3.1, PUSH’s memory overhead is directly related to the number of tracked objects, since it is due to fragmentation and metadata associated with each object. Thus, the memory overhead results can be explained using Table 3, which shows the maximum number of tracked objects in each application. For most of the benchmarks, the number of tracked objects is so small that the memory overhead caused by them is below what our measurement procedure can detect. There are a few other sources of memory overhead, such as the DTPC (§3.5). However, with our benchmarks, the overhead due to these other sources is negligible.

The memory overhead due to metadata can be calculated based on the number of tracked objects shown in Table 3. In the best case, with no read-shared objects, the size of the metadata for each object would be 48 byte (§3.1). Thus, for swaptions, memcached, and ferret the required storage for metadata would be 86KB (2% of rss), 4.8MB (5% of rss), and 6.6MB (6% of rss), respectively. This overhead is negligible in terms of vsize. Half of this overhead is due to happens-before tracking. This best-case metadata overhead is not a significant portion of the total overhead reported above.

In the worst case for metadata overhead, every object is read-shared, requiring a full vector clock for happens-before tracking (§2.2). However, with our benchmarks, due to the sticky-read sharing policy (§2.2), the number of read-shared objects was less than 100. Ferret benefits most from the sticky-read policy since 99% of its objects are read shared by multiple threads. With the sticky-read sharing policy, the metadata overhead in Ferret is only 6% of rss.

In most cases, the major source of memory overhead is fragmentation (§3.1). Several memory overhead optimization techniques are presented in §3.4. For our benchmarks, there are few opportunities to benefit from the split struct and split array optimizations. The

### Table 2: Memory overhead. V: vsize, R: rss, max #threads.

<table>
<thead>
<tr>
<th>Type</th>
<th>ett</th>
<th>ift</th>
<th>bik</th>
<th>plac</th>
<th>snat</th>
<th>swat</th>
<th>pbzip</th>
<th>mf</th>
<th>mem</th>
<th>ferret</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>~0</td>
<td>~0</td>
<td>~0</td>
<td>2.3%</td>
<td>~0</td>
<td>~0</td>
<td>~0</td>
<td>~0</td>
<td>260</td>
<td>127%</td>
</tr>
<tr>
<td>R</td>
<td>~0</td>
<td>~0</td>
<td>~0</td>
<td>64%</td>
<td>~0</td>
<td>~0</td>
<td>~0</td>
<td>~0</td>
<td>309</td>
<td>444%</td>
</tr>
</tbody>
</table>

### Table 3: Maximum number of tracked objects in each application with different thread counts. Memcached results are the same with both workloads.

<table>
<thead>
<tr>
<th>#Threads</th>
<th>ett</th>
<th>ift</th>
<th>bik</th>
<th>plac</th>
<th>snat</th>
<th>swat</th>
<th>pbzip</th>
<th>mf</th>
<th>mem</th>
<th>ferret</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>19</td>
<td>53</td>
<td>20</td>
<td>37</td>
<td>39</td>
<td>1180</td>
<td>NA</td>
<td>38</td>
<td>100K</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>25</td>
<td>71</td>
<td>26</td>
<td>49</td>
<td>75</td>
<td>1834</td>
<td>78</td>
<td>44</td>
<td>100K</td>
<td>23</td>
</tr>
<tr>
<td>16</td>
<td>33</td>
<td>95</td>
<td>34</td>
<td>65</td>
<td>113</td>
<td>1852</td>
<td>141</td>
<td>52</td>
<td>100K</td>
<td>27</td>
</tr>
<tr>
<td>32</td>
<td>49</td>
<td>143</td>
<td>50</td>
<td>97</td>
<td>219</td>
<td>1861</td>
<td>268</td>
<td>68</td>
<td>NA</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>142K</td>
</tr>
</tbody>
</table>
most common situation for split struct was where one of struct members is a lock protecting the struct and must thus be a separate tracked object. The largest benefits of split struct was with ferret, that has five instances for which split struct was useful. For most of the arrays, the number of elements was well beyond the limit of 10000 that can be handled by the split array optimization. The largest reductions in memory overhead were 512KB with swapoptions and 2MB with streamcluster.

