Schedulable Persistence System for Real-Time Applications in Virtual Machine

Okehee Goh
CSE, Arizona State University
Tempe, AZ
ogoh@asu.edu

Yann-Hang Lee
CSE, Arizona State University
Tempe, AZ
yhlee@asu.edu

Ziad Kaakani
Honeywell International Inc.
Phoenix, AZ
ziad.kaakani@honeywell.com

ABSTRACT

Persistence in applications saves a computation state that can be used to facilitate system recovery upon failures. As we begin to adopt virtual execution environments (VMs) for mission-critical real-time embedded applications, persistence service will become an essential part of VM to ensure high availability of the systems.

In this paper, we focus in a schedulable persistence system in VMs and show a prototype persistence system constructed on CLI's open source platform, MONO. By employing object serialization, the system enables concurrent and preemptible persistence operation, i.e., the task in charge of persistence service runs concurrently with application tasks and is a target of real-time scheduling. Thus, the execution of application tasks can be interleaved with the operations of persistence service, and the task timeliness can be guaranteed as the pause time caused by persistence service is bounded. The experiment output on the prototyped system illustrates that persistence service is appropriate for real-time applications because of its controllable pause time and its optimized overhead.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Realtime and embedded systems; D.4.5 [**Operating systems**]: Reliability —*Checkpoint/restart*; D.4.7 [**Operating systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms

Reliability Performance

Keywords

schedulable persistence system, checkpoint/recovery, real-time applications, virtual machine, CLI

1. INTRODUCTION

Virtual Machines (hereafter, VMs), such as JVM[17] and CLI (Common Language Infrastructure)[9], enable an abstract computing environment for program execution. Application programs are compiled into intermediate codes (bytecodes) to permit portability of "write once, run everywhere." Besides that, applications written in Java[13] or CLI-compatible languages are claimed secure due to type-safety and security sandbox model [27]. The ensured safety, portability, and reusability of OO languages make VMs attractive to embedded systems and have led to the introduction of many VMs designed for embedded applications. One example is Real-Time Specification for Java (RTSJ)[4] which has been established as a standard specification of JVM to meet the requirements of real-time embedded applications.

Along with the interests of employing VM in real-time embedded systems, persistence of applications is becoming a necessity as an approach to ensure high availability of long-running embedded applications. Persistence in applications preserves a computation state of applications beyond its lifetime. If a failure occurs, the systems can be recovered by using the preserved computation state while minimizing any loss of computation. Without a support of persistence, system failures may force the systems to restart from the beginning (cold start) or result in an inconsistent state with which computation cannot advance. Despite of the advantages of persistence, there is a concern whether making applications persistent may introduce an unpredictable latency that can impair the timeliness of real-time embedded systems. For example, Monga et al. [19] shows in the experiment of persistence service using standard object serialization of .NET framework that serialization for 100 instances of System.Int32 type takes about 100ms, which is intolerable to most real-time applications in the aspects of both performance and pause time.

Before proceeding further, we firstly define some terminologies, frequently used in the rest of this paper. *Persistence System* and *Persistence Service* refer to a subsystem in charge of making applications persistent, and an activity of making applications persistent, respectively. Additionally, *Persistent data* and *Persisted data* indicate the data that needs to be persistent among runtime data in applications, and the one that is saved through persistent service, respectively.

Due to the time constraints that real-time applications have, persistence service integrated to real-time systems should not cause unpredictable blocking delays to application tasks. To prevent unpredictable blocking delay, the service should

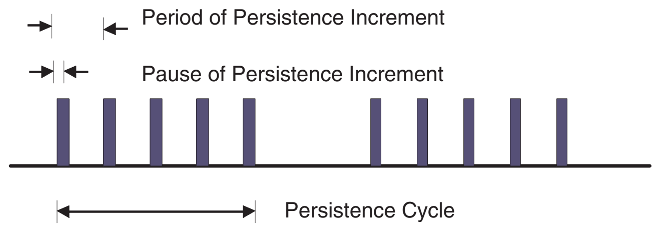
Period of Persistence Increment
Pause of Persistence Increment
Persistence Cycle

Figure 1: Scheduling Model of Schedulable Persistence System

be a schedulable object by allowing preemptivity and bounded pause time. Thus, a *Schedulable Persistence System* can be interleaved with real-time applications and both the timeliness and persistence of the real-time applications can be guaranteed.

In this paper, we depict an approach of schedulable persistence service in VM environment. The persistence service is taken by a separate task that runs concurrently with real-time application tasks. The service is divided into small portions (*persistence increment*, hereafter) to bound the pause time caused by the service. As illustrated in Figure 1, the scheduling model for the proposed approach is basically to allocate CPU cycles to consecutive persistence increments periodically. Hence, the whole persistence service becomes preemptible after each persistence increment and its execution can be treated as a periodic task invoked in a persistence cycle.

One of the issues raised while supporting preemptivity of persistence service is to guarantee the consistency of persisted data. We define that a consistent persisted data is a snapshot of all persistent data when the persistence service is triggered. With concurrent and preemptible persistence service, mutators (i.e. application tasks) are executed concurrently with the persistence service and may update the persistent data before the data becomes persisted completely. We address this issue by using write barrier in a cost effective manner.

The proposed approach of making persistence service schedulable along with real-time applications in VM environment can be viewed as a part of efforts to make VMs suitable for real-time embedded systems. However, even if we consider the emerging RTSJ's commercial products, *PERC* [1] and *Sun Real-Time Java System* [25], there is no existing VM that enables persistence of applications while considering the timeliness of real-time applications simultaneously. Beyond VMs, a few previous works [16, 8] aimed at concurrent checkpointing for general real-time systems. These approaches are to save a process memory content based on architectural support and/or by using coarse grained write barriers. Their limitations include non-portability of checkpoint data (process image) and/or the lack of semantics of preserved data for alternate recovery processing.

