A Virtual Memory Foundation for Scalable Deterministic Parallelism

Yu Zhang
University of Science and Technology of China
yuzhang@ustc.edu.cn

Bryan Ford
Yale University
bryan.ford@yale.edu

ABSTRACT

Recent deterministic execution environments promise efficient program replay and bug reproduction, but their scalability is currently limited by strictly hierarchical synchronization models or serialized thread scheduling mechanisms. To address these issues, we introduce a *single-producer multiple-consumer* (SPMC) virtual memory foundation for deterministic parallelism, which supports non-hierarchical synchronization without serialized thread scheduling. An extension to the Determinator microkernel, supporting SPMC memory regions, offers threads and processes scalable "peer-to-peer" communication while preserving the kernel's existing guarantee of system-enforced determinism. DetMP, a deterministic user-level message passing API modeled on MPI, illustrates one way to build convenient application-level parallel programming abstractions atop the SPMC foundation. Preliminary results suggest that DetMP atop SPMC may be realistic and useful, achieving near-ideal speedup for parallel matrix multiplication, and good scaling for IS, in all cases ensuring strict determinism.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Multiprocessing/multiprogramming; D.3.2 [**Language Classifications**]: Concurrent, distributed, and parallel languages

General Terms

Design, Languages, Performance

Keywords

Deterministic Parallelism, Synchronization, Message Passing

1. INTRODUCTION

A parallel program is *deterministic* if, for a given input, every execution of the program yields identical behavior and output. Determinism offers benefits for replay debugging [20], fault tolerance [10], and security [2, 14]. Current methods of executing parallel programs deterministically show promise [7, 11, 13, 22], but often incur high costs, allow buggy or malicious applications to defeat repeatability [5], and often do not actually eliminate races from the programming model [3]. All of these approaches face scalability challenges: Grace [8] and Determinator [5] support only hierarchical synchronization such as *fork/join* and *barrier*, for example, while deterministic schedulers offering a full pthreads-compatible API [7, 11, 13, 22] introduce inherently-serial thread coordination mechanisms that make scaling difficult [23].

This paper extends the Determinator microkernel [5] with *single-producer multiple-consumer* (SPMC) virtual memory, a novel operating system foundation for non-hierarchical deterministic communication and synchronization. Any process can create an SPMC memory region, then transfer to other processes a single *producer mapping* of the region, and/or any number of *consumer mappings*, in order to set up virtual "pipes" among processes. A process attempting to touch a page in a consumer mapping blocks until the producer *commits* the corresponding page. Unlike classic Unix pipes, SPMC allows the producer of a region to commit its contents in any order at page granularity, and multiple consumers can read the same pages, efficiently supporting multicast communication patterns common in parallel programming practice.

By conforming to the constraints of the Kahn process model [19], SPMC preserves Determinator's ability to enforce deterministic execution of both multi-threaded and multi-process parallel computations. Because SPMC region transfer still occurs only via direct parent/child interactions between processes and threads, SPMC also preserves the conceptual simplicity, control, and composition power of Determinator's strictly hierarchical "nested process model" [17]. Once SPMC mappings are set up, however, processes can produce and consume pages in these regions incrementally, allowing for arbitrary peer-to-peer "dialog" short-circuiting the process hierarchy, thereby removing Determinator's prior scalability-limiting restriction of requiring all inter-process communication to involve a common ancestor. Finally, by building on the virtual memory capabilities of standard processors, SPMC offers performance characteristics well-matched to modern shared memory multicore machines.

We envision using the SPMC foundation eventually to support both shared memory (SM) and message passing (MP) parallel programming models, as well as high-performance deterministic I/O. As an initial case study, however, this paper introduces and focuses on DetMP, a deterministic message passing API modeled on the well-known MPI framework [21]. Leveraging Determinator's SPMC foundation and the shared memory multicore hardware on which it runs, DetMP offers applications some of the convenience of a shared memory programming model by allowing DetMP programs to share read-only data efficiently without explicit communication. Cooperating application processes use an MPI-like API, restricted to messaging operations with deterministic semantics, to communicate mutable intermediate results.

