μsik – A Micro-Kernel for Parallel/Distributed Simulation

Kalyan Perumalla
kalyan@cc.gatech.edu

Technical Report **GIT-CERCS-TR-04-20**
Center for Experimental Research in Computer Science
Georgia Institute of Technology
May 25, 2004

Abstract

We present a novel micro-kernel approach to parallel/distributed simulation. Using the micro-kernel approach, we develop a unified architecture for incorporating multiple types of simulation processes. The processes hold potential to employ a variety of synchronization mechanisms, and could alter their choice of mechanism dynamically. Supported mechanisms include traditional lookahead-based conservative and state saving-based optimistic execution approaches, as well as newer mechanisms such as reverse computation-based optimistic execution and aggregation-based event processing, all within a single parsimonious application programming interface (API). We also present the internal implementation and a preliminary performance evaluation of this interface in ***μsik****, which is an efficient parallel/distributed realization of our micro-kernel architecture in C++.*

1. Introduction

High-performance parallel and distributed discrete event simulation (PDES) systems have traditionally been built from the ground up, for each major variant of various PDES techniques. However, it is desirable to have the freedom to add new techniques without having to develop entirely new simulation engines from scratch for every variant. To this end, we are interested in isolating the core invariant portion of PDES techniques, and provide a generalized framework for building traditional as well as newer techniques on top of the core. The core constitutes the micro-kernel, and the traditional implementations (conservative or optimistic) form the system services on top of the micro-kernel. This permits the incorporation of newer techniques on top of the core, as well as optimization of system services, without the need for system-wide changes.

The simulation micro-kernel approach is based on analogy with operating systems[1]. In operating systems that are based on micro-kernel architecture, a very basic set of services is provided by the operating system core (e.g., process identifiers and address spaces). Using such primitive services, the rest of the system services are in fact built outside the core (e.g., file systems and networking). We borrow this approach in our system. A micro-kernel operating system provides an easy and safe way of adding new system/kernel services, such as new network protocols and file systems. Similarly, a PDES micro-kernel provides an easy way to add new types of simulation processes without the need for an overhaul of the entire PDES system implementation.

The rest of the document is organized as follows. Section 2 presents the motivation and background for the design and development of the micro-kernel approach. The micro-kernel concepts for PDES are introduced in Section 3. Implementation details of the micro-kernel interface are described in Section 4. Section 5 describes the implementation of classical and newer system services on top of the micro-kernel. A preliminary performance study of μsik on a distributed platform is presented in Section 6. Finally, current status and future work on μsik are presented in Section 7.

2. Motivation and Background

In some of our current projects in collaboration with modeling experts in physical sciences, we are pursuing efficient execution of large-scale PDES models of physical phenomena such as solar wind interaction with the Earth's magnetosphere. These physics simulations are complex, and involve fine-grained event computations (typically consuming only a few microseconds per event execution). The models need a single engine that not only semi-transparently supports multiple synchronization approaches, but also entails sufficiently low overhead execution of fine-grained models. As certain models are better suited for optimistic execution, while others are appropriate for conservative processing, a single unified framework is needed for use by domain expert modelers, so that the modelers are not burdened with synchronization selection decisions *a priori* and separately for each model.

More generally, simulation processes should be free to decide for themselves whether they would like to process their events out of timestamp order, or wait until they are ascertained to be safe. Additionally, they should be free to adopt any other newer event processing scheme (e.g., aggregate event processing), or freely switch between schemes at runtime. In order to maximize communication locality and balance the load across the entire system, it should also be possible to have a mixture of processes using different mechanisms hosted in the same simulator instance. Since our focus is on very large-scale simulations, especially of physics models in our current projects, we need large-scale parallel/distributed execution capabilities.

2.1. Need for Comprehensive Approach

There is a plethora of issues to address in developing a comprehensive system for complex models. The PDES research community has developed a host of techniques for high-performance execution, but there is a need for an elegant framework for incorporating the multitude of techniques in an easy and modular fashion. Since most of the techniques are mutually orthogonal, it should be possible to support them all together in a suitably accommodating framework. Moreover, based on our past projects on conservative federated simulations[2-5] and optimistic simulation systems[6, 7], we see the need for a comprehensive framework for *incrementally* and easily incorporating various PDES techniques, as and when needed.

Simple ad-hoc prototypes do not seem to go far enough to sustain serious modeling efforts. Most complex models seem to require a wide variety of PDES techniques – e.g., multicast-based indirect communication for modularity, transparent load-balancing techniques for optimal speedup, transparent incremental state-saving techniques for complex modifications to state, and risk mitigation strategies for large-scale optimistic execution. Very few existing PDES implementations are capable of accommodating the complexity and supporting most of such techniques, and hence new systems are needed. At the same time, a "minimal risk" approach is needed for the new systems that does not preclude various optimizations down the line.

A case in point is the complexity associated with optimistic simulation. Implementation of a comprehensive modeling support for optimistic execution requires the development of a significant number of support services. This includes conversion of all traditional system services into optimistic execution mode: file I/O (open/read/write/close), dynamic memory management (malloc/free), etc. Implementation of such services is a complex endeavor.

Implementation Complexity – Example

For example, consider the implementation of optimistic memory allocation: malloc() and free() need to be made resilient to rollbacks. A generalization of this problem is called "optimistic I/O"[8]. An optimistic call to malloc() needs to be satisfied by allocating new memory. However, an optimistic call to free() should not be satisfied by immediately freeing up the memory. This is because incorrect memory reuse can occur if the call to free() happens to get rolled back later. Also, a call to malloc() can be rolled back by calling free(); however, an optimistic call to free() cannot be rolled back trivially. One simple solution (which is correct, albeit memory-inefficient) is to make free() a no-op altogether, but doing so can make the simulation consume unbounded amount of memory. At the heart of an entirely correct implementation for this problem is the need for efficient event committing primitives. By using appropriate primitives to "commit" an event when it becomes safe to do so, a call to free() can be committed safely. Similar treatment can be applied to other similar classes of problems, such as input/output operations to data files during simulation.

While general solution approaches to problems such as optimistic I/O are documented in literature, few usable systems are actually available for use in developing complex simulation models. A probable reason behind this lack of availability of usable systems is the complexity of their implementation with traditional approaches. One can speculate that existence of a comprehensive framework, like our micro-kernel, could have made implementations easier to develop and more readily available. The same can be said for a variety of other optimizations, such as incremental state saving[9, 10] and reverse computation[11].

Our thesis is that a large number of techniques in PDES can be supported *transparently* in a single *unified* framework, with a small set of fundamental primitives that are aimed at performance optimization. Based on this premise, we develop a unified application program interface (API) that encompasses most, if not all, synchronization approaches. Using this interface, simulation models can be written in a manner that is resilient to changes and optimizations to the underlying synchronization protocols.

The micro-kernel approach appears to be the most promising to satisfy all these goals. We chose the micro-kernel approach due to the number of advantages that it provides. Some of the advantages include simplicity, modularity, ease of development, ease of debugging and parsimony of interface.