For our benchmarks, the most effective memory overhead reduction technique was sliced array. Several data-parallel programs, such as fft and blackoles, have a simple sharing pattern: each thread works on a disjoint set of the array elements. Such patterns are a good match for sliced array. Since these arrays are large, without sliced array, the memory overhead would have been prohibitive. The largest memory overhead reduction was with fft. Without sliced array, a total of 4TB of memory is needed for two large arrays. However, with sliced array, the memory overhead is negligible.

Memory Overhead Comparison. The most closely-related annotation-based race detectors [11, 33], were not available to us for evaluation with our configuration. Hence, we compared the published results for several benchmarks that we also evaluated with PUSH. With SharC [11], benchmarks in common are pbzip2 and pfscan. For these benchmarks, the memory overheads of PUSH (Table 2) and the reported memory overhead of SharC were negligible. With DCOP [33], benchmarks in common are ctrace, nullhttpd, pbzip2, and pfscan. For all these benchmarks, the memory overheads of PUSH were negligible, while the reported overheads for DCOP were in the range of 6.5% to 14%.

If [11, 33] were enhanced to flag unordered sharing policy changes (§2.2), their memory overhead for every dynamically-checked 16-byte block would increase from one byte by at least 16 more bytes. PUSH requires the extra tracking metadata per object, as opposed to per fixed-size block. Thus, at least for the benchmarks evaluated, PUSH would have a much greater memory overhead advantage.

To compare PUSH’s memory overhead with that of TSsan, we evaluated TSsan on our experimental platform. The results in Table 4 are based only on rss since, due to TSsan’s implementation, vsze measurements for it are meaningless [4]. The results are that, for nine out of the ten benchmarks, PUSH’s memory overhead is lower, much lower for eight of them. With ferret, PUSH’s memory overhead is a little higher, but this must be weighed against PUSH’s dramatically lower (factor of 145) performance overhead 4.

As in Table 2, the results in Table 4 are with the maximum number of threads. With TSsan, for eight of the benchmarks, the number of threads does not affect the memory overhead. For pbzip2 and pfscan, TSsan’s performance overhead is very high (§5.5) and this affect the execution characteristics with TSsan so that the maximum memory use does decrease with decreasing thread counts. The memory overhead for pfscan is still much higher than PUSH’s—at least 300% for all thread counts. For pbzip2 the maximum memory use is significantly lower than that of the stock application for low thread counts.

### 5.5 Performance Overhead

This section presents the performance overhead of PUSH with the performance optimization discussed in §3.

Table 5 presents the performance overhead of PUSH, varying the number of threads from 1 to 32. For most of the benchmarks, the performance measure is execution time, while for memcached and nullhttpd, it is maximum throughput. The reported overheads are averages over 20 executions. The measurements varied less than 2% over 20 runs. An NA in a cell of the table indicates that the benchmark could not be run with the corresponding number of threads.

As shown in Table 5, for 7 out of 10 benchmarks, the performance overhead is under 19% for all thread counts. Memcached with YCSB Workload A is the worst case, reaching 59% overhead with two threads and 99% overhead with eight threads.

For ctrace and memcached-B, due to frequent lock operations, most of the performance overhead is caused by modifying PKRU registers (13ns) and happens-before tracking. For fft and blackoles, the main overhead is due to additional time spent accessing the array elements for sliced array (§3.4). For all the other benchmarks, most of the overhead is due to page table permission changes caused by sharing policy changes. The lower part of Table 5 presents the policy change rate for the different benchmarks. In all the cases where the performance overhead of PUSH is over 18%, that overhead is highly correlated with the policy change rate.

For most of the benchmarks, the performance overhead of the happens-before tracking is insignificant. The exceptions are ctrace and memcached-B. For ctrace, with one thread, happens-before
tracking is responsible for 33% out of the 18% overhead. When the number of threads is 16 or higher, this tracking is responsible for nearly all the overhead. For memcached-B, happens-before tracking it responsible for 17% to 47% of the overhead.

