Self-Replicating Objects for Multicore Platforms

Krzysztof Ostrowski, Chuck Sakoda, and Ken Birman

Cornell University, Ithaca, NY 14853, USA
{krzys|cms235|ken}@cs.cornell.edu

Abstract. The paper introduces *Self-Replicating Objects* (SROs), a new concurrent programming abstraction. An SRO is implemented and used much like an ordinary .NET object and can expose arbitrary user-defined APIs, but it is aggressive about automatically exploiting multicore CPUs. It does so by spontaneously and transparently partitioning its state into a set of replicas that handle method calls in parallel and automatically merging replicas before processing calls that cannot execute in the replicated state. Developers need not be concerned about protecting access to shared data; each replica is a monitor and has its own state. The runtime ensures proper synchronization, scheduling, decides when to split/merge, and can transparently migrate replicas to other processes to decrease contention. Compared to threads/locks or toolkits such as .NET Parallel Extensions, SROs offer a simpler, more versatile programming model while delivering comparable, and in some cases even higher performance.

1 Introduction

1.1 Motivation

Modern PCs are becoming massively parallel: 4-core PCs are widespread, 64-core platforms will soon become a reality [1], and the trend is likely to continue, as it is driven by compelling performance and energy considerations [2]. Unfortunately, industry benchmarks show that moving from 2 to 4 cores, whether in PC games or office applications, brings little benefit [3–5]: such systems rarely gain much speedup beyond that which can be achieved by running applications side by side. This could mean that many applications are non-parallelizable; our experience suggests otherwise, and several researchers have offered plausible explanations.

First, it is widely believed that existing concurrency abstractions are inadequate: explicitly using threads/locks is difficult; using lock-free primitives is even harder, and composing correct modules with locks can lead to deadlocks [2, 6–8]; multi-threaded programs are hard to debug [8]. Software Transactional Memory (STM) offers clean semantics, but it can be hard to exploit in applications that trigger external actions hard to "rollback" on abort [9], such as web service calls.

Second, many developers fail to use parallelism because hardware is already fast enough for their needs, and they lack incentives to optimize resource usage. Decades of increase in CPU speeds have shifted the focus from careful performance profiling towards increasingly sophisticated functionality, even at sharply higher cost [2, 10]. This is further complicated by the fact that modern development environments shield developers from so many decisions that it is difficult to understand the performance consequences of one's architectural choices [10].

These and other studies suggest that it may be unrealistic to expect developers to explicitly and deliberately express concurrency in their programs through programming language mechanisms. Instead, it may be preferable to approach concurrency in a manner similar to the way we treat garbage collection: as implicit, always present by default, controlled by the runtime, and only sporadically manipulated by the developers, possibly in a declarative fashion via annotations.

Besides abstracting away complexity, letting the runtime control concurrency is essential in avoiding the consequences of the Amdahl's law [11]. To fully exploit the potential of multi-core platforms, one needs to ensure that non-parallelizable computation represents a small fraction of the system. This is hard to achieve if concurrency has to be explicitly programmed; particularly if the developers have to manually create tasks, combine them into complex workflows, and coordinate access to shared data. If even a fraction of developers neglect to express concurrency and 10% of code is not parallelizable, the system may be limited to only a 10-fold speedup [11]. Thus, it is essential that concurrency be implicit.

In the last few years, concurrency techniques inspired by functional programming have made their way into the mainstream OO languages [12–17] in the form of *data parallelism*; this includes constructs such as parallel loops, aggregations, and workflows. In purely functional programming, concurrency is implicit [18] in the sense that computations do not share state, and can be scheduled in parallel.

Modern software technologies, however, and widely used standards, including web services and Ajax, are based primarily on the object-oriented, not functional paradigm, and notions such as mutable *state* and externally visible *behavior* with side effects are fundamental to it. Consequently, there is a limit to how much we can achieve by explicitly embedding functional abstractions in an OO language.

To a degree, parallelism is implicit in the Actor model [19], e.g., in Smalltalk [20]. Constructs such as *asynchronous methods* and *futures* [21] allow for decoupling of the caller from the callee; the two may run in parallel. Popularized by Erlang [22], this capability has been incorporated into Java [23, 24] and .NET [25–30], and we expect it to be rapidly adopted by mainstream developers. The model is attractive, in that it enables concurrency without the need for explicit synchronization; in most variants of the model, each object executes at most one method at a time, and some implementations can even enforce memory isolation between components [26], making it impossible for distinct objects to share data.

Actor-level concurrency, however, brings a new set of limitations: it enables distinct objects to run concurrently, but individual objects are sequential, and a single Actor can become a non-parallelizable bottleneck that drags performance down if it has to handle a longer sequence of method calls in a serialized fashion.

The above discussion suggests that concurrency should be implicit in the OO languages at a deeper level than simply decoupling objects from one-another: we need a way to model individual objects as implicitly concurrent. Thus, it should be possible for multiple method calls to be executed simultaneously, yet in a way that retains the clean and intuitive sequential execution semantics that makes the Actor model so attractive to developers; we need to prevent data sharing without resorting to synchronization primitives. The functional approach addressed this by eliminating the state altogether, but as mentioned earlier, state is an essential part of the OO paradigm. A less radical approach is needed: to only require that distinct concurrent activities are accessing disjoint portions of the object's state. One way to achieve this is to partition the state and allow multiple concurrently executing threads to independently operate on their coresponding partitions, in a manner similar to how the input data is partitioned by parallel loops, MapReduce [31], and other mechanisms inspired by functional programming. Indeed, if state and behavior play the same fundamental role in OO as functions do in functional programming, then partitioning state is the perfect analogue of data parallelism, and if splitting functional computation across all the available cores is automated by the runtime environment, so could be the partitioning of the object's state. This observation motivates the methodology described in the following section.

1.2 Our Approach

This paper introduces *Self-Replicating Objects* (SROs), a new concurrency abstraction that builds upon and extends the Actor model style of concurrency. An SRO resembles an ordinary object, in that it can implement arbitrary interfaces and expose user-defined methods, but unlike an ordinary object, each SRO can have multiple replicas of its state. The replicas are symmetric: each has all member fields defined by the SRO. Distinct replicas can be updated by different threads without synchronization; the values stored in their fields are unrelated.

If the SRO consists of a single replica, we say that it is in the *collapsed* state; in this case, the SRO is functionally equivalent to an ordinary Actor. If the SRO consists of multiple replicas, we call it *replicated*. In the course of its lifetime, an SRO can arbitrarily often transition between these two states; the number of its replicas can be adjusted by the runtime environment depending on factors such as the number and utilization of the available cores or the application workload. It is not possible for the programmer to refer to individual replicas or to explicitly manipulate their number; all code written by the programmer is always executed in the context of a single replica (except for two special cases discussed further).

The methods of an SRO are classified as *ordinary* and *scalable*. The former can run only in the collapsed state; if one is invoked on a replicated SRO, the runtime postpones its execution until it has an opportunity to collapse the replicas. Scalable calls can run in either state; during periods when they predominate, the SRO can be replicated to introduce parallelism. The runtime will automatically partition an SRO if many such calls follow in short succession.

SRO methods are non-blocking and asynchronous: all calls complete immediately; their execution is decoupled from invocation. The runtime transparently intercepts each call, encapsulates it as an asynchronous event, and schedules for execution on one of the available cores. The required scheduling, synchronization, and replication logic are all dynamically generated at runtime, and the required wrappers are seamlessly injected at cross-component boundaries. If desired, the

SRO could return results via asynchronous callbacks, or as *futures* (futures are compatible with SRO, but not implemented in our prototype because .NET lacks support for uniform proxies; one could implement these much as in Java [32]).

It is important to notice that in our approach, each call executes against only one replica of the SRO's state. Scalable methods can be directed by the runtime to any of the available replicas, e.g., to load-balance, based on affinity, or to leverage other considerations beyond the programmer's control. Indeed, in our model it is not possible for the programmer to invoke a specific replica. Request ordering is normally preserved, as discussed below, but in many situations a series of scalable calls submitted in succession will be executed in parallel. Moreover, the number of replicas is dynamically adjusted to control the level of concurrency.

Much as in the Actor model, each replica acts as a monitor and handles a single method call at a time; thus, each call exclusively owns the replica on which it runs, and since replicas are disjoint (no state is shared), the programmers need not use any synchronization primitives in the method bodies. Thus, in terms of ease of use and transparency, our abstraction is comparable to Actors, while potentially enabling a much more efficient use of parallel hardware. Developers never explicitly control concurrency, and although they do have to be aware of the presence of other replicas, they do not have to worry about race conditions.

Sequences of ordinary method calls issued by the same thread are executed in the order in which they were submitted, and the relative order is also preserved between pairs of ordinary and scalable method calls. Sequences of scalable calls submitted together can be arbitrarily reordered, however; in this case, ordering is undefined because the calls can be dispatched to different replicas. Ordinary calls to an SRO made by different threads are guaranteed to be causally ordered.

