Making User-level VMM for Deterministic Parallelism Nonblocking and Efficient

Yu Zhang*, Jiange Zhang, Qiliang Zhang
School of Computer Science and Technology, University of Science and Technology of China, China
Email: *yuzhang@ustc.edu.cn

***Abstract*—Many parallel programs are intended to yield deterministic results, but unpredictable thread or process interleavings can lead to subtle bugs and nondeterminism. We proposed a *producer-consumer* virtual memory—SPMC—for efficient system-enforced deterministic parallelism, and prototyped the SPMC model and its software stack entirely in Linux user space, called DLinux. This paper summarizes the implementation policies and limitations in our previous DLinux. To reduce SPMC page fault overhead and suspend/resume overhead which severely degrade the performance of DLinux, we enhance the SPMC model with nonblocking test and direct read and write primitives. Based on the extended SPMC model, we improve the implementation of upper programming abstractions. Experimental results show that relative to the previous version, the new DLinux can improve the performance of NPB workloads up to 2.33X and 1.76X on 8 and 16 processes, respectively. For CG on 8 processes, its runtime relative to MPICH2 decreases from 4.12X to 1.77X.**

I. Introduction

A parallel program is *deterministic* if, for a given input, every run of the program yields the same output [1]. Determinism offers benefits for replay debugging, and security. However, executing parallel software deterministically is challenging, because threads or processes sharing resources are prone to nondeterministic, timing-dependent races [2].

Current methods of executing parallel software deterministically [3–8] show promise but have limitations. Deterministic multithreading (DMT) systems [4–8] force multithreaded programs to always execute the same thread interleaving or schedule on the same input. These schedulers emulate conventional APIs by synthesizing a repeatable but arbitrary schedule of inter-thread interactions, incurring high overheads [9], and cannot give clear semantics implied by the program's logic. Determinator [3] enforce determinism at OS level, but only supports hierarchical synchronization such as fork/join.

Like Determinator, we would like a race-free OS-level approach to enforce deterministic parallelism with clear semantics. We proposed SPMC (single-producer/multi-consumer) virtual memory (VM) [10], a novel OS foundation for deterministic peer-to-peer "dialog" between processes or threads, short-circuiting the process hierarchy. The SPMC model integrates synchronization into VM by distinguishing two types of memory mappings. An SPMC page frame has a single producer mapping at a time, but any number of consumer mappings. SPMC VMM (VM management) system enforces that consumers cannot read an SPMC page frame until its producer explicitly publishes (called *fixes*) it, rendering it read-only. All consumers read the same SPMC page frames at their own pace, supporting asynchronous multicast communication.

With scalability and effectiveness in mind, we proposed *lazy-tree-mapping* VMM mechanism [11], which could effectively represent an infinite stream on a finite virtual address range. We wrapped SPMC regions to provide high-level synchronization abstractions [10–12], e.g.,*channel* for deterministic message passing (DetMP), hiding synchronization details.

We prototyped SPMC VMM in Determinator OS kernel first [10], then in Linux user space entirely (*i.e.,*DLinux) to support more realistic applications [11–13]. Although DLinux exhibited performance and scalability comparable to the nondeterministic system on some pipeline workloads, it had poor performance on most of NPB-MPI workloads [14].

In this paper, we summarize the implementation policies and limitations in DLinux. By evaluation, we observe that SPMC page fault and suspend/resume overheads severely degrade the performance. This paper makes the following contributions:

1) Our previous blocking SPMC model would suspend any consumer attempts to read an unfixed page. To minimize suspend/resume overhead, we enhance the SPMC and DetMP with *nonblocking* test, thus a consumer can test before reading to avoid getting suspended.

2) DLinux uses `mmap` shared VM to emulate page frames. The previous page-by-page mapping and "copy-twice" in an SPMC-page read or write cause more page faults. To reduce page fauls, we add `write`/`read` SPMC primitives to allow directly copying data without triggering SPMC page faults.