For streamcluster and pfscan, the overhead of PUSh is highly dependent on the number of threads. This is due to the fact that, in the Linux kernel, a single lock (mmap_sem) serializes all page table changes invoked by the threads of a process. To directly quantify the impact of the serialization caused by the mmap_sem, we measured the average time spent in page table changes by streamcluster. With two threads, only one of which is active when the page table change is invoked, the average latency of the operation was 1µs. With 32 threads, this latency was 89µs.

As discussed in §3.3, PUSh must periodically change the hash function used to hash thread IDs and lock addresses to MPK domain numbers. The latency of this operation contributes to the performance overhead of PUSh. We measured this latency for all the benchmarks for all thread counts. The highest latency for changing the hash function was 137ms, for memcached with 8 or 16 threads. For all the other benchmarks, the highest latency was 26ms. These results indicate that, for most benchmarks, if the hash function is changed every few seconds, the associated overhead is negligible.

The most significant factors that determine the latency of changing the hash function are the number of private or locked objects and the total size (number of pages) of those objects. Both of these factors affect the time spent resetting the protection domains. Memcached is an extreme case since it has a large number of locked objects.

**Performance Overhead Comparison.** With SharC [11], benchmarks in common, pbzip2 and pfscan, were executed with thread counts of 5 and 3, respectively, with reported performance overheads of 11% and 12%, respectively. PUSh’s corresponding overheads were 2% or less, with overheads of 6.5% or less for 8 or fewer threads (Table 5). With DCOP [33], benchmarks in common ctrace, nullhttpd, pbzip2, and pfscan were executed with thread counts of 2, 50, 5, and 3, respectively, with reported performance overheads of 27%, 0, 49% and 37.2%, respectively. PUSh’s closest corresponding overheads, as presented in Table 5, were significantly lower: 14%, 0, 1.9%, and 2%, respectively. PUSh had lower overheads even with 32 threads.

As explained in §4, we modified ctrace and nullhttpd in order to be able to obtain meaningful performance results. However, for this comparison with DCOP, we also evaluated the original versions of ctrace and nullhttpd. With the unmodified ctrace, PUSh does not incur any performance overhead. For the original nullhttpd with 50 threads, DCOP reports and overhead of 24% in CPU cycles, while PUSh does not incur any overhead in CPU cycles, throughput, or latency.

If [11, 12, 33] were enhanced to flag unordered sharing policy changes (§2.2), PUSh’s performance overhead advantage would be even greater. Specifically, for each object larger than 16/8-bytes, these operations would require checking and modifying tracking metadata for multiple blocks, as opposed to PUSh, where the metadata is per object.

Table 6 presents a performance overhead comparison between PUSh and TSan. As discussed in §1, PUSh’s overhead is at least a factor of 7.8 lower and, in several cases three orders of magnitude lower. These results reinforce the argument made in §1 that PUSh can be used in production runs while TSan is restricted to offline debugging.

### 6 LIMITATIONS AND DISADVANTAGES

Like all data race detectors, PUSh has limitations and disadvantages. Some of these are inherent (§2), some are associated with tradeoffs made for efficient implementation, and the rest are due to implementation enhancements that have not yet been done. This section summarizes these limitations and disadvantages. Despite these, as shown in other sections of this paper, PUSh, as currently implemented, is a useful tool with important advantages over other existing race detectors.

A key disadvantage of PUSh is that it requires annotation of the code. Furthermore, since PUSh relies on happens-before tracking [23, 36] (§2.2), races that, in a single execution, can escape detection by full happens-before tracking [41], can also escape detection by PUSh.

Implementation considerations motivate the mapping of LPDs to a smaller number of EPDs (§2.3). A consequence of that is that all read-shared objects are placed in a single EPD and this introduces the possibility of failing to detect races in limited specific scenarios. The locked sharing policy is also introduced as a performance optimization (§2.3). Associated with this is the fact that, under certain scenarios, a change policy from locked can result in missed races.