As noted earlier, during the lifetime of an SRO, the runtime spawns replicas as needed to execute sequences of scalable method calls in parallel, and merges replicas prior to executing ordinary method calls; during an execution, SRO goes through many such replicate/collapse cycles. The programmer cannot control when this happens, but does provide a custom per-object replicate/collapse logic. Each SRO defines a pair of special methods $\mathsf{export}(r')$ and $\mathsf{import}(r')$, where r' is a reference to a replica. After creating a blank new replica r', at first uninitialized and with empty state, the runtime invokes $\mathsf{export}(r')$ in the context of an existing replica r, to allow r to initialize r' and possibly transfer some of its state to r'. The export method is guaranteed to always run atomically with respect to other activities that might be scheduled on either of the replicas involved in the state transfer; hence, even though the method can access two independent portions of the SRO's state, the programmer need not worry about race conditions.

Likewise, when the runtime decides to demote an existing replica r', it waits for all ongoing calls to r' to complete, executes $\mathsf{import}(r')$ in the context of some existing replica r, and destroys r'. The import call allows r to transfer the state from r' and combine it with its own. By convention, import leaves r' in a blank, empty, but usable state, to allow the runtime to immediately reuse it if needed; this also allows the runtime to perform incremental state merging in parallel with scalable calls, by periodically calling import against actively working replicas.

SROs support implicit parallelism in two ways. First, to other components, their replicated nature is invisible; they behave like Actors, with asynchronous invocation semantics. SROs are fully backwards compatible with the Actor model, and can be seamlessly incorporated as a feature into any Actor-based platform. For the purpose of evaluation, we implemented SRO as a feature within our *Live Distributed Objects* (LDO) platform [33, 34], but our approach does not depend on any specifics of LDO, and we believe our findings to be universally applicable.

Second, while a developer implementing an SRO has to define export/import, she never has to spawns threads/tasks or use synchronization primitives; she can write all of her code as if it were always going to be executed by a single thread.

Example 1. Consider a priority queue with a simple interface that contains two methods: enqueue(t, w) schedules a work item w to be processed at time t, and dequeue(t) returns the list of work items scheduled to execute no later than t. If the application accesses the priority queue in a very bursty manner, scheduling and retrieving work in batches, it might be possible to parallelize it by modeling it as an SRO. In this case, each replica of the SRO represents a separate priority queue, with a disjoint set of scheduled work items. The enqueue method is marked as scalable: each work item is scheduled on one of the replicas, arbitrarily chosen by the runtime. The dequeue method is ordinary; it cannot run until the object collapses. The export method is empty; it initializes a replica without transferring any state. The import method combines two replicas, leaving one of them empty. For example, if we implement the priority queue as a splay tree (as in the example discussed in Section 3.3), the import method performs a merge on two splay trees.

In this example, modeling the queue as an SRO allows multiple enqueue calls to execute in parallel. This is desirable, since in most implementations, including the splay tree, enqueue is an $\mathcal{O}(\log n)$ operation, and can be costly if the queue is large. In contrast, accessing elements of a splay tree in order is an $\mathcal{O}(1)$ operation; hence, it is possible to merge two splay trees efficiently. Our experiments confirm that wrapping a large, heavily used queue as an SRO can make it up to 2x faster.

1.3 Related Work

While we believe SRO to be novel as a concurrent programming abstraction and an extension of the Actor model, the idea of replicating a component is not new; scalable and fault-tolerant network services have traditionally been implemented as clusters of servers that maintain sessions with and respond to requests from disjoint sets of clients [35, 36]. In some systems, service replicas used distributed protocols such as state machine replication and reliable multicast to coordinate changes to their state, but since these protocols tend to be expensive, it is more common for servers to remain only loosely synchronized, instead relying on background reconciliation mechanism to restore consistency. SROs use a similar idea: rather than require all methods to run against shared state, we allow the replicas to diverge, and rely on the import method to combine the results of their work.

The behavior of an SRO is reminiscent of MapReduce [31, 37, 38]. Scalable calls handled in parallel by replicas can be viewed as analogous to *map* followed by *reduce* that combines the results of the call with the state of a replica. Import is analogous to *reduce*, whereas export resembles the MapReduce deployment step. As the SRO replicates and collapses, it goes through a series of MapReduce-like stages: its state is repeatedly partitioned, its portions are processed in parallel, and then merged together. Most of the classic MapReduce workloads are easily expressed as SROs; we discuss some of them in Section 3.4.

Compared to MapReduce, SRO is more general. First, MapReduce is inspired by functional programming, and most intuitive when used in a functional style, to represent one-off computations that run to completion. It is less obvious how to use it in an elegant way to model objects such as the priority queue in Example 1, or agents that continuously respond to requests while maintaining internal state accumulated in the course of computation. Second, MapReduce models data as sets of (*key,value*) pairs [31]. SRO does not use keys; replica states and arguments of scalable calls can assume any form. Third, in an SRO that has multiple scalable methods, different types of reductions interact with one-another as they execute, by altering replica states. Finally, SROs are backwards compatible with the Actor model and can be easily incorporated into legacy code; transforming an Actor into an SRO requires only adding import and export, and annotating some methods as scalable; all existing code remains unchanged.

Some of these points also apply to other abstractions inspired by functional programming or data parallelism. This includes parallel loops and reduces, work-flows, and graphs of interconnected tasks; some of the most popular examples include Intel's Thread Building Blocks (TBB) [14], Microsoft's Parallel LINQ (PLINQ) [15] and Task Parallel Library (TPL) [16] (also known as Parallel Extensions and integrated into .NET 4.0), and others [12, 13, 17, 39]. As mentioned earlier, using these mechanisms involves a conscious and deliberate effort on behalf of the programmer, and legacy code would have to be significantly modified to be parallelizable; this conflicts with the implicit parallelism approach we postulated earlier. Also, while the abstractions are certainly extremely useful, they are either low-level, or they belong to a different paradigm. We believe that OOP needs its own approach to concurrency, and SRO is a step in this direction.

Our proposal builds upon a rich body of work on the Actor model [19] and its implementations: Erlang [22], Scala [23, 24], Timber [40], Salsa [41], and others. Much prior work focused on the equivalence of threads and events [42] as a way of implementing Actors; particularly on implementing asynchronous methods that contain blocking calls and infinite loops without allocating a thread per Actor, e.g., by using tail recursion [23–25], or by automatically transforming blocking code to continuation-passing style by the compiler [28]. In most implementations of the model, Actors assume control via a blocking *receive* or a similar call [22–25, 27–30, 43, 44]; much work focused on features such as multi-way joins that allow multiple events to be corelated and received atomically in a rendezvous-style. Our contribution is complementary to all the above work. Support for loops and blocking calls and features such as multi-join receive could be easily incorporated into SRO using any of the published techniques. Likewise, automatic replication could be added without much difficulty to most of the aforementioned platforms.

A number of lightweight thread architectures have been proposed to support very large numbers of Actors, e.g., Kilim [45], Actra [46], and Capriccio [47]. Our prototype uses a thread pool and assumes that methods do not contain blocking code. Some have criticized event-driven programming as unintuitive [48, 49, 23]; others argued the opposite [6, 7], and the widespread adoption of standards such as Ajax and the turn towards asynchronous APIs [50] strongly support this point of view. Some of the prior work focused on ways to combine the models [23, 51].

Many implementations of the Actor model use a custom work-stealing scheduler [52, 53]. In principle, SRO is compatible with any scheduling infrastructure, although some of the functionality provided by our prototype, such as replicating SROs across processes and machines, are difficult to combine with work-stealing.

The idea of an implicitly parallel computation has also been explored, e.g., in Fortress [54], but implicit parallelism in earlier work applies primarily to control structures such as loops, whereas our work targets fundamental OO abstractions. Other work focused on mechanisms such as speculative parallel method calls [55].

State replication as a way to introduce concurrency has been explored in [56]; their system performs static analysis and transforms sequential code by injecting threads and locks. SRO uses dynamic reflection, and injects wrappers with lock-free event scheduling code. We focus on different types of optimizations, such as remote replication, incremental imports, or flow control, and different workloads. Our technique is complementary to, and could be used in combination with [56].

We believe SROs to be the first to model Actors, mapreduce, and many forms of task- and data-level parallelism as special cases of a single unified programming abstraction. This, the ease of use, the backwards compatibility with Actors, and the flexibility left to the runtime in deciding when to spawn or merge replicas and where to direct its method calls, make SRO an attractive alternative as a basic building block for massively parallel applications.

1.4 Contributions

This paper makes the following contributions:

1. It introduces SRO, a new concurrent programming abstraction that extends Actor-style parallelism with the ability to parallelize individual Actors, while being fully backwards compatible and easy to integrate into legacy systems.
2. It offers an approach to implicit concurrency that more naturally fits with the OO paradigm than those inspired by data flow functional and programming.
3. It combines elements of multiple paradigms (Actors, data parallelism, MapReduce) within a single unified abstraction and offers a versatile programming model that enables forms of concurrency typically achieved with threads and locks. It discusses two broad classes of workloads that benefit from SRO and are hard to model using the existing tools: stateful components that perform continuous, heavy batch-style processing, and producer-consumer scenarios.
4. It evaluates a working system (http://liveobjects.codeplex.com/); it shows that SRO matches cutting edge technologies such as PLINQ on MapReduce workloads, and in some scenarios overperforms it thanks to techniques such as remote replication. It shows substantial gains on workloads native to SRO.