As a preliminary evaluation of SPMC and DetMP, we examine two well-known benchmarks, matrix multiplication (MM) and integer sorting (IS). Comparing MM atop DetMP against MM on De-

APSys'11, July 11-12, 2011, Shanghai, China.

terminator's original hierarchical shared memory model (DetSM), DetMP offers speedup close to ideal, scaling better than DetSM. Comparing IS from NPB-MPI [12] implemented atop DetMP against IS for MPI on Linux, we find that IS-DetMP offers 110% of the speedup of IS-LinuxMPI on 2 CPUs, and 97% and 78% of that of IS-LinuxMPI on 4 and 8 CPUs, respectively, offering good scalability while (unlike Linux) guaranteeing determinism.

This paper makes three main contributions. First, we introduce SPMC, a virtual memory foundation for non-hierarchical deterministic parallelism. Second, atop SPMC we develop DetMP, a deterministic asynchronous message passing API fully compatible with existing languages such as C, and similar to familiar message-passing frameworks such as MPI. Third, our preliminary performance results offer evidence that DetMP atop SPMC may be a realistic and useful approach to deterministic parallel programming.

The remainder of the paper is structured as follows. Section 2 summarizes relevant background work, then Section 3 presents the SPMC model and DetMP. Section 4 examines the prototype and reports preliminary results, and Section 5 concludes.

2. BACKGROUND AND RELATED WORK

As the most common forms of parallel programming models, SM and MP models provide different synchronization concepts and constructs, bringing different issues on determinism.

Deterministic Shared Memory Programming.

In the shared memory (SM) model, parallel tasks (usually threads) share an address space to which they read and write concurrently. Although *fork/join* and *barrier* are naturally deterministic synchronization patterns among threads, synchronization mechanisms such as locks and semaphores popularly used in controlling concurrent access are semantically nondeterministic and invite heisenbugs [1].

Deterministic schedulers [7,8,11,13,22] can make bugs reproduciable, but allow misbehaved software to defeat repeatability, and face scalability challenges [23]. DMP [13] and CoreDet [7] interleave threads' synchronization and memory access operations on an artificial schedule. Because this deterministic schedule is semantically arbitrary—not implied by anything in the program's logic—this approach makes race conditions reproducible but does not eliminate them: slight input changes may still reveal schedule-dependent bugs. TERN [11] attempts to address this problem of instability across inputs by reusing past schedules for similar future inputs. Kendo [22] provides deterministic guarantees for data-race-free programs by ordering lock acquisitions and releases. Grace [8] provides deterministic execution for C/C++ fork/join parallel programs using paged-based software transactional memory techniques.

New programming languages and type systems, such as Deterministic Parallel Java [9], offer determinism but require programmers to rewrite legacy code and perhaps adopt unfamiliar concepts.

Rather than designing new languages or deterministic schedulers, the WC memory model [4] allows concurrent threads logically sharing an address space but never seeing each others' writes, except when they synchronize explicitly and deterministically. The experimental Determinator OS [5] is one implementation of WC, but previously supports only strictly hierarchical synchronization.

Deterministic Message Passing.

In the message passing (MP) model, parallel tasks (usually processes) exchange data through by messages to one another, maintaining private memories. Communications among processes are non-hierarchical and may be *asynchronous* (in which the sender does not wait for the receiver to be ready) or *synchronous* (requiring the sender and receiver to wait for each other to transfer a message).

Although MP separates processes' data spaces, it does not automatically offer determinism: results can depend on order of message reception. A message passing API following the constraints of Kahn process networks [19] can offer deterministic execution, but popular MP frameworks such as MPI [21] do not satisfy these constraints. Furthermore, the requirement to marshal and unmarshal data into messages can be both less convenient to programmers and less efficient, even if optimized for shared memory hardware [24].

SHIM [15,16] offers deterministic MP in the Kahn network model, but requires adoption of a new programming language.

3. SPMC MODEL AND USAGE

This section describes the proposed SPMC model for deterministic parallelism, first covering the SPMC region primitive provided by the Determinator kernel, then the user-level DetMP message passing API that builds a familiar environment atop this primitive.

