TCASM: An asynchronous shared memory interface for high-performance application composition

Douglas Otstott[a,b,*], Latchesar Ionkov[b], Michael Lang[b], Ming Zhao[a]

[a] School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ, USA
[b] Ultrascale Systems Research Center, Los Alamos National Laboratory, Los Alamos, NM, USA

ARTICLE INFO

Article history:
Received 22 March 2016
Revised 1 December 2016
Accepted 23 January 2017
Available online 24 January 2017

Keywords:
Shared memory
Composite applications
Checkpointing
In situ analysis

ABSTRACT

This paper addresses the growing need for mechanisms supporting intra-node application composition in high-performance computing (HPC) systems. It provides a novel shared memory interface that allows composite applications, two or more coupled applications, to share internal data structures without blocking. This allows independent progress of the applications such that they can proceed in a parallel, overlapped fashion. Composite applications using in-node shared memory can reduce the amount of data to be communicated between nodes, allowing checkpointing and data reduction or analytics to be performed locally and in parallel. The approach is implemented in Linux, and evaluated using benchmarks that represent typical composite applications on a large HPC testbed. The results show that the proposed approach significantly outperforms the traditional ones (up to a 15-fold speed increase on a 200 node machine).

1. Introduction

As high performance computing (HPC) systems increase in scale, scientific workflows utilizing such systems are becoming increasingly complex. With extreme scale systems, it is no longer feasible to run large simulations and post-process the results. One of the key drivers for this change is the explosion of data resulting from large-scale simulations. It is becoming increasingly difficult to move the data from the supercomputer to the permanent storage and back again for additional processing steps [1]. This pressure is pushing simulation developers to couple these previously separate processing steps into composite applications [2].

Composite applications consist of two or more distinct applications coupled with scripting or glue-code. Composition techniques often enable greater functionality and absorption of additional processing steps. An example of a composite application is the Community Climate System Model (CCSM) [3], a large-scale weather/climate simulation consisting of multiple distinct applications. There are also simulations paired with uncertainty quantification (UQ) or visualization [4–6]. Multi-scale physics involves modeling systems at multiple distinct granularities. This is an area where coupled application models are highly applicable [7]. Such composite applications could consist of large-scale applications sharing each node or partitioned on distinct sets of nodes. Another use case would be a single large application, coupled with a smaller utility application that has less of a performance impact on node resources. A utility application could manage the movement of checkpoint data to stable storage [8,9], communication of data to another large-scale coupled application (like coupling the

* Corresponding author.
E-mail addresses: dotstott@asu.edu, dotst001@fiu.edu (D. Otstott), lionkov@lanl.gov (L. Ionkov), mlang@lanl.gov (M. Lang), mingzhao@asu.edu (M. Zhao).

XGC1 fusion code to the reduced-model code XGC-1A [2,10]), application frameworks [11–13], data filtering, data translation, region-of-interest (ROI) extraction [14], or any number of other functions [2].

The current hardware trends are pushing scientific workflows towards assembling composite applications. Large memory capacity in SMPs is leading to immense data sets that are difficult to store and retrieve from permanent storage. Additionally, the multitude of cores on current and future processors and the promise of heterogeneous cores [15–17] on a single socket will soon enable configurations with specialized cores [18,19] and the idea of spare-cores [20]. Spare-cores are cores on the node that cannot directly improve the performance of an application due to resource constraints, but can be assigned other tasks in support of the application. The large amount of memory, the cost of data movement, and the possibility of spare-cores and heterogeneous cores all make composite applications an attractive option for scientific workflows.

Composite applications are currently being developed in an ad-hoc manner with little system support. The driving need is to quickly couple existing applications with a small amount of glue-code to tie them together. This promotes replication of work, leads to fragile frameworks which do not scale, and can result in large amounts of data having to be moved off node. As most of the compute nodes in large HPC systems do not have local storage, the most commonly used interface for interaction between weakly coupled applications is the parallel file system, but this does not scale well. A better mechanism to facilitate composition is shared memory, usually used together with some other synchronization mechanisms (semaphores, named pipes, etc.) to ensure consistency between the applications. These techniques usually lead to tight coupling, and lock-step execution of the processes that form the composite application. The most common implementations, using one or more buffers shared by processes, require proper buffer-use accounting as well as knowledge in every application of what other applications are accessing the data. A promising solution to support the development of composite applications is a publish/subscribe mechanism. This has been used successfully before, but usually across networks rather than between applications sharing a single node [21,22]. An important advantage of this solution is the process isolation, such that errors in one application would not necessarily be propagated to the other process space.

As one solution to remedy the lack of system support for application composition, we have developed a new shared-memory-based asynchronous interface, TCASM— Transparent Consistent Asynchronous Shared Memory, for flexible and efficient application composition. TCASM allows disjoint applications to share regions of memory in a publish/subscribe architecture with minimal effort and no need for mutual exclusion. TCASM facilitates memory sharing in a way that guarantees consistency between the publisher and its subscribers by carefully restricting when and how changes to shared memory propagate. In this way, several applications can be coupled without the need for multiple shared memory buffers, mutual exclusion, explicit copying, or file I/O. TCASM was implemented by modifying the Linux shared memory system to support a new `msync` flag which would essentially make any changes to privately mapped memory publicly available, while preserving distinct versions of the original data in the subscriber's private memory.

The goal of TCASM is to facilitate application composition, which can be useful when implementing data reduction, filtering, region-of-interest extractions, and coupling of common analytics and visualization applications for existing software. These can then be reused with many applications. TCASM demonstrates several advantages over existing composition methods including failure isolation, parallel execution, simplified application interface, OS management of data sharing, locking with copy-on-write protection, small performance overhead and, in some cases, a reduction in memory usage and runtime.

In the paper, we present two illustrative use cases of TCASM, non-blocking check-pointing and in situ data analysis, to demonstrate the usage of the simple TCASM interface and the benefits of this approach. These two examples were chosen because they represent pivotal use cases for HPC. Application-specific checkpointing is the HPC community's key resilience mechanism and it is greatly impacted by the cost of data movement. Additionally, in situ techniques are increasingly important for performing data analysis operations that would have normally been handled by post-processing. However, post-processing is becoming prohibitively expensive due to ever-growing data-set sizes.

We provide evidence to validate this model as an OS approach to support the development of composite applications. We modified two real world scientific applications, SNAP [23] and miniFE [24], to perform the checkpointing and in situ analysis using various established application coupling techniques which we compare to our TCASM implementation as well as the baseline (unmodified application). TCASM performs well both at scale (200 nodes), where the cost of a large checkpoint is reduced by a factor of 10, and on single node experiments, where the runtime of checkpointing and analytics use cases in TCASM is reduced to 64% and 50%, respectively, compared to traditional solutions. Both use cases also perform close to the baseline when using TCASM to share memory. Furthermore, the amount of work required to implement a TCASM-based solution is small, requiring on average only four extra lines of code in the application source.

The key contributions of the work are as follows:

- Design of a novel asynchronous shared memory interface for building composite applications;
- Implementation of this design in the widely used Linux kernel;
- Libraries for commonly used C, FORTRAN, and C++ languages to allow HPC application developers to use TCASM in user-space;
- Two representative use cases to demonstrate feasibility of TCASM composition—non-blocking checkpointing and in situ data analysis.

The rest of this paper is organized as follows: Section 2 introduces the background and related work. Section 3 describes the design and implementation of TCASM. Section 4 presents the use cases. Section 5 discusses our experimental results

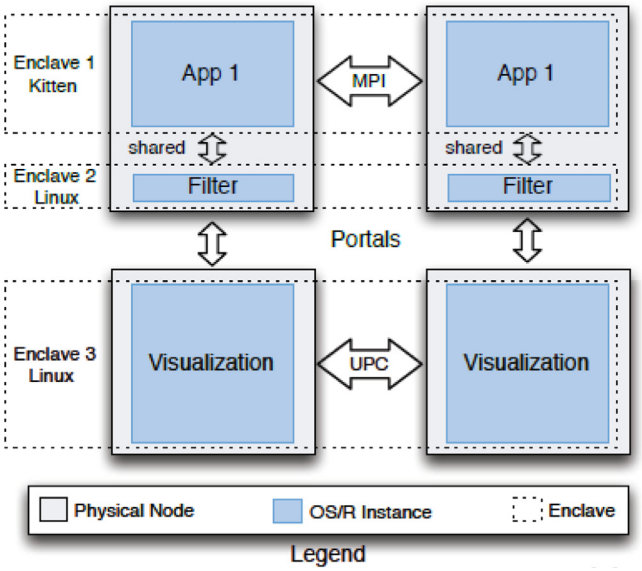


Fig. 1. Hobbes' proposed architecture [2].

examining the performance and trade-offs of using our solution. Finally, Section 6 concludes the paper and outlines our future work.

2. Background and related work

2.1. Need for application composition

Typically, ultrascale systems partition nodes into two groups: compute and data nodes. Compute nodes perform the computational work and each node usually has dozens of processors and vast memory. Data nodes lack computational power and are usually coalesced into a distributed or parallel file system. Compute nodes may lack local persistent storage altogether and rely entirely on the data nodes for persistent storage needs.

As system sizes increase from petascale (10^{15} operations per second) to exascale (10^{18} operations per second), the data associated with large simulations is exploding (32–64 petabytes [25]). Future systems will have tens of thousands to millions of nodes and billion-way parallelism across the machine [2].

These huge data sets are difficult to move to and from persistent storage. Space sharing of the compute nodes is a common strategy to reduce I/O and network traffic. By placing related applications on the same compute nodes, no data has to be moved off node if the applications are able to share data. Co-resident applications can be combined, or "coupled", in several ways as described in Section 2.2.

Applications may be coupled for many reasons. By sharing data directly, coupling relevant applications may reduce the amount of data going to persistent storage or over the interconnect. Coupled applications can provide powerful data inspection functionality such as debugging, data analytics, or visualization of intermediary results. Finally, coupled applications can facilitate decoupled I/O, allowing the computation to continue while a checkpoint is written.