We evaluate DLinux and a nondeterministic popular MPI implementation–MPICH2 [15] using NPB. Experimental results show that relative to the previous version, the optimized DLinux improves the performance up to 2.33X and 1.76X on 8 and 16 processes, respectively; for `CG` on 8 processes, its runtime relative to MPICH2 decreases from 4.12X to 1.77X.

II. The SPMC Model

The SPMC model allows a process to establish direct "peer-to-peer" SPMC regions between its arbitrary descendants, eliminating it as a scalability bottleneck in subsequent computation. The SPMC region is introduced by distinguishing *producer mapping* and *consumer mapping*, to the same shared page frames, shown on Fig. 1.

A. Protocols for Race Freedom

SPMC VMM system should follow the protocols below:

978-1-5090-5081-9/16 $31.00 © 2016 IEEE
DOI 10.1109/PDCAT.2016.41

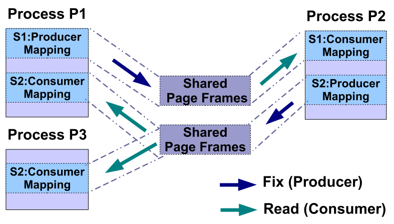


Fig. 1. Processes sharing SPMC regions.

Single-producer-only: To any SPMC region, there must be only one producer at any moment, accordingly avoiding WAW (*write-after-write*) races.

Consumed-after-fixed: Any consumer can read an SPMC page only after the producer explicitly publishes (*fixes*) it. Any attempts to access an unfixed SPMC page shall be **blocked**, avoiding RAW (*read-after-write*) races.

Fixed-at-most-once: The producer can explicitly fix each page frame mapped to its virtual page at most once, and then lose write permission to the page. If the producer wants to rewrite the page with new data, it can explicitly remap the page to new page frame without affecting old page frames, accordingly avoiding WAR (*write-after-read*) races.

B. Software Stack atop the SPMC Model

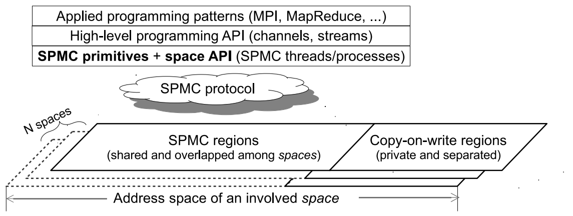

Applied programming patterns (MPI, MapReduce, ...)
High-level programming API (channels, streams)
SPMC primitives + space API (SPMC threads/processes)
SPMC protocol
N spaces
SPMC regions (shared and overlapped among spaces)
Copy-on-write regions (private and separated)
Address space of an involved space

Fig. 2. Software stack on the SPMC model.

Fig. 2 shows the software stack we have built on the SPMC model. We proposed new user-level thread and process models atop the SPMC model, and use the term "*space*" to abstract our "thread" and "process". Each space has its own CPU register state for a single control flow, as well as copy-on-write memory by default containing code and data directly accessible within the space. As in a nested process model [16], a space cannot outlive its parent, and can directly interact with its immediate parent and children via proposed space API (shown on Table I), or with other spaces via explicitly declared SPMC regions. Currently there are only per-space copy-on-write heaps, occupying distinct virtual address ranges.

With scalability and effectiveness in mind, we have proposed a *lazy-tree-mapping* for SPMC VMM [11], which contains two orthogonal techniques—*lazy page mapping* and *space extension*. The former introduces a special anchor page table to support lazily allocating page frames "on demand". The latter enables `extend` primitive to remap a finite virtual address range to lazily-generated page frames organized in

TABLE I
MAIN SPACE AND CHANNEL FUNCTIONS

`sid = space_alloc(gid)`	Allocate a child space of global ID and return its internal ID.
`space_start(childsid, fn, args)`	Start a child space with specified internal ID to run `fn(args)`.
`void space_sync(childsid, restart)`	Wait for the specified child to return and restart the child if needed. *space_join(c)* is a syntactic sugar of *space_sync(c, false)*, which waits for the child to terminate.
`void space_ret()`	Return to its parent space.
`cino=chan_alloc()`	Allocate a channel and return its ID.
`chan_setprod(cino, childsid, ascons)`	Set a given child space producer of the channel.
`chan_setcons(cino, childsid)`	Set a given child space consumer of the channel.
`chan_send/sendLast(cino, buf, sz)`	Send a message stored in `buf` of `sz` bytes via the channel.
`sz=chan_recv(cino, buf)`	Receive a message of `sz` bytes from the channel and save it in `buf`.

the form of an arbitrarily deep tree, effectively representing an infinite stream or infinite versions of shared data.

