Digital Object Identifier: 10.1109/ICPADS.2016.0132

URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7823847

Application-level Determinism in Distributed Systems

Christopher Boelmann, Lorenz Schwittmann, Marian Waltereit, Matthäus Wander and Torben Weis
Distributed Systems Group
University of Duisburg-Essen
Duisburg, Germany

***Abstract*—Deterministic and reproducible program execution eases the development and debugging of distributed systems. However, deterministic execution comes at high performance costs and is hard to achieve, especially when running on different hardware. In this paper we introduce the concept of *application-level determinism* and describe how the parallel programming model *Spawn & Merge* can be used for scalable and deterministic distributed computation. Application-level deterministic applications yield reproducible deterministic results independent of the number of nodes participating in the computation, even though intermediate tasks may be executed in an unpredictable schedule. To achieve consistency independent of the order in which operations have been applied we present a new Operational Transformation algorithm, which mitigates the performance loss of introducing determinism with Spawn & Merge. We show that such deterministic processing can scale across a cluster of compute nodes and discuss for which kind of workload the programming model is feasible. Furthermore, for high and low workloads, we evaluate the cost of adding determinism to be 28% and 40% higher than perfect parallel computation.**

***Keywords*-Application-level Determinism; Deterministic Distributed Systems; Reproducible Program Execution; Parallel Program Execution; Operational Transformation**

I. Introduction

Deterministic program execution is a useful tool for testing and debugging parallel and distributed systems, and it ensures that a computation reproduces the same result when repeated multiple times, which is essential e.g. for scientific computations. However, the typical synchronization techniques, which build on semaphores and message passing, are not inherently deterministic, because they are sensible towards timing. Thus, a test with perfect test coverage must trigger each possible interleaving of parallel computations. Since the number of possible execution paths can rise exponentially, a perfect test coverage is often impossible. Furthermore, enforcing a certain interleaving in a test is technically demanding for test-case developers. Hence, the risk of race conditions at runtime remains. Debugging is subject to the same problem. When a bug has been found which is caused by a race condition, it can be difficult to reproduce it for the above-mentioned reasons.

In contrast, a deterministic program execution is by definition free of race conditions, but it can become very costly to ensure that every aspect of a computation is deterministic. For example, in a non-distributed setting either the operating system or a framework must ensure that locks are always acquired in the same order. In the distributed case, messages must be ordered deterministically, to ensure that program execution is not influenced by jitter on the network. While this is technically possible [1], such systems do not scale on large clusters, because the degree of parallelism is limited by the deterministic lock order and message order. Furthermore, such systems cannot dynamically adapt to non-deterministic timing. For example, a process spawns four child processes for some computation and merges their results with its own data structures in a deterministic order. If the first spawned child process is the last one to finish, the parent process must not handle the result of the other child processes on a first-come-first-serve basis. Instead, it must wait, because it has to stick to its deterministic order. Thus, determinism can cause waiting cycles as the possibility to dynamically adapt program execution is limited.

Our work aims at gaining the advantages of determinism, but without paying the performance cost caused by full determinism. For this purpose we introduce the concept of *application-level determinism*. We assume that an application is structured as a task hierarchy and uses the synchronization primitives Spawn & Merge [2]. We ensure that every task is eventually executed with the same input and yielding the same output, but we allow for out-of-order execution of these tasks. Hence, neither lock-order nor message-order is deterministic. Thus, from an application programmer's point of view, all application logic is executed deterministically. From a system's point of view, however, tasks are distributed non-deterministically on nodes in a cluster. This allows us to exploit all parallelism offered by the application logic while yielding reproducible application-level executions, even if the execution environment changes (e.g. from a supercomputer to a personal computer for debugging purposes).

The difficult aspect of dynamically scheduled tasks is that a parent task sees the completion of its children in a non-deterministic order. Therefore, we have developed a technique based on the concept of *Operational Transformation* [3]. This ensures that the data structures of the parent task always end up in the same state, no matter in which order the results of the child tasks have been merged in.

We analyzed our system for various scenarios to demonstrate that deterministic systems can be built to scale and that the dynamic scheduler and out-of-order merging enhance the system performance. Of course, some applications lend themselves better to the Spawn & Merge paradigm than

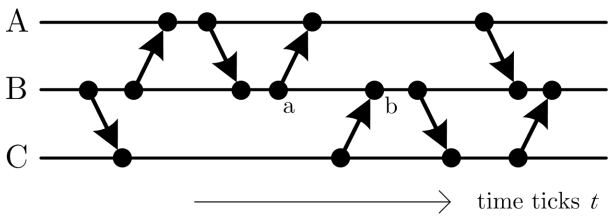


Fig. 1. Example schedule of accessing a data object shared at B.

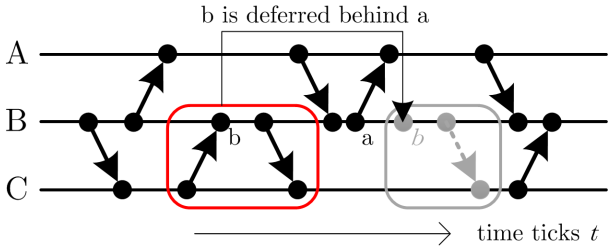


Fig. 2. Full determinism introduces waiting times.