Application composition is also an important concern of the new operating system and run time (OS/R) frameworks for exascale systems. For example, the Hobbes system, currently being developed at Sandia National Labs [2], requires kernel-level support for application composition. Fig. 1 depicts this architecture with a visualization use case [2]. The scientific application (App1) is coupled with the filter application, which feeds data to a visualization application on a separate set of nodes. Note that App1 and the filter are co-located on their partition of physical nodes; App1 and the filter are loosely coupled, allowing the application to run independently from the filter and, therefore, from the visualization application. The composed (meta) application consists of the three loosely coupled applications which, in the Hobbes example, have the option of running under different operating systems.

2.2. Related application composition techniques

We summarize the comparison between TCASM and the related application composition methods in Table 1:

1. "Single Application" refers to solutions that directly embed the coupled functionality into the same process at compilation time. These solutions are straightforward to implement, but require tight coupling of code, which is detrimental to both performance and reliability. It can significantly slow down the application if the added function is expensive (such

Table 1
Comparison of composition methods.

Sharing mechanism	Sync required	Isolation	Copy required	Attach anytime	Auto-versioning
Single application	Y	N	N	N	No
Shared file	Y	Y	Y	Y	Yes
MPI	Y	Y	Y	N	No
Shared memory	Y	Y	Y (for multi-buffer)	Y	No
TCASM	N	Y	N (except for modified data)	Y	Yes

as a large file write operation). There is also no isolation between main application and embedded functions, where the crash of one can directly compromise the other. Moreover, users have to manually implement advanced features such as versioning—the ability to preserve multiple versions of the shared data. Single application solutions have the advantage of not being limited by costly I/O or synchronization. However, as the embedded function is a blocking call, it explicitly prohibits parallel execution (since any modifications made by the application would violate the consistency of the data set in the embedded function view) and is thus considered slower than a shared memory approach.

2. "Shared File" refers to reading and writing from a shared file in order to communicate the relevant data from separate cooperating applications. File-based solutions require synchronization to prevent concurrent data accesses, provide process isolation, and do not require that all processes be launched at the same time. A separate copy of the data must be stored in the filesystem; however, no additional memory is necessary for this solution. With some additional work, auto-versioning can be implemented by creating additional files. Unlike other solutions, those leveraging the filesystem are bound by file I/Os and are thus considered slower than shared memory solutions. However, they do allow co-allocations to run in parallel. The main drawbacks are the poor scaling and performance of accessing the data stored on a shared file resource.
3. "MPI" refers to solutions where the different application functionality is implemented in a separate MPI rank, or set of ranks. While relatively easy to implement, MPI-based solutions require mutual exclusion to protect data operations and extra copies of data in memory. Additionally, all processes must be started at the same time and explicit copy operations may be necessary. They do not provide process isolation for faults, and do not support auto-versioning. MPI solutions allow co-applications to run in parallel and MPI inter-process operations are relatively fast. Thus, MPI implemented solutions have generally good performance.
4. Traditional "Shared Memory" based solutions, which use single or multiple shared buffers to couple applications, share all the pros and cons of an MPI-based solution, except that shared-memory co-applications can be attached/detached/reattached at anytime. The cooperating applications are required to manage the costly and complex synchronization themselves. When only a single buffer is used, slower co-applications will block the main application, whereas in multi-buffer solutions, multiple complete copies of data need to be saved in memory. Costly synchronization aside, shared memory is one of the fastest ways to facilitate application composition.
5. Finally, our proposed "TCASM" approach provides a new shared-memory interface which does not require application synchronization, provides process isolation, does not need to copy unmodified data, allows cooperating applications to be attached and disconnect at anytime, and supports auto-versioning. We will show in this paper that TCASM-based solutions will outperform traditional shared memory implementations of scientific co-applications as well as provide all the advantages (and none of the shortcomings) of all previously mentioned solutions.

There is related work on providing in-transit coupling of applications in which the simulation application writes intermediate results to the local storage of a set of nodes that run an in-transit application which analyzes the data on its way to permanent storage [4]. In this case, applications are running on separate sets of nodes and linked by temporary files as is done in many-task frameworks like Falkon [12]. The ADIOS infrastructure [5] allows applications to be coupled by using either files or memory regions, but in case of the latter, it uses RDMA to transfer data between applications. In comparison, TCASM supports in situ coupling of applications that run on the same set of nodes and can use the shared memory interface to realize high-performance data sharing. In situ and in transit coupling are thus complementary application composition techniques and are expected to co-exist in HPC systems.

2.3. Related work on shared memory interface

Previous OS-based memory sharing methods include System-V or POSIX shared memory [26] and tmpfs [27] — which allows shared files in a RAM disk. They do not provide a mechanism to ensure that the data is consistent and they require additional knowledge of all processes that share the data in order to synchronize accesses to it.

Knem [28] is specifically designed for inter-process/intra-node memory sharing for point-to-point communication and collective operations, as seen in MPICH2. The Boost [29] libraries have an inter-process section which includes message queuing, memory sharing, and file locking functionality. Xpmem [30] is an ongoing project that facilitates inter-process memory sharing by allowing processes to map the virtual address space of different processes running on the same node. However, all these require synchronization between processes or a multi-buffer framework which allow processes to work

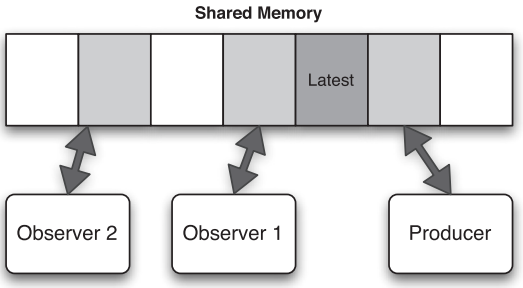


Fig. 2. The design of multi-buffer-based application composition.

on different copies of the data stored in separate buffers. Synchronization adds extra overhead to the participating processes, and can result in severe performance degradation if one process blocks the other for extended periods. Using extra memory buffers means the size of the data set is multiplied by the number of processes in question. In contrast, TCASM simplifies application programming by removing the need for complicated synchronization and makes efficient use of memory by not copying unmodified data. COSH [31] implements memory sharing abstractions in systems with heterogeneous cores and atypical memory hierarchies.

Finally, a preliminary study of TCASM [32] presented its basic functionality and initial results. Built upon that, this paper provides a more comprehensive and in-depth discussion of TCASM. It also includes new designs and implementations for auto-versioning and out-of-memory handling, new discussions on the non-blocking checkpointing and in situ analysis use cases, and new experimental evaluation results.

3. Design and implementation

3.1. Architectural design

To explain our design we will use a simple yet typical coupled application example, a producer application and an observer application. The producer performs calculations on a data set with a certain set of computational steps which are then repeated for the next iteration. At the end of each set of steps the data is in a consistent state. Observers need read-only access to the most recent consistent state of the data which should not change until they are done processing it. They may work at a different rate than the producer and can be launched at anytime after the producer has been started. This model covers many scenarios, such as observer applications that do data reduction or data restructuring before communication or observers that pass a snapshot of the data set to a third application.

The simplest way to support a producer/observer scenario is to have a single copy of the data in memory shared by the processes. This requires read-write locking and leads to lock-step progress between producers and observers and very slow progress for both.

A more scalable approach uses multiple buffers or a buffer queue as depicted in Fig. 2. The producer process iterates through the set of buffers; advancing to the next buffer each time it reaches a consistent state. Therefore, every buffer the producer has used and left is in some previous consistent state. The observers either iterate through the set, behind the producer (blocking if they catch up) or skip to the newest buffer that the producer has left, depending on the nature of the utility. Fig. 2 depicts this model with two observers and one producer. The producer is advancing to the right, and the buffer immediately to the left of the current buffer is marked “latest”. This allows the producer and observers to make independent progress at different rates, but it requires N times the shared data’s memory, where N is the number of buffers.

Our approach tries to preserve the asynchronous aspect of the multi-buffer approach and minimize the memory overhead by eliminating the need of copying unmodified data. The main idea is to avoid data duplication by sharing the pages that contain unmodified data between the multiple versions in memory. In the worst case, if all pages of the data set are modified every time and each observer is advancing at a unique rate, the memory consumption will be equal to that of $N+1$ multi-buffers with N observers. The best case is one in which no or very few pages are touched. On average, we expect a subset of the total pages to be modified on any given iteration, but not all. Specifically, TCASM provides a write-protected, shared memory region, mapped by both the producer and observer(s). Since the region is write-protected, any change made by the producer will trigger a copy-on-write (COW) where the modified memory pages are cached within the producer’s address space, hidden from the observers. In this way, changes to the region are not immediately reflected in the working set, but are held locally in the producer’s address space. When the producer is ready to share a consistent state with its observers, it has the ability to flush the previously copied and modified pages back to the original ones. The observer’s perspective of data is preserved by copying the over-written pages into the observer’s address space, thus maintaining consistency. The producer’s most recently published data is made visible to the observer only after it indicates that it is ready for the new data.

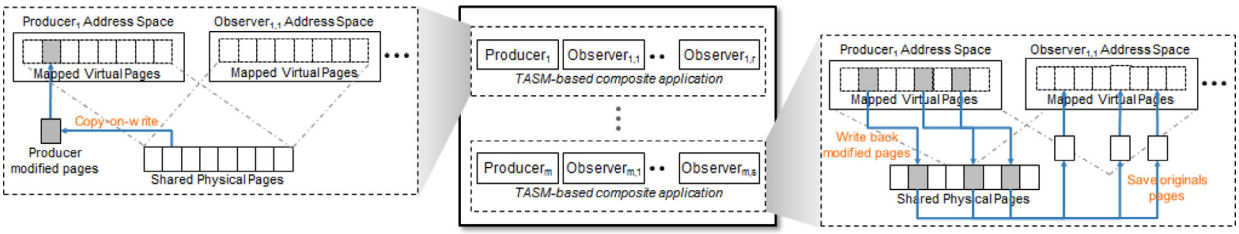


Fig. 3. TCASM architecture.