At programming level, by wrapping an SPMC region we proposed two kinds of high-level synchronization abstractions: SPMC *channels* for message passing (called DetMP) [10, 11] and SPMC *streams* for deterministic stream sharing (called DStream) [12]. Based on the API shown on Table I, we have implemented a subset of standard MPI which has deterministic semantics (called DetMPI) and a deterministic MapReduce library for a single multicore system (called DMR) [13].

At implementation level, we first enhanced Determinator OS kernel to prototype SPMC primitives and implemented DetMP library atop it, showing promise [10]. Since Determinator is an experimental OS and falls short of drivers and `libc` functions etc., it can only support limited small programs. To support more realistic applications, we then retrofitted the SPMC into Linux, called DLinux. For quick prototype, DLinux is emulated entirely in user space. Although DLinux exhibits performance and scalability comparable to the nondeterministic system on some pipeline and MapReduce workloads [11–13], it suffers from poor performance for HPC programs such as NPB.

C. SPMC Primitives

Spaces manipulate SPMC regions via explicit primitive calls and implicit SPMC page faults.

1) Primitives to set up or destroy an SPMC region: A space can call `va = spmcR_alloc(size)` to create an SPMC region of address range [*va*, *va*+*size*) in the current space, setting it the producer of the region. A producer can invoke `spmcR_transown(child,sva, dva,size)` to transfer producer mappings in its address range [*sva*, *sva*+*size*) to the given child's address range [*dva*, *dva*+*size*), making itself become a consumer. A producer or consumer can call `spmcR_copycons (child,sva,dva,size)` to copy consumer mappings from its address range to the child space's destination range. Finally, a space can call `spmcR_clear(va, size)` to clear its mappings in the given SPMC address range.

The whole produce-consume relationship of an SPMC region is set up by several parent-to-child interactions, *i.e.*, `transown` or `copycons` primitive calls, which make the relationship conceptually simple and deterministically manageable. Once the relationship of an SPMC region is set up, peer-to-peer dialogs can be done among all involved spaces, short-circuiting the communication hierarchy.

2) Accessing an SPMC region: Any attempts to access an SPMC page with invalid mapping would raise an SPMC page fault. If raised from a producer, the fault handler will actually allocate a real page frame and update the faulting page with a writable producer mapping. If a consumer is attempting to read an unfixed SPMC page, due to blocking-based "consumed-after-fixed" protocol, the handler will suspend the consumer.

3) Primitive to fix an SPMC region: The producer can invoke `spmcR_setfix(va,size)` to fix the given SPMC region [*va*,*va*+*size*), setting each SPMC page in the region read-only, then awaken each waiting consumer.

4) Primitive for space extension: A producer or consumer can call `spmcR_extend(extva, va,size)` to extend its address range [*va*,*va*+*size*) via an extension page starting from *extva*. For the producer, the `extend` primitive will remap the address range to new generation of page frames and save remapping information into the extension page. While for the consumer, the primitive will get remapping information via the extension page, then remap its address range to the next generation of page frames according to the information.

III. Implementation Policies and Limitations

We now describe DLinux, a system built in Linux user space to offer SPMC foundation for deterministic parallelism.

A. Spaces

DLinux executes application code in an arbitrary hierarchy of *spaces*, and emulates a space using a single-threaded Linux process to achieve cross-space memory isolation by default. Each space can call SPMC primitives to explicitly set up SPMC regions for deterministic communication among spaces.