PUSh’s current implementation relies on placing every object in a separate virtual page (§3.1). This can lead to prohibitive memory overhead. PUSh’s optimizations that mitigate this problem (§3.4), result in PUSh generally having low memory overhead relative to competitive schemes (§5.4). However, some of these optimizations are constrained by limitations of the Linux memory system (§3.4), and we have not yet implemented kernel changes to overcome these limitations. PUSh’s implementation relies on MPKs (§3.2). The problem of potential missed races due to the limited number of MPKs is mitigated by periodically changing the hash function (§3.3). While this rehashing mechanism is highly effective (§5.3), there is, still, a small probability of missing races due to hash collisions, especially for races that occur only once during the execution of the program. The pkey_mprotect_j optimization (§3.5) requires threads to be pinned to cores. If this cannot be done, in some cases, the performance overhead may increase significantly.

---

**Table 6: Performance overhead comparison: additional execution time as percentage of the stock application execution time. P: PUSh vs. T: TSan.**

<table>
<thead>
<tr>
<th>Type</th>
<th>ctrace</th>
<th>ft</th>
<th>blksch</th>
<th>pfscan</th>
<th>stimul</th>
<th>tt</th>
<th>phip2</th>
<th>rhibpd</th>
<th>mem$A</th>
<th>mem$B</th>
<th>ferret</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>7</td>
<td>3.0</td>
<td>2.0</td>
<td>6.5</td>
<td>13</td>
<td>1.1</td>
<td>1.9</td>
<td>~0</td>
<td>99</td>
<td>39</td>
<td>8.2</td>
</tr>
<tr>
<td>T</td>
<td>423</td>
<td>507</td>
<td>473</td>
<td>13K</td>
<td>2567</td>
<td>746</td>
<td>4245</td>
<td>529</td>
<td>780</td>
<td>384</td>
<td>1612</td>
</tr>
<tr>
<td>Max #threads</td>
<td>8</td>
<td>12</td>
<td>3.3</td>
<td>2.1</td>
<td>33</td>
<td>99</td>
<td>1.4</td>
<td>3.9</td>
<td>~0</td>
<td>82</td>
<td>23</td>
</tr>
<tr>
<td>T</td>
<td>304</td>
<td>392</td>
<td>1420</td>
<td>36K</td>
<td>6588</td>
<td>693</td>
<td>3940</td>
<td>2485</td>
<td>873</td>
<td>356</td>
<td>1595</td>
</tr>
</tbody>
</table>
7 RELATED WORK

There is a large variety of works on data race detection based on static analysis, dynamic checking, or a combination of the two. Tools based on static analysis do not incur overhead during normal operation and can find data races in rarely-executed code sections. However, such tools are susceptible to a high rate of false positives and/or negatives when applied to C [21, 37, 45]. Dynamic checkers generally perform lockset analysis [20, 41], or happens-before analysis [23, 28, 42, 52] based on instrumentation of normal memory accesses, thus potentially incurring high performance overhead. There are also techniques that rely on customized hardware support [31, 38, 53]. Some mechanisms combine static analysis and dynamic analysis to reduce the runtime overhead [11, 12, 16, 20, 33].

Tools based on sampling [14, 32, 43, 51] target potential use in production, sacrificing soundness for low performance overhead. For such tools, the accuracy is generally in the range 2%-70%, some with performance overheads in a range similar to PUSH’s ([§5.5]). Unlike such tools, PUSH continuously checks all memory references and, based on §5.3, provides higher accuracy. However, the sampling tools do not require annotation and have negligible memory overhead.

There are several works that utilize page-level protection to facilitate data race detection [34, 35, 39]. MultiRace [35] uses page-level protection to detect the first access to a shared memory location in each time frame. Aikido [34] develops per-thread page tables to identify pages shared by multiple threads and provide machinery to instrument memory accesses to these shared pages. The closest work to PUSH is ISOLATOR [39], as discussed in §1.