In the following, we give a brief background on object serialization and a discussion of related works in Section 2. The design approach for a schedulable persistence system is introduced in Section 3. In Section 4 and Section 5, we present the details of the prototyped schedulable persistence system and the experimental results. Finally, Section 6 draws a simple conclusion.

2. BACKGROUND AND RELATED WORKS

2.1 Checkpointing & Object Serialization in VMs

Checkpoint based roll-back recovery is a fault-tolerant technique to minimize the lost computation due to failures. A checkpoint can be taken during failure-free execution and a state of computation is saved. Upon a failure, the recovery operation uses the checkpoint to restore the computation state up to the moment that the checkpoint was taken. Then, the computation can proceed.

Checkpoint can be done in the level of VM to save the state of live objects which represent the computation state of applications. Also, as suggested in [5] and [23], the applications' execution state can be save as portable checkpoints of thread objects. The saved state of data and thread objects can be serialized by following VM's standard data format and then become readable by a VM in any architecture.

Object Serialization in Java and CLI-compatible languages [22, 18] is a significant functionality used for lightweight object persistence, and data marshaling/demarshaling in *RMI* (Remote Method Invocation) or .NET *Remoting* programming. Object serialization transforms the state of objects in memory into a sequence of bytes or other representations suitable for transmission to file systems or communication media. It can be a simple approach to support object persistence. According to object reachability, *Persistent Objects* or *Persistent Data* include not only a persistent root object, but also all reachable objects from the root object. During *serialization* all reachable objects from the root object must be traced to generate persisted data. Then, *deserialization* restores objects from the persisted data. However, the current approach for object serialization suffers several drawbacks [10]:

- The procedure for serialization or deserialization runs as a sequential operation of the invoking threads. Hence, the normal thread operation can only proceed after the serialization operation and may suffer a long pause (the serialization operation cannot suspend in the middle of the procedure).

- Object serialization does not consider consistency of persisted data. For example, if a thread serializes an object graph (all reachable objects from a persistent root object) while other threads mutate any objects in the graph, the consistency of the persisted data in the graph is not guaranteed.

Apart from the aforementioned drawbacks, the performance of the existing serialization operation in terms of time and space is also a concern. The object serialization in both JVM and CLI heavily uses *Reflection* mechanism of VM, which allows managed code to retrieve information of fields, methods, constructors for objects, classes etc. Reflection is basically an interpreted operation on object meta data and may incur a significant overhead and performance penalty.

2.2 Efficient Object Serialization

There were several attempts to improve the performance of Java Object Serialization in order to enhance *RMI* mechanisms. Haumacher et al. [20] suggested a serialization function in pure Java code. To remove the overhead in a wire protocol, it minimizes the size of metadata in the serialized data by omitting data fields' type information, name, etc. To avoid any costly invocation of reflection mechanism in a generic serialization operation, it supports only user-defined serialization, which requires programmers to specify methods to serialize/deserialize objects of each class. Breg et al. [6] implemented a serialization function in native code by using JNI. Their study shows that the overhead from Reflection and JNI is still high although the implementation in native codes improves the performance to a certain extent. In addition, a type-cache was employed in their approach to reduce the overhead caused by repetitive invocations of Reflection functions.

2.3 Orthogonal Persistent Object Systems

Making data persistent beyond the lifetime of applications may require extra programming effort to save and restore data using specific programming constructs, for example, DB languages to access DBMS. *Orthogonal Persistent System* [2] aims at a transparent programming such that no significant programming effort is required to handle persistent data in a database. In addition, it adopts *Lazy pointer swizzling* and *Incremental updates* techniques to reduce the latency due to restoring and saving the persistent objects. Lazy pointer swizzling is to update the loaded object's address (from persistent storage address to memory address) on demand. Furthermore, with incremental updates, only updates, instead of the whole data objects, are saved to a persistent storage since the latest persistence service. Through the techniques, the orthogonal persistent object system provides scalability on enterprise applications that handle massive data. However, its unpredictable latency still makes the system inappropriate for real-time applications.

2.4 Checkpointing for Real-Time Applications

There are few investigations on checkpointing memory content of real-time application processes in OS level. Li et al.[16] present a concurrent checkpointing algorithm in which mutators are allowed to interleave with a checkpointing thread. The algorithm incurs a reduced latency by checkpointing memory pages one at a time rather than checkpointing the whole memory space of mutators at once. The consistency of checkpointed data is addressed by employing a copy-on-write mechanism. The mechanism places write-protection on memory pages when checkpoint starts. When mutators try to update a memory page, the page is checkpointed before the update is applied. Although this approach can reduce the latency modestly, two limitations follow. Firstly, the granularity in a level of a memory page is too coarse to satisfy the timely requirements of real-time applications. Secondly, there is a significant initialization delay to enable MMU (Memory Management Unit) write protection on the whole memory space of the application.

Cunei et al. [8] propose a concurrent checkpoint mechanism by employing a mirror copy of memory blocks. The mirror copy works as follow: the checkpoint mechanism maintains an auxiliary memory block as a mirror copy of main memory. In order to have the mirror copy constantly updated, update operations of mutators apply on both the main memory and the mirror copy. When a checkpoint starts, the updates on a mirror copy are suspended, and then the data on the mirror copy is saved to a persistent storage. Write barriers can be used to allow the mirror copy updated in software, rather than depending on a specific hardware. To reduce the cost of write barriers, the mirror copy is mapped to separate physical memory pages while checkpoint is underway. Hence, write barrier operations do not need to be changed according to the presence of checkpoint. Employing a mirror copy can be a solution of checkpoint while concern the timeliness of applications. However, the cost having extra memory blocks may not be tolerable to resource constraint embedded systems, and the approach is also limited to checkpoint memory content in OS level.