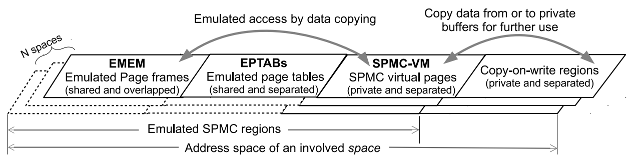
1) Deterministic space ID: Linux does not guarantee the value of a process ID deterministic. To avoid exposing this nondeterminism to spaces, DLinux gives each space a unique internal space ID. The space ID is used to identify a child to be operated in parent-to-child functions shown on Table I.

2) Global structure: The global structure is kept in an `mmap` shared memory so as to be accessed by SPMC VMM system. It is initialized in `spmc_init(nspaces)`, which is directly or indirectly called by application code. The global structure holds shared metadata for each space and each SPMC region.

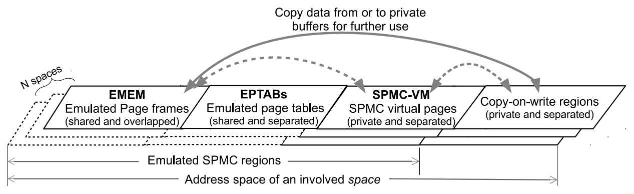
3) Space functions: `space_alloc()` allocates a free space from spaces in the global structure, and `space_start()` wraps a Linux `fork` call with additional operations on SPMC memory and metadata. DLinux uses Linux pipes to synchronize between parent and child spaces. For example, the parent writes a signal indicating whether to restart to a pipe in `space_sync()`, while the child reads the signal from the pipe to decide whether to continue running in `space_ret()`.

4) Heaps: Each space has a separate sub-heap, managed by a variant of Doug Lea's malloc [17] with about 170 modified lines of code. DLinux initializes a `mmap` private memory as the whole area for all disjoint sub-heaps.

B. SPMC VMM system



(a) Old: copy-twice in an SPMC page read/write



(b) New: direct read/write between private buffer and epages

Fig. 3. Sketch of space implementation in Linux user space.

Due to the user-level implementation, DLinux cannot control actual page mapping in Linux kernel, but uses Linux virtual memory to emulate SPMC VMM. As shown on Fig. 3(a), DLinux creates two chunks of `mmap` shared memory to emulate page frames (EMEM) and per-space page tables (EPTABs), as well as a chunk of `mmap` private memory for SPMC-VM.

1) EMEM: Each emulated page frame ("epage" for short) has a corresponding per-page metadata (*i.e.,* "pageinfo"). Both epages and pageinfos are stored as two arrays. DLinux maintains a freelist of epages to provide a simple epage allocator.

2) EPTABs: For simplicity, each space's emulated page table ("eptab" for short) is just one-level page table, mapping SPMC virtual pages ("vpage" for short) to epages. Each vpage has a 32-bit page table entry (PTE) in the eptab, which consists of `P`, `W`, `O` bits and a 28-bit address indicating the mapped epage. Bits `P`, `W` and `O` indicate whether the epage is present, writable or produced by the current space, respectively.

3) SPMC-VM: To control access privilege to SPMC regions for different spaces, DLinux creates a chunk of fixed-size `mmap` private memory to serve as SPMC-VM and controls access permission to SPMC vpages via disciplined use of `mprotect` system calls. Thus some read/write operations on SPMC vpages from application code would trigger segmentation faults, the control is then transferred to the segmentation fault handler, emulating SPMC page faults and their handler.

4) Access control and SPMC page faults: DLinux currently adopts lazy-tree-mapping mechanism. When an SPMC region is created via `spmcR_alloc` or transferred to a space via `spmcR_copycons` or `spmcR_transown`, real data epages have not yet been allocated, so SPMC vpages in the region will be set to PROT_NONE via `mprotect`. Any attempt to access a PROT_NONE vpage would trigger a segmentation fault.

If a PROT_NONE vpage fault comes from a producer, the fault handler would allocate a data epage and map it to the faulting vpage, then upgrade the vpage to PROT_WRITE via `mprotect`, so that the producer can directly write. The `spmc_setfix` handler would copy the content from the vpages to the relative epages so that consumers can access later, then downgrade the vpages to PROT_READ, finally awaken any consumer waiting for them via `kill` system call.