Compared with many other dynamic race detectors, the overhead of PUSH is quite low. For comparison, relevant results can be found for IFRIt [19]; TxRace [52]; FastTrack [23] and ThreadSanitizer [4, 42], with the data reported in [29]. With eight threads, for blackscholes, streamcluster, swaptions, and ferret, these mechanisms incur overheads of 1.82x-79.2x, while with PUSH no overhead is greater than 1.13x. For streamcluster with 32 threads, the overhead for three-state-of-the-art tools are 20x-400x [29]. With PUSH, the overhead is 1.99x.

PUSH is closely related to SharC [11], Shoal [12] and DCOP [33]. As discussed in §1, the fundamental differences between PUSH and these works are (1) PUSH uses hardware to enforce sharing policies, and (2) PUSH uses happens-before tracking to identify conflicting concurrent sharing policy changes that can result in false negatives.

To reduce the overhead during execution, SharC uses static analysis to enforce the private and readonly policies. Other sharing policies are enforced during runtime by instrumenting memory accesses to those objects. To facilitate static analysis, the following requirements are imposed: (1) the sharing policy of an object can only be changed when its reference count is 1, and (2) after the policy change, the pointer pointing to an object whose sharing policy is changed is set to NULL. These requirements make SharC difficult to apply with data structures such as linked lists and trees. Shoal [12], partially mitigates the impact of SharC’s restrictions by introducing the concept of groups of objects whose sharing policy can be atomically changed. Shoal also adds the barrier sharing policy, which allows an object to be either read only or write by one thread between two barrier operations. However, there remain restrictions on dynamic policy changes, which increase the burden on the programmer. Furthermore, for both SharC and DCOP, there is a need for a runtime mechanism to track object reference counts and this incurs a performance overhead of up to 30% [12].

DCOP [33] instruments memory accesses to enforce the specified sharing policies in the runtime. Static analysis is used to eliminate redundant instrumentation. The core annotations of PUSH (§2.1) are essentially identical to DCOP [33]. Furthermore, DCOP also uses the sticky-read annotation. However, the purpose of sticky-read in DCOP is to reduce annotation burden, while in PUSH it is also used to reduce the memory overhead. The differences in annotations between DCOP and PUSH are discussed in §5.2. In general, DCOP has more annotation overhead than PUSH, as well as higher performance and memory overheads.

8 CONCLUSION

Decades of development of data race detectors have resulted in a rich design space of techniques and tools that vary widely in terms of precision, soundness, types of races detected, performance overhead, memory overhead, scalability, burden imposed on programmers, and applicability to various programming languages. A useful subspace of this design space covers race detectors that require explicit specification of intended sharing and then identify violations of these intentions. Since such tools don’t have to infer the intended sharing, they have the potential of reduced complexity, increased precision and soundness, and reduced overhead.

PUSH advances the state-of-the-art in the design subspace described above by introducing the use of happens-before tracking to check for conflicting concurrent sharing policy changes. A second key novel feature of PUSH is the way it uses off-the-shelf hardware to reduce the performance overhead for detecting unintended sharing. A key aspect of the design and implementation of PUSH is the use of memory protection keys (MPKs). PUSH uses a universal family of hash functions to overcome the limitations of the widely-available implementation of MPKs. Additional optimizations that range from enhanced annotations to OS kernel changes further reduce memory and performance overheads. In many cases, the performance overhead of PUSH is at a level that allows its use during production runs.

We evaluated PUSH with ten benchmarks and up to 32 threads. PUSH detected ten races. Comparison with results from ThreadSanitizer [4, 42] shows that, excluding races due to standard library calls, no data races were missed. PUSH’s memory overhead was under 2.4% for eight of the benchmarks. PUSH’s performance overhead exceeded 18% for only three of the benchmarks, reaching 99% overhead in the worst case. Our work included targeted evaluations of the various optimization introduced in this work. For example, we have shown that, in an extreme case, PUSH’s novel use of MPKs reduces the performance overhead from a factor of 32.2 to just 18%. We have also shown how additional annotations can dramatically reduce PUSH’s memory overhead.

ACKNOWLEDGMENTS

Michael Mitzenmacher provided the upper bound presented in §3.3 and reference [44]. We also thank the reviewers for their helpful feedback and George Varghese for the computing resources.