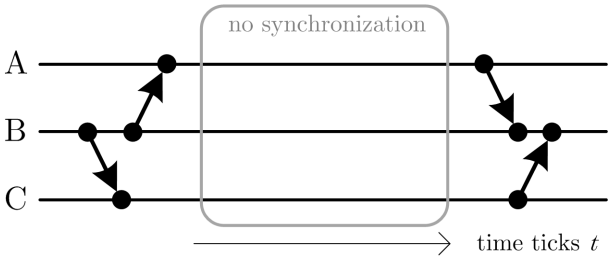


Fig. 3. Remove synchronization points to avoid waiting times.

others. Therefore, we analyzed in which settings our approach provides the greatest benefit.

The **contributions** of this paper are: 1) the notion of application-level determinism, 2) a programming model for deterministic, distributed computation, 3) an efficient Operational Transformation control algorithm for out-of-order merging of data.

The paper is structured as follows: In the next section we define the term application-level determinism and discuss why it allows us to better exploit parallelism. In Section III we describe the application scenario we use to illustrate examples within our paper. In Section IV we show that deterministic distributed systems can be conveniently implemented by applying the Spawn & Merge paradigm to distributed systems. In Section V we discuss how the out-of-order merging is realized. In Section VI we analyze our systems with measurements in various settings. Finally, we conclude the paper and discuss further related research topics in Sections VII and VIII.

II. Application-level Determinism

In a *fully deterministic* program flow each program statement is executed in the same order. As the program transits the same internal states for every execution, this yields not only a reproducible end result but also reproducible intermediate results. The disadvantage of this approach is the cost for achieving full determinism, especially for distributed applications, which will lose the bulk of their potential for parallel computation [1]. The reason for limited parallelism is the enforcement of a deterministic execution schedule, which is most likely not optimal. Consider the example in Fig. 1, in which two processes A and C access a data object shared at process B. The actual execution varies due to unpredictable CPU or network timings and may lead to the schedule shown in Fig. 2. However, full determinism defers the early execution of C until A has finished its data access, which causes waiting times. The runtime enforces the happens-before relations depicted in Fig. 1, since it lacks the knowledge whether a different ordering of events would yield the same result. The potential slowdown of full determinism increases with the number of happens-before relations between processes.

We argue that full determinism is not necessary to achieve reproducible intermediate results and propose a more coarse-grained approach called *application-level determinism*. With application-level determinism, each program run will execute the same set of tasks or function calls with identical inputs and outputs. The execution order and timing of each function call does not need to be identical in each program run. This allows the runtime to schedule tasks with a higher degree of freedom and to exploit parallelism more efficiently.

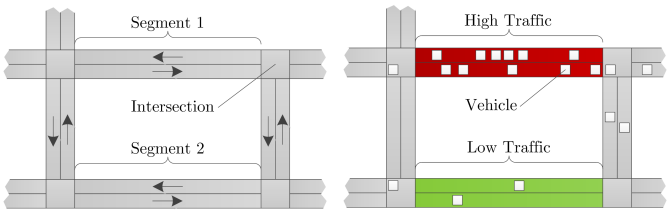
One of the measures for efficient application-level determinism is to reduce the number of happens-before relations and dependencies between processes. This is achieved by removing synchronization points, as shown in Fig. 3. Instead of locking shared data objects, processes operate concurrently on their own copy of data. Data-structures track write operations as data changes, which can be merged together with another process. This allows for concurrent writes without additional synchronization points for locking. With this approach, the cost of synchronized data access is replaced with the cost of merging data changes. However, merging can take place in any order once a function has finished without introducing additional waiting times. We present an algorithm for out-of-order merging, which guarantees not only a consistent but also a deterministic result.

III. Example Scenario

In this paper we consider road traffic simulation for testing advanced driving-assistance systems (ADAS) as an example application scenario. Simulations are used to ease the development and validation of ADAS, since testing requires driving several million kilometers to validate the system [4].

One way to decompose the simulation is an agent-based approach where every car in the simulation is simulated on its own. Since the agent simulation requires information about other agents (i.e. cars) within its proximity, many shared data accesses are necessary to synchronize the agents resulting in limited parallelism as stated in Section II.

Another way to decompose the traffic simulation into smaller computational units is to divide the road system model into segments that can be simulated in parallel [5], e.g.



(a) Road Segments by Intersections. (b) Traffic Jam vs. Empty Segment.

Fig. 4. Road segments and different workloads.

between intersections or traffic lights (see Fig. 4a). Such road segments typically differ in computational complexity subject to the number of vehicles in a segment (see Fig. 4b). The workload shifts to adjacent segments as vehicles move in the simulation. For a segment-based simulation it is sufficient if the segment simulation process has access to the data for its own segment, as well as the data of adjacent segments.

Copying this data for each simulation process removes the shared data access necessity and allows for a higher degree of parallelism since during one simulation step no synchronization is needed. However, conflict resolution of data modifications (e.g. cars moving to adjacent segments) after simulation steps is more expensive, which is a trade-off between higher parallelism and higher merge cost and is discussed in Section VI. The programming model Spawn & Merge eases the development of this type of applications. In the next section we apply it to deterministic distributed systems.

IV. Distributed Spawn & Merge

Spawn & Merge has been introduced as a programming model for deterministic synchronization of multi-threaded programs [2]. In this paper, we argue that the programming model can also be used to ease the development of deterministic distributed systems. The programming model introduces the synchronization primitives `Spawn()` and `Merge()`. `Spawn()` calls a function as a child *task*. Tasks are executed in parallel and operate on their own copy of input data. Upon completion the output of a task is deterministically merged back into its parent.