2 Implementation

2.1 Live Distributed Objects

This section presents the architecture of the LDO platform as a point of reference; the implementation of SRO as an extension to LDO is discussed in Section 2.2. Our priority queue SRO can be defined as a component in LDO as shown below (the code is simplified for readability; the actual namespaces and syntax differ).

```
01: [Serializable] [ComponentClass("6F0DB8E4D16443C28A71D39CB04A763E")]
02: class PQueue : IPQueue, IReplicated<PQueue>, ISerializable {
03:     PQueue(IContext c) { e = c.NewEndpoint<IPQClient,IPQueue>(this); }
04:     private IEndpointInternal<IPQClient,IPQueue> e;
05:     [Endpoint] IEndpoint<IPQClient,IPQueue> E { get { return e; } }
06:     [Option(Async|Multi|Scalable)] void IPQueue.Enqueue( ... ) { ... }
07:     [Option(Async|Multi)] void IPQueue.Dequeue( ... ) { ... }
08:     void IReplicated<PQueue>.Export(PQueue other) { ... }
09:     void IReplicated<PQueue>.Import(PQueue other) { ... } }
```

A component in LDO is an annotated .NET class that exposes some *endpoints*. An endpoint is a small data structure that stores two interfaces: one exposed by the component that defines it (here **IPQueue**), and one that should be exposed by another component (**IPQClient**). Endpoints serve as connectors; all interactions between any pair of components in LDO are channeled through their endpoints, which have to be explicitly connected. As soon as this happens, the interfaces are exchanged, and the runtime dynamically generates, compiles, and transparently injects various wrappers that enable features such as automatic data conversion or event scheduling. Replication has been implemented as one of such wrappers.

Each component creates its endpoints in the constructor (line 3), stores them as fields (line 4), and provides on demand via annotated .NET properties (line 5). Any entity that holds a reference to a component can query it for endpoints and connect them to endpoints exposed by other components (including itself). The connect/disconnect operations are the only ones supported by endpoints. Thus, to use a component, one has to connect a client endpoint of a complementary type.

Wrappers are hidden from the programmer. To expose an interface (**IPQueue**), the component implements its methods (**IPQueue**.Enqueue and **IPQueue**.Dequeue in lines 6-7), and points to itself as the entity that handles all calls arriving via the endpoint (**this** in line 3). Under the hood, such interface is actually implemented by a wrapper, which follows the façade pattern, and can forward the call directly to the target component, or schedule it for later (Fig. 1). Method bodies can call the other connected component through the endpoint (e.g., e.Interface.Result(...)).

The set of injected wrappers is determined based on annotations ([**Option**(...)] in lines 6-7). For example, a method could be annotated as asynchronous (Async), multi-threaded (Multi, similar to COM multi-threaded apartments), and scalable (Scalable). By default, methods are synchronous (no scheduling code is injected).

Whenever the application process loads a new DLL, the LDO runtime uses .NET reflection and introspection to scan all types, looking for annotations, and generates wrappers as a C# code, which is then compiled in memory and loaded.

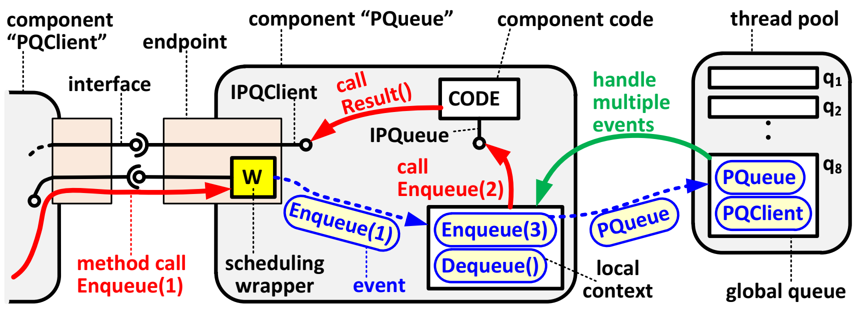

Fig. 1. To invoke method Enqueue on the component *PQueue* connected to it, component *PQClient* calls its local endpoint, which routes the call to the endpoint exposed by *PQueue*. The latter passes the call to an auto-generated wrapper, which encapsulates it as an event and schedules it on a lock-free queue in *PQueue*'s local context. The local context schedules itself on one of the global queues associated with scheduler threads.

For each asynchronous method, an event class is created that stores all its input and output parameters and programmer's custom annotations. For each interface exposed via a wrapper, a façade class is created that defines all its methods, and in their bodies, instantiates appropriate events and passes them to the scheduler.

Each component in LDO, at the time of creation, receives its private runtime context (**IContext** in line 3), a small object that holds all scheduler data structures associated with the individual component. When an endpoint is created (line 3), its wrappers are configured to pass their events to the component's local context. The local context has its own lock-free queue, on which all events associated with the component are deposited. If a new event is placed on the local context queue, but the queue is not actively processed by any scheduler thread, the local context registers itself with the scheduler by posting an event on a global lock-free queue (Fig. 1). Each thread in the thread pool has one global queue associated with it.

Each thread in the pool monitors its queue. When it finds an event posted by a local context, it enters the context to handle calls scheduled in it in a batched manner, one after another. The number of calls and time spent in a single local context are bounded to prevent starvation. If the scheduler thread exceeds this limit, it puts the context back on one of the other global queues, and moves on to the next local context. The advantage of this two-level queue hierarchy is that events associated with the same component are processed together, with better locality, and that scheduler threads consult their queues less often, thus reducing contention. A major disadvantage is that this push-based approach can result in unevenly balanced load. In practice, allocating 2x more scheduler threads than there are CPU cores suffices to avoid imbalance in all workloads we have tested.

2.2 Self-Replicating Objects

In the absence of scalable calls, an LDO component behaves as described in the preceding section; it acts similarly to a classical Actor, except with asynchronous methods that run to completion. Replication gets first activated when a scheduler thread enters the component's local context and dequeues an event representing a scalable method call from the component's local queue. The thread determines the type of the call from the annotations tagged onto the event by the scheduling wrapper. Now, depending on the situation, the thread can proceed in three ways.

First, if the component is in a collapsed state (as indicated by a status variable in the local context), the thread may simply ignore the annotation and execute the call as ordinary: scalable calls are legal in any state. This will be the case if the scheduler believes that the component is not under heavy workload, and the overhead of replication would outweigh its advantages. The scheduler maintains statistics in the local context, e.g., the average duration of a call in a recent time period or the number of scalable calls encountered in close succession; these can be used to devise adaptive policies for when to replicate. The discussion of such policies is outside the scope of this paper. In all our experiments we adopted an aggressive replication policy: new replicas are spawned (up to a predefined limit) when possible if all existing replicas are currently busy processing scalable calls.

Second, if the SRO is in a collapsed state, running an ordinary call, the event is put back at the head of the queue and the context is frozen; no new events are processed until all ordinary calls complete. The context also freezes if an ordinary call is dequeued in a replicated state, while scalable calls are still executing.

Finally, if the component is in a collapsed state, but not processing ordinary calls, or if it is already in the replicated state, the event is scheduled for execution on one of the *slave* replicas (the initial replica is designated as *master* and plays a special role). If no slave replicas exist, or if all existing slave replicas are busy, and their number is still below the predefined threshold, a new replica is created.

Slave replicas are copies of the entire component: as mentioned earlier, each has a copy of all fields/methods in the master. Each slave replica also includes its own local context and its own local event queue independent from the master; it is a completely separate component in LDO, and behaves as described in the preceding section. The only difference is that all features pertaining to replication are disabled: slave replicas cannot further replicate the calls. Also, slave replicas are not visible to other application components, and cannot directly receive calls: they can only receive calls forwarded from the master. The slave replicas may be able to submit calls to other components through one of the master's endpoints, although we have not needed such functionality in any of the tested workloads.

If a new replica is created, it is now added to the master's pool, and `export` is executed in the context of the master to transfer state. If not, one of the existing replicas is selected using a simple round-robin policy. In either case, the event is subsequently placed onto the designated replicas's lock-free queue; it is executed later, when a scheduler thread enters the replica's local context. Meanwhile, the master keeps processing events in its local context, and if more scalable calls are encountered, they are immediately rescheduled to execute on different replicas.

Notice that in this scheme, every scalable call goes through two event queues, and is scheduled twice. The disadvantage of this is that calls suffer from increased latency. On a very heavily loaded system, it can take tens of milliseconds for such event to be processed. The advantage is that if many scalable calls are submitted in short succession, they can be much more efficiently distributed across threads. It might be possible to reduce the overhead by moving the scheduling logic to the scheduling wrapper, but we found that it dramatically increases the complexity of the scheduler: even in the current version, the scheduling and replication logic includes 30+ CAS-style operations, and in practice, replication can be adaptively enabled only on bursty workloads, where throughput matters more than latency. Indeed, we experimented with a simple *bypass* technique that avoids intermediate scheduling on the master; it made no impact on the results reported in Section 3.