2.2. Related Work

The High Level Architecture (HLA)[12] defined by

the US Department of Defense provides services for integrating a wide variety of simulator implementations, including space and/or time parallel (conservative, optimistic) discrete event simulations, and time-stepped continuous simulations. However, the architecture has been designed for interoperation of coarse integration entities, such as distributed programs communicating over the network. As such, it is not optimized for integration of fine-grained entities, as in the hosting of multiple event-oriented logical processes and/or threads within a single UNIX process. In particular, primitives to facilitate efficient process scheduling are not addressed in the standard; such primitives turn out to be the key to efficient execution of fine-grained autonomous entities.

The work more closely related to our present subject is by Jha and Bagrodia[13] in which a unified framework is presented to permit optimistic and conservative protocols to interoperate and alternate dynamically. (A variation of Jha and Bagrodia's protocols is later discussed in [14], but in the context of VLSI applications). High-level algorithms are presented in [13] that elegantly state the problem along with their solution approach. However, they do not address implementation details or performance data. Their treatment provides proof of correctness, but lacks an implementation approach and a study of runtime performance implications†. Our work differs in that we are interested in defining the interface in a way that guarantees efficient implementation, and we describe details for a high-performance implementation of such a unified interface. Some of our terms share their definitions with analogous terms in their work, but our interface uses fewer primitives and diverges in semantics for others. For example, our interface does not require the equivalent of their Earliest Output Time (EOT). Similarly, in contrast to their need for lookahead, we do not require that the application always specify a non-zero lookahead.

A variety of parallel/distributed software systems are available to support distributed conservative execution. However, very few software systems exist that support *distributed* optimistic simulation. Even fewer operational systems (almost none that we are aware of) are available for switching between conservative and optimistic modes at either at compile time or runtime.

SPEEDES[15] is a commercial optimistic simulation framework that is capable of distributed execution; however, it has not been shown to be suitable for high-performance execution of fine-grained applications. In fact, some evidence exists that indicates that its runtime and memory performance are not optimized for fine-grained distributed applications. GTW[7] and ROSS[16] are representative of high-performance implementations of optimistic simulators, but they are restricted to parallel execution on symmetric shared memory multiprocessor (SMP) platforms. This constraint limits the user's choice of hardware as well as scalability. An exception is the WARPED simulator[17], a shared-memory time warp system extended to execute on distributed memory platforms, but it has been evaluated on relatively small hardware configurations. We are interested in scalable execution on large-scale computing platforms, such as large clusters (hundreds) of quad-processor SMP machines typically available in supercomputing installations for academic research. The cluster-of-SMPs platform is more appealing since it is relatively less expensive as compared to a comparable SMP system for large number of processors.

We note that, while the possibility of switching between types of protocol is not new, our parsimonious API and our high-performance implementation approach are novel.

3. Micro-Kernel Concepts

In the micro-kernel view, simulation processes‡ are fully autonomous entities. Simulation processes communicate by sending and receiving events to/from other processes. They are free to determine for themselves when and in what internal order they would process their received events.

The micro-kernel does not process events in and by itself – it only acts as a router of events. In particular, it does not generate, consume or buffer any events. It does not examine event contents, except for the event's header (source, destination and timestamp§). The micro-kernel does not distinguish between regular events, retraction events, anti-events or multicast events. It also does not perform event buffer management (memory reuse, fossil collection, etc.), in contrast to traditional parallel/distributed simulation engines. The distinctions among event types and their associated optimizations are deferred to protocol-specific functionality of services outside the kernel proper. The responsibility of a micro-kernel is restricted to only providing services to the simulation processes such that the processes can efficiently communicate with each other, and collectively accomplish "asymptotic" time-ordered processing of events.

---

†It is commonly acknowledged that, in high-performance parallel/distributed execution, "the devil is in fact in the details".

‡Traditional PDES literature refers each distinct communicating entity in a simulation as a "logical process". We use the terms "logical process" and "simulation process" interchangeably.

§The timestamp of an event (also called its "receive timestamp") is the simulation time at which its receiver processes it.

3.1. Core Services

The micro-kernel core consists of *naming*, *routing* and *scheduling* services, as follows:

- **Naming**: The naming services of the micro-kernel provide a uniform way for simulation processes to locate and refer to each other, within and across processors in a parallel/distributed execution setting. The micro-kernel maintains a list of valid identifiers, and provides a way to map identifiers to processes and vice versa.
- **Routing**: The routing services ensure that events are transparently forwarded to the receiver process, regardless of whether the sender and receiver are co-located or distributed. It does so in a manner that ensures that no event timestamp is ever omitted in global timestamp-ordered processing.
- **Scheduling**: The scheduling services of the micro-kernel take care of allocating CPU cycles among multiple simulation processes in a manner that best promotes simulation progress, and ensures absence of livelock or deadlock.

A wide variety of PDES mechanisms can be built around this parsimonious set of core services, as outlined in Figure 1.

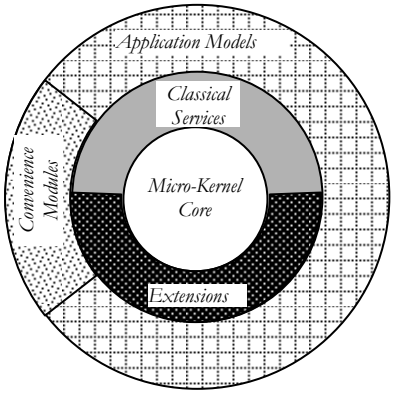

Figure 1: Elements of the micro-kernel architecture, and their inter-relationships.

Classical services include support for conservative and optimistic processing – event processing/commitment, rollback support and lookahead specification services. They also include kernel process support for remote (direct) communication, retractions and multicast (reflector-based) communication. Extensions are envisioned to accommodate newer techniques in the future, such as aggregate event processing. Convenience services include initialization, wrap-up, timers, and routines such as reversible random number generation.

3.2. Event Lifecycle and Categories

Events can be considered to go through different stages in their life cycle. First an event is allocated and scheduled by a sender simulation process. Next, the receiver simulation process performs initial processing of the event. This stage includes executing application (model) code associated with that event type. Eventually, in a following stage, final actions associated with the event are committed. Finally, the memory used by the event is released and recycled.

Based on the disposition of event lifecycle stages, at any given moment during simulation, all events belonging to a simulation process can be categorized into four distinct classes – committed, committable, processable and emittable. The first set of events (committed set) is those that have been processed, committed and whose memory has been released for reuse. The second set (committable set) consists of those that have been processed but are waiting to be committed. The third set (processable set) consists of events received by this simulation process that are waiting to be processed. The final set (emittable set) is a logical set that comprises those events that are schedulable by this simulation process to other simulation processes (excluding itself) during the processing of its current set of committable and processable events.

The event categories and their mutual ordering are illustrated in Figure 2.

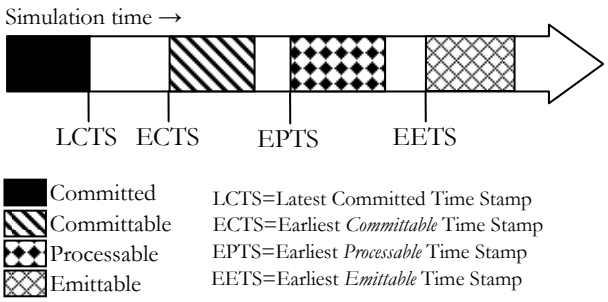

Figure 2: Illustration of the simulation timeline and important event categories in each simulation process. The relation LCTS≤ECTS≤ EPTS≤EETS always holds.