Listing 1 shows a (simplified) example of how the traffic simulation scenario sketched in Section III can be realized using Spawn & Merge. Once the road segments and their connections are initialized in lines 6 and 8, the simulation loop (line 10) is started. In every loop iteration for every road segment contained in `segments` a simulation task is spawned (line 13) by calling `Spawn()` with the function `SimulateSegment` in combination with the necessary arguments, i.e. the segment to simulate and its adjacent segments. On task-creation the passed arguments are copied for local usage in the tasks to minimize synchronization points caused by shared data access (see Section II). `MergeableSegments` are special data structures that include the logic for being copied and deterministically merged. The `SimulateSegment` tasks are executed in parallel. Finished tasks will finally return their output to their parent task. Once `Merge()` is called by the parent task (line 17), it waits for all child tasks to finish and merges their data-copies into the original data deterministically. The merge has to be explicitly triggered which is necessary for achieving application-level determinism which will be further discussed in Section IV-C. Once all child tasks have been merged, `Merge()` unblocks and the loop starts over.

Listing 1. Simulation Example using Spawn & Merge.

```
1  void SimulateSegment(MergeableSegment* self,
2    MergeableSegment* neighbors[]);
3
4  void mainTask(){
5    MergeableSegment* segments[] =
6      InitSegments();
7    MergeableSegment* neighbors[][] =
8      InitAdjacentSegments();
9
10   for (int simStp=0;simStp<simEnd;++simStp){
11     for (int i=0;i<segments.size();++i){
12       // Spawn one Task for every Segment
13       Spawn(SimulateSegment, segments[i],
14         neighbors[i]);
15     }
16     // Wait for all children to finish
17     Merge();
18   }
19 }
```

A. Copying Data and Operational Transformation

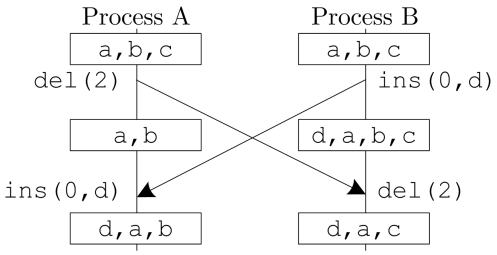
Using traditional concurrency control mechanisms, programmers have to manually synchronize data access to make the program execution deterministic. This can be hard to achieve, since the sequence of lock acquisitions depends on non-deterministic timings. The number of possible locking sequences raises exponentially in complex applications. `Spawn` and `Merge` ease the synchronization of parallel processes, however it necessitates copying of data[1]. Consequently Spawn & Merge requires no locking on the application level, because there is no shared data access and thus less synchronization points. This in turn allows for a higher degree of parallelism.

To resolve conflicting data modifications of the parent task and the child tasks in a deterministic manner Spawn & Merge utilizes *Operational Transformation* (OT) [3]. Systems for concurrent document editing are the prime use cases for OT algorithms. For example, Google Wave [6] used OT and demonstrated that OT can work efficiently and on a large scale.

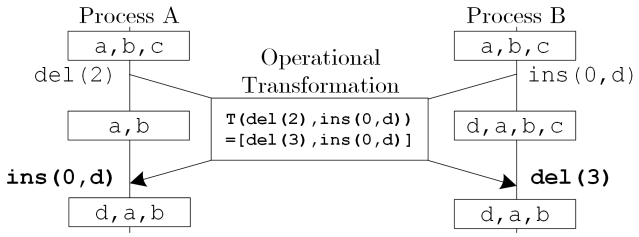
Data structures that shall be merged using OT need to track *operations* (i.e. modifications) performed on them as well as to provide an algorithm for deterministic conflict resolution. This algorithm depends on the data structure only and is independent of the application. Such data structures are in this paper referred to as *mergeable*[2].

[1] A limitation of copying is that non-serializable objects (e.g. sockets) cannot be used when spawning tasks since they cannot be transmitted to another node for execution. This limitation applies to distributed applications in general.

[2] A set of standard data types is provided with the programming model.



(a) Inconsistent results.



(b) Consistent with Operational Transformation.

Fig. 5. Operational transformation example.

Fig. 5 shows an example of a concurrent list modification by two processes A and B. If the operations are applied by the other process without being transformed, the resulting lists will be inconsistent (see Fig. 5a). By applying the Operational Transformation function T the operation `del(2)` is transformed to `del(3)` since the offset of element `c` has shifted due to the insert operation of process B (see Fig. 5b).

Operational Transformation algorithms can be categorized by their *Transformation Properties (TP)* [7]. TP1 algorithms achieve a consistent conflict resolution between two concurrently modified copies. This is because the transformation result depends on the non-deterministic order in which operations are received by the process owning the "master copy". Hence, both copies are consistent, but the state is non-deterministic. TP2 algorithms achieve a consistent and deterministic conflict resolution between any concurrently modified copies.

The use of TP1 or TP2 algorithms both have drawbacks for our approach. When using a TP1 algorithm it is necessary to merge the children in a deterministic order, because the result of the merge depends on this order. However, this means that the parent has to delay the merge of a finished child C until all children that have to be merged before C have also finished and been merged. This can potentially lead to undesired waiting cycles at the parent task. Using a TP2 algorithm, however, is expensive in terms of memory requirement and transformation complexity. Version vectors are necessary to track for every operation which operations of other processes it already knew about [8]. In Section V we present an algorithm that leverages Spawn & Merge specific properties to overcome the stated limitations of typical OT algorithms.