There have been a lot of research work focusing in the scheduling issues for fault-tolerant real-time applications with time-redundancy schemes [15, 12, 11], and [21]. Because the time-redundancy schemes require extra time for fault detection and fault recovery, scheduling fault-tolerant real-time applications are resorted to finding the WCET (Worst Case Execution Time) of the applications with failure detection and recovery, conducting schedulability study, and establishing efficient scheduling algorithms in the presence of faults.

3. DESIGN APPROACHES

The design of the proposed persistence system is focused on achieving three goals: to enable concurrent and preemptible persistence service with adjustable pause time while ensuring consistency of persisted data, to make persistence service efficient for resource constrained systems, and to provide essential features necessary for fault recovery as well as data recovery. We will limit the scope of the persistence service to data persistence but not thread persistence for applications. However, we claim that the characteristics of real-time embedded applications, and inevitable initialization for external data during recovery make data persistence the foremost concern for failure recovery.

We design a schedulable persistence system by employing object serialization and extending the functions of VM environment. We use MONO, an open source development platform of .NET framework running on Linux platforms [28], as an example platform to illustrate the details of our design. The benefit of employing object serialization is portability because object serialization generates persisted data with VM-aware standard format. Thus, the persisted data can be used in both HW and time redundant fault tolerant architectures. Serialization and Deserialization are implemented in native codes as a subsystem of CLI. This decision is drawn to overcome the limitations of serialization provided as managed code including poor performance and the inadequacy to be extended for preemptible and concurrent serialization. Since we employ object serialization to achieve persistence, we will refer serialization and serialized data interchangeably with persistence service and persisted data, respectively.

3.1 Efficient Serialization Algorithms

The serialization operation consists of three steps, firstly to obtain the state of objects, secondly to transform the state into binary sequences, and then to write the serialized data to memory, or files (persistent storage) [14]. The factors affecting the performance of serialization are:

- Reflection allows managed code to retrieve information of persistent objects' fields. It is only way to obtain the state of objects in serialization implemented in managed code, but it is very costly.

- The serialized data consist of metadata as well as states of persistent objects. The metadata include type information, headers necessary to parse the states for deserialization. The excessive metadata increases IO overhead as well as the size of serialized data. The benchmark on object serialization implemented for .NET Compact Framework [6] indicates that reducing the size of metadata overhead helps improving the performance.

- The execution performance in managed code is much slower than native codes. To improve the performance, Breg et al. [6] implemented serialization in native codes using JNI. However, the overhead associated with JNI is not negligible.

An efficient serialization should avoid any reference to reflection mechanism and minimize the meta data information saved in persistence storage. To further optimize the performance, we can adopt a native code implementation of serialization service as internal functions of VM. To begin the design, persistence class and fields should be declared. Hence, programmers can apply their knowledge of the applications to select the class and field of data that must be made persistent. In C#, this can be done with additional attributes, such as *[SPersistent]* and *[SPersistentField]*. As shown in Listing 1 of a C# class declaration of a persistent class with two persistent fields. *Attribute* in C# is associated with managed code in a form of *Class Metadata*, but does not alter the managed code's semantics. When managed code is loaded, the associated class metadata is placed as in-memory data structure. In Java, similar to *Serializable* interface, a persistence interface can be defined to indicate a class whose instances may be persistent.

```
[SPersistent]
class Foo
{
    [SPersistentField]
    public int i;

    public double d;

    [SPersistentField]
    public Bar o;

}
```

Listing 1: Example declaration of a persistent class

To implement serialization/deserialization as internal functions in VMs, we will not be able to use reflection mechanism to retrieve underlying information of persistent classes/fields. Our approach is to maintain a *Persistent Class Map* internally for persistent classes. The map facilitates the introspection function of a reflection mechanism and contains persistent fields' type information and offset. For example, the map for *Foo* class in Listing 1 lists two entries for a field x and a field o. The map helps efficiently locating persistent fields from persistent objects during serialization and deserialization. A persistent class and its persistent fields are identified by accessing the persistence attribute information in class metadata. The map can be built when the class is loaded or when the class' persistent objects are serialized for the first time.

Serialized data includes not only the states of persistent objects but also object header, which describe class information, fields' types, fields' names, and other header information, etc. We design a *Serialization Protocol* to define the format of serialized data. The protocol minimizes the amount of serialized data by reducing the amount of the metadata to be saved with persisted data. This is done by, firstly, not carrying a verbose format of metadata to specify names and types of its fields in serialized data. Those information can be retrieved through persistent class map, which is also generated during deserialization. Secondly, persistent classes are distinguished with unique class identification (*Class ID*) and are maintained in a *Persistent Class Cache* with detail information such as assembly version etc. Serialized data of persistent objects carry only *Class ID* representing its class instead of verbose format of its class information. The cache is also serialized so that deserialization can generate the same type of persistent class cache. The serialized data is converted into binary stream, which is independent of the underlying hardware platforms.

To invoke serialization/deserialization operations, we employ a simple API of three static methods of class *SPC*. *AddRoot* registers a persistent root object and returns a persisted data ID. When the data needs to be persisted, the second method *SendPData* is called. For deserialization, *RecvPData* is invoked with the ID of a persistent root object which is used to locate the corresponding persisted data.

```
Foo oKey, oValue;
int nRootID1, nRootID2;

//Initialization Phase for cold start
if(IsWarmRestart()==false)
{
    oKey   = new Foo();
    oValue = new Foo();

    //Register persistent root objects
    nRootID1 = SPC.AddRoot(oKey);
    nRootID2 = SPC.AddRoot(oValue);
}
//Initialization Phase for warm restart
else
{
    oKey   = SPC.RecvPData(nRootID1);
    oValue = SPC.RecvPData(nRootID2);
}

//Periodic Operation Phase
while()
{
    //Main operations
    ...
    //Serialize all persistent object graphs
    SPC.SendPData();
}
```

Listing 2: An example program of using persistence service

Listing 2 is an example of a persistent application to support warm start using the API. In many real-time control applications, function blocks are invoked repetitively using sensor inputs and to control actuators. The applications are typically structured with *Initialization Phase* and then *Periodic Operation Phase*. In the initialization phase, resource required to execute the function blocks are initialized. Then, in the operation phase, function blocks are invoked periodically. Once applications' data is persisted at the end of each period, failure recovery can be done by restarting the applications at a new period with the latest persisted data.