If a vpage fault comes from a consumer, the fault handler checks whether the corresponding data epage has been fixed or not. If not, `sigsuspend` system call is invoked to block the consumer. If so or the blocked consumer is resumed, the handler would copy the content from the epages to the consumer's SPMC vpages, and set the SPMC vpages PROT_READ. Thus the consumer can read data directly from its SPMC vpages.

C. Channels

For user-level libraries atop the SPMC mentioned in Section II-B, we only introduce the implementation of channel and its API (DetMP) for lack of space. Each channel (multicast or unicast) has a single producer and one or more consumers, which is implemented by wrapping a fixed-size SPMC region (512KB by default) to hold messages sent by the producer. The keys to implementing channels are how to enforce determinism at message granularity, how to transfer unknown number of messages of unknown size via a fixed-size SPMC region.

1) Channel metadata and inodes: DetMP separates available SPMC-VM into multiple fixed-size regions to serve as channel *inodes*, and creates a `mmap` private memory to store per-space metadata for each channel inode. Due to space extension capability introduced by lazy-tree-mapping, a channel only needs to wrap one inode of at least 8KB, in which the tail 4KB-page serves as an extension page. The per-space metadata of a channel records its status such as offset at which a producer puts or a consumer gets the next message, so that each producer or consumer can run at their own pace.

2) Setting up a channel: A space can call `alloc` (omitting prefix "`chan_`", see Table I) to create a channel with a globally unique ID, in which an inode is internally created by an `spmcR_alloc` call. A space can invoke `setprod` or `setcons` to set a child space producer or consumer via the `spmcR_transown` or `spmcR_copycons` calls. Once the produce-consume relationship of a channel is set up, the producer then can call `send` to put a message from its private buffer to the inode, while a consumer can call `recv` to receive the next message from the inode to its private buffer.

3) Handling a message of unknown size: Each message is stored in the inode in turn. By means of space extension, when there is insufficient room in the inode to save a large message (even exceeding the inode size), the message can be internally split into several sub-messages. Each sub-message is put and fixed on the inode in turn, mingled with an `spmcR_extend` call to remap the inode to new generation of epages. To let consumers know the length of each message, each message—which might be a sub-message—is stored on the inode with an internal header, *i.e.,* (a, b), where a and b indicate the message length and the whole message length, respectively.

4) Deterministically transferring a message: DetMP introduces two types of fix policies – *eager-fix* and *lazy-fix* – to enforce determinism at message granularity.

Eager-fix policy fixes a message via `spmcR_setfix` call immediately after it is written to the inode, thus it can be consumed just in time, but requires each message should be page-aligned. Eager-fix increases the demands for more epages, which will further lead to high cache misses.

In some cases, we can consider about storing several short messages on the same epage, and take lazy-fix policy, *i.e.,* fixing an SPMC page only when it is filled or it stores a message to be published in time. This lazy-fix policy can make full use of pages, but increase the access latency. Furthermore, a special `sendLast` function needs to be introduced into the channel API, and programmers have to invoke it to notify the runtime system that it is putting a message which should be fixed in time, accordingly restricting programming.

5) Channel management in DetMPI: DetMPI implicitly creates channels for communication among MPI processes. It classifies channels into several kinds: (a) $n(n\text{-}1)$ 1:1 channels for point-to-point communication; (b) n 1:(n-1) channels for multicast. DetMPI reuses these channels to support various MPI point-to-point and collective communication functions, which decreases the number of channels to be created, but increases the coupling of processes, causing performance slowdown due to increased waiting. To avoid deadlock, DetMPI currently adopts eager-fix based channels for simplicity.

D. Performance Analysis and Limitations

DLinux ensures determinism but has poor performance on NPB-MPI except `EP` compared with MPICH2. To find performance bottlenecks in user-level SPMC VMM system and its upper channel layer, we deeply evaluate DLinux using 8 NPB workloads with A input class, where `BT` and `SP` require the process count to be a square, e.g., 9 or 16. We find DLinux has at least three limitations on implementation policies.