This process is illustrated in Fig. 3. Initially, the producer's and observer's virtual memory pages are mapped to the same set of physical pages. On the left side of the figure, the producer is modifying a page, so a new page is allocated in the producer's virtual address space for COW, before modifications are applied. On the right side, the producer has published its modified pages to the shared region, so the original pages must be preserved in the observer's address space. Before the shared pages are overwritten by the producer, they are copied out of the shared region and saved for the observer. The observer's corresponding virtual pages are remapped to these copied pages so it preserves the original view of data. This allows observers to advance at different rates than the producer and one another since they always keep their own versions of the data. Because of this behavior, we run the risk of causing out of memory errors. For this reason, we implemented a special case handler for out of memory situations. This special case recognizes the TCASM producer/observer paradigm, and utilizes a novel approach. Details on how the handler was implemented will be explained in Section 3.2.1.

TCASM leverages the "publish/subscribe" paradigm to minimize the coordination required among the producer and observers. The producer "publishes" each consistent state of the shared memory region without any knowledge of the observers. The observers, then, have the option to "subscribe" to the producer and receive each publication, much like a customer can subscribe to a magazine. The key is that the producer is completely unaware of the subscribers and works the same regardless of how many there are (including when there are none).

The benefit of the TCASM approach is that it provides a shared memory interface that avoids application synchronization by letting the operating system intervene when collisions on the data would occur. By leveraging COW, the operating system mediates data changes to present consistent data to the sharing processes and eliminates the need for complexity within the applications sharing the data.

The memory overhead of TCASM depends on how much shared data is modified by the producer between flushes and how many versions of the data are held by the observers. TCASM presents a trade-off between application performance and memory usage. Take the checkpointing use case as an example. The most memory-conserving checkpointing solution is to synchronously checkpoint the data modified by the application. However, this has the worst impact on performance as it requires the application to halt execution until the expensive checkpointing operation has completed. Compared to other asynchronous checkpointing solutions, e.g., the multi-buffer-based solution, TCASM is much more memory efficient because it saves only the data that is modified by the application. Moreover, as emerging non-volatile memory (NVM) technologies find their way into HPC systems, TCASM can leverage the high bandwidth and high capacity of NVMs to store the shared data with potentially negligible memory overhead, which will be studied in our future work.

3.2. Implementation details

TCASM's implementation is based on several existing mechanisms in Linux: private vs. shared mapped memory, `msync`, COW and anonymous and file-backed pages. User-space memory allocations in Linux can be described as *file-backed* or *anonymous*. The former is named so because these allocations allow tasks to directly map the file contents to memory with demand-paging. Executables, shared libraries, and other file-based mappings are of this type. Multiple tasks can map the same file and use it as a shared medium. Anonymous mappings, on the other hand, are not backed by any file and are the result of empty allocations. Linux kernel does not allow explicit sharing of anonymous mappings except for a special case created by the `fork` system call.

Memory allocated via `mmap` can be flagged as either *private* or *shared*. Changes made to *shared* memory are flushed to the mapped file and propagated to other processes mapping the same file, though a call to `msync` may be necessary first. If the mapping is *private*, then modifications to the content are only visible to the process that made them. Private mappings can consist of a mix of file-backed and anonymous pages. When the content of a privately mapped page is first modified by the process, the kernel performs COW of that page: a new anonymous page is allocated and its mapping replaces the original in the address space. Changes made to privately mapped memory are never propagated to the file, even during calls to `msync`.

The existing shared memory mechanism has limitations for our solution since Linux mappings either allow all processes to see all changes or no changes whereas TCASM requires that the observer sees changes only after the producer has reached a consistent state. Therefore, we require a system in which changes *are* propagated but only *after* a specific set of operations has been completed. To achieve consistent memory sharing we added a new flag, `MS_UPDATE`, to the `msync` system call,

which allows the producer to make private data available to any process as though it were public. When `msync` is called on a privately mapped region with the new `MS_UPDATE` flag, the kernel flushes any COW generated pages to the shared mapped memory file. In this way, inconsistent pages are kept solely in the producer's address space, unseen by the observers which can only access the last set of consistent memory published by `msync`.

Initially, both producer and consumer allocate a memory region using `mmap`. The producer process should create a shared memory file in the `/dev/shm/` file system and instantiate its memory region as private from the memory file. An observer wishing to access the producer's publications should open the same file (using `shm_open`), and instantiate its memory similarly, except as read-only. Note that an observer can be dynamically attached and detached from a producer at any time while a producer is running. Changes to the mapped files, however, should not be immediately reflected in an observer's view since it is assumed to be conducting operations that require a consistent state (e.g., checkpointing), and may be in the middle of an iteration. To solve this, `msync` was modified to account for pages that are still mapped by other processes. When the `MS_UPDATE` flag is used, `msync` will copy pages mapped by other processes into a new anonymous region and update any relevant mappings before allowing them to be overwritten. As such, versions published by the producer will not be reflected in the observer's working set until after the observer has mapped the shared memory file with `mmap` again. As mentioned earlier, the observer will see the last publications synchronized by the producer before the observer's call to `mmap`. This is acceptable, since most target applications for TCASM require the observer to see the most recent update.

With some additional overhead, finer-grained versioning can be created where the producer and observers can share multiple versions of the data. This ability enables important use cases such as offline analysis of application execution. By persisting each iteration of the critical data sections, the progress of the application can be scrutinized by a separate process, after the application has finished. This can be useful for a variety of reasons, depending on the nature of the application: debugging and understanding how or why a particular answer was reached or pin-pointing the exact moment when a certain condition was met. This approach can be very attractive to users, as it does not require modifications to the producer and does not require a new full run of the producer to gain new insight.

In order to support auto-versioning, when a producer publishes a consistent state, it must store it in a new `shm` file whose name is uniquely determined by the name of the data set and the version of the data. For example, in SNAP [23], one of our use cases, we used the iteration numbers of the two nested loops to indicate version number. Note that this does not mean that the first version will be numbered from zero. Rather, the first version will have the number of the iteration on which it was made. Furthermore, version numbers may not be sequential; if the producer is not syncing at every iteration, the version numbers may skip.

The observers, who must be aware of the `shm` file naming conventions, can iterate over the `shm` file names until they find the version that they want. However, because `shm` files are stored in memory, the number of versions is limited by the size of system memory as a multiple of the working set size. In an HPC environment, this may not be a very large number, since scientific applications tend to take up a large amount of the memory. To address this, a simpler solution for auto-versioning that is less memory intensive was proposed: let the observers do it. As mentioned before, as long as an observer holds a page, it cannot be overwritten or `munmapped`. A simple checkpointing observer can be modified to not overwrite a preexisting checkpoint file, by creating a new file for each iteration as described earlier. If new versions are coming too quickly for the observer to get them all, a multi-threaded solution can be implemented to hold pages for the observer until it is ready to write them. This was achieved quite easily using pthreads in C. The main function continuously checks shared memory for a new version, when it finds one, it spawns a new pthread to handle the copying and persisting of the data, and continues waiting for the next version. When the pthread has completed its checkpoint, it frees its version in memory and terminates.

This interaction over time is depicted in Fig. 4. We show mappings of memory pages in the producer's virtual memory, the observers' virtual memory, and the file content. The file content view always represents the most recent published data from the producer. The rows of the figure represent increasing time downward. Initially all views share the same page, Page A. Once the producer performs a write operation, the kernel copies Page A into Page B (a new anonymous page) and applies the producer's changes. At this point, Page B is only visible to the producer, while the file and the observer still see Page A. Once the producer calls `msync`, Page B becomes file-backed and Page A becomes anonymous, mapped only in the observer's virtual memory. The producer starts modifying the data again, mapping new anonymous Page C to its virtual memory. When the observer maps the file again, the anonymous Page A, with the old data, is discarded and replaced with the file-backed Page B. While this example shows only one observer, multiple concurrent observers would work in the same fashion.

With the TCASM approach, the producer task incurs a COW overhead for each page that it modifies. However, we argue that the overhead of COW will be insignificant compared to the saving that we achieve from eliminating the need of explicit synchronization for manually copying the data. In addition, expensive memory copy operations can be avoided if the producer does not modify all pages in the shared region, giving an advantage to the COW mechanism over unconditionally copying the entire shared range of pages in a multi-buffer solution.

The TCASM approach provides significant decoupling between producers and observers. Since, in the multi-buffer approach, observers and producers can access data concurrently, special care must be taken to prevent observers from accessing inconsistent data, and producers from modifying data observers are still using. Typically, some mutual exclusion has to be implemented to protect against these cases. Conversely, TCASM does not require the composite applications to implement explicit synchronization, and warrants far fewer changes to the original producer code in order to achieve the same

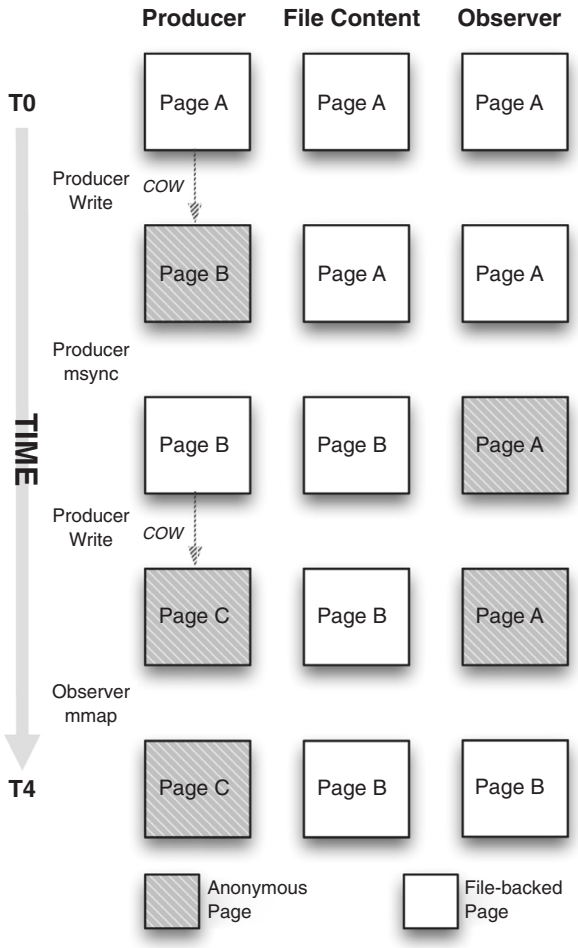


Fig. 4. An example of a producer and an observer sharing memory asynchronously using TCASM.

end. Instead, TCASM guarantees consistency by leveraging Linux's COW mechanism and anonymous filesystem (explained earlier in this section and depicted in Fig. 3) to cache changes to the observer's view of the shared memory in addition to the shared memory itself. In this way, the producer can ensure only consistent data is pushed to the shared file and the observer's view is protected until it is ready for the next iteration.