3.2 Concurrent Persistence Mechanism

The existing serialization in JVM or .NET is an atomic procedure, i.e., the applications have to wait until the last object of persistent object graph gets serialized. This may result in a long pause on applications. To make persistence service schedulable for real-time embedded applications, we assign a separate task responsible for the service (hereafter, *SP task*) and make the task preemptible by dividing the service into small increments. As a consequence, real-time scheduling algorithms can be applied to dispatch urgent tasks once the SP task performs a bounded persistence service per increment and relinquishes CPU cycles to other tasks.

We define that a consistent persisted data is a snapshot of all persistent data when the persistence service is triggered. With concurrent and preemptible persistence service, the mutators may modify the persistent data before it is completely serialized. We address this issue by using a write barrier. The write barrier traps the update operations of mutators on persistent objects. If the persistent object yet gets serialized, the write barrier performs serialization of the object before the update is effective on the object.

Ideally, a write barrier should be only applied to the update operations on the persistent data when persistence service is in progress. However, unless the object graph starting from the root object is scanned, there is no straightforward approach that can identify whether an object of persistent class needs to be serialized. To avoid the application of write barrier to all objects, we may take either of the following two approaches: the first approach is to apply write barrier on methods that update persistent fields of a potential persistent object, and the second approach is to mark persistent objects in advance.

3.2.1 Annotating Methods that Update Persistent Objects

Under this scheme, the methods that update persistent objects are annotated (hereafter "Annotating SP") such that a write barrier becomes effective when the methods are invoked during a persistence service interval. Then, the write barrier performs two operations on the object encountered: a test operation for the serialization status of the object and a serialization operation if it is not yet serialized. The annotation can be done by the programmer or by compiler when a definition of persistent fields is detected. When it is done by the programmer, the approach relies on users' descriptions to effectively restrict the range of write barrier to likely update operations on persistent fields of persistent objects. Because not all fields in persistent classes are persistent, applying write barrier on methods that update persistent fields can narrow down the effective range. As shown in Listing 3, the attribute *[SPersistentUpdate]* is used to specify the annotated method *set*. This scheme can be also applied in JVM given that annotation is supported since JDK 1.5 as a result of JSR 175, "a metadata facility for the Java programming language" [24].

```
[SPersistent]
class Foo {
    [SPersistentField]
    public int x;

    [SPersistentField]
    public IxVertex PNodes;

    public int XP {
        get {
            return x;
        }

        [SPersistentUpdate]
        set {
            x = value;
        }
    }

    public void Bar() {
        ...
        XP=10;

        PNodes[i]=Baz;
        ...
    }
}
public class IxVertex {
    [SPersistentField]
    public Vertex[] nodes;

    public IxVertex(Vertex[] param1){
        nodes = param1;
    }
    public Vertex this [int index] {
        get {
         return nodes[index];
        }

        [SPersistentUpdate]
        set {
         nodes[index] = value;
        }
    }
}
```

Listing 3: An example persistent class with method annotations

However, if the methods specified with the *[SPersistentUpdate]* attribute conduct update operations on nonpersistent fields as well as persistent fields, the write barrier is unnecessarily applied to the update operations on nonpersistent fields. The solution is to use a wrapper method designated for updating a persistent field, and to make other program codes call the method to update the field. To realize that solution, we recommend to define *Property* for each persistent field, and access the field through its corresponding property. *Property* in C# is an interface to provide a direct access of a class field through two methods: *Get* method to return the value of the field, and *Set* method to write a value to the field. Thus, specifying *[SPersistentUpdate]* to the property's set method can confine write barrier to the persistent field's update. For example, in Listing 3, an integer type field x is a persistent field and XP is a property to the field x. The attribute *[SPersistentUpdate]* on a set method of the property XP allows to activate write barrier on the set method while a persistence service is in progress.

If a persistent field is array (actually a reference to an array object in CLI), the update operation on each element of the array should trigger the write barrier. However, the set method of the property corresponding to an array reference does not invoke a write barrier on access to the element of the array. A solution is to define an *Indexer* class for an array in C#. This approach allows the instances of a class or struct being indexed in the same way as arrays. Indexer consists of get/set methods similar to properties except that their accessors take the indices to the elements such that the elements are guarded. In Listing 3, for example, a class *IxVertex* is an Indexer class defined to provide get/set methods to an access to array's elements.

The limitation of the Annotating SP is that it may consider nonpersistent objects as persistent objects. For example, a nonpersistent object whose class happens to be a persistent class but it is not reachable from the persistent root objects can be still protected by the write barrier. Although it can cause unnecessary overhead, it does not violate the consistency or integrity of persisted data because the nonpersistent objects mistakenly serialized will be deserialized as dead objects that are unreachable via any live objects and are the subject of garbage collection. The another limitation of this scheme is that any updates to persistent fields must be done through properties or indexers although in C#, properties and indexers are recommended to provide encapsulation. However, we can also rely on compiler techniques to insert write barrier on updates to persistent fields or define properties or indexers as shown in Listing 3.