Notice also that our scheme is push-based; there is no work-stealing. Once an event is forwarded to one of the replicas, it is assigned to the replica and cannot be further reassigned. To alleviate the potential for uneven distribution of work, we provide a *flow control* policy. Each replica adaptively chooses the maximum number of events it can accept: it monitors the duration of the subsequent calls, and adjusts the threshold so that it has enough events to continue for δ seconds, where δ is a configuration parameter. If all replicas are at their thresholds, the event cannot be rescheduled and the master context is frozen, as discussed earlier. In practice, we found that on the tested workloads, flow control is not necessary, since the durations of subsequent scalable calls were similar enough. Our scheme could be modified to use a work-stealing policy, if desired, although some of the features discussed later rely on the push approach and would have to be disabled.

Moving scalable calls to slave replicas has one additional benefit: the master is free to perform export and import at any time while a long sequence of scalable calls is being processed. Normally, the master executes import calls only when the SRO is in a replicated state, and its context is frozen on a pending ordinary call. When a slave replica finishes processing all calls scheduled on it, and detects that the master is frozen in such a state, it awakens the master, which subsequently executes import on the replica, then disposes of it. While some replicas are getting demoted, other replicas might still continue processing their scheduled calls; this allows for the processing and merging phases to partially overlap.

In some workloads discussed below, postponing the entire merge phase until the end is undesirable, e.g., in situations where the costs of merging rise rapidly as a function of the number of events processed by a replica. Accordingly, we implemented optional *incremental* imports. When this feature is enabled, each slave replica runs a timer with timeout θ, where θ is a configuration parameter; it checks the timer after processing each event. Whenever the timeout expires, the replica checks if another replica designated as its *parent* is ready to accept an incremental import. The replicas are ordered into a tree-like hierarchy, with the master at the root. A replica can always accept an incremental import if there are no incremental imports currently enqueued on it, as discussed below. If the parent cannot accept an import, the timed-out replica resets the timer to a shorter *retry* interval θ' (also a parameter), and retries the incremental import later. If the parent is ready for incremental import, the process works as follows.

To perform incremental import, the timeout-out replica r spawns a new *transient* replica r' outside of the replication limits. The transient replica r' has no local context, no queue, and it cannot accept events; it exists solely for the purpose of transferring state. The timed-out replica r first executes $\mathsf{export}(r')$ it its own context to initialize r', and follows with $\mathsf{import}(r)$ in the context of r', thus allowing all its state to be moved to r'. Replica r now remains in an empty state, and can continue to accept new calls, whereas r' is enqueued with the parent on a dedicated *transfer queue*. Each replica checks its own transfer queue after each scalable call, and if it finds a transient replica r' enqueued on it, it immediately performs $\mathsf{import}(r')$, then disposes of r'. To prevent the transient replicas from piling up and keep the overhead low, a replica is considered not ready for import if its transfer queue is non-empty. Thus, transient replicas are spawned at most at the speed at which they can actually be consumed by their respective targets.

The last feature we discuss in this section is the ability to export slave replicas to other processes or to other machines; we refer to this as *remote* replication (as opposed to *local* replication discussed so far). To support this, the LDO scheduler optionally spawns a number of child processes and establishes TCP connections with them. Each of the child processes runs a separate hosting environment, and can run components on behalf of the master process controlling it. When remote replication is enabled, the master replica on the master process, after creating a slave replica, can optionally serialize and send it over TCP to the child process. If a replica is exported out of process, instead of rescheduling scalable calls, as described earlier, the master likewise serializes all calls and sends them over TCP to the child for remote execution. The child remotely executes all calls, and can periodically export its current state and send it back to the parent process in a way similar to how we implemented incremental import. Otherwise, the process is virtually identical to what was discussed so far, except that all control requests such as the intent to import and demote a replica, are transmitted over the network. Since all scheduling logic is already event-driven, asynchronous, and push-based, we did not need to significantly change our scheduler to support this.

As discussed in Section 3, remote replication allows replicas to run in separate CLRs, with their own schedulers and as a result, much less contention. The main overhead factor is the cost of serialization/deserialization. To reduce it, we used a serialization stack from our QuickSilver Scalable Multicast (QSM) engine [57], with performance-improving features such as scatter-gather and mechanisms that eliminate the need to repetitively pass class/namespace names and other metadata. While not as versatile and functional as .NET serialization, this scheme let us decrease communication overheads by an order of magnitude.

3 Evaluation

3.1 Configuration and Performance Metrics

All measurements reported here were performed on a Dell PowerEdge 2950 server with 2 quad-core Intel Xeon X5355 2.66GHz/8MB cache CPUs (in total 8 cores), 16GB RAM, running Microsoft Windows Server 2008 R2 with .NET Framework 3.5sp1/4.0beta2. The LDO runtime and the test components were compiled with optimizations and ran as 64-bit code. A typical workload lasted for minutes and involved a warmup phase. Time was measured using a combination of DateTime, QueryPerformanceCounter, and for detailed profiling on Fig. 7, 8, also the RDTSC instruction. We took precautions against the known RDTSC pitfalls: take samples on the same physical thread pinned to a CPU core; disable power management, check for CPU frequency changes; issue memory barriers; reboot between series of experiments; analyze distributions and check for negative/very large samples. We rely on RDTSC only for increased resolution on very short intervals. Statistics such as % CPU utilization were taken using OS performance counters.

The two primary performance statistics we employ are *speedup* and *efficiency*. Both are relative with respect to the baseline we use in the particular experiment. We define speedup s as $s = t_0/t$, where t is the time to finish all computation with SRO, and t_0 is the time to finish in the baseline scenario; the total amount of work is the same. We define efficiency as relative speedup divided by relative cost: $e = (t_0/t)/(c/c_0)$, where c is the total CPU utilization across all cores if using SRO, and c_0 is the CPU utilization in the baseline scenario. Note that since efficiency is relative to the baseline, it can be higher than 1 (more is better). This is the case if the baseline scenario involves high contention.

Another statistic we use is the relative utilization of CPU cores. To account for the fact that threads migrate, in each set of measurement samples we order cores from the most to the least loaded and average samples within each rank. We use the term k^{th} *core* to refer to the k^{th} busiest core, not a physical one.

3.2 Performance with Synthetic Workloads

We begin with the analysis of idealized synthetic workloads in order to quantify the overheads incurred by the SRO infrastructure and the .NET CLR. We also evaluated realistic workloads and report them later, but those results are easier to interpret in light of the information from the synthetic ones. The main takeaway from this section is that when a method call to an SRO lasts for at least a few μs (as is the case for Enqueue calls in a large priority queue, as shown on Fig. 3, and all MapReduce workloads discussed in Section 3.4), the overheads of replication are negligible, and scalability is limited by factors independent of SRO: memory bottleneck and .NET CLR overheads, particularly memory allocation, garbage collection, and contention in .NET data structures.

In the first experiment (Fig. 2), the tested component declares two methods: Work() annotated as scalable and an ordinary Done(). Work performs a single type of operation repeatedly, in a tight loop, as shown below. By varying the number of loop iterations (constant *count*), we control the duration of a single Work call.

```
01: void Work() { for (int i = 0; i < count; i++) { operation } }
```

The benchmark also invokes Work multiple times in a loop, sequentially and from a single thread, and follows with a single call to Done, as shown below. Done calls back into the benchmark code to signal that all work has completed.

```
01: for (int i = 0; i < num; i++) { sro.Work(); }
02: sro.Done();
```

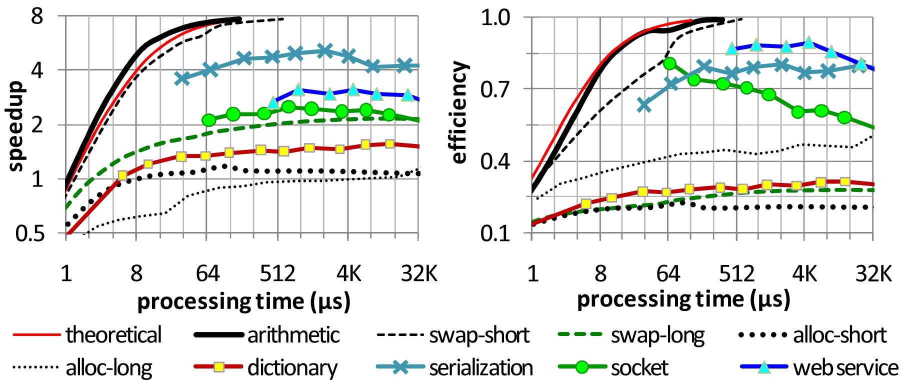

Fig. 2. Performance with synthetic workloads: one type of operation, no state to merge. Processing time and speedup are plotted on a logarithmic scale to improve readability.

In the baseline scenario, the tested component acts like an ordinary .NET object; each call to Work and Done runs synchronously. With SRO enabled, Work becomes asynchronous and is excuted in parallel by the replicas, and Done waits for all the Work calls to complete and for all the replicas to be merged. In this experiment, there is no state to merge; the body of import is empty. The benchmark thus only measures the maximum speedup that can be achieved by parallelizing Work calls. In realistic workloads, much of this gain is subsequently lost in import.