3.2.1. Out of memory handling

While the TCASM approach is memory efficient in most cases, out of memory situations may still happen in the worst case (when the observers are holding onto too many previously published versions of data and do not release the allocated memory). To address this, we investigated the existing Linux out-of-memory (OOM) killer. The OOM killer is a function which is invoked during page faults. If there are no free pages to allocate, the function selects processes to kill based on a value assigned to each process, `OOM_SCORE`. In most cases, the producer should be prioritized over the consumer. This can be easily achieved by assigning a higher `OOM_SCORE` to the observer processes, indicating that they should be killed first in a out-of-memory situation.

However, due to the nature of the producer/observer paradigm, it may not be preferable to kill the observer. A common scenario is one in which neither the observer nor the producer allocates any memory as part of its processing loop (note that `mmap`ping a previous shared region does not allocate pages). Therefore, after the producer and observer have passed their initialization phase, the only operation which would allocate a page in memory is the COW of protected pages. This means only the producer will ever trigger an out of memory error.

Furthermore, we expect the majority of observers to follow a specific pattern of execution: `mmap` a shared memory region, perform its function, `munmap` the shared memory, and repeat this pattern several times. Take for instance, the checkpointing use case: the observer will map the critical region, perform a streaming write to persistent storage, unmap the region, and wait for a new version to be published. In this scenario, it is not necessary to kill the observer. In fact, it is

far more preferable to let the observer finish and unmap its memory, as a half written checkpoint is of no benefit at all. In order to handle this case, the Linux OOM killer was modified.

The kernel modifications involved adding two new states to the special values used by the OOM killer: `TCASM_PRODUCER_APP(−19)` and `TCASM_OBSERVER_APP(−18)`. Then, a new wait queue was added to the memory management subsystem. A condition was added to the `out_of_memory` function: if the current process is a TCASM producer, put the producer in the new wait queue. Finally, the new wait queue is woken during the `__free_one_page` and `do_munmap` functions.

In this way, if a user knows a particular producer and observer follow this paradigm, they can notify the OOM killer by writing −18 and −19 into the observer and producer processes' `OOM_SCORE` fields, respectively. If the system arrives at an out of memory error, rather than kill either application, it will put the producer to sleep, until the observer `munmap`s the shared region, freeing a version of the data, and allowing the COW allocation to be completed successfully.

3.3. TCASM application interface

TCASM's functionality is exposed to user-space via the `mmap` and `msync` systems calls. Thus, using TCASM requires some simple application modifications. The data required by any observer needs to be placed in shared regions, allocated via `mmap`. For this reason, we developed a C-based FORTRAN library, and a custom memory allocator for C++ to facilitate shared-memory allocation using `mmap`. These contributions make TCASM available to a wide range of HPC applications, written in C, C++ and FORTRAN, the primary languages used by HPC developers. The FORTRAN library contains several FORTRAN wrapper functions for the C system calls. Since the interface is general, developers who wish to use this interface need only allocate c-type pointers (a data type in FORTRAN), and pass them to one of the functions in the C library. These pointers can then be used to allocate data types in FORTRAN. For C++, the custom memory allocator uses `mmap` to create objects. This way, any critical containers could simply be initialized using the shared memory allocator in place of the standard allocator.

The producer needs to be modified to publish the new versions at points where data is consistent, by calling `msync`. Simple application code would look as follows. Algorithm 1 depicts a typical producer. At initialization, it must allocate critical shared memory using `mmap`. Note that this requires a shared memory file, which the producer can create or may already exist. During each iteration, after doing some modifications to the shared region, it has reached a consistent state, it must call `msync` to publish the current version. Once work has completed, it frees the memory using `munmap`. Algorithm 2 depicts a sample observer. At the beginning of each iteration it must map the producer's latest publication by calling `mmap` on the memory file created by the producer. After performing its function, the observer must close (`munmap`) and reopen (`mmap`) the file to see the newest version published by the producer.

Algorithm 1: Producer application.

```
mmap(filename, data);
while not_finished do
    work();
    msync(data);
end
munmap(data);
```

Algorithm 2: Observer application.

```
while not_finished do
    mmap(filename, data);
    work();
    munmap(data);
end
```

The producers and observers need to agree on the naming convention for the shared memory files and the data structures inside of the files. The files are typically named by the MPI ranks of the producer processes. For example, the MPI rank 0 process of SNAP [23] calls `mmap` on the file named `snap_0.shm`. It is also possible for a producer to save its shared data into separate files, each storing a memory region with one or more variables. Then the names of the shared memory files will include the region IDs. Finally, as discussed earlier, each shared memory file may have multiple versions, so the names of the files may also include the version numbers.

The application using TCASM to share its data also needs to provide some metadata to hold application-specific descriptions of the structures to be shared. In our prototype, such metadata was communicated through a reserved portion at the head of its corresponding shared memory region. Information such as the number of regions and their respective sizes, the variables and their types, data versions, and important iterators from the application, may be required in order to allow

the observers to interpret the data. For example, a checkpoint co-application would need all of the information that is required to restart the application from a checkpoint such as the input deck and iterator values. For an analytics observer, the extents and offsets of the data structures of interest for the calculation would need to be included. Any constants can be included here as well. Use of a defined data description standard such as HDF5 [33] or netCDF [34] could easily be supported. In all of our experiments, metadata required less than one physical page of memory. It should be noted that this metadata is necessary in every aforementioned solution, with the exception of compilation-time coupled code, where the observer procedure would have access to the data values directly.

To present this new asynchronous shared memory interface to applications, TCASM requires changes to the operating system kernels used by HPC systems, but its implementation is based on the existing memory management interface which is relatively stable across different kernel versions. The specific target of TCASM is to support the Hobbes [2] project and be deployed on the new Trinity machine [35]. However, its general approach is also applicable to any kernel that implements copy-on-write and virtual memory mappings. For example, Hobbes' authors have already ported TCASM to Kitten [36], a lightweight kernel for HPC systems, via a set of extensions to the memory management layer. Moreover, systems such as BEE [37] leverage virtualization and Docker [38] containers to allow users to deploy specific kernels for their individual work without root permissions. Finally, if an application chooses to not use the TCASM APIs, it will not be affected by the kernel modifications.

4. Use cases

TCASM can enable a wide variety of composite HPC applications. In this section, we present two important use cases of TCASM, non-blocking checkpointing and in situ data analysis, to demonstrate its real-world applicability.

4.1. Non-blocking checkpointing

Traditionally, checkpointing is performed without help from the operating system. The simplest way is for the application to implement its own checkpoint by manually writing all data necessary for a restart. This approach is often slow as it requires synchronous I/O, but it does have the advantage of only writing the necessary data, as opposed to all data (e.g., BLCR [39]). TCASM offers the same benefits of application-level checkpointing by providing applications an interface to 1) specify which ranges of memory are critical for a checkpoint/restart, and 2) indicate when, in the runtime of the application, the critical data is a consistent state. Additionally, by leveraging the copy-on-write protection mechanism of the operation system, TCASM's publish mechanism only copies pages that have been written since the last publish. The time that it takes to checkpoint an application is dependent on the amount of modified pages that the producer publishes and the medium to which the observer saves the data. More importantly, TCASM addresses the drawback of traditional application-level checkpointing solutions by allowing applications to checkpoint the necessary state in a non-blocking manner with minimal impact on their performance. To use TCASM for implementing non-blocking checkpointing, the applications need to be modified to publish all data necessary for a restart at regular intervals, using `mmap` and `msync`, and coupled with an observer application responsible for persisting the published memory region. Note that, for most applications, a general-purpose checkpointing observer can be used. Unlike costly I/O write operations required for a full checkpoint, the producer only needs to call `msync`, which is a relatively quick memory operation. Similar to traditional application-level checkpointing, the parallel processes of an application typically synchronize among themselves (either explicitly using a barrier operation or implicitly via data exchanges). But the key difference is that, instead of implementing the checkpointing operation as part of its logic, the application uses msync to request TCASM to perform the checkpointing asynchronously and continues its execution without being blocked. Once the data has been synced back to the shared region, the observer can begin the slow synchronous file I/O required to persist this data, while the producer continues, unabated or aware. There can be two types of observers: one simply opens a file and writes the data to disk, and the other uses an I/O forwarding layer such as IOFSL [40] to forward the data over the network to an IOFSL server. In many HPC environments, compute nodes may not have local storage devices, and must hand data off to some data servers. In many cases, I/O forwarding libraries such as IOFSL are used to write data over the network to the data servers.

Fig. 5 depicts the difference between TCASM-based non-blocking checkpointing to the traditional approach. A traditional checkpointing application must stop its operation and write all relevant checkpoint information to persistent storage. It is important that the application pauses during the checkpoint, because any modification to the data set may result in an inconsistent state which cannot be restarted. In contrast, TCASM facilitates the coupling with a separate observer process which always sees the most recent consistent state from the checkpointing application and can write checkpoint data asynchronously without interrupting the execution of the application.