3.2.2 Marking Persistent Objects In Advance

To identify precisely the persistent objects, the persistence service can mark persistent objects in advance before serializing the objects. The approach (hereafter, "Marking SP") takes two phase: *Marking Phase* and *Serialization Phase*. In marking phase, SP task traces and marks all persistent objects in the object graph. In serialization phase, all the marked objects get serialized. Both phases are preemptible and are protected by a write barrier. The operations of the write barrier is different according to the phases. In marking phase, the write barrier tests the serialization status of the object encountered, and then in serialization phase, write barrier tests the mark status of the object, and serialize the object if it is marked and is not yet serialized. The write barrier in making phase does not check the mark status because marking on objects is underway. It may generate some false-serialized objects, but they may not be many because the marking phase usually takes a short amount of time comparing to the serialization phase.

This scheme can confine the write barrier on any update operations to persistent objects. However, the overhead is that all update operations in marking phase conducts test operation for the object encountered. That can be implemented with a fast path like inline codes. The study on cost of write barrier using a fast path [3] shows that the cost is tolerable. Unlike Annotating SP, this does not give programming burden to limit access to persistent fields through properties or indexers.

In summary, either Annotating SP or Marking SP can be used to distinguish persistent objects from nonpersistent objects and to efficiently narrow down the effective range that the guard operation by a write barrier is applied. The choice among two approaches might depend on the amount of checking operations and the annotations provided by programmers.

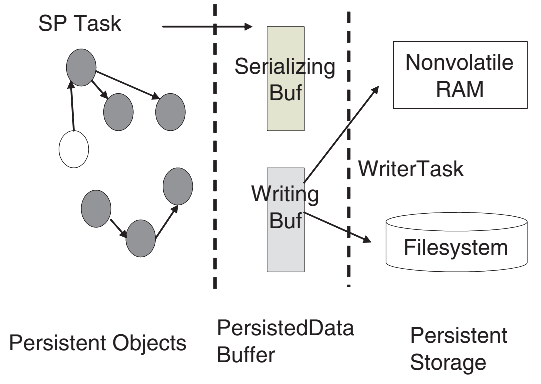

Figure 2: Structure of the proposed Schedulable Persistence System

3.3 Structure of Data Persistence

The architecture for schedulable persistence system is devised to aim at a general persistence system that can be applied regardless the types of persistent storage. The data persistence system consists of two tasks, *SP task*, and *Writer task* in Figure 2. SP task traces persistent objects, serializes the state of the objects, and then writes the persisted data into a buffer. On the other hand, Writer task moves the data from the buffer to a persistent storage. The system maintains two buffers: one buffer can be the *SerializingBuf* to where SP task writes persisted data; and the other buffer will be the *WritingBuf* read by Writer task to move persisted data to persistent storage. The roles of the buffers are switched back and forth such that the write and read operations are performed concurrently. Hence, a smooth data transfer of persisted data to storage devices can be accomplished.

4. IMPLEMENTATION

We implemented a prototype of schedulable persistence system in CLI's MONO platform version 1.1. The major effort in the implementation includes the construction of the SP task designated for persistence service and write barrier schemes. The SP task starts to serialize objects when *SendPData* API gets called as stated in Listing 2. As illustrated in Figure 1, persistence service is performed by conducting small increments of the service periodically. Runtime parameters are set with the targeted pause time of each persistence increment and the period of persistence increment.

During each persistence increment, the minimum work unit of serialization is a single object; that is, the pause time of a persistence increment can be as small as the duration taken to serialize a single persistent object. Persistent object graphs are traced through a breadth-first search, and any object types including strings and array of strings are treated as single persistent object to keep a minimum work unit of a single object. An internal queue is employed to buffer the persistent objects encountered when the object graph is traced. During each persistence increment, depending upon the target pause time, a number of objects from the queue are traced and serialized. To access the fields of a persistence object, persistent class map is used to quickly locate persistent fields within persistent objects. The map is organized as a linked list of all persistent fields specified with attribute *[SPersistentField]*. In our prototype implementation, the map is built while loading a persistent class in order to avoid any delay during serialization.

To serialize persistent objects, the state of objects as well as their reference relationship must be preserved. Each reference points to a corresponding object using an actual memory address (MID). In persistence storage, each MID must be converted to a unique logical address (LID). We maintain the pairs of MID and LID for all persisted objects in a hash table *MID2LID*. Whenever an object is encountered during the breadth-first search of persistent object graph, *MID2LID* is looked up such that references to shared objects can get converted by the same LID. In addition, MID2LID table keeps the status of each object indicating whether it has been serialized or not. The status is encoded by using MSB (most significant bit) of LID in the table to save memory space.

To invoke write barrier when a serialization gets started, we consider the instructions defined in Common Language Instruction (CIL). Among the 250 instructions, only *stfld* and *stelem.{i,i1,i2,i4,i8,r4,r8,ref}* are used to store a new value in a field of an object and in a vector element, respectively[1]. To protect any update operations through write barriers, MONO's JIT compiler is modified such that the two instructions are translated into CLI's internal functions. Each of these functions calls indirectly to a update function through a pointer which is altered according to the presence of concurrent persistence service. In other words, the update function with write barrier is invoked when concurrent persistence service is underway, otherwise the one with no write barrier is called. This approach is applied to both Annotating SP and Marking SP. However, the former modifies the implementation of *stfld* and *stelem* instructions when they are a part of the methods annotated with *[SPersistentUpdate]*. The translated code is not altered if the *stfld* and *stelem* instructions are not done by an annotated method. On the other hand, in the latter scheme, the translated codes for all *stfld* and *stelem* instructions are modified. Then, object mark is tested to trigger a write barrier.

The operation of de-serialization is completed in two phases. In *construction phase*, objects are re-constructed based on serialized data and a hash table with pairs of LID and MID is established. In the following *fix-up phase*, the LIDs are fixed up with corresponding memory address (MID) to recover the reference relationship between objects. Only the objects with reference member fields are enqueued into a fix-up cache in construction phase so that the fix-up operation is limited to the objects with reference fields.