3.1 SPMC Region Kernel Primitive

Determinator initially constrained inter-process communication to direct parent/child relationships [5]. To alleviate this limitation, we extend Determinator's virtual memory system with *SPMC regions*, which allow a process P to establish communication directly among different children or descendents of P, eliminating P as a scalability bottleneck in subsequent computation. The kernel constrains SPMC regions to guarantee determinism despite in the presence of this "peer-to-peer" communication. SPMC regions thus form communication primitives analogous to Kahn process networks [19], expressed in a virtual memory API.

Determinator originally allows only read-only sharing of physical memory, through the kernel's copy-on-write (COW) mechanisms. SPMC generalizes the kernel with a restricted form of read/write sharing, by distinguishing two types of memory mappings. A given physical page can have only one *producer* mapping at a time, in one process's address space. The page may have any number of *consumer* mappings in multiple processes, however. The kernel enforces a protocol in which consumers have no access to a page while the producer is writing to it, but once the producer explicitly *fixes* an SPMC page using a system call described below, the producer loses write access while all consumers gain read access. Figure 1 illustrates a simple scenario in which two processes use two SPMC regions for bidirectional communication.

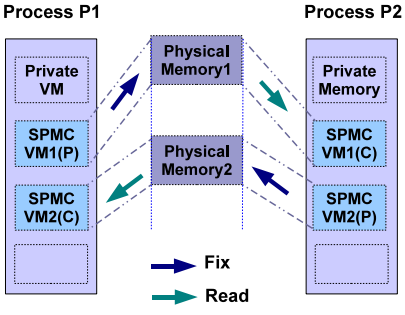

Figure 1: Processes can share several SPMC virtual memory regions, each of which has only one producer mapping, and any number of consumer mappings. Consumers can access an SPMC page only after the producer explicitly fixes it.

Table 1: Extended options to the system calls in Determinator.

Put	Get	Option	Description
✓	✓	Own	Put/Get the ownership of all or part of an SPMC region to/from child.
	✓	Fix	Fix all or part of an SPMC region.

To support SPMC regions, we add two optional arguments to Determinator's existing Put/Get/Ret system calls, as shown in Table 1. A user process P calls Get with the Zero and Own options to get a Zero-filled SPMC region, and becomes the owner (producer) of that region. P can transfer ownership of all or part of this SPMC region to any child, by invoking Put with the Own and Copy options; the parent then becomes a consumer of the transferred region. P can also hand out consumer mappings of the region to any number of child processes, by calling Put with the Copy option.

As in Determinator's current semantics, if at the time of a Get/Put call the specified child is still executing, the kernel blocks the parent until the child stops due to a Ret call, processor trap, or deterministic quantum expiration. This cooperative synchronization is necessary for determinism, but occurs only at SPMC region "setup" time. After region setup, the processes holding SPMC mappings can interact and synchronize in a more fine-grained, peer-to-peer fashion, unrestricted by the process hierarchy.

The kernel maps SPMC regions of the producer and all consumers to the same physical memory pages, as shown in Figure 1. To enforce determinism, the kernel never permits consumers to read a memory page while the producer is writing to it. Instead, the kernel gives *only* the producer access to a given page, until the producer explicitly *fixes* the page with a system call. The producer then loses the ability to write to the page, but all consumers gain the ability to read the page, and the page remains read-only for the rest of its lifetime. If a consumer attempts to read an SPMC memory page before the producer fixes it, the kernel blocks the consumer, to be awoken later when the producer fixes that page. Each process can unmap its SPMC regions when it does not need the mapped pages anymore, and the kernel frees each underlying physical page when all the mappings of that page are removed or replaced.

Once an SPMC region has been fixed, a process P can call Get with the Own option to "regain" the ownership of the region. The kernel then remaps the SPMC region of P to other newly-allocated physical pages, then copies the content from the original physical pages to the new ones, so as not to affect other consumers of the original SPMC pages. P can then create new consumer mappings of those pages by invoking Put with the Copy option, or transfer ownership to a child by specifying the Own and Copy options.