Note that the gaps between the event segments represent potential time discontinuities in discrete event simulation – since events are instantaneous actions, time "jumps" can exist between them.

A Lower Bound on Time Stamp (LBTS) value is defined as a distributed snapshot[18, 19] of the least EETS value among all processes in the simulation. In parallel/distributed execution, the exact value of the least EETS is difficult to compute due to network delays, and hence a lower bound on the least EETS is computed and safely used as the LBTS.

When simulation time advances to/beyond the timestamp of a committable event, examples of actions to be performed when committing the event include the following:

- **State vector release**: Release of state vectors, if any, used for state saving during optimistic processing. The state associated with the last committed event is noted as the most recently valid state.
- **Input/Output**: Operations such as conservatively printing output to the terminal, or reading from a file.
- **Memory allocation/release**: Finalizing the effect of operations by the application, such as memory allocation and memory release.

3.3. Process Scheduling

Without loss of generality, assume that the events in a process are logically stored in two data structures: FEL and PEL. The Future Event List (FEL) is the set of events in the process' processable event set. Processed Event List (PEL) is the set of events in the process' committable event set. Let FEL_i^{top} be the minimum timestamp in FEL_i (infinity if FEL_i is empty) and PEL_i^{top} be the minimum timestamp in PEL_i (infinity if PEL_i is empty). Note that PEL_i^{top} is always infinity for conservative simulation processes.

The earliest time stamp for each event category is computed as follows:

1.	$ECTS_i$	=	*Min*(FEL_i^{top}, PEL_i^{top})
2.	$EPTS_i$	=	*infinity* if conservative FEL_i^{top} if optimistic
3.	$EETS_i$	=	*Min*(FEL_i^{top} + Lookahead$_i$, PEL_i^{top})

In the preceding equations, $EETS_i$ is defined as a simple expression, but it could be expanded to include additional complexity, as needed. For example, if lookahead is highly variable across events, $EETS_i$ could be defined on a per-event basis: $EETS_i=\min(E_j+LA_j)$ for each event E_j in FEL_i, and LA_j is the lookahead for event E_j. Similar refinements can be made based on limiting it by the set of destination processes of process *i*.

In fact, $EETS_i$ for optimistic processes can be easily tightened further by noting that PEL_i^{top} is the timestamp of a local event. Hence, the earliest emittable timestamp is the event *generated* by this event, and not its own timestamp. Hence the $EETS_i$ can be refined to specify the minimum timestamp among all processed events that are locally *generated*, destined to other processes. A simple lower bound on this timestamp is PEL_i^{top}+Lookahead$_i$. This could be used instead of PEL_i^{top} in computing $EETS_i$.

On each processor, the scheduling algorithm proceeds by executing the code in Figure 3 within a loop (a formal proof of correctness of progress is being outlined in a separate document):

```
1.  if( ECTS_min < LBTS )
2.      Process_ECTS-min.advance( LBTS )
3.  else
4.      Process_EPTS-min.advance_opt( EPTS_min2 )
```

Figure 3: Body of micro-kernel's scheduler loop.

$ECTS_{min}$ is the minimum ECTS among all processes on that processor. $Process_{ECTS\text{-}min}$ is the process with the minimum ECTS value. $Process_{EPTS\text{-}min}$ is the process with the minimum EPTS value. $EPTS_{min2}$ is the second least EPTS value among all processes on that processor. The method P.advance(T) conservatively processes all events of process P with timestamps less than or equal to time T. The method P.advance_opt(T) optimistically processes all events of process P with timestamps less than or equal to time T. Either method is a no-op if P is null.

The LBTS itself is computed as the minimum EETS among all processes across all processors. Any transient event (in transit across processors) is considered to belong to the sender process' queues until the event reaches its receiver process. The LBTS computation can be performed concurrently with the scheduler (e.g., in a separate thread). Alternatively, cycles can be allocated inside the scheduler for LBTS computation just before optimistic processing (line 4).

3.4. Conservative Processing

During normal processing, the micro-kernel only schedules conservatively executable actions in increasing order of their committable timestamps. Only those processes whose ECTS values are less than or equal to the LBTS value are considered for conservative scheduling. The process with the least ECTS value is scheduled, and it is permitted to advance up to and including the current LBTS value. When that process is finished with its processing, the micro-kernel schedules the process with the next minimum ECTS value, and so on. Note that new events, if any, generated by the scheduled process will have timestamps greater than or equal to the current LBTS value. This is because LBTS is guaranteed to not exceed the minimum among EETS values of all simulation processes, which in turn implies that no process can emit an event with a smaller timestamp.

If no process exists whose ECTS value is less than or equal to the current LBTS value, then the micro-kernel initiates a new LBTS computation. A new LBTS value typically takes time to be computed, due to communication latency across processors. It is this

delay that induces blocking of conservative computation. This blocking period can be utilized as an opportunity to perform optimistic event processing. Hence, while a new LBTS value is being computed, the micro-kernel schedules those processes that are capable and willing to perform optimistic event processing, as described next.

3.5. Optimistic Processing

In optimistic execution mode, the micro-kernel schedules the process that has the least EPTS value. Recall that the EPTS value for conservative processes is infinity, and it is equal to the minimum timestamp among unprocessed events for optimistic processes (or, infinity if FEL is empty). Thus, if there are any optimistic processes, their EPTS values can make them schedulable for optimistic processing.

When at least one optimistic process exists for scheduling, optimistic execution is scheduled as follows: two processes with the minimum and the next minimum EPTS values (say, $EPTS_{m1}$ and $EPTS_{m2}$) are selected. If only one optimistic process exists, $EPTS_{m2}$ is set to infinity (in this case, this limit needs to be customized, if necessary, to throttle unbounded optimism). Then, the process with $EPTS_{m1}$ is allowed to optimistically process its events with timestamps less than or equal to $EPTS_{m2}$.

The constraint on optimistic time advance is designed to avoid the possibility of unnecessary rollbacks caused by violation of dependencies among local simulation processes. To elaborate, suppose the first process (with $EPTS_{m1}$) advances to beyond $EPTS_{m2}$, and later the second process (with $EPTS_{m2}$) sends an event with timestamp $EPTS_{m2}$ to the first process. The first process will then needlessly incur a rollback to $EPTS_{m2}$, which could be avoided in the first place by constraining the first process to advance only up to $EPTS_{m2}$.

Also, initiating optimistic execution only when all conservative processing is blocked ensures that the time spent in correct execution is maximized, and the possibility for incorrect execution (due to optimistic execution) is minimized.

3.6. Aggregate Processing

Yet another event processing mechanism is called "aggregate event processing." In this, events are processed in groups, rather than one event at a time. The advantages of aggregate processing can include reduced event handling overheads, and faster event execution. For example, dequeueing multiple events at the same time from the future event list could incur lesser cost than dequeueing them one at a time. The bodies of event-handling procedures could be better optimized by the compiler when the bodies are merged together (e.g., due to better use of common sub-expression evaluation).