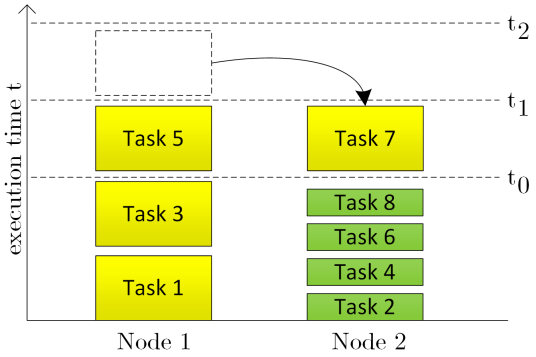


Fig. 6. Rescheduling of task 7.

B. Dynamic Scheduling and Load Balancing

The original Spawn & Merge concept was designed for multi-threaded programs. Thus threads were directly scheduled by the operating system. For the execution of tasks as a distributed system Spawn & Merge needs to be extended with a scheduler. In order to provide distribution transparency, the scheduler is able to distribute tasks without any code annotations by the programmer.

We use a central *dynamic scheduler* for automated task distribution in our approach, since a static scheduler cannot schedule tasks efficiently whose computational complexity varies subject to intermediate runtime results (e.g. moving traffic in the example application in Section III). Since even a dynamic scheduler is not aware of the actual execution time of a newly spawned task, several computationally intensive tasks might be assigned to one compute node, while another node receives lightweight tasks. To cope with this imbalance, we include a workstealing algorithm to schedule efficiently despite differing task complexities and execution environments. The approach is illustrated in Fig. 6, where Node 2 steals a pending task from Node 1, decreasing the overall execution time from t_2 down to t_1. This way applications are able to cope with poor schedules by redistributing tasks automatically if necessary.

C. Application-level Determinism in Spawn & Merge

The program execution of a Spawn & Merge based application is not fully deterministic. This is because the task scheduling depends on unpredictable timings and concurrency, e.g. the order of incoming tasks. To reduce the amount of synchronization points and thus increase the degree of potential parallelism, passed arguments copied for every task instead of using shared data access.

To meet the guarantees of application-level determinism, all spawned functions need to receive the same input data (i.e. function arguments), perform the same computations within the function body and yield the same results for every execution (as stated in Section II).

When a function is spawned using `Spawn`, the arguments and the function identifier are serialized and sent to a compute node for execution. Thus the current state of the arguments is preserved as a snapshot and even if the task is not executed

immediately, it still has the intended starting state, independent from changes to the data structures in the meantime. Thus the function inputs are deterministic, as long as `Spawn` is called in a deterministic manner.

The function body of a task is executed sequentially in its own thread and is therefore deterministic[3]. Spawning and merging are the only synchronizations with other tasks that are running concurrently. They are also the interface when the deterministic function body communicates with the non-deterministic Spawn & Merge framework internals. For a deterministic execution of the function body it is crucial that `Spawn` and `Merge` behave in the same way, independently of non-deterministic internals of the framework. `Spawn` calls do not affect the function body determinism since they are not able to modify data within the function body. `Merge` however receives the results of its child tasks in an arbitrary order, since the child task execution (scheduling) and return timings are non-deterministic. To achieve a deterministic outcome for `Merge`, a *deterministic conflict resolution* of concurrent task results is necessary. As stated in Section IV-A, we utilize Operational Transformation to resolve conflicting modification of copies. The logical order used for conflict resolution in the transformation algorithm is determined by the (deterministic) spawning order of child tasks. Thus, conflicts are always resolved deterministically with the same logical ordering.

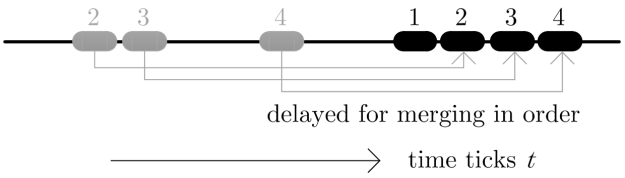
Finally, since the function input is deterministic and the function body is deterministic, the function output (i.e. the operations performed on copies) is deterministic, too.

For task hierarchies this means that the merging parent task will also be deterministic as long as every child task is deterministic. Thus the determinism propagates upwards in the task hierarchy, which implies that all observable function calls are deterministic and that the overall program execution is application-level deterministic.

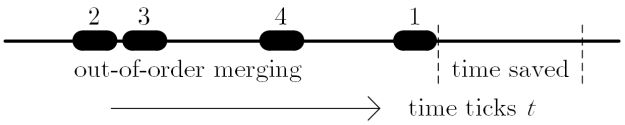
V. Out-Of-Order Merge

To avoid the TP2 Operational Transformation costs, the original Spawn & Merge concept enforced that finished child tasks would always be merged (i.e. the transformation applied) in the same deterministic order for every execution and thus would also be deterministic when using a TP1 algorithm. This, however, can result in waiting time at the merging parent task since it has to wait for the next task to merge before the potentially computation-intensive OT algorithm can proceed, even though other tasks may already have finished. For example in Fig. 7a the merge order is $< 1, 2, 3, 4 >$. The tasks 2, 3 and 4 finished earlier than task 1 and thus merging of tasks 2–4 has to be delayed until task 1 finishes. Even though TP1 OT algorithms can be used in distributed Spawn & Merge, the ability to merge out-of-order, like in TP2 algorithms, would mitigate waiting induced performance losses to a certain degree (see Fig. 7b).

We argue that to realize application-level determinism and an out-of-order merge we do not need an algorithm that



(a) Waiting cycles due to delaying for merging order.



(b) Time saving due to out-of-order merge.