4.2. In situ analysis

Analysis functions are often written tangential to main applications in order to infer some extra meaning from the data produced over the course of execution. Traditionally, there are two ways to perform these additional data analysis. One way is via post-processing where analytics is performed after the main application has finished running. This usually involves

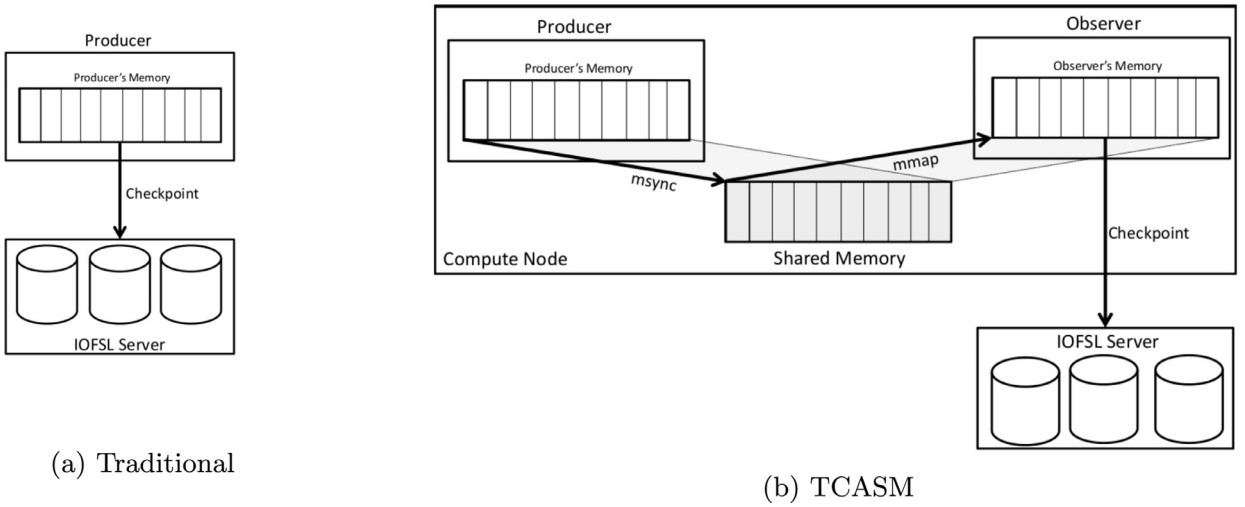


Fig. 5. Traditional synchronous checkpointing vs. TCASM-based non-blocking checkpointing.

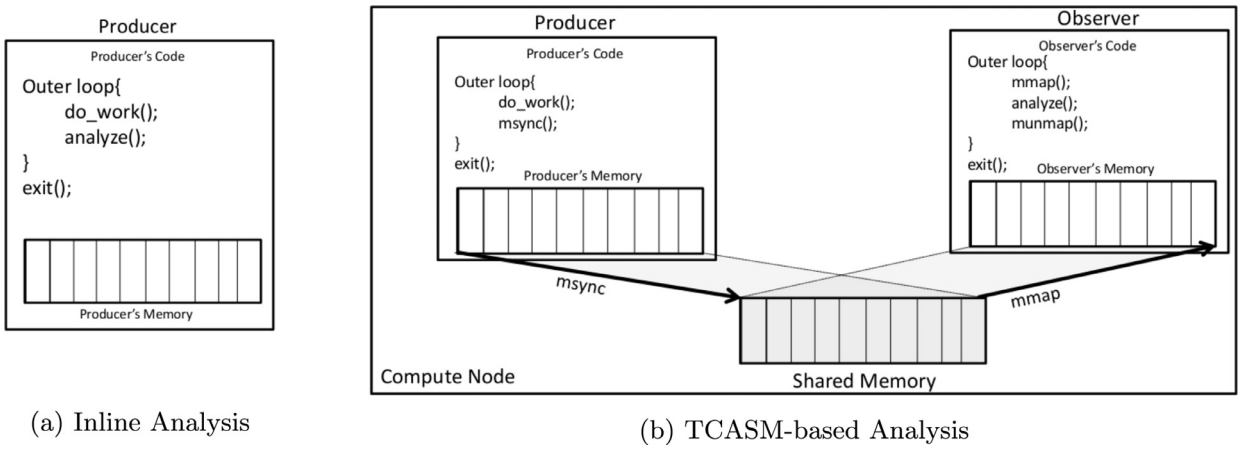


Fig. 6. In situ data analysis.

writing large data sets to persistent storage, as part of regular execution, and later, retrieving this data for analysis. As scientific application data sets have become rather large, this writing and rereading can be expensive. In these cases, in situ analysis has become the preferred method. In situ analysis is performed alongside the main application on the same set of resources while the main application is running and producing the data, usually via some form of application composition. The most common way is to directly embed the analytic functions into the main application's source code, where the invocation of the analytic operation will block the main application from proceeding. Other less common approaches could be implemented using any of the application composition methods listed in Table 1. As discussed in Section 2.2, TCASM-based analysis has the advantage of automatic versioning, parallel execution, process isolation, no synchronization requirement, and minimal memory overhead.

To use TCASM for in situ analysis, the producer application must share all relevant data for the analytic function by instantiating it from a shared memory region, similar to the checkpoint use case. Then, when the data has reached a consistent state, the producer must publish this region. While executing on the same node, an observer process must map the shared memory region to see the data published by the producer. It may then execute the analytic operation on the data.

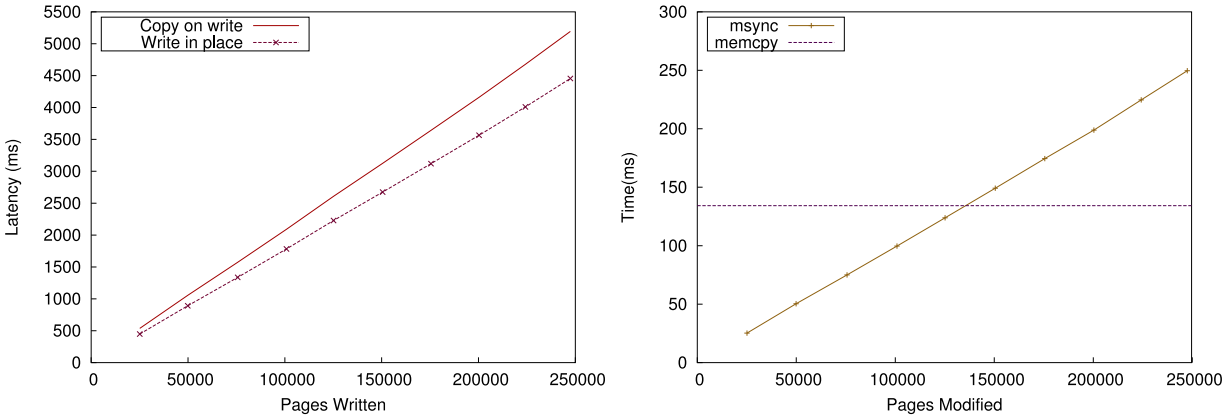
Fig. 6 depicts two versions of the data analysis implementation. In the inline analysis approach, the application calls an `analyze` function directly at the end of each iteration, interrupting its execution before returning. In the TCASM approach, the application is coupled with an observer that implements the analysis. The data producer calls `msync` to publish its most recent data at the end of each iteration and immediately continues its execution without having to coordinate with the analytic observer. In a separate process, the observer calls `mmap` to read the data published by the producer and, calls the same `analyze` function to perform the analysis, in parallel to the producer's execution, as opposed to in series.

Table 2
Experimental environments.

Name	Processor	Cores	Speed	Arch	DRAM	Disk
Jungla	AMD Opteron 6272	64	2.1 GHz	64 bit	126GB	1 TB HDD
PRObE	AMD Opteron 252	2/node	2.6 GHz	64 bit	4GB/node	2× 1TB HDDs/node
Haswell	Intel Xeon(R)	20	2.6 GHz	64 bit	126GB	200GB SSD
Magny Cours	AMD Opteron 6128	32	2.0 GHz	64 bit	64GB	200GB

Table 3
Overhead of critical operations.

Operation	Latency
`msync`	0.98 microseconds/page
`memcpy`	0.63 microseconds/page
COW	2.71 microseconds/page



(a) Latency of write operations with and without COW enabled by pages written

(b) TCASM's msync vs multi-buffer's memcpy with 1 Gigabyte of Memory by pages written

Fig. 7. Overhead of TCASM operations.

5. Evaluation

In our experimental evaluation, we first employ custom micro-benchmarks to quantify overheads and compare various sharing mechanisms. We then use real-world applications to investigate the performance benefits of TCASM. SNAP [23] is an MPI/FORTRAN90 application which simulates nuclear reaction experiments. MiniFE [24] implements the kernels representative of implicit finite-element applications, written in C++. Both are frequently used by the HPC community to benchmark computing systems. We modified these two applications to implement checkpointing and in situ analysis and evaluate the performance of TCASM.

Table 2 lays out the different machines used in the rest of the evaluations. From here on, the testing environments will be referred to as their names given in the first column of Table 2. Note that PRObE is a 600 node cluster; all other entries are single-node machines. Since TCASM is a kernel-level solution, all machines are running the TCASM kernel, a modification of Linux version 3.7.0.

5.1. Microbenchmarks

In this section, we report the observed performance degradation incurred by TCASM's COW and `msync` mechanisms using a microbenchmark. The benchmark involves modifying a set of memory pages iteratively. In one experiment, the pages are write protected (thus resulting in a COW) and in another they are not. Write protected pages are synchronized after each iteration, returning them to their original write-protected state. All data for Table 3 and Fig. 7 is collected on Jungla.