Our framework permits one to add such an aggregate processing implementation without having to change the entire PDES infrastructure. Since a process is autonomous, it is free to process its events in any order it deems fit, as long as its time advances are performed in accordance to the micro-kernel interface. More importantly, such an implementation will seamlessly co-exist with the rest of the existing mechanisms.

In fact, aggregate processing could by itself be implemented as two variants: conservative aggregate processing, and optimistic aggregate processing. In conservative aggregate processing, only events whose timestamps are less than LBTS are permitted to be merged. In optimistic aggregate processing, this restriction is relaxed (aggregated events are permitted to span beyond LBTS), provided the application supplies an aggregated-undo procedure (e.g., aggregated reverse computation or state-saving) to rollback incorrect aggregate processing.

4. Micro-Kernel Implementation

We now describe our implementation of the micro-kernel interface.

A naïve implementation of the micro-kernel approach could entail significant overheads, as compared to the traditional monolithic simulator implementations. In a monolithic simulator, it is possible to optimize the implementation by employing centralized data structures such as event buffers, event lists and state vectors. On the other hand, in a micro-kernel, the key data structures are, by design, encapsulated inside simulation processes. The challenge is to find efficient ways of implementing the micro-kernel framework so as to minimize or eliminate overheads in (a) synchronizing time across all processes (b) inter-process event communication.

A key issue is the problem of keeping accurate ordering among processes with respect to their ECTS, EPTS and EETS values. For example, when a new event is sent from one simulation process to another, the receiver's ECTS, EPTS and EETS values can change. Similarly, a simulation process will have its values changed at the end of processing an event. Event retractions need to be dealt with appropriately, as timestamp-ordered events. Anti-events generated during rollback of optimistic simulation need to be accurately accounted for in time synchronization. The choice of data structures determines the efficiency of micro-kernel operation.

As its main components, the micro-kernel maintains a list of local user processes, a hash table for mapping process identifiers to processes, and a list of kernel processes. For scheduler operations, three important priority queues are maintained. Each of these components is described next.

4.1. Naming Services

To provide naming services, the micro-kernel maintains a mapping of process identifiers to process instances. The mapping is dynamic, and can contain gaps in identifiers. Since processes can be added and deleted at runtime, gaps can arise due to deletion of processes at runtime. Identifiers are assigned to new processes using a sequentially incremented counter. As a design decision, we opted for a minimal complexity solution by always using new process identifiers, and to never reuse identifiers even though some identifiers might become unused due to process deletion. We believe this will not create the problem of identifier exhaustion in applications of interest to us, since they are *not* envisioned to create/delete processes with very high frequency.

Simulation processes can be kernel processes or user processes. Kernel processes are used for internal implementation of services on top of the micro-kernel (see Section 4.3). User processes are part of application model.

User processes are assigned positive identifiers, starting at 0, while kernel processes are assigned negative identifiers, as shown in Figure 4. The rationale behind this scheme is that it allows applications to rely on their processes being identified from 0 to *n*-1 (this is a common way in which models are written). Using negative identifiers for kernel processes makes them transparent to the application, and will not interfere with the traditional modeling methods.

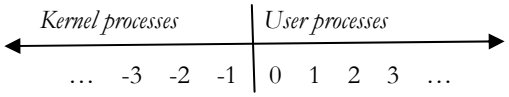

Figure 4: Every simulation process is assigned a locally unique identifier as soon as it is added to the simulation. User processes are assigned positive identifiers starting with 0, while kernel processes are assigned negative identifiers starting with -1. Identifiers are assigned from an incrementing counter, and are *not* recycled when processes are deleted.

Special identifiers are also defined for specifying an invalid identifier, and to specify multicast destinations.

4.2. Scheduling Services

The scheduler is implemented as a loop inside a micro-kernel method.

Process Ordering

Three in-place *min-heaps* are used, one each for tracking the ECTS, EPTS and EETS values of simulation processes. Each heap maintains the minimum time-stamped process at the top. For example, the process with the least ECTS value is always available as the top of ECTS heap. The heaps are designed to rapidly update and readjust the data structure when the key of an element is increased or decreased. This rapid update is essential to quickly keep the heaps consistent before and after every scheduling action by the scheduler.

Readjusting Timestamp Orders within Scheduler

When events are sent or received by simulation processes, their relative ordering can change with respect to their ECTS, EPTS, and EETS values. The heaps of the micro-kernel scheduler need to be readjusted to restore correct timestamp order. This readjustment is accomplished via a before_dirtied() and after_dirtied() pair of methods within simulation process. These methods keep track of whether any changes occurred to the key timestamps. If (and only if) any of the ECTS, EPTS or EETS values of an affected process changes, the corresponding scheduler heap is updated. The affected process that needs to be updated could be the active (sending) process that is currently scheduled. Additionally, it could also be the set of processes to which the currently scheduled process generates new events.

Distributed Time Synchronization

To compute LBTS values, we employ the distributed snapshot algorithm described in [20]. Our latest implementation [21] of this algorithm employs synergetic time management. In synergetic time management, multiple distinct synchronization algorithms are concurrently initiated and superimposed. Their combined performance can be expected to be better than each of them running individually and separately. Our current implementation includes two different modules: one is based on efficient global hierarchical reductions[20, 22], while the other is based on an optimized variant[23] of the Chandy-Misra-Bryant null message algorithm[24]. These have been tested on large-scale platforms, and have been demonstrated to scale very well, even up to supercomputing configurations of more than 1500 processors[22, 23, 25].

A nuance to be carefully considered in synchronization is related to retractions. It is important to ensure that LBTS does not advance as far as a retraction's timestamp, even if a retraction has the least timestamp in the entire simulation. Consider the case in which an event is sent from processor A to processor B, and later processor A initiates a retraction of that event. Suppose no other event exists in the entire system. If LBTS is permitted to advance up to the timestamp of a retraction, processor B could (incorrectly) commit the event before the retraction request reaches processor B (note that no lookahead constraints are violated). Such an incorrect situation is avoided by permitting the LBTS to advance to a value that is strictly less than the least timestamp of all retractions in the system. This ensures

that events are not incorrectly committed.

4.3. Routing Services

Local communication is trivially handled by enqueueing the event in the local destination process. Remote communication is implemented via a special delegation mechanism using kernel processes (see below). Similarly, multicast (reflector) services are implemented using additional kernel processes. The reflector kernel processes themselves use the other kernel processes for remote communication. As indicated earlier, the micro-kernel itself never stores or buffers any events at any time. Every event routed through the micro-kernel is immediately delegated either to the destination process (if it is a local user process), or delegated to a local kernel process (if the destination is a remote process or a multicast group).

Kernel Processes

Kernel processes are used to implement remote federate communication and reflector-based event exchanges. The reason they are implemented this way is that the functionality can be quite seamlessly implemented using the scheduling services provided by the micro-kernel core. Operations such as retraction services, secondary rollbacks, and multicast exchanges are all easier to implement as libraries of simulation processes.