Unlike serialization, in our current implementation, deserialization does not work in a concurrent mode; that is, when deserialization for recovery is requested due to an application's failure, the recovered application's execution can resume once deserialization over the entire serialized data completes. If the recovery, due to large amount of serialized data, places a long latency beyond the application's timely requirements, the timeliness of the application cannot be guaranteed even with persistence service. One of solutions over this problem might be to resume the execution of a recovered application with a partial recovery once the partial recovery initializes the application enough to resume. To allow a partial recovery, we can apply on-demand object deserialization and lazy pointer swizzling [2]: on-demand object deserialization deserializes objects when they are referred, and lazy pointer swizzling converts LID to a correspondent MID when the object represented with the LID is actually accessed. Other technique to be considered is *In-Place Object Deserialization* which is suggested as a method of serialization/deserialization for Java RMI (Remote Method Invocation) in [7]. The in-place object deserialization conducts deserialization without allocation and copying of objects by reusing a buffer allotted to serialized data for deserialized object. We leave considering a partial recovery as a future work.

[1]CIL defines additional store instructions such as *stsfld*, *stind.⟨type⟩*, *starg.⟨length⟩*, and *stloc*. However, these store instruction do not require a write barrier because their store operation is not conducted on instance objects.

5. EXPERIMENTS

Experiments in the prototype persistence system are conducted to examine the performance of the proposed design approaches. The experiment is done in a PC workstation with 1.5GHz Pentium IV processor and 256MB memory. To have a high resolution timer and preemptive kernel, TimeSys' Linux/Real-Time(v4.1.147)[26] is used.

The experiments basically follow a comparative study among the performance measures of serialization and deserialization operations. MONO's serialization library and the proposed schedulable persistence (SP) service with the Annotating SP and Marking SP write barrier schemes are tested. We collect both the maximal and average measures after running the experiments 40 times. If a specific measure is common to Annotating SP and Marking SP schemes, we simply denote it as a *SP* measure for the proposed schedulable persistence system. Whenever SP service and applications run concurrently in Linux threads, the SP thread is assigned with a higher priority than application threads.

5.1 Performance of Serialization and Deserialization

The performance of serialization and deserialization operations using MONO's serialization library, Annotating SP, and Marking SP, are collected by running in the stop-the-world mode. Since the time needed to complete the operations depends upon the object types as well as the number of persisted objects, the serializations of a red-black tree of composite objects (consisting of multiple primitive fields), and an array of a primitive type (integer) are benchmarked. The average response times of serialization and deserialization for the red-black tree and the array of a primitive type are shown in Figure 3, Figure 4, Figure 5, and Figure 6 respectively.

The results in the figures clearly indicate that, in terms of the response time of the serialization and deserialization operations, the proposed schedulable persistence system (with either Annotating SP and Marking SP) outperforms the MONO's serialization library significantly. The performance improvement is derived from two main factors. First, the SP serialization is implemented as a subsystem of CLI by defining native functions which is much more efficient than the serialization library of managed code. The second factor is the use of persistence class map which allows us to avoid costly reflection mechanism.

Given that the experiments were run in the stop-of-the-world mode, the effect of write barrier schemes is not presented in the results. However, comparing the Annotating

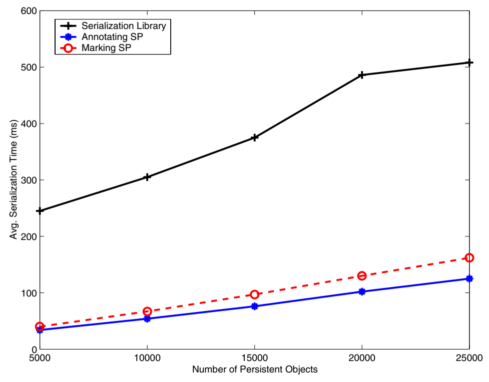

Figure 3: Serialization of Tree

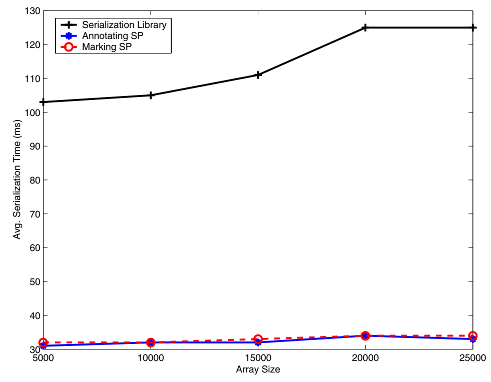

Figure 5: Serialization of Array of Primitive type

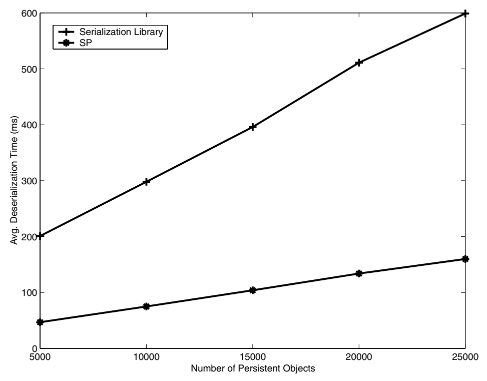

Figure 4: Deserialization of Tree

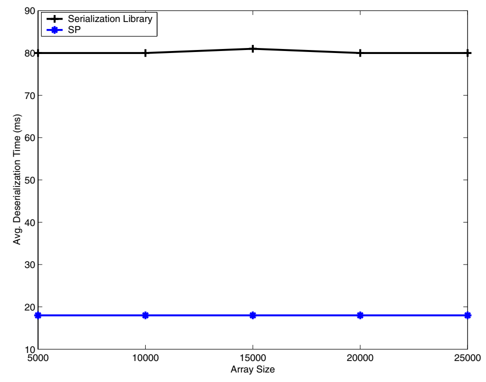

Figure 6: Deserialization of Array of Primitive type

SP and Marking SP schemes, we found that Marking SP has a 30% more overhead than Annotating SP especially for applications with many reference objects. This is mainly due to the difference of the invocation numbers of internal functions translated from *stfld*, and *stelem*. More detail is given in 5.2. Furthermore, the extra marking phase before conducting serialization in Marking SP scheme also contributes to that. On the other hand, in this experiment, the size of the serialized binary data in SP is not much different from that of serialization library because in each of the benchmarked applications there is only one persistent class and the metadata preserved is only a small portion of serialized data.