Fig. 7. Merging order.

provides all properties of TP2 algorithms. With the Spawn & Merge programming model it is only necessary to merge modifications by child tasks back into the parent task, not into any other task. Hence, the merge graph is a tree, whereas TP2 supports arbitrary merge graphs. In this section we present a novel merge algorithm for lists that is more efficient than typical TP2 algorithms. Our algorithm can perform out-of-order merging of modified copies while still yielding deterministic merge results. The algorithm leverages Spawn & Merge specific properties. First, in Spawn & Merge a task can only merge with its direct children (and thus with its parent), i.e. a task always knows all children and their deterministic spawning order[4] and at which version they have been spawned. Thus, it is only necessary to track a single version counter for each spawned data structure instead of expensive version vectors. Second, the order for merging is also deterministically provided by the used `Merge` primitive, i.e. the spawning order of tasks for basic merge.

The basic idea of the algorithm is twofold. First, the transformation algorithm receives an ordered list of the child tasks that are currently being merged. Thus the algorithm is able to ignore element modifications that it should not see yet (since the operation was applied by a child task that had a lower order[5]). Second, we introduce tombstoning [8] for deleted elements since a lower order task might have deleted an element that a higher order task (merged later) also intended to delete. Without tombstoning the higher order task would not be capable of finding the element to delete anymore. Furthermore, we switch from a list of operations to a list of elements that contain the operations (i.e. modifications) performed. These elements are 4-tuples $(element, version, creatingEntity, tombstoned)$, where $element$ is the value of the element, $version$ is the version in which the element has been created, $creatingEntity$ identifies which task created the element and $tombstoned$ is set to the version in which an element has been deleted.

The transformation algorithm is shown as Algorithm 1. The

[3]I.e. if no inherently non-deterministic functions are called, e.g. RNGs or non-deterministic user inputs.

[4]Since the spawning order directly depends on the static program code.

[5]A lower order in this context applies to a later merge position in the order list. A higher order applies to an earlier merge position.

Algorithm 1 Deterministic Out-Of-Order Merge

```
1:  function TRANSFORMATION(pList, cList, order)
2:      var
3:          cTaskId          ▷ ID of currently merged child
4:          lOrderList       ▷ IDs with lower order than child
5:          cIdx                                ▷ cList index
6:          pIdx                                ▷ pList index
7:          steps  ▷ Steps to modified child element position
8:          element                  ▷ Stores created element
9:      end var
10:     lOrderList ← LowerOrder(cTaskId, order)
11:     pIdx ← 1                            ▷ Start index of pList
12:     cIdx ← 1                            ▷ Start index of cList
13:     while cIdx ≤ |cList| do
14:         steps, cIdx ← NextModifiedElement()
15:         pIdx ← FindPosInPList(steps)
16:         if IsTombstone(cList[cIdx]) then
17:             pIdx ← NextKnownElement()
18:             if IsNotTombstone(pList[pIdx]) then
19:                 MakeTombstone(pList[pIdx])
20:             end if
21:         else     ▷ Modified element is an inserted element
22:             pIdx ← ResolveConflicts(lOrderList)
23:             element ← CreateElement(cTaskId)
24:             InsertElement(pIdx, element)
25:         end if
26:         pIdx ← pIdx + 1
27:         cIdx ← cIdx + 1
28:     end while
29: end function
```

transformation function takes three parameters: the parent list $pList$, the modified list $cList$ of the currently merged child, as well as the logical order of children $order$ for conflict resolution, determined by the calling `Merge` primitive.

Initially, the algorithm stores the IDs of all tasks that have a lower order than the currently merged task in $lOrderList$ (line 10) and initializes the list indexes $cIdx$, and $pIdx$ to the first elements of $cList$ and $pList$. The while-loop (line 13) runs as long as the child list index did not reach the end of $cList$. Within the loop, the algorithm searches for the next modified child element in the child list $cList$. This is done by skipping all child elements that were either not modified or elements that have been created and already been deleted by the child. The search yields two outputs, the index $cIdx$ of the modified element in list $cList$, as well as $steps$ (line 14). $step$ states how many element steps over elements that were known to the child at spawn-time have to be performed on the parent list. If there is no further modification in $cList$, the algorithm will exit the while-loop. $steps$ is used in line 15 to advance the index $pIdx$ until $pIdx$ points directly at the first position of $pList$ that potentially conflicts with the upcoming modified element $cList[cIdx]$.

If the modified child element is a tombstone (line 16), then the element has been deleted by the child. In this case the parent list index $pIdx$ is advanced to the next element that was known to the child at spawn-time (line 17). If the resulting element in the parent list $pList$ is not yet a tombstone, then then the element will be converted into a tombstone by setting the tombstone-flag (line 19). Otherwise, nothing is done, since the element has already been deleted.