This is fairly analogous to operating system micro-kernels. Services such as networking, file I/O, etc. are implemented as processes outside the micro-kernel core, which themselves utilize many of the services that user processes utilize.

Remote Event Communication

Events exchanged by simulation processes across processor boundaries are handled via a special mechanism that is consistent with the micro-kernel approach. Analogous to networking support in operating system micro-kernels, the remote event exchanges are handled by special kernel processes. The kernel processes for remote communication act as local representative proxies for the corresponding remote processors.

These kernel processes are responsible for maintaining a mapping from event identifiers to event buffers. Such a mapping is necessary in order to implement event retractions (during conservative and/or optimistic execution) and anti-events (to realize secondary rollback/cancellation in optimistic execution). The kernel process is also responsible for periodically flushing the hash table when events are committed and can no longer be retracted or canceled.

The use of kernel processes to implement remote communication aids in easily adding various optimizations. Sophisticated variants can be incorporated with few changes to the rest of the system. Here we briefly discuss a few possibilities:

Optimistic Sends: This is the most common method, in which an event scheduled to a remote process is immediately sent over the wire to its corresponding remote processor. A downside with this scheme is that the network communication cost becomes a wasted overhead if the event is later retracted. The event retraction could be initiated either by the user (in conservative or optimistic processing) to take back a previously scheduled event, or by the kernel for event cancellation (anti-messages for secondary rollbacks in optimistic processing).

Lazy Sends: Instead of forwarding the event immediately over the wire to the remote processor, the event could be withheld within the kernel process for dt simulation time units, where $0<dt\leq(T_{event}-T_{now})$. Delaying the event longer will postpone the network communication cost, which is beneficial in case the event is retracted later. On the flip side, it might increase the event communication latency, and stall the receiving processor waiting to receive the event for its own progress. Adaptive schemes could be devised and implemented in the kernel process to exploit this "lazy send" optimization.

Non-aggressive Sends: The kernel process can also be used to easily implement non-aggressive sends – i.e., to send remote messages if and only if they cannot be retracted in the future. This is a well-known PDES variant in optimistic simulation to separate risk and aggressiveness [26], in which events are processed optimistically *locally*, but only "correct" events are propagated *across* processors. The kernel process adds the event in its FEL, and "processes" the events in a conservative fashion. The event is actually sent over the wire to the remote processor only when it is committed. Since events are committed only when they are guaranteed to be not retracted, non-aggressiveness is ensured.

Message Bundling: To amortize the cost of network communication, it is possible to bundle multiple events into one message. The cost savings can be good especially when events are small in size, as compared to network message headers (e.g., TCP header size). Again, such bundling techniques can be incorporated into the kernel process responsible for remote communication.

5. Simulation Process Base

While conforming to the API required by the micro-kernel, the simulation processes have the flexibility to be implemented in a variety of ways. Here, we describe one such implementation, whose methods can be used for both conservative as well as optimistic application

processes. Our implementation encompasses both types of processes.

5.1. Base Execution Framework

The simulation process interface has three tiers. Tier I consists of methods invoked by the micro-kernel on simulation processes on various occasions, as described in preceding sections. Tier II consists of implementation-specific methods provided by the simulation process to its subclasses, for a variety of synchronization modes, including conservative and optimistic execution. Tier III consists of some convenience methods, such as for initialization and termination.

Tier I	enqueue() dequeue()	advance() advance_opt()	ects() epts() eets()
Tier II	dispatch() undispatch()	undo_event() commit_event()	save_state() free_state()
Tier III	init() execute() wrapup()	set_timer() timedout()	retract()

Figure 5: A subset of methods of μsik simulation processes. Tier I methods define the interface that the micro-kernel expects from all simulation processes. Tier II methods define the services provided by the base implementation of a simulation process to its subclasses. Tier III are convenience services provided for models.

All events belonging to a simulation process are maintained in two data structures encapsulated within that process: FEL and PEL, as shown in Figure 6. Unprocessed events (previously processed events that are later rolled back, or new incoming events that are not processed yet) are stored in the FEL, which is a *min-heap* ordered by events' receive timestamps. Processed events are stored in PEL, which is a doubly-linked list stored in increasing timestamp order from head to tail. Newly processed events are appended to PEL tail. The PEL is rolled back to the point of timestamp fault before a new unprocessed event from FEL is processed.

Lookahead

Lookahead can be specified on a per-destination basis: add_dest() method can be used to specify a destination process ID and associated lookahead. A generic lookahead can be given by specifying a wildcard process ID. If a simulation process never adds any processes in its destination list, it is assumed to specify zero lookahead to every other process.

State Saving

State saving is supported in the base process implementation via calls to two abstract methods: save_state() and free_state(). The base implementation for an optimistic process can utilize these two hooks, in addition to commit_event(), to implement most variants of state saving – e.g., copy, incremental and periodic.

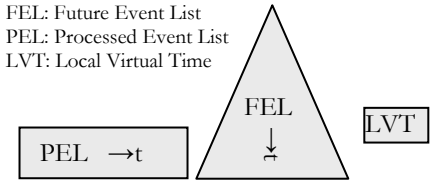

Figure 6: Internal state of the base simulation process implementation. FEL is implemented as a *min-heap* priority queue, and PEL is implemented as a linear linked list of events ordered by their timestamp. LVT is the timestamp of the most recently processed (or being processed) event that has not been rolled back.

Reverse Computation

Similar to state saving, the base process implementation provides hooks to add reverse event handlers that are automatically invoked if/as needed for rollback. The application provides reverse event handlers by overloading the undo_event() method.

Aggregate Event Processing

We are currently in the process of incorporating support in our base implementation for both the variants of aggregate event processing, as described in Section 3.6. This is essentially achieved by overloading the advance() or advance_opt() methods to process multiple events at a time.

Process Migration and Load Balancing

Process migration is realizable as time-synchronized deletion of a process at source processor and addition of a new process at the destination processor. Since the deletion is time-synchronized, the process is guaranteed to have received all events up to the time of deletion. The new process at the destination processor can be appropriately initialized with the same set of events as with the deleted process at the source processor.

5.2. Event Retraction

In order to implement event retraction in a distributed memory setting, it is necessary to map event pointers to memory-independent identifiers, and vice versa. In each kernel process for remote communication (see Section 4.3), a mapping between event pointers and event identifiers is maintained in a hash table to quickly locate the event to retract across distributed memory/address-spaces. Event identifiers are assigned by the kernel processes just before events are sent out on the wire. These event identifiers are specified in retraction requests if/when the events are

retracted later. Since the mapping table can grow unboundedly over the length of simulation, memory is recycled by flushing obsolete mapping entries when events are committed by the kernel processes.

5.3. Event Reflectors

Multicast (indirect) communication among simulation processes is enabled by the concept of "reflectors". A reflector embodies the state required to maintain the list of currently subscribed processes. Reflectors are implemented as kernel processes, one process per reflector. When an event is sent (posted) to a reflector, the corresponding reflector process takes care of forwarding the event to all processes that are subscribed to that reflector. Subscriptions, un-subscriptions, publications and reflections are all time synchronized. For example, if a process at time T_p posts an event with timestamp T_e timestamp, and another process subscribes to the reflector at time T_s, $T_p \leq T_s \leq T_e$, the reflector ensures that the newly joined process receives a copy of the event (since that process is present in the list of subscribed processes by the event's timestamp).