5.2 Controlling the Pause Time of Persistence Increments

The aforementioned experiments illustrate the execution time of the schedulable persistence system. To support real-time applications, we need to examine whether the proposed schedulable persistence system has a behavior following the scheduling model of Figure 1. The experiments conducted are to apply the two write barrier schemes in two different execution modes, stop-the-world mode (STW) and preemptible modes. For the preemptible mode, the targeted pause time of each persistence increment and the period of persistence increment are set to $500\mu s$ and 3ms, respectively.[2]

The benchmarked application mainly constructs a directed weight graph, which contains 1326 persistent objects. The total amount of serialized data is about 43KBytes. The application works as follows. First, the application constructs a directed weight graph of nodes. After the construction is complete, it triggers a persistence service. Then, the application starts to make changes on the connections of nodes and then delete all the nodes of the graph one by one. In other words, in preemptible mode, the SP task serializing the graph runs concurrently with a mutator that updates the persistent objects.

The experiment results are shown in Table 1. It indicates that the schedulable persistence system by using both write barrier schemes is able to control the pause time of each persistence increment to meet the targeted bound of $500\mu s$. The average execution time of serialization in preemptible mode is slightly higher that that in the stop-the-world mode. This difference is somehow expected given additional overhead caused by context switches and controlling the pause time of each increment.

The total execution times of the application of the Marking SP in both stop-the-world mode and preemptible mode are higher in than that of the Annotating SP. In both write barrier schemes, the CIL update instructions, *stfld* and *stelem*, are translated into internal functions during JIT (runtime compilation). Then, depending upon whether a serialization is in progress or not, the internal functions call actual update functions with either a write barrier or without a

[2]This experiment focuses on measuring the flexibility of persistence increment's granularity and the overhead due to persistence service's preemptivity so that these numbers, $500\mu s$ and 3ms, are arbitrarily chosen in that sense.

	Annotating SP		Marking SP		Marking SP(v2)	
Properties	STW	500μs	STW	500μs	STW	500μs
Max. pause time per increment(μs)	5171	453	6162	478	6344	478
Avg. pause time per increment(μs)	5056	401	6084	388	6257	398
Avg. persistence cycle (ms)	NA	39	NA	48	NA	48
Max. serialization time(μs)	5171	6056	6162	6758	6344	7367
Avg. serialization time(μs)	5056	5632	6084	6612	6257	6859
Max. application exec. time(ms)	300	301	374	390	302	309
Avg. application exec. time(ms)	279	282	351	366	272	284
Avg. No. of Increments	1	14	1	17	1	18
Avg. No. of WB invocations	0	1284	0	185762	0	16284
Avg. No. of WB Serializations	0	24	0	41	0	50

Table 1: Pause time and overhead of schedulable persistence system

write barrier, accordingly. The main difference between the two schemes is the number of CIL update instructions translated into the internal functions. Annotating SP limits the instructions to those in the methods annotated with *[SPersistentUpdate]* whereas all *stfld* and *stelem* instructions, including the ones in standard class libraries, are converted into the internal functions in Marking SP. The overhead incurred in the execution of the internal functions results in a longer execution time of the application in Marking SP. Furthermore, that Marking SP has a marking phase in advance which also affects the execution time for serialization.

To find out a compromised solution for the added cost of internal functions for write barrier, we experiment a modified Marking SP scheme in which only the *stfld* and *stelem* instructions in the methods of persistence classes are translated into the internal functions. The performance is shown in the Marking SP(v2) column of Table 5.2. This modification reduces the number of invocations of the internal functions significantly and leads to a total execution time of the applications similar to that of Annotating SP. It indicates that even though the execution time in Marking SP is high, it can be adjusted by defining the scope of the update operations to persistent fields. If persistent fields are updated through the methods of persistent class, a write barrier on the methods of persistent class is able to provide a sufficient guard without an excessive overhead.

Besides execution times, the numbers of increments to complete a persistence cycle and the length of a persistence cycle are presented in Table 1. In addition, the numbers of invocations of write barrier operations, and the numbers of objects serialized during write barrier operations are given. The distinct characteristics revealed on the two measures is discussed in the following subsection 5.3.

5.3 Objects Serialized by Annotating SP and Marking SP

In this experiments, we look into how the two schemes, Annotating SP and Marking SP, efficiently distinguish persistent objects from nonpersistent objects. To have mixed persistent and nonpersistent objects in heap memory, the benchmark application consists of three threads and each of which firstly constructs a directed graph of nodes and then deletes all the nodes one by one. Each graph is different in the perspective of persistence. The first one makes all objects in the graph persistent by making its root node a

	STW	Annotating SP	Marking SP
Avg.No. of persistent objs.	1326	1326	1326
Avg.No. of serialized objs.	1326	1677	1326
Avg.No. of WB invocations	0	2643	240912
Avg.No. of WB serialization	0	339	13

Table 2: Checking and serialization operations of the two write barrier schemes

persistent root object before deleting any nodes. The second one has no persistent objects but using the same persistent class as the first thread. In the third thread, a different class is used to instantiate the objects in the graph and no persistence service is invoked. Among three threads, only the first one triggers a persistent service after the construction of the graph. The application has 1326 persistent objects out of a total of 3978 objects.