In case of a purely computational workload, when the body of Done consists of floating point operations (data series "arithmetic" on Fig. 2), speedup > 1.0 is achievable when the duration of the call is higher than $\epsilon = 1\mu s$, approximately the amount of overhead introduced by the SRO scheduling and replication logic. Both speedup and efficiency reach their maximum with calls approaching $100\mu s$, but even calls lasting $5\mu s$ can benefit from a 4-fold speedup and a 70% efficiency.

The observed values match our predictions (data series "theoretical"), captured by the formulas below, in which n is the number of cores, t is the duration of the call in a baseline scenario, and κ is scheduling contention discussed below.

$$\text{speedup } s(t) = \frac{n}{1 + n\epsilon/t}\ , \qquad \text{efficiency } e(t) = \frac{1}{1 + \kappa\epsilon/t}\ . \tag{1}$$

To see how the formulas are derived, note that scheduling overhead can be viewed as mostly sequential. If all work is perfectly balanced across cores, m calls to an SRO take $m\epsilon + mt/n$, compared to mt in the baseline case. We obtain $s(t)$ after dividing the latter by the former. To compute cost, assume that in the scheduler, κ cores on average compete for resources. The number of cores used by the SRO is an average of κ and n, weighted by $m\epsilon$ and mt/n, correspondingly. We arrive at the formula for efficiency $e(t)$ by dividing speedup $s(t)$ by that weighted average.

The value of κ will depend on the implementation. In our platform, contention occurs primarily in the implementation of lock-free queues spinning on CAS-style operations; it involves the application thread issuing calls and a scheduler thread dispatching them to the replicas, hence $\kappa \approx 2$; this is the value we used on Fig. 2.

The next two workloads in this experiment are heavy on memory operations: swapping the contents of random cells in a 1.6MB array that fits entirely in the cache (the *swap-short* series) and in a 100MB array that does not (*swap-long*). Replicas in the SRO scenario work on disjoint parts of the array, and use different patterns of random access; the random indices of cells to swap are pre-computed.

When all data fits entirely in the L2 cache, the achievable speedups are nearly the same as in case of the purely computational workload, although for calls that last under $100\mu s$ efficiency is slightly lower. In contrast, if the data does not fit, performance is slashed by the system memory bottleneck, the speedup converges to 2 even with very long calls, and the efficiency remains low, at the level of 25%.

Contention is even more pronounced with memory allocations. The next two workloads allocate small objects, combine them into linked lists, and restart each time the lists reach a predefined maximum size: 160 elements (*alloc-short*), or 2.5 million (*alloc-long*). With SRO, replicas build separate lists, and their maximum length is scaled down to ensure that the garbage collection overheads are similar. In either case, we cannot speed up sequential code: allocation is too expensive.

While workloads consisting solely of memory operations and allocations may seem artificial, they turn out to be good predictors of the performance we observe with several of the standard collections. In particular, manipulating hash tables (*dictionary*) involves small allocations and access over large portions of memory, and indeed it lies somewhere in between with its 1.5 speedup and a 30% efficiency.

In contrast to manipulating large data structures and collections, operations such as manipulating network sockets, calling SOAP web services, serialization, and deserialization are quite scalable: we can achieve 2.5 to 5-fold speedup with up to 80% efficiency. This suggests that contrary to the popular trend, in managed environments the benefits of parallelism may be easier to find, e.g., in web application components as opposed to typical data-heavy MapReduce workloads.

In the experiment just discussed, all calls to the tested component were made from within a single thread, but in realistic parallel applications, there will often be concurrency on the client-side as well. In the second experiment, we split the calls evenly among 8 client threads and let them all make their calls concurrently. The final Done is called once, after all client threads run through their for loops. In the baseline scenario, we additionally protect Work with a method-wide lock.

For the sake of brevity, we omit performance data and just briefly summarize the results. For the most part, speedup is not affected by client-side concurrency; in the baseline scenario, all work is anyway serialized due to locking and the SRO is isolated from client threads through its local context queues. What does change is efficiency: with calls under $500\mu s$, the baseline scenario experiences high lock contention and saturates all CPU cores with useless work. SRO efficiency is >1 even with $1\mu s$ calls for most workloads, and reaches 8 at $500\mu s$ with *arithmetic*.

3.3 Performance with Stateful Components

In this section, we move on to more realistic workloads. We discuss two examples of components that have state and a nontrivial merging phase: the *priority queue* introduced in Section 1.2 (Example 1), and the *append log* (Example 2, discussed below). The former represents a broader class of workloads that involve batched processing, where data is continuously accumulated and accessed in phases. The latter represents the broader class of producer-consumer scenarios. The examples demonstrate the versatility of SRO. While implementing these components using Actor-style parallelism or data-parallel abstractions is possible, we found it hard to do this in an elegant manner; typically, one resorts to low level programming, with explicit use of threads and fine-grained locks. The examples are also important in that compared to MapReduce workloads discussed in Section 3.4, the components discussed here are lightweight: a typical call lasts only a few μs (Fig. 3), yet SROs can offer measurable performance benefits even at this level.

To evaluate the performance of the priority queue, we preload the queue with various numbers of elements, ranging from a few to a few millions, and perform a burst of `enqueue` calls in a loop followed by a single `dequeue`. As in the preceding section, if SRO is disabled, all calls run synchronously, and if SRO is enabled, the `dequeue` is postponed until all `enqueue` calls complete. By varying the number of enqueued elements, we vary the size of the data structures and the overheads incurred by each call: parameter *processing time* on Fig. 3. Additionally, we vary the number of arguments passed to `enqueue`: parameter b on Fig. 3. When $b > 1$, the benchmark is requesting multiple events to be scheduled in one call, passing the desired times together as arguments to `enqueue`. The event times are random and computed in advance to avoid polluting the experiment. To simulate the use in a realistic application, the k-th event out of the total n is scheduled to run at time $k/n + \alpha \cdot X$, where X is a [0,1] uniformly distributed random variable and α is a parameter that determines how much the series is unordered. The results are similar for most choices of α; those on on Fig. 3 were obtained with $\alpha = 0.1$.

As seen on the charts (Fig. 3, top), higher speedups and efficiency are possible with longer calls, when the SRO scheduling overheads and the cost of allocating splay tree nodes are less pronounced. Longer calls work on larger data structures, and spend proportionally more time on comparisons and memory operations. As explained in the preceding section, this is where gains can potentially be made.

With 8 CPU cores, one might expect speedup higher than 2, but if we compare the results on Fig. 3 with those on Fig. 2, it becomes clear that performance is limited by memory bottleneck and non-scalable memory allocation overheads; the two types of operations dominate in the code, and indeed, speedup/efficiency curves on Fig. 3 lie close to those of *alloc-long* and *swap-long* reported on Fig. 2.

The speedup is better explained if we look at the performance of SRO separately in the working phase (when performing `enqueue`), and in the merge phase (when collapsing the replicas). In the working phase, the SRO utilizes only about 3-4 cores (not shown) because of the contention factor mentioned above. Because work is split among multiple replicas, each call is also slightly cheaper. Overall, SRO completes work in 20-30% of the time it takes the baseline implementation to finish (Fig. 3, bottom left). The non-parallelizable merge phase takes a similar amount of time (Fig. 3, bottom right), thus bringing speedup down to 70-100%.

Notice that while the gain achieved in the work phase converges to a constant level and depends mainly on the number of cores and contention in the CLR, the

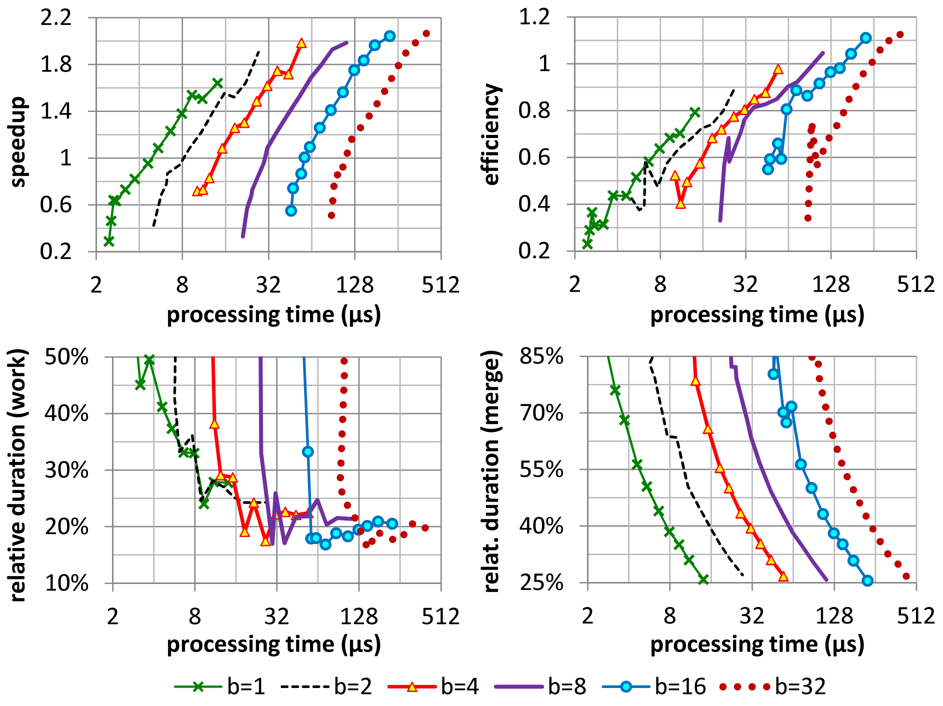

Fig. 3. Top charts: performance of the priority queue implemented as SRO, relative to the baseline. Bottom charts: the durations of the enqueuing (*work*) vs. import (*merge*) phases with SRO enabled, expressed as relative to the duration of the baseline scenario.

relative duration of the merge phase is sharply decreasing: as the data structures get larger and more costly to access, the per-element merge overhead eventually becomes insignificant compared to the per-call overhead during the work phase.