The framework is capable of accommodating both conservative as well as optimistic operations on reflectors. In the interest of space and time, the details of an actual implementation are deferred to a later report. A comprehensive coverage of time-synchronized multicast implementations is presented in [27]. We believe that all the techniques described in that work can be easily implemented in our micro-kernel framework using reflector kernel processes.

5.4. Kernel Events

Certain base functionality, such as process initialization and timer services are best provided as convenience functions as part of the base class for simulation processes. Implementation of these services requires the use of time-stamped events, but those events are best kept inside the system implementation. Kernel events are used for implementing such internal functionality. Kernel events are just like regular events, and are in fact instantiated as subclasses of the base event type. However, they are not exposed to the application/model and as such are treated as part of the simulator system services.

5.5. Optimized Queues and Lists

For a high-performance implementation, it is important to design the data structures carefully to minimize all runtime overheads. Traditionally, to incur minimal overheads, the norm has been to avoid encapsulated data types, and instead manipulate elements explicitly via pointers. For example, GTW[7] and ROSS[16] both use pointers and macros extensively to manipulate various data structures, such as free state-vectors, lists of processed events, etc. While such implementations help keep the runtime costs very low, they however result in source code that is extremely hard to study, understand, debug, test, enhance and maintain.

One solution to avoid the pitfalls of low-level pointers-based implementation is to use standard template libraries. The templates help capture errors at compile-time, as well as encapsulate operations on complex data structures. However, the use of templates comes at a runtime cost. The internal implementation of standard templates uses dynamic memory allocation/de-allocation, which adds to runtime cost. The challenge is to retain the advantages of encapsulation and compiler-assisted type checking (as with templates), but at the same time minimize runtime overheads.

In order to avoid and eliminate overheads associated with dynamic memory allocation/de-allocation in handling priority queue and list data structures, we define our own heap and list data types. Our definitions are different from other library templates in that our definitions permit the same object to be linked into multiple instances of the same container type, without the need to allocate container headers to hold the elements. Standard template libraries are difficult to use or inefficient when the same element needs to belong to multiple instances of the same type of container.

For example, in our micro-kernel, we need to link each simulation process into three different priority queues *simultaneously*. This is to order the processes along their three basic timestamps: ECTS, EPTS and EETS. Thus, the key used for ordering in each queue is different. Moreover, the keys are overloaded member methods, rather than member variables, of the simulation base class. However, the container data type is exactly the same: a priority queue (implemented as an in-place *min-heap*).

With our approach, we are able to employ the same priority queue implementation for ordering processes by their ECTS/EPTS/EETS timestamps, as well as for ordering events within a process by the events' receive timestamps.

6. Performance Study

Our implementation currently runs in parallel on a network of shared-memory multiprocessors, and is portable across homogeneous configurations of Windows, Mac and Unix/Linux platforms. As of this writing, it has been tested on Intel Pentium and Itanium architectures.

We present preliminary performance data of μsik next, using a standard PDES benchmark known as Phold[28]. In our Phold implementation, *NLP* simulation processes are evenly mapped to all available

processors. A fixed population of events, *NLP**R, is generated at initialization, with random destinations (*R*, an integer, is the ratio of number of events to number of processes). When a process receives an event, it schedules a new event into the future to another random destination (possibly to itself) with a minimum time increment called lookahead. With probability *L*, the destination is on the same processor as the source. We use a fixed increment equal to a lookahead of value 1.0, and use a uniform random number generator to randomly determine event destinations.

The Phold benchmark is a fine-grained application, with very little computation performed per event. As a result, it represents a worst-case scenario that can reveal runtime overheads of the simulation engine.

We report our performance numbers on two clusters: (1) the Sith cluster, which is a cluster of Itanium systems, each system containing two Itanium II 900MHz processors and 8GB memory (2) the Jedi cluster, which is a cluster of Intel SMPs, each system with 8-CPUs of Intel Xeon processors and 4GB memory.

6.1. Sequential Performance

Sequential execution of the Phold application can help reveal the overheads associated with process scheduling as well as event exchanges. Figure 7 (for the Sith cluster) and Figure 8 (for the Jedi cluster) show the average time taken to process an event in Phold, for increasing number of simulation processes and events. The time per event includes both send/receive costs, as well as process scheduling costs.

The process scheduling costs are accentuated when the event population is low. For example, when *R*=1, on average, each simulation process has a single event, and holds a high probability that its next send is not to self. This forces the update of time queues for the scheduled process as well as the destination process for the newly scheduled event. When *R*=10, it is ten times more likely that the newly scheduled event by a simulation process could be sent to itself.

Note that the simulation processes are plotted in logarithmic scale along the *x*-axis. The data demonstrates that our micro-kernel implementation scales excellently with the number of simulation processes, without drastic overheads for the maintenance of ECTS, EPTS and EETS values.

In the largest sequential configuration, we are able to simulate an event population of 10 million events and 100,000 simulation processes.

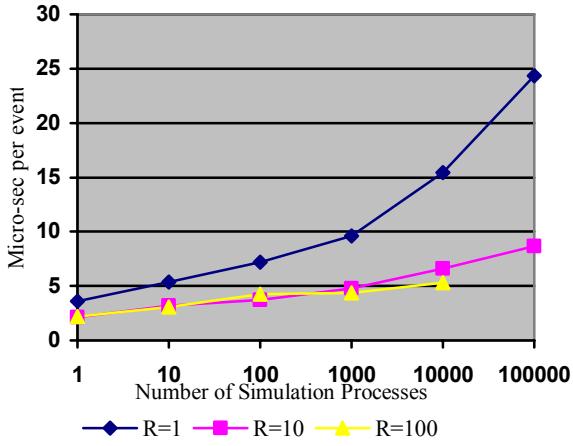

Figure 7: Sequential performance of µsik on Phold, demonstrating scalability of implementation up to million events and hundred thousand simulation processes, all active at the same moment on one processor (on the Sith cluster).

The performance for low event population, however, appears to be less than perfect. While such a low event configuration is not common in real applications, we are investigating ways to minimize the overhead in such situations as well.

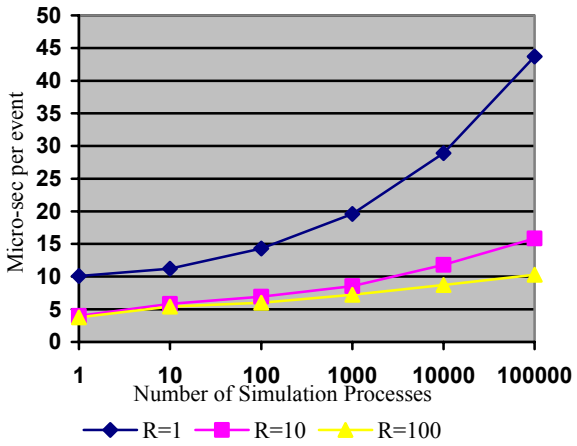

Figure 8: Sequential performance of µsik on Phold, demonstrating scalability of implementation up to 10 million events and hundred thousand simulation processes, all active at the same moment on one processor (on the Jedi cluster).