The results of the experiment is shown in Table 2. The data indicates that Marking SP is able to precisely pinpoint persistent objects. On the other hand, extra 351 nonpersistent objects accessed by the 2nd thread are serialized in Annotating SP. These extra objects happen to be the same class of the persistent objects and are updated by the annotated methods when the 1st thread requests a persistence service. However, because Annotating SP limits the write barrier to the annotated methods, the number of invocation to the write barrier operations is much less than that in Marking SP. Note that, in Annotating SP, the average number of objects serialized by the write barrier operations, i.e. 339, is less than the average number of the extra nonpersistent objects serialized by the scheme. This should not happen if nonpersistent objects can only be serialized (mistakenly) by the write barrier operations. In fact, an object serialized by the write barrier operations may have references to other objects. These referenced objects must also be serialized and are placed in the internal queue such that the SP task can trace the object graph through the breadth-first search. Thus, extra nonpersistent objects may be preserved by the SP task during the subsequent persistence increments.

6. CONCLUSION & FUTURE WORKS

In this paper, we aim for a schedulable persistence system to support object persistence for real-time applications in VMs. The system is designed with a concurrent persistence service task to avoid any long pause delay caused by sequential operation of serialization. To guarantee the consistency of persistent objects due to the concurrent operations of persistence service task and application tasks, write barrier schemes are devised. A prototype system, involving the modification of JIT and rewriting the serialization operation, is constructed in CLI's open source platform, MONO. The experiments show a significant performance gain resulted from the native function implementation for serialization operation and a replacement of reflection mechanism for class introspection. In particular, we are able to demonstrate the bounded pause time for each persistence increment in the prototype system and the modest overhead incurred in write barrier operations.

Based on the insight gained from this study, our future work is to establish a cost model for the persistence service. The model is to estimate the worst case execution time of persistence service based on number of persistence objects, the class information of the objects, and their reference dependency. The model can then be used to schedule persistence cycle and increments, as well as to design a suitable scheduling algorithm for application tasks subject to recovery constraints.

7. ADDITIONAL AUTHORS

Additional authors: Elliott Rachlin (Honeywell International Inc., email: `elliott.rachlin@honeywell.com`).

8. REFERENCES

[1] Aonix North America, Inc. PERC, 2006. http://www.aonix.com/perc.html.

[2] M. Atkinson and M. Jordan. A review of the rationale and architectures of PJama: a durable, flexible, evolvable and scalable orthogonally persistent programming platform. Technical Report TR-2000-90, Sun Microsystems Laboratories and Dept. Computing Science, Univ. Glasgow, UK, 2000.

[3] S. M. Blackburn and A. L. Hosking. Barriers: friend or foe? In *ISMM*, pages 143–151, 2004.

[4] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[5] S. Bouchenak, D. Hagimont, S. Krakowiak, N. D. Palma, and F. Boyer. Experiences implementing efficient java thread serialization, mobility and persistence. *Softw., Pract. Exper.*, 34(4):355–393, 2004.

[6] F. Breg and C. D. Polychronopoulos. Java virtual machine support for object serialization. In *Java Grande*, pages 173–180, 2001.

[7] C.-C. Chang. *Safe and Efficient Cluster Communication with Explicit Memory Management*. PhD thesis, Cornell University, 1999.

[8] A. Cunei and J. Vitek. A new approach to real-time checkpointing. In *Proceedings of the 2nd ACM Virtual Machine and Execution Environments Conference (VEE 2006), June 14-16, 2006 Ottawa, Canada*. ACM, 2006. (to appear).

[9] ECMA. Ecma-335 common language infrastructure, 2002.

[10] H. Evans. Why Object Serialization is Inappropriate for Providing Persistence in Java. Technical report, Department of Computing Science, University of Glasgow, Glasgow, 2000.

[11] S. Ghosh, R. G. Melhem, D. Mosse, and J. S. Sarma. Fault-tolerant rate-monotonic scheduling. *Real-Time Systems*, 15(2):149–181, 1998.

[12] S. Ghosh, R. Mellhem, and D. Mosse. Enhancing real-time schedules to tolerate transient faults. In *IEEE Real-Time Systems Symposium*, pages 120–129, 1995.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[14] M. Hericko, M. B. Juric, I. Rozman, and A. Zivkovic. Object serialization analysis and comparison in Java and .NET. *ACM SIGPLAN Notices*, 38(8):291–312, 2003.

[15] H. Lee, H. Shin, and S.-L. Min. Worst case timing requirement of real-time tasks with time redundancy. *rtcsa*, 00:410, 1999.

[16] K. Li, J. F. Naughton, and J. S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2nd PPOPP'90), SIGPLAN Notices*, pages 79–88, Mar. 1990.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[18] Microsoft Corp. Microsoft .NET framework binaryformatter serialization format, 2002.

[19] M. Monga and A. Scotto. A generic serializer for mobile devices. In *Proceedings of the 20th annual ACM symposium on applied computing*, Santa Fe, New Mexico, USA, 2005.

[20] M. Philippsen and B. Haumacher. More efficient object serialization. In *IPPS/SPDP Workshops*, pages 718–732, 1999.

[21] S. Punnekkat, A. Burns, and R. Davis. Analysis of checkpointing for real-time systems. *Real-Time Systems*, 20(1):83–102, 2001.

[22] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat. Pickling state in the Java(TM) system. *USENIX, Computing Systems*, 9(4):291–312, 1996.

[23] T. Suezawa. Persistent execution state of a java virtual machine. In *Java Grande*, pages 160–167, 2000.

[24] Sun Microsystems, Inc. JSR175 : a metadata facility for the java programming language, 2005. http://www.jcp.org/en/jsr/detail?id=175.

[25] Sun Microsystems Inc. Sun Real-Time Java System, 2005. http://java.sun.com/j2se/realtime.

[26] TimeSys Corporation. Timesys linux/real-time user's guide, version 2.0, 2004.

[27] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 1999.

[28] Ximian. MONO. http://www.go-mono.com.