1) Blocking-based SPMC model: As mentioned in Section III-B, any consumer attempts to read an unfixed SPMC page would be suspended until the producer later fixes the page. Experimental results show that the suspend/resume overhead significantly degrades the performance of NPB workloads. From the lowest part of each bar on Fig. 4, we observe that some workload (except `EP`) process can spend 14∼27% of the total runtime during suspension period when running on 8 or 9 processes, but rise to 16∼40% on 16 processes.

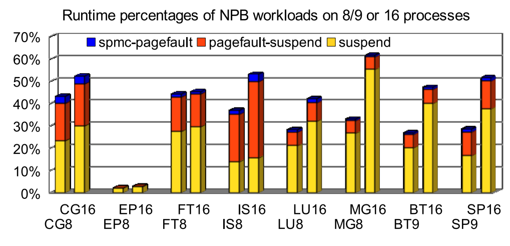


Fig. 4. Detailed runtime percentages of NPB workloads on DLinux.

2) Page-by-page mapping via SPMC page faults: Current SPMC VMM system only supports mapping an SPMC vpage to a lazily-allocated epage via triggering an SPMC page fault (*i.e.,* segmentation fault on the SPMC vpage) page by page. Thus the upper channel layer has to write/read a message of n pages accompanied with n SPMC page faults.

TABLE II
TRAFFIC COUNTS FOR NPB WORKLOADS

	D-pages	**S-pages**	**pagefaults**	**sends**	**D-pages/sends**
CG8	71830	623	205716	21910	3.3
EP8	111	512	280	111	1
FT8	230069	1863	690084	693	332.0
IS8	80328	684	240653	1788	44.9
LU8	479224	3831	1374702	316664	1.5
MG8	44616	407	131505	7928	5.6
BT9	570504	4573	1698843	32892	17.3
SP9	1002152	7956	2990006	65276	15.4

To understand the traffic in each NPB workload, columns 2∼5 of Table II list total numbers of epages for application data (D-page), epages for VMM (S-page, serving as page tables or extension pages), SPMC page faults and `chan_send` calls in all processes of each NPB workload. Column 6 lists the quotient of the D-page count divided by the send count, reflecting the average page-aligned message size in each workload. From the table, we understand the reason why `EP` has good performance is due to its less communication. For other 7 workloads, their average message sizes all exceed 1, up to 332.0 in `FT8`, and SPMC page fault counts are very high, at least twice of D-page counts—one for the producer and others for consumers. Fig. 4 further reflects that time of SPMC page fault handling in some process occupies large proportion of the time cost by all SPMC operations.

3) "Copy-twice" in an SPMC-page read/write: As shown in Fig. 3(a), an SPMC page write is emulated by a write to the producer's vpage and a write from the vpage to the corresponding epage; while an SPMC-page read also causes a write from the epage to the consumer's vpage and a write from the vpage to its private buffer. Such "copy-twice" in an SPMC-page read/write also brings more memory copy overhead.

IV. NEW DESIGN AND IMPLEMENTATION

A. Nonblocking Support

To minimize suspend/resume overhead, we introduce non-blocking test at either SPMC or upper channel layers:

1) spmcR_test(va): A consumer can invoke this explicitly to check whether a given vpage *va* is mapped to a fixed epage, accordingly avoiding getting suspended.

2) chan_test(cino): At channel layer, this function is added to allow a consumer checking whether the next message is available in the channel *cino* before reading. The `chan_test` handler first checks whether the caller is a consumer, and abort if not. If so, it then invokes `spmcR_test` and returns the result.

3) Nonblocking chan_recv: The previous blocking-based `chan_recv` handler itself needs to do the following things:

S1. Check whether the caller is a consumer, abort if not.

S2. Check whether the remainder address range in the consumer's inode can hold the internal header of the next message, and do space extension if not.

S3. Read the internal header from the inode, in which the consumer might be suspended if the header is not ready.

S4. According to message length a and the whole message length b in the header, read the message body from the inode and write it to the private receive buffer. This stage may involve multiple sub-message acquisition and space extension operations, depending on whether a is less than b and the remainder address range in the inode.