With SPMC regions, the kernel thus allows inter-process synchronization to "short-circuit" the process hierarchy while preserving a deterministic execution model. The copy-on-write mechanism optimizes large copies to avoid physically copying read-only pages. We intend this communication model to serve as a deterministic foundation to support multiple deterministic parallel programming models, as well as scalable deterministic I/O capabilities in the future. For the present, however, we focus on using SPMC for message-passing parallel applications, as described next.

3.2 DetMP User-Space Parallel API

DetMP is a user-space library built atop the kernel's SPMC region primitive, offering a high-level *channel* abstraction for message passing, and an MPI-like set of collective communication operations.

A channel is a pipe-like abstraction implemented and managed in user space by the DetMP library. Each channel has a globally-unique ID, `cino`, and has a unique producer and one or more consumers. The producer can asynchronously send several messages to a given channel `cino` without waiting for a response, as follows, where each message of `size`-byte length stored in a buffer `buf`:

```
chan_send(cino, buf, size).
```

A consumer can asynchronously receive whole or prefixes of messages in the order they were sent, storing the received data into `buf`, by calling `size = chan_recv(cino, buf)`. In the following simple program fragment, the current thread creates two child threads `pch, cch` and a channel `cino`, and let the two children communicate directly via the channel.

```
spmc_init(); // initialize SPMC meta-data
int cino = chan_alloc(); // allocate a channel cino
thrd_args a={.cino = cino,...};// user-defined arguments

pch = thread_alloc(pID);//allocate a child for producing
cch = thread_alloc(cID);//allocate a child for consuming

chan_setprod(cino, pch, 0);//set pch the producer of cino
chan_setcons(cino, cch); //set cch the consumer of cino

thread_start(pch, prod, &a));//start pch to run prod(&a)
thread_start(cch, cons, &a));//start cch to run cons(&a)
```

DetMP implements each channel as a fixed-size SPMC region holding a series of messages sent by the producer. Each message occupies a contiguous range of pages in the region, and the producer fixes one or more pages for each message sent. To support asynchronous communication, DetMP maintains channel meta-data recording the status of active channels, such as IDs of the producer/consumers, and offsets at which the producers and consumers put/get the next message in the SPMC region.

Collective Communication.

DetMP provides a set of collective communication functions similar to those in MPI, offering convenient high-level communication among a group of threads. Unlike MPI, collective communication functions in DetMP explicitly specify the relevant channels explicitly. A group of n threads, represented as a thread pool, consists of a master thread and its $(n-1)$ child threads.

DetMP implements various MPI-like collective communication operations atop the SPMC primitive. Assuming a group with three threads, G={T0,T1,T2}, Figure 2 shows how these functions map to basic channel send/receive operations on channels.

Barrier: At a barrier, each DetMP child thread makes a Ret system call to stop and wait for the master. The master makes $(n-1)$ Get system calls to synchronize with each child, after which the master restarts all children to proceed past the barrier.

Broadcast: The root thread T0 broadcasts to all threads of a group via a *root-to-all* $(1:(n-1))$ channel c, as shown in Figure 2(a), where T0 broadcasts message 1 to T1 and T2 via channel c.

Scatter: The root thread T0 scatters data to all threads in a group. For n threads, the function needs $(n-1)$ *root-to-thread*(1:1) channels. The root thread sends each item to its corresponding thread in turn (except the root thread itself) on the appropriate channel. In Figure 2(b), T0 sends message 2 to T1 via channel c1 and message 3 to T2 via c2, leaving message 1 for itself.

Gather: This function is the reverse of the *scatter*, which gathers data from all threads to the root thread. For n threads, the operation needs $(n-1)$ *thread-to-root*(1:1) channels; each thread except the root sends a message to the root via the appropriate channel. In Figure 2(c), T0 receives 2 from T1 and 3 from T2 in turn via channels c1 and c2, respectively, gathering all messages in its buffer.