Time values are collected before and after the write and synchronization operations. Pages that are not write protected are copied (using `memcpy`) to another region in place of synchronizing. Time values are also collected before and after `memcpy` calls. All operations are repeated 100 times and mean values are presented in Table 3. Latency of calls to `memcpy` is included for comparison. Time values presented are aggregate times for contiguous data sets, represented as time per page.

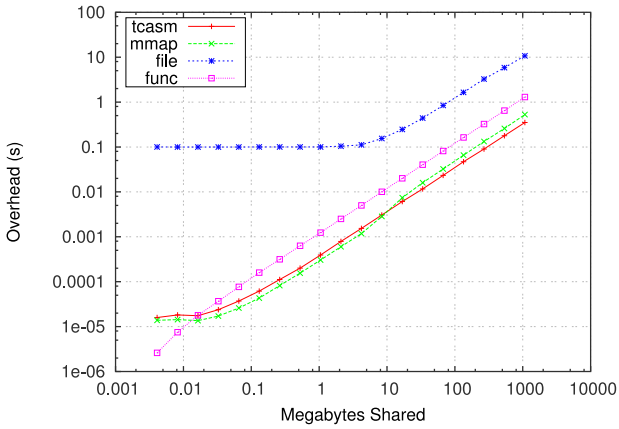


Fig. 8. Comparison of composition mechanisms relative to a producer not sharing any data.

The results show that COW accounts for the most significant overhead, almost three times that of `msync` calls. However, this experiment does not tell the whole story. Writes to protected pages only trigger a COW once per page. Once a page has been copied, the new page is not protected and the application may continue to modify it indefinitely without incurring overhead. Therefore, sparse or infrequent writes results in a much greater slowdown than dense or continuous writes when modifying privately mapped memory. This is because the initial cost of the copy is amortized over a much larger set of operations in the case of dense writes. Similarly, calls to `msync` only affect pages that have been previously modified such that the expected latency of an `msync` call would be proportional to the number of dirty pages.

Copy-on-write overhead is demonstrated in Fig. 7a. In this experiment, an application attempted to write a contiguous stream of integers linearly over two regions of memory, one protected (labeled "Copy on write") and the other not (labeled "Write in place"). The time values encompass only the actual write operations, as experienced by the application. Each time a new page is written, the kernel performs a copy and updates the virtual addresses, which has considerable overhead. However, the cost of COW is amortized by subsequent operations on the already copied page, which advance normally. As such, COW only accounts for a 16% increase in latency over the unprotected writes.

During calls to `msync`, non-trivial bookkeeping is performed by the kernel for each page which has been modified by the producer. In comparison, the multi-buffer approach copies all pages in one buffer and places them in another, regardless of whether they have been modified. Fig. 7b compares the overhead of `msync` as opposed to the traditional multi-buffer approach which uses `memcpy`. For this experiment, a fixed buffer of size 1GB was allocated and different percentage of the data, ranging for 10–100%, was modified. Once the specified number of pages had been modified, the region was either copied to another buffer or was flushed via `msync`. Since `msync` only copies pages that are modified by the application, largely unmodified data sets are actually faster to sync than copy. After the amount of modifications exceeds around 55% of the buffer, this advantage is negated. When the entire buffer is modified `msync`'s latency is about 1.85 times of `memcpy`'s. Many scientific applications touch somewhere between 50% and 70% of their pages each iteration [41], with the larger application touching closer to 50%. This observation means that typical applications should not experience much, if any, overhead from using msync over memcpy. In Sections 5.2 and 5.3, we explore real-world applications to test this hypothesis.

Fig. 8 shows data from a set of in-house microbenchmarks (collected on the Haswell machine) comparing TCASM to similar composition techniques: a semaphore protected queue of `mmap`ed regions (labeled "mmap"), a single application with the observer function implemented directly within the producer application (labeled "func"), and a shared file written to the local ext4 filesystem, protected with a semaphore, opened by both applications (labeled "file"). The producer iterates over a buffer writing double precision floating point elements. The producer then shares this data with a separate process which reads the written values and quickly sums them. The plots on the figure represent the total completion time for the producer and observer relative to a producer with no sharing mechanism enabled. Our results show that TCASM's COW synchronization performs increasingly better than the other approaches as the data-set size increases; typically a few fractions of a percent for `mmap`, and several seconds for file-based application composition.

5.2. Checkpointing

5.2.1. MiniFE

In this section, we evaluate the performance of TCASM-based non-blocking checkpointing as discussed in Section 4.1. First, we investigate the overhead incurred by TCASM when used for checkpointing. Fig. 9 shows the runtimes of three flavors of MiniFE (collected on Jungla): 1) the original implementation (labeled "Unmodified"); 2) an implementation utilizing the TCASM memory allocator and synchronizing shared memory at regular intervals but without actual observers (labeled "Syncing"); and 3) TCASM-based MiniFE with IOFSL observers checkpointing to a storage server at regular intervals (labeled

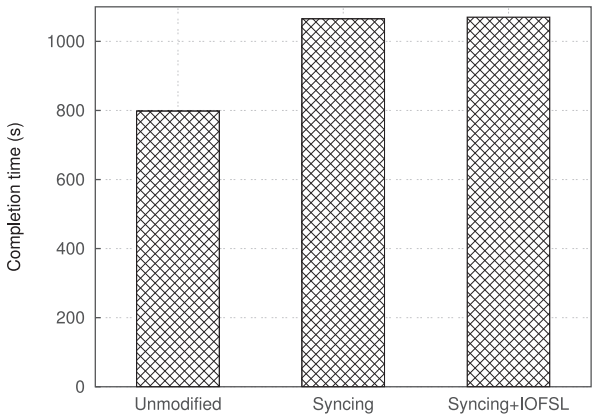


Fig. 9. MiniFE performance: unmodified compared to TCASM synchronizing data every iteration with and without observers copying the data off node using IOFSL.

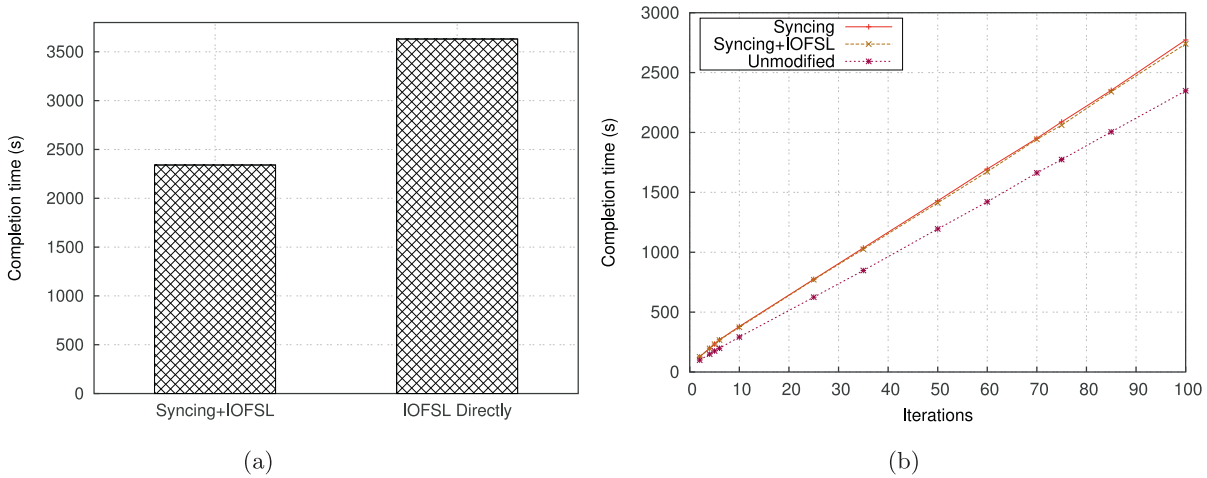


Fig. 10. (a) Runtime of SNAP's TCASM implementation with checkpointing observers compared to direct implementation of check pointing) on a single node with 64 ranks. (b) Runtime of unmodified SNAP as compared to TCASM-based SNAP with and without checkpointing observers.

"Syncing+IOFSL"). All three experiments use 32 MPI ranks to run MiniFE. In Case 3, 32 MPI ranks of observers run along with MiniFE and space-share the available 64 cores on Jungla. We study the impact of space-sharing vs. time-sharing of CPUs between the TCASM producer and observers in the next section. The storage server runs IOFSL server and the observers use IOFSL library calls to write data over the network. The third case represents a functional checkpointing use case, where data necessary to restart MiniFE is being persisted on the storage server via the observer processes. We included the second case to quantify the cost of using TCASM independently from any observer functions. The results show that TCASM allocation and regular synchronizing account for a 34% runtime overhead while the presence of observers adds nothing to execution time. Most iterations modified less than 10% of the pages allocated through TCASM.

5.2.2. SNAP

Next, we use SNAP to compare the performance of TCASM-based checkpointing using an IOFSL observer (labeled "Syncing+IOFSL") to the traditional approach which invokes IOFSL directly within SNAP to implement checkpointing (labeled "IOFSL Directly") in Fig. 10a.

The impact on performance is tremendous. By migrating IOFSL to the observer, we effectively make the I/O asynchronous, since the actual write operation has been moved to the observer process. This allows the producer to advance without waiting for the checkpoint to be persisted. TCASM's contribution is facilitating this departure from synchronous I/O with no need for explicitly copying the data, tightly-coupled synchronization, or heavily modifying the source code. Clearly, there are significant performance gains from leveraging shared memory for data sharing with the TCASM implementation finishing in about 64% of the total runtime of direct checkpointing implementation. This particular experiment was done on a relatively small data set (500MB per process across 32 MPI processes) on the Magny Cours machine.