If the modified child element is an inserted element (line 21), potential insert conflicts will have to be resolved. This is the crucial part for achieving determinism for out-of-order merging. All merging tasks will have to resolve the insert conflict at this position in the parent list the same way, independent of the order in which they are merged (1). Here, $lOrderList$ is used to decide where the new element has to be inserted, by skipping the conflicting elements until either a conflicting element with a lower order is reached (i.e. all elements with a higher order are skipped) or the next element that was already known to the child at spawn-time is reached. For conflicting elements this means that conflicts at one point within the parent list $pList$ always result in a deterministic order from highest order to lowest order, independent of the order they were merged (2). Once the right insert position $pIdx$ is determined, the new element will be created (line 23) and inserted into $pList$ (line 24). Finally the indexes $cIdx$ and $pIdx$ are advanced over the current modification and the while-loop starts over in line 13.

$$x_{k,n} : n\text{-th element known by child}$$
$$x_{c,x} : \text{conflicting element with order } x$$
$$x_{c,2} : \text{element to insert with order } 2$$
$$cList : [\dots, (x_{k,n}), \overbrace{(x_{c,5}), (x_{c,1})}^{pIdx \,\rightarrow\, x_{c,2}}, (x_{k,n+1}), \dots] \quad (1)$$
$$cList : [\dots, (x_{k,n}), \overbrace{(x_{c,5}), \overbrace{(x_{c,2}), (x_{c,1})}^{pIdx}}^{\text{deterministic order}}, (x_{k,n+1}), \dots] \quad (2)$$

Since both lists, $pList$ and $cList$ are only traversed once, this algorithm enables the transformation to be applied with a complexity of $\mathcal{O}(n)$ where n is the list size. However, the complexity of inserting and deleting elements in the list at runtime also becomes $\mathcal{O}(n)$, since the lists have to be traversed to skip tombstoned elements, to determine the right position to insert or delete an element in the list. Since we have to traverse the complete lists, our algorithm performs poorer for larger lists than classic Operational Transformation algorithms that only need to consider the operations performed. In contrast, most classic Operational Transformation algorithms have a complexity of $\mathcal{O}(m^2)$ where m is the number of operations, which has to be performed completely at the task performing the merge (i.e. the parent task). Our proposed algorithm mitigates the $\mathcal{O}(n)$ list modification penalty since this is still running on the child tasks and thus benefits of being executed in parallel.

VI. Analysis and Evaluation

We have implemented the proposed system in C++11 based on MPICH [9] to show practical feasibility on top of a message-passing middleware. We ran the evaluation on a cluster of 10 virtual machines[6] to analyze its applicability to different application scenarios. The analysis is twofold: First, we use a parameterized synthetic benchmark to show that the distribution with Spawn & Merge achieves a reasonable performance compared to a hypothetical ideal distributed computation. Second, we analyze for which application types Spawn & Merge is a feasible choice by evaluating the communication and computational overhead for achieving application-level determinism.

A. Scalability

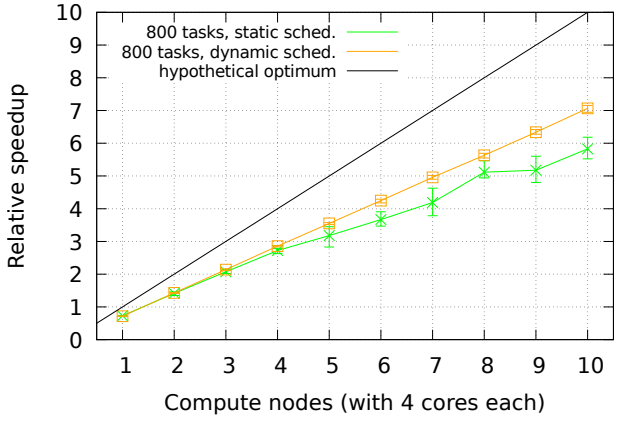
The synthetic benchmark consists of a CPU-bound computation, where the parameter l determines the amount of artificial workload in CPU cycles. A parent task spawns k tasks and distributes the workload across n compute nodes with c CPU cores each. In reality, tasks vary in computational complexity. We take this into account by classifying these k tasks as *low* ($0.65k$ tasks), *medium* ($0.3k$ tasks, 5 times the workload of *low*) and *high* ($0.05k$ tasks, 30 times the workload of *low*). The tasks' starting order is chosen by a statically seeded PRNG and is therefore deterministic. After a child task has finished its workload share, it will be merged by the parent.

a) Measurement: We execute the computation for two different workloads (low and high load[7]) with a single-threaded implementation on one node to acquire a reference computation time r. A hypothetical perfectly distributed application using $n \cdot c$ CPU cores requires at least $\frac{r}{n \cdot c}$ time units. Using this fraction, we have an ideal reference value to compare our measurements to.

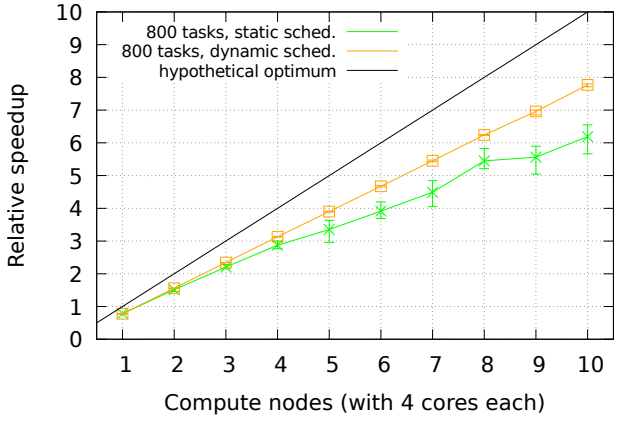
For the same workloads, we measure the performance of our proof of concept implementation with varying numbers of compute nodes from $n = 1$ to $n = 10$ and $k = 800$ tasks. To be able to evaluate the dynamic scheduler, we also measure execution times with a static round robin scheduler.

Fig. 8 shows the average speedup of five measurement runs for $n = 1$ to $n = 10$ (with $c = 4$ CPU cores per compute node) compared to the ideal reference. The measurement results of both workloads (Fig. 8b and Fig. 8a) show the characteristic linear graph of a scalable system. Compared to the ideal reference, our proof of concept requires 40% more time for a low workload and 28% for a high workload. This difference is expected: the relative overhead of our approach diminishes with an increased workload. In general, the ideal reference is unobtainable due to unparallelizable parts of the code.