Reduce: This function combines data items sent by each thread using a specified reduction operation, such as *sum, maximum, minimum, product*, into the root thread's reduction buffer. The function

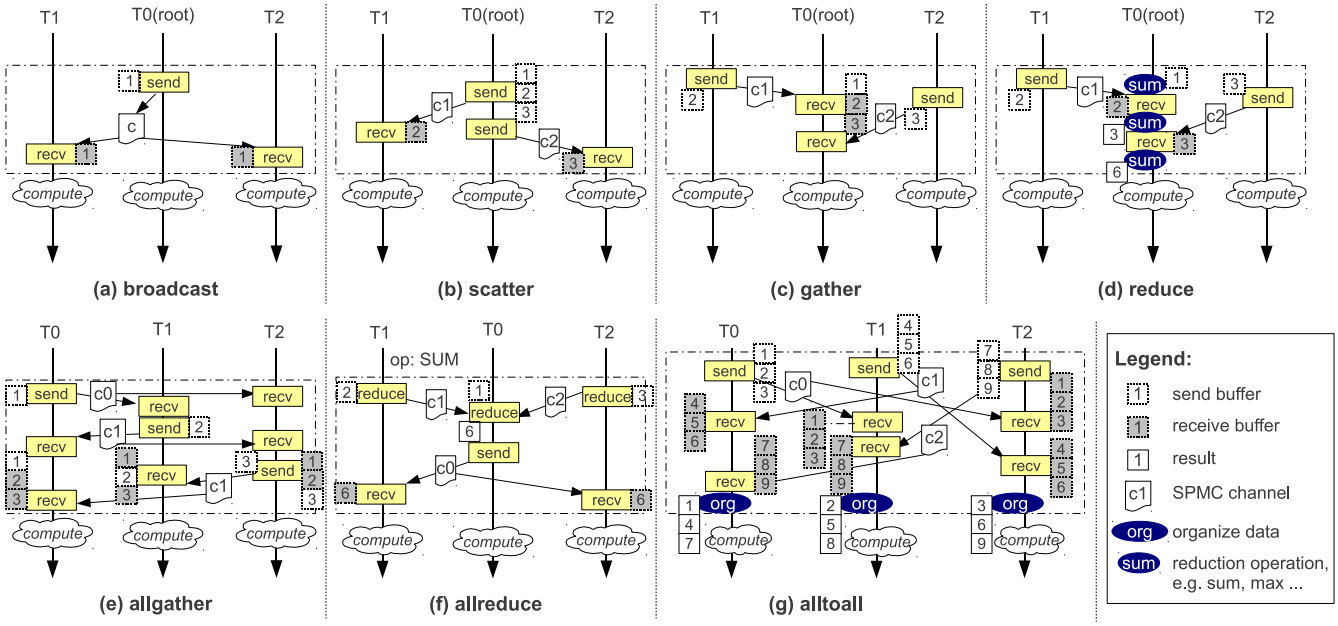

Figure 2: Mapping of collective communication operations to SPMC channel send/recv pairs

needs $(n-1)$ *thread-to-root*(1:1) channels for n threads. In Figure 2(d), T0 first initializes the buffer as 1, then receives 2 from T1 and executes *sum* with 1, then receives 3 from T2 and executes *sum*, finally getting the reduction result 6.

Allgather: This function is similar to the *gather*, but all threads receive copies of all messages, not just the root thread. For n threads, the function needs n *thread-to-all*(1:(n-1)) channels: each thread sends a message and receives messages from other $(n-1)$ threads in the specified order. In Figure 2(e), all threads get the gathering result containing a sequence of 1, 2 and 3.

Allreduce: This function combines the data sent by each thread using a reduction operation, and all threads receive the result. DetMP implements this by combining *reduce* and *broadcast*: in Figure 2(f), the three threads first call *reduce* to let the root T0 get the reduced result 6, then T0 broadcasts 6 to the others via channel `c0`.

All-to-all: This is an extension of *allgather*, where each thread sends distinct data to each receiver. The $j-$th block sent from thread i is placed in the $i-$th block of thread j's receive buffer. For simplicity, as shown in Figure 2(g), each thread first broadcasts all blocks in its sending buffer to all others, and receives all other threads' blocks, then puts the relevant blocks in its receive buffer.