6.2. Time Synchronization Cost

In small parallel executions, the cost of time synchronization across processors is low, and is shown to scale with the number of processors. Figure 10 shows the average processing time per event while the

number of processors is varied. This experiment is intended to measure time synchronizing cost in isolation from remote event exchange costs. This is achieved with *L*=100% by choosing random destinations only from among local processes (i.e., no event goes across processors). The entire distributed execution, however, is still time managed – LBTS computations are performed, and time advances of simulation processes are permitted only upon LBTS advances.

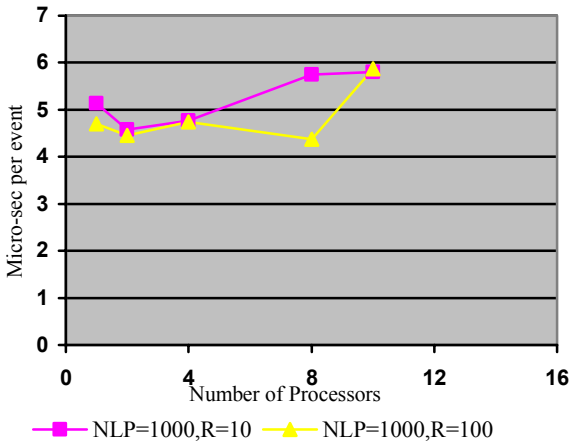

Figure 9: Conservative parallel performance of µsik on Phold with localized communication (on the Sith cluster).

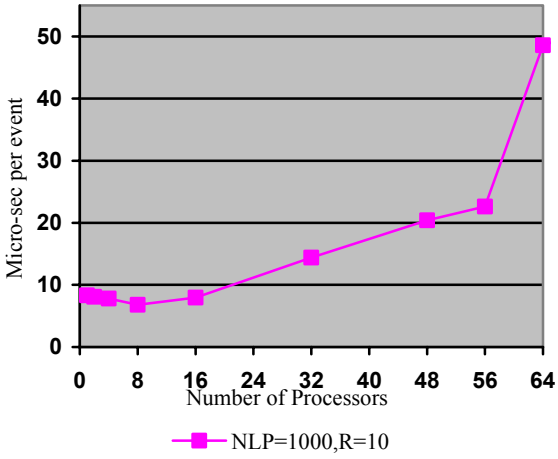

Figure 10: Conservative parallel performance of µsik on Phold with localized communication (on the Jedi cluster).

By relating this data to the sequential performance shown in Figure 8, it is clear that the time synchronization overhead is low up to 56 processors. The performance seems to weaken at 64 processors. We are investigating the reasons behind this degradation at larger number of processors.

6.3. Optimistic Simulation Performance

In the optimistic configuration, each of the Phold processes executes its events optimistically ahead in time. Rest of the application is unmodified. In fact, the only change between the conservative and optimistic executions is the invocation to set the optimistic execution flag in the simulation process.

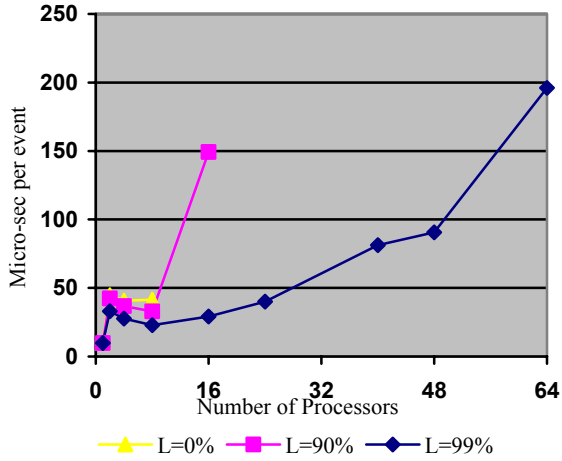

Figure 11: Optimistic parallel performance of µsik on Phold (on the Jedi cluster). NLP=1000, R=10.

Figure 11 shows the performance of optimistic parallel execution on an example configuration of Phold. This execution is intended to demonstrate the capability as a working proof-of-concept of our prototype. Further work is needed to reduce overheads in larger parallel executions, especially by incorporating flow-control mechanisms that can adaptively throttle over-optimistic execution.

6.4. Mixed Simulation Performance

A simple change of the configuration yields an example in which every alternate process in Phold is conservative, and every other process is optimistic. This configuration is once again intended to serve as proof-of-concept demonstration of the micro-kernel approach that can accommodate both types of processes. Figure 12 shows the performance of such a mixed configuration executing in parallel on up to 64 processors.

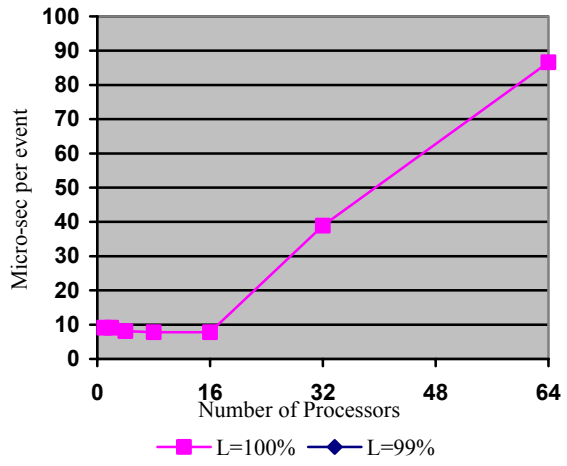

Figure 12: Mixture of optimistic and conservative processes parallel performance of µsik on Phold (on the Jedi cluster). NLP=1000, R=10.

6.5. Memory Usage

Figure 13 demonstrates that memory is recycled efficiently on large configurations. Events are committed (and hence freed) as fast as possible, in a scalable fashion.

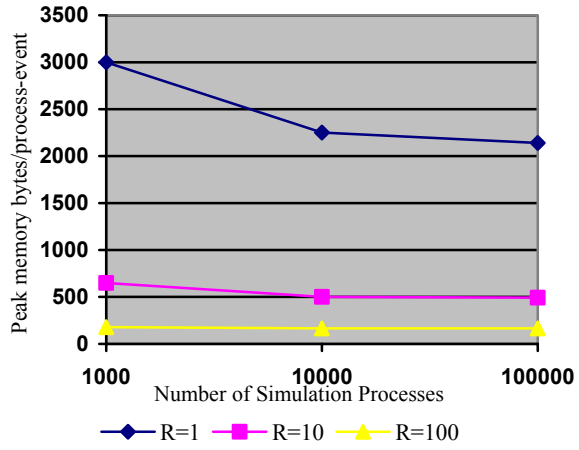

Figure 13: Memory usage of µsik in sequential execution of Phold (on the Jedi cluster).

Memory usage in bytes on the *y*-axis is computed as: $(M_{RP}-M_1)/(R*P+P)$. Here, M_{RP} is the memory usage for a Phold execution of P simulation processes and $R*P$ event population. M_1 is the memory usage of *1* process and 1 event. This essentially computes the average combined memory occupied by all processes and events. For example, when R=1, processes and events are equally weighted in the average. When R=100, event memory usage dominates, showing that each event on the average consumes 165 bytes, which is roughly twice the size of each base event. Each base event is 84 bytes long, and each base process is 168 bytes long. Memory for each event consists of the base event plus application data for each event, rounded to the next nearest multiple of 64.