With the nonblocking primitive support, for each read from the inode in S3 and S4, we can add corresponding macro SPMCR_TEST(spmcva) immediately before the read (defined below). We introduce this macro definition in order to flexibly support either blocking or nonblocking by compiling DLinux with or without -DSPMC_NONBLOCKING option.

```
#ifndef SPMC_NONBLOCKING
#define SPMCR_TEST(spmcva)
#else
#define SPMCR_TEST(spmcva)      while (!spmcR_test(spmcva));
#endif
```

B. Direct Read/Write Support

There are two kinds of page faults on the SPMC memory, *i.e.,* segmentation faults on SPMC-VM and normal page faults on epages. We denote the former as SPMC page faults. To reduce SPMC page fault overhead mentioned in Section III-D, we add `spmcR_read` and `spmcR_write` primitives to allow directly copying data without triggering SPMC page faults.

1) spmcR_write(va, buf, size): The producer can call it to copy data of *size* bytes from a given private buffer to the SPMC region [*va*, *va*+*size*), where *va* and *size* should be page-aligned. Accordingly, SPMC VMM system can know how many epages are required to store a message, thus the primitive handler can directly map the producer's vpages to a set of newly-allocated epages, set the vpages writable, then copy data from the buffer to the vpages without triggering SPMC page faults.

2) spmcR_read(va, buf, size): Similarly, a consumer can call this to copy data from the SPMC region to a given private buffer. The primitive handler would directly map the consumer's vpages to the corresponding epages, might suspend the consumer if one of the epages is not present or unfixed, set the vpages readable if the mapped epages are fixed, and then copy data from the vpages to the buffer.

3) Avoiding "copy-twice" in an SPMC page read/write: In implementing the above two primitives, we can further optimize to avoid "copy-twice" shown on Fig. 3(a), *i.e.,* directly copying data between the private buffer and the epages shown on Fig. 3(b), rather than the vpages.

4) Refactoring `chan_send` and `chan_recv`: We can refactor `chan_send` and `chan_recv` to flexibly support three modes: nwr – unoptimized mode in previous DLinux, wr – direct read/write with "copy-twice", and wr2 – direct read/write without "copy-twice". We introduce macros MEMCOPY_TOSPMC and MEMCOPY_FROMSPMC to copy to/from the inode.

```
#ifdef SPMC_WR_DIRECT
#define MEMCOPY_TOSPMC(a, b, c) \
          spmcR_write((void*)(a), (void*)(b), c)
#define MEMCOPY_FROMSPMC(a,b,c) \
          spmcR_read((void*)(a), (void*)(b), c)
#else
```

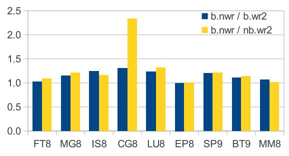
```
#define MEMCOPY_TOSPMC          memmove
#define MEMCOPY_FROMSPMC        memmove
#endif
```

V. Experimental Evaluation

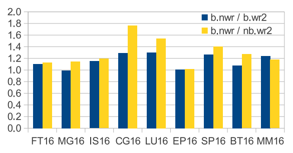
All evaluation experiments are conducted on a 32-core 4×Intel Xeon E7-4820 system with 128GB RAM, running Ubuntu 12.04. We use glibc v2.15 and GCC v4.6.3.

1) Methodology

We use three versions of DLinux to evaluate: b.nwr–blocking plus nwr mode in previous DLinux, b.wr2–blocking plus wr2 mode, and nb.wr2–nonblocking plus wr2. We compare them with `mpich2-1.4` using unmodified MPI programs, including 8 workloads with A input class in NPB-MPI 3.1 and an MPI matrix multiplication `MM`. Each workload is executed 10 times. To reduce the effect of outliers, the lowest and highest runtimes for each workload are discarded, so each result is the average of the remaining 8 runs.



(a)

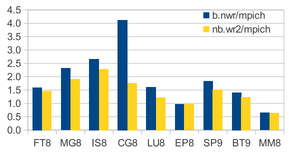


(b)

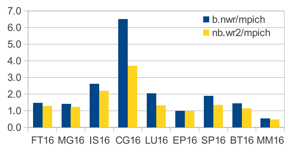
Fig. 5. Runtime overhead relative to b.nwr on various workloads.