Now, we shift attention to our second example: the *append log*. As mentioned earlier, this type of logic is often expressed using low-level abstractions: threads, locks, asynchronous I/O. SROs are easier to use, and as shown here, just as fast.

Example 2. Consider a logging component with one method log(m) that appends an update m to the end of a log file on a disk, e.g., to enable rollback and replay in a system that can crash and reboot. The operation needs to serialize m, which is computationally intensive, and could be parallelized, and perform I/O, which requires synchronization to ensure that the appended segments do not overlap. We can model the component as an SRO, in which a scalable call log(m) serializes the object and puts it on the list of objects pending I/O, the import call merges the lists, and if it occurs on the master, import also invokes I/O in a synchronous manner. By activating SRO's incremental imports, we ensure that the serialized blocks are written out regularly, at a speed at which the disk can accept them.

The benchmark creates a .NET dictionary m with n elements in it, and then repeatedly invokes log(m) in a loop, measuring the time for all calls to complete.

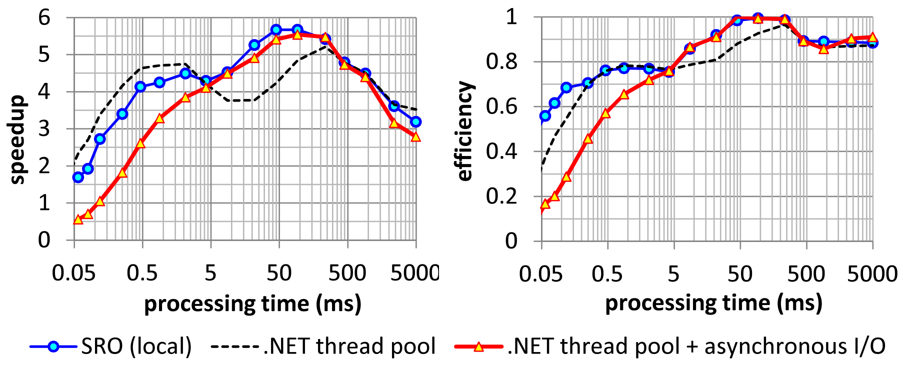

Fig. 4. Performance with *append log*, relative to the single-threaded sequential baseline: SRO compares favorably to hand-crafted code with threads/locks + asynchronous I/O.

As in the previous sections, by controlling the number of elements n, we change the duration of a single log call. The baseline implementation is again sequential, singe-threaded, and invokes the log calls one after another.

We compare the speedups achieved by SRO with two alternative implementations. The first (*.NET thread pool*) performs serialization in parallel by scheduling work items on the .NET thread pool, and performs disk I/O synchronously, protected with a **lock**. The second (*.NET thread pool + asynchronous I/O*) also uses asynchronous I/O to parallelize disk writes and minimize lock contention.

SRO compares favorably to both implementations: it makes the most efficient use of CPU (Fig. 4), and always performs at least as well as asynchronous I/O. The slight differences compared to *.NET thread pool* may be caused by different behavior of the .NET and LDO schedulers; overall, performance is comparable.

3.4 Performance with MapReduce Workloads

In this section, we shift attention to classic MapReduce workloads, modeled after prior work on MapReduce on multi-core platforms [37]. As a baseline scenario, we have implemented each workload as a program that explicitly spawns threads and makes efficient use of fine-grained locking to protect shared data structures; we refer to this baseline as *locking*. We demonstrate that performance achieved with SRO always matches, and in several cases beats that achieved with a hand-written code; replication overhead is lower than the hard to avoid lock contention. Furthermore, SROs perform as well as, and in several cases better than state of the art parallel MapReduce abstractions offered by the latest versions of .NET.

We evaluated the performance of SROs with three parameter settings. First, we used in-process replication: all replicas are spawned in the same address space as the benchmark code; we refer to this setting as *local* replication on the charts. We disabled flow control and incremental aggregation; they had minimal effect on performance. Second, we used out-of-process (remote) replication: replicas are exported to child processes, with all communication over TCP. Since serialization in this scenario affects performance, we evaluated two variants: SRO with .NET binary serialization stack (*remote*), and with the QSM [57] stack (*remote - QS*).

SROs are coded in much the same way as in the preceding sections: scalable map calls are issued in a **for** loop, followed by a single ordinary done call to mark the end of the computation, and the final reduction step is performed in import.

We compared our results with two state of the art approaches based on .NET Parallel Extensions: one that directly uses Parallel.For from .NET Task Parallel Library (TPL), denoted as *ParallelFor* on the charts, and another expressed with .NET PLINQ. The *ParallelFor* version adheres to the following general pattern.

```
string[] paths = ...;
IDictionary<string,int> result = new Dictionary<string,int>();
Parallel.For(0, paths.Length, new ParallelOptions(),
    () => new Dictionary<string,int>(),
    (index, loop, counts) => Reduce(counts, Map(paths[index])),
    counts => { lock (mylock) result = Reduce(counts, result); });
```

The above shows the WordCount implementation; the others are similar. Notice the lock statement in the last line; there, partial results generated by each worker thread are integrated into the main data structure at the end of the computation.

The *PLINQ* version is expressed in a purely functional style; hence, it does not require any explicit locks. It adheres to the following general pattern.

```
IDictionary<string, int> result =
    (from index in ParallelEnumerable.Range(0, paths.Length)
        select Map(paths[index])).Aggregate((x, y) => Reduce(x, y));
```

With PLINQ, aggregation is performed incrementally (during the computation).

Most of the following discussion focuses on analyzing the classic WordCount workload (Fig. 5), to explain in depth the reason for the performance advantages of SRO over the manually written code, and on the benefits of remote replication. Other classic MapReduce workloads are discussed at the very end of this section.

We discuss two WordCount implementations: the most intuitive (*hash table*) that uses a .NET Dictionary to count word occurrences, and one (*sorting*) that is less heavy on memory allocation and the use of .NET collections. The latter is computationally more expensive, but incurs less contention and performs better across the board; in particular, it is 4x faster in the baseline scenario.

As shown on the charts, regular SRO (*local*) performs as well as hand-written code with the *sorting* implementation, and almost 2x faster in case of *hashtable*. In both cases, SRO utilizes fewer CPU resources. The results are almost identical to those achieved by PLINQ, and better than ParallelFor in one of the scenarios.

When work is offloaded to child processes, SRO performs significantly better than all other implementations: with QuickSilver serialization, it yields an almost 2x speedup over PLINQ in the sorting scenario and more than 3x speedup over PLINQ in the hashtable scenario. In the latter case, SRO provides a 5.7x speedup over the hand-written code, and makes a 4x more efficient use of CPU resources. This is despite the considerable overhead incurred by serialization/deserialization and exchanging data over TCP with child processes. Performance and efficiency

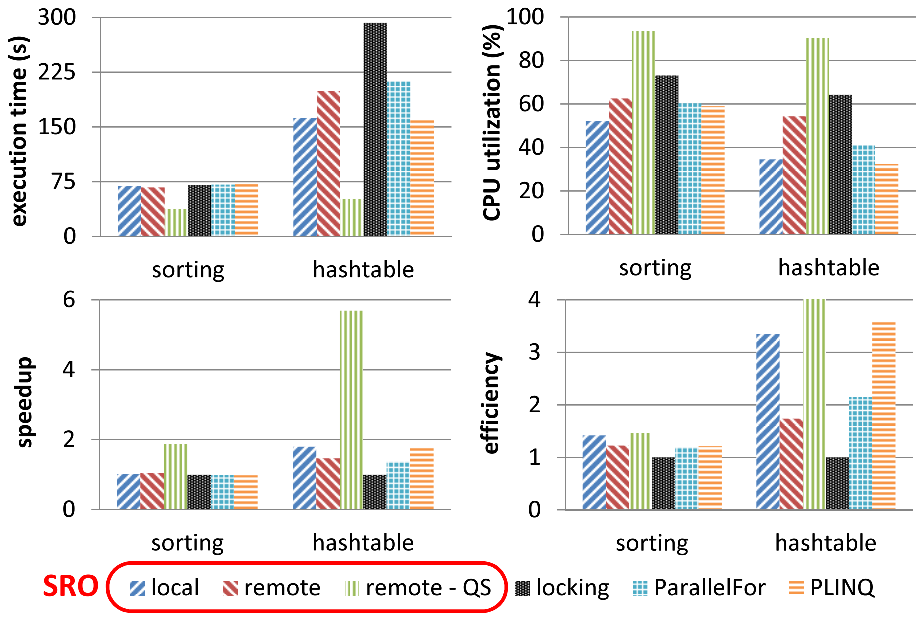

Fig. 5. Performance of WordCount: we compare code that uses .NET Dictionaries (*hash table*) with code that sorts words to facilitate counting and avoids memory allocations (*sorting*). Performance metrics on the bottom charts are relative to the *locking* baseline. On these charts, all performance metrics including speedup are plotted on a linear scale.

gains are due to the fact that remote replication greatly reduces contention, and the redundant work involved in inter-process communication is smaller than the overhead otherwise incurred by spinning locks, lengthy garbage collections, etc.