Fig. 10b demonstrates the growth rate of SNAP's runtime for the same input by calling `msync` with varying number of iterations (ranging from 2 to 100). The figure features three curves: the original implementation of SNAP (labeled "Unmodi-

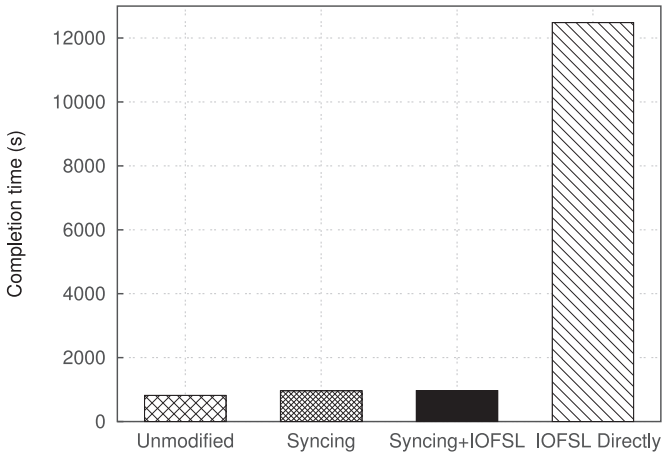


Fig. 11. Performance of SNAP, scaled on 200 compute nodes.

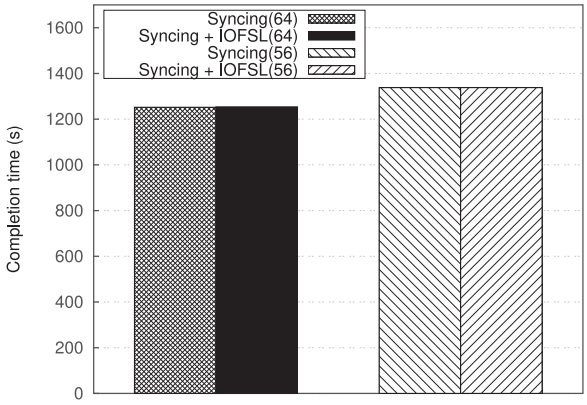


Fig. 12. Performance of TCASM-based SNAP with 64 ranks (time sharing) and 56 ranks (space sharing).

fied"), the TCASM version of SNAP syncing at regular intervals, but without observers (labeled "Syncing"), and TCASM-based SNAP with IOFSL observers checkpointing to the storage server at regular intervals (labeled "Syncing+IOFSL)"). In all three cases, SNAP was run with 32 MPI ranks and the third was run with 32 additional observers on the Magny Cours machine.

As illustrated, in all three cases, the runtime of SNAP grows linearly. The runtime differences between the TCASM implementations and the unmodified SNAP reveal the overhead from the COW and syncing mechanisms. At the full 100 iterations, TCASM syncing accounts for about 17% overhead, over-writing about 72% of SNAP's pages during each iteration. The cost of the observer running alongside SNAP is negligible (around 1%).

To demonstrate TCASM's effectiveness at scale, several experiments were conducted on 202 cluster nodes using the PRObE system [42]. Fig. 11 illustrates the relative performance of four implementations of SNAP. The first case is the original implementation of SNAP with 400 ranks running across 200 nodes (labeled "Unmodified"). The second case is the TCASM version of SNAP syncing at regular intervals, but without observers (labeled "Syncing"). The third case is TCASM-based SNAP with 400 observers checkpointing to the storage server at regular intervals using IOFSL (labeled "Syncing+IOFSL"). The fourth case (labeled "IOFSL Directly") represents an implementation of SNAP that invokes IOFSL itself to write to the storage server directly.

Fig. 11shows the difference among the first three cases is minuscule. Since `msync` is a local operation, only affecting the memory on the node its called on, its overhead is related to how many processes are calling it, and how much data is being synced. In this particular setup, the cost of two ranks calling `msync` on 1 Gigabyte simultaneously is negligible. However, comparing to the fourth case where SNAP has to checkpoint synchronously, the real benefit of TCASM-based non-blocking checkpointing becomes evident as shown by the 15-fold speed-up, because as the IOFSL clients compete for time on the server, SNAP may advance unimpeded.

Finally, we use SNAP to quantify the overhead of having observers and producers time-sharing vs. space-sharing the cores. We ran SNAP with four different setups on Jungla, and Fig. 12 depicts the results. The first bar is the TCASM implementation of SNAP with no observers on 64 ranks, each rank pinned to a core. The second bar shows the same setup, except that there are 64 observers present. The observers are not pinned to any core; they are free to float anywhere. The

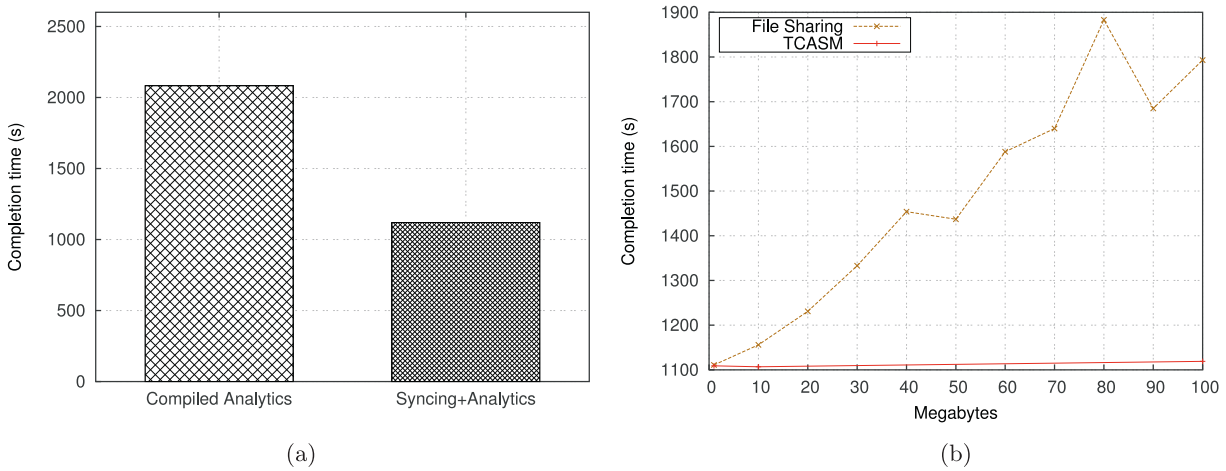


Fig. 13. (a) Runtime of SNAP with an embedded spectrum analysis procedure vs. a TCASM implementation sharing necessary data to an analytic observer. (b) Runtime of SNAP using the file system to share data as compared to TCASM.

third and fourth bars show TCASM SNAP running 56 ranks, with no observers vs. 56 observers, respectively. The 56 ranks of SNAP were packed seven per NUMA domain (of which Jungla has eight). The remaining eighth core, was either left empty or held the seven observer processes which communicated with producer ranks on the same NUMA domain. The inputs of both experiments are identical, and the size of the checkpointed region is 49 Gigabytes. We found that the overhead of both strategies is negligible—adding observers increased the runtime by about 0.16% (two seconds) when time-sharing with 64 ranks, whereas space-sharing with 56 ranks resulted in no overhead. When comparing the absolute performance between the two strategies, time-sharing CPUs with the observers allows SNAP to use all the CPU cores and finish 6.6% faster than space-sharing. Therefore, time-sharing is preferable to space-sharing.

5.3. In situ analysis

Next, we evaluate TCASM in the context of the in situ analysis use case discussed in Section 4.2 using SNAP. The data analysis calculates the energy spectrum of the entire experiment at any given instant in execution and prints it to standard output. This is accomplished by extracting key variables and arrays from the shared memory region.

Fig. 13a shows the comparison between the two cases, an implementation that embeds the analysis function directly as part of the SNAP code (labeled “Compiled Analytics”), and the TCASM-based implementation which uses coupled observers to perform data analysis (labeled “Syncing+Analytics”), on Jungla with 64 MPI ranks of SNAP and 64 observer processes. Using TCASM for spectrum analysis cuts the execution time of SNAP down to 53% by moving the actual computation involved in the analysis to the observer processes and performing it asynchronously. 100% of all shared pages were modified by each iteration. However, the data required to implement this solution was fairly small (on the order of a few dozen pages per rank for the largest experiment).

Fig. 13b compares TCASM-based in situ analysis (labeled “TCASM”) to another implementation using a shared file where SNAP writes intermediary data to a file stored on the file system and the observer reads data from the file to perform analysis (labeled “File Sharing”). The size of the shared region was increased to show growth. The x-axis represents the size of the actual shared region necessary for the spectrum computation. Each experiment was conducted with 64 MPI ranks on Jungla.

The results show that TCASM scales very well, with little increase in runtime as the size of the experiment increases, whereas the file sharing model continues to grow with the file size. The speed of file write operations is restricted by the speed of the underlying IO device, while `msync` calls are limited only by the speed of memory. Hence, any application composition technique that depends on persistent storage is going to be significantly slower than TCASM.

5.4. Development overhead

Finally, we estimate the amount of development effort required to implement TCASM in operating systems and to use TCASM to compose applications by counting the lines of code that we added to Linux and the benchmarks, respectively. In total, TCASM added about 500 lines of code to the Linux 3.7 kernel. The majority of this was added to the `mm/msync.c` file to facilitate the new MS_UPDATE flag with a few lines added to `mm/rmap.c` and `mm/swap.c`. The FORTRAN syscall library, which enables TCASM in SNAP, is 199 lines. To use TCASM, only six lines were added to SNAP and about 50 were modified, the majority of which are removed allocation statements and adjusted variable declarations; and only four lines

of code were modified in MiniFE. The TCASM C++ allocator used by MiniFE is 180 lines. These results demonstrate the ease of development of TCASM for other operating systems, languages, and applications.

6. Conclusions and future work

In this paper we presented TCASM, an asynchronous shared memory interface for enabling high-performance composite applications. With the rapidly growing size of HPC applications and their I/O demands, traditional distributed/parallel file system based storage architecture becomes increasingly insufficient for many cases, e.g., for handling the intensive checkpointing data from a simulation application or to support the high-speed data exchange among a simulation application and an analytic/visualization application. Hence, there is an urgent need for solutions that allow such data to be handled asynchronously without slowing down the simulations. Such a solution should also be convenient for scientists to use without requiring much effort to revamp their existing code. TCASM provides this much needed solution by allowing composite applications to asynchronously and efficiently share data using a shared memory interface.