The data shows that µsik's usage of memory per event is bounded in proportion to the actual number of events, and scales with the both number of events as well as the number of simulation processes.

7. Status and Future Work

µsik is a general-purpose parallel/distributed simulation kernel built upon a micro-kernel architecture consisting of autonomous simulation processes. Simulation processes are autonomous in the sense that they hold and manage their own events, and can be optimistic or conservative in their event processing, or adopt other techniques such as aggregate event processing. The micro-kernel overhead is kept very low by design, and runtime and memory are scalable with both the number of events as well as the number of logical simulation processes. µsik also uses the concept of kernel processes, which serve to push kernel-functionality to outside the micro-kernel, as simulation processes themselves (e.g., for inter-processor event exchanges and retractions).

The current implementation is portable across UNIX/Linux and Windows platforms. The micro-kernel source-code is compact, comprising less than 3000 lines of C++ code. The implementation is built on top of a thin software layer called libSynk[21] which provides efficient communication and virtual time synchronization across processors in shared and/or distributed memory platforms.

The µsik software release includes the micro-kernel source code, example applications, and a user's manual. The most recent version is available for download from the following URL:

http://www.cc.gatech.edu/computing/pads/kalyan/musik.htm.

We are currently working on profiling the run-time performance on benchmark applications, in order to identify the most time-critical paths in execution. We envision being able to further reduce runtime overheads for process scheduling, and distributed synchronization on larger number of processors. Optimistic execution also needs further enhancement, such as incorporating flow control and additional system services.

Acknowledgements

This work has been supported by National Science Foundation grant ATM-0326431.

References

[1] J. Liedtke, "On Micro-Kernel Construction," presented at ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado, USA, 1995.

[2] R. M. Fujimoto, T. McLean, K. S. Perumalla, and I. Tacic, "Design of High-performance RTI Software," presented at Workshop on Distributed Simulations and Real-time Applications, 2000.

[3] R. M. Fujimoto, S. Ferenci, M. Loper, T. McLean, K. S. Perumalla, G. F. Riley, and I. Tacic, "FDK Users Guide," College of Computing, Georgia Institute of Technology 2001/02/14 2001.

[4] T. McLean and R. M. Fujimoto, "The Federated Simulations Development Kit: A Source-Available RTI," presented at Spring Simulation Interoperability Workshop, 2001.

[5] G. F. Riley, M. Ammar, R. M. Fujimoto, A. Park, K. S. Perumalla, and D. Xu, "A Federated Approach to Distributed Network Simulation," *ACM Transactions on Modeling and Computer Simulation*, vol. 14, pp. 116-148, 2004.

[6] S. L. Ferenci, K. S. Perumalla, and R. M. Fujimoto, "An Approach for Federating Parallel Simulators," in *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*: IEEE Computer Society, 2000, pp. 63-70.

[7] S. R. Das, R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, "GTW: A Time-Warp System for Shared Memory Multiprocessors," presented at Winter Simulation Conference, 1994.

[8] M. Hybinette and R. M. Fujimoto, "Latency Hiding with Optimistic Computations," *Journal of Parallel and Distributed Computing*, vol. 62, pp. 427-445, 2002.

[9] D. West and K. Panesar, "Automatic Incremental State Saving," presented at Workshop on Parallel and Distributed Simulation, Philadelphia, Pennsylvania, USA, 1996.

[10] R. Ronngren, M. Liljenstam, R. Ayani, and J. Montagnat, "Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation," presented at Workshop on Parallel and Distributed Simulation, Philadelphia, Pennsylvania, USA, 1996.

[11] C. Carothers, K. S. Perumalla, and R. M. Fujimoto, "Efficient Optimistic Parallel Simulations using Reverse Computation," *ACM Transactions on Modeling and Computer Simulation*, vol. 9, 1999.

[12] "IEEE Std. 1516: High Level Architecture," in *Institute of Electrical and Electronic Engineers*, 2000.

[13] V. Jha and R. Bagrodia, "A unified framework for conservative and optimistic distributed simulation," presented at Workshop on Parallel and Distributed Simulation, 1994.

[14] D. Lungeanu and C.-J. R. Shi, "Distributed simulation of VLSI systems via lookahead-free self-adaptive optimistic and conservative synchronization," presented at IEEE/ACM International Conference on Computer-Aided Design, San Jose, California, United States, 1999.

[15] Metron, "SPEEDES: Synchronous Parallel Environment for Emulation and Discrete-Event Simulation", Last accessed 2004/05/01, http://www.speedes.com/.

[16] C. Carothers, D. Bauer, and S. Pearce, "ROSS: A High-Performance, Low Memory, Modular Time Warp System," *Journal of Parallel and Distributed Computing*, vol. 62, pp. 1648-1669, 2002.

[17] G. D. Sharma, R. Radhakrishnan, U. K. V. Rajasekaran, N. Abu-Ghazaleh, and P. A. Wilsey, "Time Warp Simulation on CLUMPS," presented at Workshop on Parallel and Distributed Simulation, Atlanta, Georgia, USA, 1999.

[18] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transaction on Computer Systems*, vol. 3, pp. 63-75, 1985.

[19] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation," *Journal of Parallel and Distributed Computing*, vol. 18, pp. 423-434, 1993.

[20] K. S. Perumalla and R. M. Fujimoto, "Virtual Time Synchronization over Unreliable Network Transport," presented at Workshop on Parallel and Distributed Simulation, 2001.

[21] K. S. Perumalla, "libSynk Home Page", Last accessed 2004/05/01, http://www.cc.gatech.edu/computing/pads/kalyan/libsynk.htm.

[22] K. S. Perumalla, A. Park, R. M. Fujimoto, and G. F. Riley, "Scalable RTI-based Parallel Simulation of Networks," presented at Workshop on Parallel and Distributed Simulation, San Diego, 2003.

[23] A. Park, R. M. Fujimoto, and K. S. Perumalla, "Conservative Synchronization of Large-scale Network Simulations," presented at Workshop on Parallel and Distributed Simulation, 2004.

[24] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, vol. 24, pp. 198-205, 1981.

[25] R. M. Fujimoto, K. S. Perumalla, A. Park, H. Wu, M. Ammar, and G. F. Riley, "Large-Scale Network Simulation -- How Big? How Fast?," presented at IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunication Systems (MASCOTS), 2003.

[26] P. F. Reynolds, Jr., "A Spectrum of Options for Parallel Simulation," in *Proceedings of the 1988 Winter Simulation Conference*, 1988, pp. 325-332.

[27] I. Tacic and R. M. Fujimoto, "Synchronized Data Distribution Management in Distributed Simulations," in *Proceedings of the Workshop on Parallel and Distributed Simulation*, 1998.

[28] R. M. Fujimoto, "Performance of Time Warp Under Synthetic Workloads," in *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, *SCS Simulation Series*, 1990, pp. 23-28.