2) Performance Relative to Previous DLinux

We first evaluate how direct read/write and nonblocking influence the performance. Fig. 5 shows the performance of various workloads on the b.wr2 and nb.wr2 relative to the previous b.nwr. By only using direct read/write without "copy-twice", the b.wr2 version improves the performance up to 1.31X on 8 processes and 1.3X on 16 processes. By further using nonblocking test before reading from a channel, the nb.wr2 version improves the performance up to 2.33X on 8 processes and 1.76X on 16 processes. The nb.wr2 version increases the performance of `MG`, `CG`, `IS`, `LU`, `SP` and `BT` by more than 13% on 8 or 16 processes. We observe that nonblocking test significantly improves the performance of `CG`.



(a)



(b)

Fig. 6. Runtime overhead relative to MPICH2 on various workloads.

3) Performance Relative to MPICH2

We next compare DLinux with MPICH2. Fig. 6 shows the performance of workloads on both previous DLinux and optimized nb.wr2 relative to MPICH2. The `MM` on various DLinux versions always have better performance than MPICH2. For NPB workloads except `EP`, the optimization proposed in this paper indeed improves the performance, e.g., the runtime of `CG8` (`CG` on 8 processes) on DLinux decreases from 4.12X to 1.77X of that on MPICH2.

VI. Conclusion

We have already showed that nonblocking support and direct read/write primitives improve the performance of NPB workloads on DLinux. Nonblocking test can be further used to improve various N:1 communication patterns, which we have not applied and evaluated in the current DLinux, so there is still more room to improve DLinux. In the future we intend to improve limitations in the upper DetMPI layer. We will explore how communication features and traffic influence the performance, and how lazy-fix channel can be used in DetMPI.

Acknowledgments. This research was supported by the National Key R&D Program of China (2016YFB1000403).

References

[1] Robert L. Bocchino et al. Parallel programming must be deterministic by default. In *HotPar*, March 2009.

[2] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *13th ASPLOS*, pages 329–339, March 2008.

[3] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *9th OSDI*, October 2010.

[4] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for C/C++. In *24th OOPSLA*, October 2009.

[5] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *14th ASPLOS*, March 2009.

[6] Heming Cui, Jiri Simsa, Yi-Hong Lin, et al. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *24th SOSP*, pages 388–405, New York, NY, USA, 2013. ACM.

[7] Joseph Devietti et al. DMP: Deterministic shared memory multiprocessing. In *14th ASPLOS*, March 2009.

[8] Tongping Liu, Charlie Curtsinger, and Emery Berger. Dthreads: efficient deterministic multithreading. In *23rd SOSP*, pages 327–336, October 2011.

[9] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Scaling deterministic multithreading. In *2nd WoDet*, March 2011.

[10] Yu Zhang and Bryan Ford. A virtual memory foundation for scalable deterministic parallelism. In *2nd APSys*, July 2011.

[11] Yu Zhang and Bryan Ford. Lazy Tree Mapping: Generalizing and scaling deterministic parallelism. In *4th APSys*, July 2013.

[12] Yu Zhang, Zhaopeng Li, and Huifang Cao. System-enforced deterministic streaming for efficient pipeline parallelism. *Journal of Computer Science and Technology*, 30(1):57–73, 2015.

[13] Yu Zhang and Huifang Cao. DMR: A deterministic MapReduce for multicore systems. *International Journal of Parallel Programming*, pages 1–14, October 2015.

[14] Rob F. Van der Wijingaart. NAS parallel benchmarks version 2.4. Technical Report NAS-02-007, NASA Ames Research Center, October 2002.

[15] Mathematics and Computer Science Division Argonne National Laboratory. MPICH2-1.4: a high-performance and widely portable implementation of the MPI standard (both mpi-1 and mpi-2), June 2011.

[16] Bryan Ford et al. Microkernels meet recursive virtual machines. In *2nd OSDI*, pages 137–151, 1996.

[17] Doug Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html, 2000. [Online; accessed 24-Sep-2012].