To better illustrate the reasons for the observed performance gains, particularly with remote replication, we report the relative durations of different phases of computation (Fig. 6). As seen on the chart, serialization/deserialization with the QSM stack takes on the order of a few seconds compared to the tens of seconds saved in the working phase thanks to the lower contention among replicas. Unlike the priority queue, merging replicas in WordCount is very cheap: the time spent in import accounts for about 5% of the total duration of the experiment.

We attributed SRO's performance gains to a reduced contention. To measure it directly, we profiled different sections of the code (Fig. 7). First, in the baseline scenario 50% of the time is spent acquiring locks, which hurts efficiency. Second, GC activity with the baseline is especially high in the *hash table* scenario, where SRO turns out to be particularly beneficial. Remote replication slashes the cost of memory-related operations and the use of collections (*add* and *GC*); it makes less impact on sections that are less sensitive to contention (*find words* and *sort*).

To further elucidate the dynamics of the workload, we micro-profiled selected individual lines of code that contain representative operations, such as individual dictionary lookups and string comparisons (Fig. 8). Each of these operations took

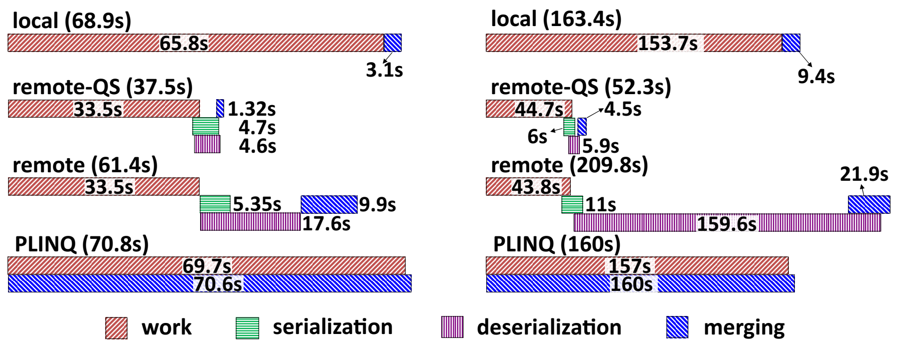

Fig. 6. The time spans of the different phases of computation in the WordCount workload (left: *sorting*, right: *hash table*). In case of PLINQ, merging overlaps with computation; the length of the *merging* bar is thus not indicative of the actual work invested.

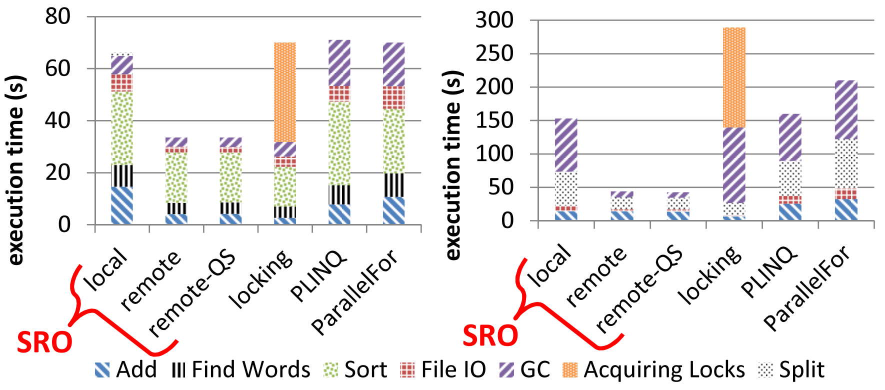

Fig. 7. The relative contributions to the total execution time in Word Count (left: *sorting*, right: *hash table*) coming from different sections of the benchmark code: adding new words to a dictionary (*add*) and looking up existing ones (*find words*), sorting words to eliminate duplicates (*sort*), splitting lines of text into word-sized strings (*split*), file I/O (*file IO*), acquiring locks (*acquiring locks*), and the garbage collector activity (*GC*).

significantly less time in SRO's remote replication scenarios despite the fact that the total number of threads (and thus the amount of preemption) was identical.

It is worth noting that SRO performance matches that of PLINQ despite the fact that SRO employs an aggressive push-based scheduling scheme instead of a work-stealing scheduler [53]. In all scenarios we evaluated, allocating an adequate number of scheduler threads and object replicas guarantees even distribution of work; as a rule of thumb, we typically set these to 2x the number of CPU cores.

As mentioned before, the reason for using a push-based scheme is motivated by the remote replication performance. Work-stealing scheduling is hard to combine with remote replication without an explicit support from the .NET runtime; the scheduler would have to span across and migrate events between processes. In our push-based scheme, a call once dispatched stays assigned to the same process. Imbalance in work assignment can be bounded by the flow control scheme.

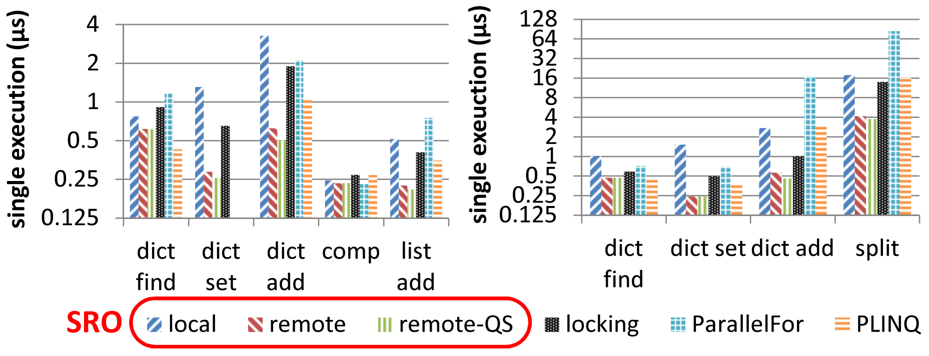

Fig. 8. The result of micro-profiling individual lines of code in Word Count (left: *sorting*, right: *hash table*): dictionary TryGetValue (*dict find*), assignment d[key]=value (*dict set*), dictionary Add (*dict add*), a comparison of two strings (*comp*), list Add (*list add*), and string Split (*split*) operations. Longer times point to increased contention in .NET.

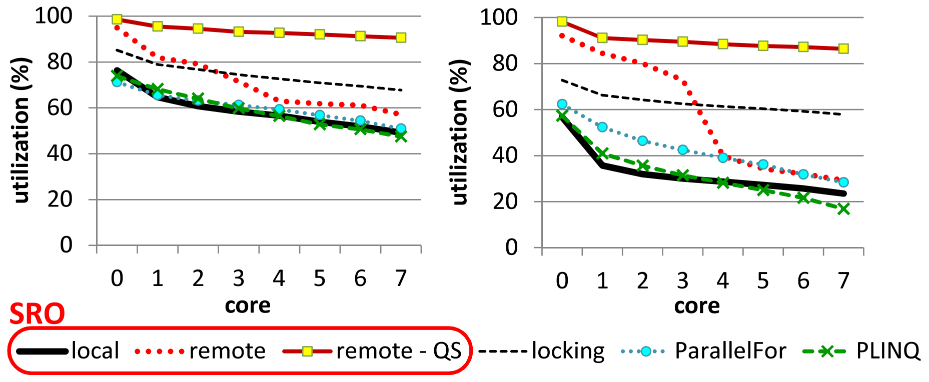

Fig. 9. Relative CPU core utilizations with WordCount for the *sorting* (left) and *hash table* (right) implementations: the k^{th} data point shows the load on k^{th} most busy core.

To put this discussion in context, let's look at the relative utilization of CPU cores (Fig. 9). As seen on the chart, locally replicated SROs balance the workload roughly as well as PLINQ. The distribution of work is actually slightly more even on all cores except for the busiest one; the load on the busiest core is higher than on the others due to the (mostly sequential) replica splitting and merging phase. Remote replication spreads work even more evenly despite the mostly sequential deserialization phase. This is due to the fact that with out-of-process replication, different replicas are using independent garbage collectors. The baseline *locking* implementation also loads cores evenly, but as noted earlier, it wastes resources.