For one node, static scheduling is faster than dynamic scheduling. For example, we observed an execution time of 1301.2 seconds for static scheduling and 1320.3 seconds for a dynamical schedule (for $n = 1, k = 800$). This overhead



(a) Low workload l.



(b) High workload l.

Fig. 8. Ideal Distributed Computation vs. Spawn & Merge

is caused by the dynamic scheduler tracking capacity of all nodes which is not redeemable with clever scheduling due to a lack of nodes (see Section IV-B).

As soon as more than one node is used, dynamic scheduling becomes faster than static scheduling. However, the gain varies since the static scheduler results have a large deviation. Schedules created by it can be favorable or distribute actual workload unevenly despite distributing tasks round robin. While such an assignment can also occur using a dynamic scheduler, it is able to reschedule tasks (see Section IV-B). Therefore, the deviation of our dynamic scheduler is significantly lower.

We conclude that our proposed approach scales with a reasonable static overhead. In contrast to static scheduling, our dynamic scheduler achieves shorter execution times with lower spread across repeated runs.

b) Influence of Dynamic Scheduling and Merge Order: We perform additional measurements to evaluate the impact of dynamic scheduling together with out-of-order merging in detail. We use the afore described setup ($n = 10, c = 4$, $k = 800, l =$ *high*, referred to as *scenario 1* here) and add a parameter stating whether the out-of-order merge optimization is in use. Since the execution time depends highly on the actual schedule, we run four executions with different seeds.

[6] Amazon EC2 *m4.xlarge* as of Feb. 2016. 4 × 2,4-GHz Intel Xeon E5-2676 v3 (Haswell), 16 GByte Memory, 750 Mbit/s.

[7] In the context of this subsection low and high workload correspond to $l = 4.064 \cdot 10^{10}$ and $l = 9.7536 \cdot 10^{12}$ CPU cycles.

scheduling	merge	Execution time in s			
		scenario 1		scenario 2	
static	fixed	163.9 ± 10.3		188.4	
	out-of-order	164.0 ± 10.2	$(+0.07\%)$	173.4	(-8.0%)
dynamic	fixed	130.5 ± 0.2	(-25.5%)	176.8	(-6.2%)
	out-of-order	130.7 ± 0.1	(-25.4%)	160.5	(-14.8%)

TABLE I
IMPACT OF SCHEDULING AND MERGE ORDER.

Furthermore, we create a second scenario (*scenario 2)* with a highly uneven workload distribution. 763 tasks are classified as *low*, 36 as *medium* (5 times the workload of *low*) and one task as *high* (50 times the workload of *low*). As stated in Section V, out-of-order merge is only beneficial if there is a long running task which its parent has to wait for. To resemble such conditions, the longest running task has to be merged first. That way, it is highly likely that merges from other tasks have to wait if out-of-order merge is disabled (cf. Fig. 7a,7b). To be able to measure this effect more clearly, we significantly increase the modifications on shared data structures.

Table I shows the average execution times of 5 measurement runs and standard deviations. As shown in the previous section, scenario 1 measurements profit from dynamic scheduling; in total, a speedup of 25.4% can be achieved. However, switching from a fixed merge order to an out-of-order merge does not have a positive impact. Instead, it causes a small decrease in performance of 0.07% for static and 0.15% for dynamic scheduling. Since these are within the standard deviation, we do not consider this to be significant but merely an outlier.

In contrast, in scenario 2 out-of-order merging causes a speedup of 8.0% for static and 9.2% for dynamic scheduling. If both optimizations are employed, a total speedup of 14.8% occurs. The moderate effect of enabling dynamic scheduling (in comparison to scenario 1) results from the single big task in scenario 2. This is because the overall execution can never be faster than the single execution of any task, even with a perfect schedule.

The results in Table I show that the influence of out-of-order merge highly depends on the task distribution and merge order. Merging out-of-order enables the application to prepone the merge for early tasks instead of delaying it. Thus, the positive effect can only apply for scenarios where merge computations are not negligible and where there is a possibility for tasks to finish significantly earlier than other tasks (as in scenario 2). However, we did not find a significant disadvantage of enabling out-of-order merge also for scenarios in which these conditions are not met (e.g. scenario 1).

B. Feasibility of Distributed Spawn & Merge

To analyze the applicability of Spawn & Merge to various application types, we consider certain application characteristics as parameters of another benchmark. The applicability depends on the expected amount of conflicts to resolve (application-dependent) and the costs for conflict resolution (data structure-dependent). In this benchmark, a parent tasks spawns a child task with a mergeable list consisting of s integers. Both tasks perform m concurrent modifications on this list (i.e. insert and remove operations) to assess overhead between parent and child task induced by Operational Transformation. In addition, the child processes a workload of l.[8]

list size s	operations m	Execution time in s (OT ratio)			
		$l =$ high		$l =$ low	
empty	50	103.4	(0.1%)	51.5	(0%)
	50k	104.1	(0%)	52.5	(0%)
	200k	115.7	(0.1%)	63.9	(0%)
1k items	50	103.1	(0%)	51.6	(0%)
	50k	104.3	(0%)	52.4	(0%)
	200k	116.0	(0%)	64.4	(0%)
1M items	50	104.2	(0%)	52.4	(0%)
	50k	202.2	(4.3%)	148.4	(3.5%)
	200k	604.5	(5.9%)	544.6	(5.6%)

TABLE II
SPAWN & MERGE PERFORMANCE (WITH RELATIVE AMOUNT OF OT).