The power of TCASM comes from two important design decisions. First, by leveraging existing virtual memory features, particularly COW, and extending them to meet the needs of composite applications, TCASM enables applications to share data with efficient memory usage. Second, by following the publish/subscribe model, it further allows the data-sharing applications to be largely decoupled and progress at different rates without requiring costly synchronization. The combination of these two designs allows TCASM to provide an efficient and asynchronous data-sharing interface and support a variety of composition applications including non-blocking checkpointing and in situ data visualization. Moreover, TCASM is provided as simple memory allocators for the commonly used languages (FORTRAN, or C++) so scientists can conveniently use it to create composite applications.

We prototyped TCASM in the widely used Linux kernel and evaluated it using benchmarks representing two important use cases on large-scale testbeds. Our results show that the overhead of TCASM is small for typical applications and in the worst case, it performs the same as the commonly used multi-buffer approach. More importantly, TCASM-enabled asynchronous checkpointing and in situ analysis substantially outperform the traditional synchronous solutions, by 11.96 and 1.86 times, respectively.

Based on these results, we will broaden the impact of TCASM along several directions in our future work. First, we would like to see the new functionality introduced by TCASM to be added to the main-line Linux kernel, and we will help port this feature to other HPC operating system kernels of interest such as Kitten [43]. Second, we will apply TCASM to support large-scale applications such as the Community Climate System Model (CCSM) [3] and other coupled physics applications. Finally, we will provide TCASM as a building block for new HPC operating systems and runtimes, such as Hobbes [2] and Argo [44], designed for many-core processors and extreme-scale environments.

References

[1] D. Rogers, K. Moreland, R. Oldfield, N. Fabian, Data Co-Processing for Extreme Scale Analysis, Technical Report, SAND2013-1122, Sandia National Laboratories, 2013. Level II ASC Milestone 4745.

[2] R. Brightwell, R. Oldfield, A.B. Maccabe, D.E. Bernholdt, Hobbes: Composition and virtualization as the foundations of an extreme-scale os/r, in: Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers, ROSS '13, ACM, New York, NY, USA, 2013, pp. 2:1–2:8, doi:10.1145/2491661.2481427.

[3] P.R. Gent, G. Danabasoglu, L.J. Donner, M.M. Holland, E.C. Hunke, S.R. Jayne, D.M. Lawrence, R.B. Neale, P.J. Rasch, M. Vertenstein, et al., The community climate system model version 4, J. Clim. 24 (19) (2011) 4973–4991.

[4] K. Moreland, R. Oldfield, P. Marion, S. Jourdain, N. Podhorszki, V. Vishwanath, N. Fabian, C. Docan, M. Parashar, M. Hereld, et al., Examples of in transit visualization, in: Proceedings of the 2nd International Workshop on Petascale Data Analytics: Challenges and Opportunities, ACM, 2011, pp. 1–6.

[5] N. Podhorszki, S. Klasky, Q. Liu, C. Docan, M. Parashar, H. Abbasi, J. Lofstead, K. Schwan, M. Wolf, F. Zheng, et al., Plasma fusion code coupling using scalable i/o services and scientific workflows, in: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science, ACM, 2009, p. 8.

[6] A. Sayed, H. El-Shishiny, Computational experience with nano-material science quantum monte carlo modeling on BlueGene/L, in: MEMS, NANO, and Smart Systems (ICMENS), 2009 Fifth International Conference on, IEEE, 2009, pp. 213–217.

[7] N. Belcourt, R.A. Bartlett, R.P. Pawlowski, R.C. Schmidt, R.W. Hooper, A Theory Manual for Multi-Physics Code Coupling in LIME., Technical Report, Sandia National Laboratories, 2011.

[8] K. Li, J.F. Naughton, J.S. Plank, Low-latency, concurrent checkpointing for parallel programs, IEEE Trans. Parallel Distrib. Syst. 5 (8) (1994) 874–879, doi:10.1109/71.298215.

[9] K. Ferreira, Keeping checkpoint/restart viable for exascale systems (2012).

[10] Center for edge physics simulation, (http://epsi.pppl.gov/).

[11] W.R. Elwasif, D.E. Bernholdt, A.G. Shet, S.S. Foley, R. Bramley, D.B. Batchelor, L. Berry, et al., The design and implementation of the swim integrated plasma simulator, in: Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on, IEEE, 2010, pp. 419–427.

[12] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, M. Wilde, Falkon: a fast and light-weight task execution framework, in: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, 43, ACM Press, Reno, NV, 2007. 10.1145/1362622.1362680.

[13] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, M. Wilde, Swift: Fast, reliable, loosely coupled parallel computation, in: Services, 2007 IEEE Congress on, IEEE, 2007, pp. 199–206.

[14] J.W. Berry, B.A. Hendrickson, Graph Analysis with High Performance Computing., Technical Report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2007.

[15] J.D. Brown, S. Woodward, B.M. Bass, C.L. Johnson, Ibm power edge of network processor: a wire-speed system on a chip, Micro, IEEE 31 (2) (2011) 76–85.

[16] B. Burgess, B. Cohen, M. Denman, J. Dundas, D. Kaplan, J. Rupley, Bobcat: Amd's low-power x86 processor, Micro, IEEE 31 (2) (2011) 16–25, doi:10.1109/MM.2011.2.

[17] S. Keckler, W. Dally, B. Khailany, M. Garland, D. Glasco, Gpus and the future of parallel computing, Micro, IEEE 31 (5) (2011) 7–17, doi:10.1109/MM.2011.89.

[18] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, A. Singhania, The multikernel: a new os architecture for scalable multicore systems, in: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09, ACM, New York, NY, USA, 2009, pp. 29–44, doi:10.1145/1629575.1629579.
[19] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, J. Kubiatowicz, Tessellation: space-time partitioning in a manycore client os, in: Proceedings of the First USENIX Conference on Hot topics in Parallelism, HotPar'09, USENIX Association, Berkeley, CA, USA, 2009. 10–10.
[20] J.C. Sancho, D.J. Kerbyson, M. Lang, Characterizing the impact of using spare-cores on application performance, in: Euro-Par 2010-Parallel Processing, Springer, 2010, pp. 74–85.
[21] R. Gupta, P. Beckman, B.-H. Park, E. Lusk, P. Hargrove, A. Geist, D. Panda, A. Lumsdaine, J. Dongarra, Cifts: A coordinated infrastructure for fault-tolerant systems, in: Parallel Processing, 2009. ICPP '09. International Conference on, 2009, pp. 237–245, doi:10.1109/ICPP.2009.20.
[22] P.T. Eugster, P.A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Comput. Surv. 35 (2) (2003) 114–131, doi:10.1145/857076.857078.
[23] J. Zerr, R. Baker, Snap: Sn (discrete ordinates) application proxy - proxy description, 2013.
[24] M.A. Heroux, Mini finite element.
[25] G. Grider, Exascale fsio/storage/viz/data analysis can we get there? can we afford to?, 2011.
[26] J. Moran, Sunos virtual memory implementation, in: Proceedings of the Spring 1988 European UNIX Users Group Conference, 1988.
[27] P. Snyder, tmpfs: A virtual memory file system, in: Proceedings of the Autumn 1990 EUUG Conference, 1990, pp. 241–248.
[28] B. Goglin, S. Moreaud, Knem: a generic and scalable kernel-assisted intra-node {MPI} communication framework, J. Parallel Distrib. Comput. 73 (2) (2013) 176–188. http://dx.doi.org/10.1016/j.jpdc.2012.09.016.
[29] Boost C++ libaries, 2007,
[30] xpmem: Cross-process memory mapping, 2014.
[31] A. Baumann, C. Hawblitzel, K. Kourtis, T. Harris, T. Roscoe, Cosh: clear os data sharing in an incoherent world, in: 2014 Conference on Timely Results in Operating Systems (TRIOS 14), 2014.
[32] D. Otstott, N. Evans, L. Ionkov, M. Zhao, M. Lang, Enabling composite applications through an asynchronous shared memory interface, in: Big Data (Big Data), 2014 IEEE International Conference on, 2014, pp. 219–224, doi:10.1109/BigData.2014.7004236.
[33] The hdf5 group, 2016, (https://www.hdfgroup.org/HDF5/).
[34] The hdf5 group, 2016, (http://www.unidata.ucar.edu/software/netcdf/).
[35] Trinity, 2016.
[36] B. Kocoloski, J. Lange, H. Abbasi, D.E. Bernholdt, T.R. Jones, J. Dayal, N. Evans, M. Lang, J. Lofstead, K. Pedretti, et al., System-level support for composition of applications, in: Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers, ACM, 2015, p. 7.
[37] Q. Guan, Bee: Build and execution enviroments, 2016.
[38] Docker, 2016.
[39] P.H. Hargrove, J.C. Duell, Berkeley lab checkpoint/restart (blcr) for linux clusters, in: Journal of Physics: Conference Series, vol. 46, IOP Publishing, 2006, p. 494.
[40] Iofsl: I/o forwarding scalability layer.
[41] J.C. Sancho, F. Petrini, G. Johnson, E. Frachtenberg, On the feasibility of incremental checkpointing for scientific computing, in: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, IEEE, 2004, p. 58.
[42] G. Gibson, G. Grider, A. Jacobson, W. Lloyd, Probe: a thousand-node experimental cluster for computer systems research, USENIX ;login: 38 (3) (2013).
[43] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, R. Brightwell, Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing, in: Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, 2010, pp. 1–12, doi:10.1109/IPDPS.2010.5470482.
[44] P. Beckman, Argo: An exascale operating system and runtime, in: Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers, ACM, 2013, p. 2.