Having thoroughly analyzed *WordCount*, we now briefly summarize the performance achieved by SRO on other examples of MapReduce workloads (Fig. 10). First, as was the case for *WordCount*, SRO's local replication generally performs at least as well as hand-written code, PLINQ, and ParalellFor. Remote replication does not help on workloads, in which the method calls are very short (*String*

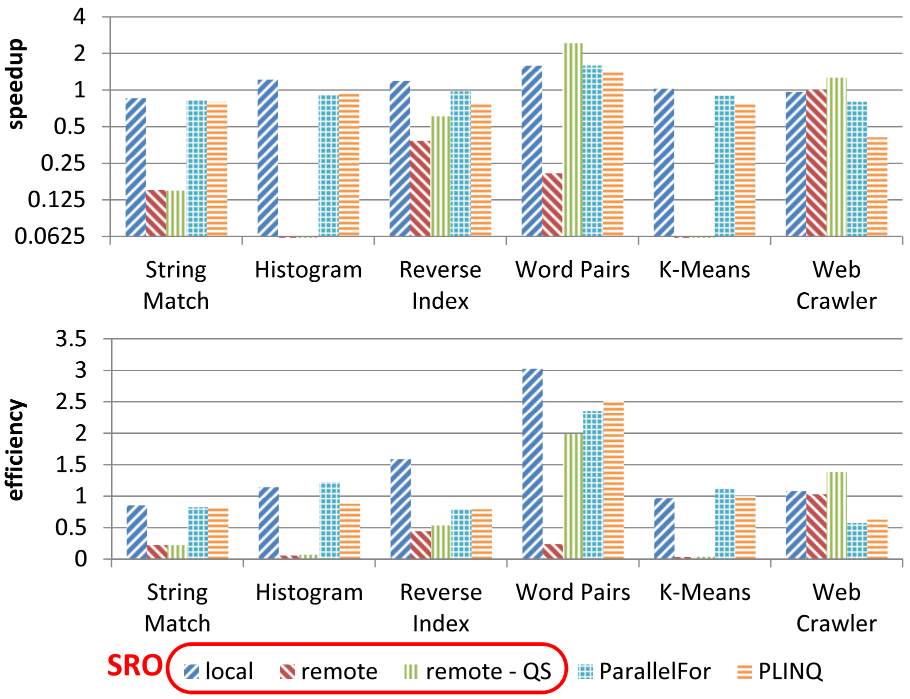

Fig. 10. Performance with other typical MapReduce workloads [37]. *Word Pairs* counts the frequencies of bi-grams (two-word sequences) instead of single words. *Web Crawler* compiles a list of hyperlinks starting at *http://google.com* in an iterative manner; each iteration explores the next level. *K-Means* is also iterative, and uses randomly generated data points. *Reverse Index* operates on an offline, locally saved snapshot of Wikipedia.

Match and *K-Means*), and where the amount of data transferred in each method call is relatively large *Histogram* and *Reverse Index*). In the former case, batching calls can help significantly; after doing so, one can still demonstrate performance gains due to the reduced contention (not reported here due to space limitations).

4 Conclusions

SROs are an elegant, easy to use programming abstraction that addresses shortcomings of the Actor model concurrency, and unifies multiple paradigms. SROs are versatile; they can match, and in some cases exceed the performance of both manually-written code and dedicated mechanisms such as the .NET thread pool, asynchronous I/O, or PLINQ, on several types of workloads, whether functional (MapReduce) or not (*priority queue*, *append log*). SROs are easy to incorporate into existing OO environments, and seamlessly integrate with the existing code.

Acknowledgments. This work was supported, in part, by grants from AFRL, NSF, and Intel. We would like to thank Andrew Myers for his helpful comments.

References

1. http://www.intel.com/pressroom/archive/releases/20090526comp.htm
2. Sutter, H.: The concurrency revolution. http://www.ddj.com/cpp/184401916
3. Case, L.: Does quad core matter? extremetech.com (2007)
4. Hagedoorn, H.: Cpu scaling in games with dual and quad core processors. guru3d.com (2008)
5. Intel: Digital Office Performance: Business Productivity with SYSmark 2007 preview. intel.com (2009)
6. van Renesse, R.: Goal-oriented programming, or composition using events, or threads considered harmful. EW-8 (1998)
7. Ousterhout, J.: Why threads are a bad idea (for most purposes). Invited talk at USENIX (1996)
8. Sutter, H., Larus, J.: Software and the concurrency revolution. Queue **3**(7) (2005)
9. Harris, T.: Exceptions and side-effects in atomic blocks. Science of Computer Programming **58**(3) (2005)
10. Larus, J.: Spending moore's dividend. CACM **52**(5) (2009)
11. Amdahl, G.: Validity of the single processor approach to achieving large scale computing capabilities. (2000)
12. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. IJHPCA (2007)
13. Ghulou, A., Sprangle, E., Fang, J., Wu, G., Zhou, X.: Ct: A flexible parallel programming model for tera-scale architectures. techresearch.intel.com (2007)
14. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media (2007)
15. Duffy, J., Essey, E.: Parallel LINQ: Running queries on multi-core processors. msdn.microsoft.com (2009)
16. Leijen, D., Hall, J.: Parallel performance: Optimize managed code for multicore machines. msdn.microsoft.com (2009)
17. Blelloch, G.: NESL: A nested data-parallel language. CMU Tech Report (1993)
18. Burton, F.: Functional programming for concurrent and distributed computing. Computer Journal **30**(5) (1987)
19. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. IJCAI (1973)
20. Yokote, Y., Tokoro, M.: Experience and evolution of concurrent smalltalk. SIGPLAN Notices **22**(12) (1987)
21. Baker, h., Hewitt, C.: The incremental garbage collection of processes. AIPL'77
22. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Prentice Hall (1996)
23. Haller, P., Odersky, M.: Event-based programming without inversion of control. JMLC (2006)
24. Haller, P., Odersky, M.: Actors that unify threads and events. scala-lang.org (2007)
25. Pickering, R.: Concurrency in f# - part iii - erlang style message passing. strangelights.com (2007)
26. Fähndrich, M., et al.: Language support for fast and reliable message-based communication in singularity os. EuroSys (2006)
27. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for c#. TOPLAS (2004)
28. Chrysanthakopoulos, G., Singh, S.: An asynchronous messaging library for c#. SCOOL (2005)

29. Podwysocki, M.: Introducing maestro – a dsl for actor based concurrency. weblogs.asp.net/podwysocki (2009)
30. Gustafsson, N.: Axum: Language overview. msdn.microsoft.com (2009)
31. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. SOSP (2004)
32. Eugster, P.: Uniform proxies for java. OOPSLA (2006)
33. Ostrowski, K., Birman, K., Dolev, D., Ahnn, J.: Programming with Live Distributed Objects. ECOOP (2008)
34. Ostrowski, K., et al.: Live Distributed Objects. liveobjects.cs.cornell.edu (2007)
35. Zhang, X., Hiltunen, M., Marzullo, K., Schlichting, R.: Customizable service state durability for service oriented architectures. EDCC (2006)
36. Marian, T., Balakrishnan, M., Birman, K., van Renesse, R.: Tempest: Soft state replication in the service tier. DSN (2008)
37. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating mapreduce for multi-core and multiprocessor systems. HPCA (2007)
38. Lämmel, R.: Google's mapreduce programming model - revisited. Sci. Comput. Program. **68**(3) (2007) 208–237
39. Thies, B., Karczmarek, M., Amarasinghe, S.: Streamit: A language for streaming applications. ICCC (2001)
40. Black, A., Carlsson, M., Jones, M., Kieburtz, R., Nordlander, J.: Timber: A programming language for real-time embedded systems. timber-lang.org (2002)
41. Varela, C., Agha, G.: Programming dynamically reconfigurable open systems with salsa. OOPSLA (2001)
42. Lauer, H., Needham, R.: On the duality of operating system structures. ACM SIGOPS OSR (1979)
43. Fournet, C., Gonthier, G.: The join calculus: A language for distributed mobile programming. APPSEM (2000)
44. Sulzmann, M., Lam, E., van Weert, P.: Actors with multihea- ded message receive patterns. COORDINATION (2008)
45. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for java. ECOOP (2008)
46. Thomas, D., et al.: Actra: A multitasking / multiprocessing smalltalk. SIGPLAN Notices 24(4) (1989)
47. von Behren, R., Condit, J., Zhou, F., Necula, G., Brewer, E.: Capriccio: scalable threads for internet services. SOSP (2003)
48. Cunningham, R., Kohler, E.: Making events less slippery with eel. HotOS (2005)
49. von Behren, R., Condit, J., Brewer, E.: Why events are a bad idea (for high-concurrency servers). HotOS (2003)
50. http://msdn.microsoft.com/en-us/devlabs/ee794896.aspx
51. Li, P., Zdancewic, S.: A language-based approach to unifying events and threads. Tech Report, U. of Penn. (2006)
52. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. PPoPP (1995)
53. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. OOPSLA (2009)
54. Allen, E., et al.: Project fortress: A multi-core language for multi-core processors. linux-mag.com (2008)
55. Balakrishnan, S., Sohi, G.: Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. SIGARCH (2006)
56. Rinard, M., Diniz, P.: Eliminating synchronization bottlenecks using adaptive replication. TOPLAS (2003)
57. Ostrowski, K., Birman, K., Dolev, D.: Quicksilver scalable multicast. NCA (2008)