4. PROTOTYPE IMPLEMENTATION

We have extended Determinator with SPMC regions at kernel level and DetMP at user level. Though early and incomplete, the prototype is sufficient to explore the feasibility of our design.

4.1 Design

Determinator is written in C with small assembly fragments, currently runs on the 32-bit x86 architecture, and supports up to 1GB of physical memory directly mapped into kernel space.

We modified the Determinator kernel's virtual memory system and system calls to support SPMC regions. The kernel uses x86 page-based address translation [18] to give each process an independent user-level address space, enforcing inter-process protection and isolation. A classic x86 two-level page directory/table hierarchy maps virtual to physical addresses in 4KB pages. Each page table entry (PTE) points to physical memory page. We use an available PTE bit to record whether a mapping is to an SPMC page. The kernel sets this bit when creating an SPMC region via the Get system call, and clears this bit at a producer's Fix request.

At user level, we reserve an address range for channel meta-data and channel internal nodes or *inodes*. For simplicity each active channel inode occupies a 4MB SPMC region. To support messages with different sizes, a message in an inode consists of its real length and its content. Messages in an inode are page-aligned so as to leverage the kernel's copy-on-write optimizations. To keep meta-data consistent across processes, we separate the creation and execution of a user process as in Java, and let a parent process copy or transfer the consuming/producing right of an SPMC region to a child between the creation and execution stages of the child, as in the program fragment in Section 3.2.

We provide each process a private heap for dynamic memory allocation (`malloc`) in applications.

4.2 Preliminary Results

We first compared DetMP with Determinator's original shared memory programming model (DetSM) using **matmult (MM)**, which multiplies 1024-by-1024 matrices. Both implementations divide work in the same way and use a master/slave coordination model. In MM-DetMP, the master uses channels to communicate with each child sending the task and receiving the result. We divide total execution time into *computation* (CP), *communication* (CM), and *verification*. In MM-DetSM, the master uses *copy* at the *fork* stage to transfer a task to a child, and *merge* at the *join* stage to collect the result from a child. We split total running time into *computation*, *merging* (MG), and *verification*. We omit verification time as it represents a negligible percentage of the total.

Figure 3(a) shows each version's speedup (SP) relative to single-CPU execution on the extended Determinator ran under QEMU [6] on a 48 core, 1.7GHz AMD Opteron PC. Table 2(a) lists the specific statistics, including time percentages of computation, communication and merging occupied by each thread on average, and the specific speedup and speedup ratio. MM-DetMP scales better than

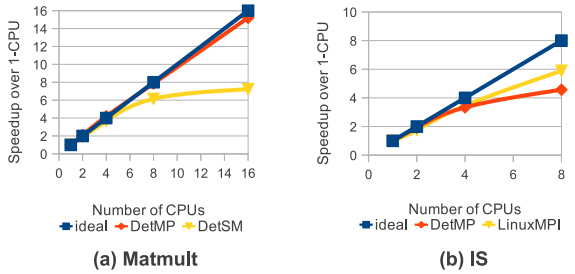

Figure 3: Parallel speedup over its own single-CPU performance on various benchmarks.

Table 2: Scalability and time distribution comparison.

(a) MM-DetMP vs. MM-DetSM.

CPUs	DetMP			DetSM			DetMP-SP/
	CP	CM	SP	CP	MG	SP	DetSM-SP
1	99.9%	0.0%	1	100.0%	0%	1	1
2	99.6%	0.3%	2.09	98.7%	1.3%	1.89	1.11
4	99.2%	0.5%	4.19	97.0%	2.9%	3.67	1.14
8	98.9%	0.8%	7.86	93.6%	6.4%	6.14	1.28
16	98.4%	1.3%	15.21	87.5%	12.5%	7.23	2.1

(b)IS-DetMP vs. IS-LinuxMPI.

CPUs	DetMP			LinuxMPI			DetMP-SP/
	CP	CM	SP	CP	CM	SP	LinuxMPI-SP
1	92.3%	7.5%	1	92.6%	7.3%	1	1
2	73.0%	26.8%	1.93	41.6%	57.6%	1.75	1.1
4	67.6%	32.0%	3.33	30.1%	69.5%	3.45	0.97
8	48.3%	51.6%	4.57	24.2%	75.3%	5.89	0.78

MM-DetSM, and achieves close to ideal scalability. The main reason is that merging in MM-DetSM occupies more time percentage than communication in MM-DetMP with increasing thread count, as shown in Table 2(a).

We also compared DetMP with Linux MPI by porting the Integer Sort (IS) benchmark from NPB3.3-MPI [12] to DetMP. IS is a bucket sort, where the number of keys ranked, number of processors used, and number of buckets employed are all powers of two. Communication costs are dominated by an *Alltoallv* operation.

Figure 3(b) shows the speedup of IS-DetMP running on the extended Determinator, and IS-LinuxMPI running on Linux, relative to the single-CPU performance of each. Table 2(b) lists specific measurements of time distribution, speedup and speedup ratio. IS-LinuxMPI ran on Ubuntu 10.10 with MPICH2, and both Ubuntu and Determinator were run in the same QEMU environment on the above 48-core PC. We show only results with up to 8 CPUs because when running under QEMU, Ubuntu uses at most 8 CPUs regardless of the number of CPUs specified to QEMU. Speedup of IS-DetMP is lower than that of IS-LinuxMPI on 4 and 8 CPUs (which are 97% and 78% of IS-LinuxMPI's, respectively), but it still scales well and, unlike Linux, guarantees determinism. We also see that the communication in IS-DetMP takes a smaller time percentage than in IS-LinuxMPI with an increasing number of thread.

5. CONCLUSION

We believe that SPMC offers a promising virtual memory foundation for implementing scalable, deterministic, and convenient parallel programming models, for today's and tomorrow's massively multicore processors. The kernel's SPMC primitive offers non-hierarchical inter-process communication and synchronization while ensuring system-enforced determinism. DetMP has shown good scalability on the benchmarks evaluated so far. In the future we expect to explore more efficient channels and other parallel programming models atop SPMC.

Acknowledgments.

This research was supported by China's Fundamental Research Funds for the Central Universities, and by the U.S. National Science Foundation under grant CNS-1017206.

6. REFERENCES

[1] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *VVEIS*, pages 82–93, April 2003.

[2] Amittai Aviram et al. Determinating timing channels in compute clouds. In *CCSW*, October 2010.

[3] Amittai Aviram and Bryan Ford. Deterministic OpenMP for race-free parallelism. In *3rd HotPar*, May 2011.

[4] Amittai Aviram, Bryan Ford, and Yu Zhang. Workspace Consistency: A programming model for shared memory parallelism. In *2nd WoDet*, March 2011.

[5] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Determinator: OS support for efficient deterministic parallelism. In *9th OSDI*, October 2010.

[6] Fabrice Bellard. QEMU, a fast and portable dynamic translator, April 2005.

[7] Tom Bergan et al. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *15th ASPLOS*, March 2010.

[8] Emery D. Berger et al. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, October 2009.

[9] Robert L. Bocchino et al. A type and effect system for deterministic parallel Java. In *OOPSLA*, October 2009.

[10] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *3rd OSDI*, pages 173–186, February 1999.

[11] Heming Cui, Jingyue Wu, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *9th OSDI*, October 2010.

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

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

[14] George W. Dunlap et al. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th OSDI*, December 2002.

[15] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *Transactions on VLSI Systems*, 14(8):854–867, August 2006.

[16] Stephen A. Edwards, Nalini Vasudevan, and Olivier Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *DATE*, March 2008.

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

[18] Intel Corporation. IA-32 Intel architecture software developer's manual, June 2005.

[19] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475, Amsterdam, Netherlands, 1974. North-Holland.

[20] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX*, pages 1–15, April 2005.

[21] Message Passing Interface Forum. MPI: A message-passing interface standard version 2.2, September 2009.

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

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

[24] Angela C. Sodan. Message-passing and shared-data programming models – wish vs. reality. In *19th HPCA*, May 2005.