Table II shows the overall execution time, which includes the communication and computational overhead depending on the parameters. The relative costs of Operational Transformation are given in brackets after each execution time. The initial list size s has little impact on the execution time as long as few modifications are made (at most 1.7% for low workload). This overhead is caused by serialization and network induced latency.

Overhead caused by data structure modification depends highly on the list size. Consider an empty list: Under high workload the execution time increases between $m = 50$ and $m = 200$k by 12% while the execution time for a list with one million items increases by 480% under these circumstances.

However, only 5.9% of this time is used for Operational Transformation. Instead, we could identify the underlying data structure `std::vector` as the primary reason for this increase: every insert or remove operation has a complexity of $\mathcal{O}(n)$. Hence, modifications of big data structures are the main cause of slowdowns in our measurements. The Operational Transformation algorithm is only responsible for up to 5.9% of the execution time.

VII. RELATED WORK

We now review prior work, which falls either in the category of deterministic program execution or in the category of conflict resolution.

A. Local Determinism

Various research has been done on deterministic execution of parallel programs running on a single host. *DejaVu* [10] enables the deterministic replay of Java programs by modifying the Java VM to capture a logical thread schedule, which is enforced when replaying the program. *Samsara* [11]

[8]In the context of this subsection low and high workload correspond to $l = 3.05 \cdot 10^{11}$ and $l = 6.1 \cdot 10^{11}$ CPU cycles.

also enables the deterministic replay of program executions. It utilizes the hardware-assisted virtualization extensions of commodity processors to reduce the overhead of logging the memory read-set and write-set necessary for replay. *CoreDet* [12] is a compiler and runtime system for the deterministic execution of arbitrary multithreaded C and C++ programs, which serializes threads by using a deterministic scheduler. Using the runtime system *dThreads* [13] each process has private and shared views of shared memory. Changes in the shared memory are ordered deterministically and applied at deterministic synchronization points to enforce the deterministic execution of arbitrary programs. The software framework *Kendo* [14] enforces a deterministic order of synchronization operations to achieve a race-condition free execution. *Grace* [15] is a runtime system for fork-join programs that copes with concurrency errors by committing all updates on shared memory deterministically and by executing threads in program order to achieve a deterministic schedule. The *deterministic Operating System* (*dOS*) [16] enables determinism for arbitrary multithreaded programs. Threads and processes are executed as single deterministic units. Internal non-determinism is eliminated by using a deterministic scheduler and external non-determinism is recorded. Hence, the system can be replayed deterministically.

Deterministic Parallel Java (DPJ) [17] uses an annotation based type and effect system that divides the heap into hierarchical regions and manages read and write access to these regions for each task. *MELD* [18] integrates DPJ as a deterministic language within a deterministic execution system built upon CoreDet to combine the strengths of both components. *Concurrent Collections* [19] is a programming model for deterministic parallel execution. It features a distinction between application logic and parallelization on a language level. The task of merging data is left to the application developer. *Cilk++* [20] uses hyperobjects to reduce concurrently modified shared data structures deterministically. This is similar to mergeable data structures used in our approach.

Utilizing functional programming languages for deterministic parallel computing is another interesting approach due to their inherent absence of side effects. Coutts and Löh [21] demonstrated its applicability with a reasonable performance compared to an OpenMP implementation written in C.

All of these techniques cope with multithreading but are limited to local execution, whereas our approach targets deterministic distributed systems.

B. Distributed Determinism

Determinator [22] is a deterministic proof-of-concept OS that enforces deterministic execution of arbitrary programs written in any existing language. Determinator is able to transparently distribute the computation on a cluster of homogeneous compute nodes by space migration. *DDOS* [1] is a distributed system build upon dOS. DDOS delays the delivery of network messages to ensure a deterministic distributed execution.

Determinator and DDOS offer deterministic execution of arbitrary programs, suitable e.g. for debugging purposes. The performance penalty of this fully deterministic approach is about one order of magnitude [1]. We argue that the cost of full determinism is not required for reproducibility of results. With a programming model tailored towards application-level determinism, applications behave deterministically with a smaller performance penalty.

C. Conflict Resolution

Pingali et al. [23] introduce an approach that abandons determinism to allow different but valid intermediate results (don't-care non-determinism). Compared to our notion of *application-level determinism* this potentially increases the degree of parallelism but does not provide a deterministic program execution.

Burckhardt et al. [24] introduced *isolation types*, which support application-specific deterministic merge functions. The details, however, are left to the programmer. Unlike in Spawn & Merge, the approach does not provide an operation centric view on concurrently modified data structures, which complicates writing an intention preserving merge function.

VIII. Conclusion

In this paper we introduced application-level determinism as a more efficient alternative to full determinism. An application-level deterministic application has several benefits for developers of distributed applications. First, the application yields reproducibility and determinism for observable function calls despite concurrency and unpredictable timings of the invocations. Second, reducing synchronization points and permitting non-deterministic execution order for function calls enables a higher degree of parallelism. This allows for dynamic scheduling of function calls to benefit from all available resources (e.g. compute nodes in a compute cluster) without losing the application-level determinism guarantees. Thus, the distributed application remains application-level deterministic, even if the execution environment changes, e.g. from a supercomputer to a personal computer for debugging purposes, and thus furthermore allows the application to scale by adding further computation resources without sacrificing determinism. By building on the Spawn & Merge programming model we ease the development of scalable deterministic distributed applications.

We introduced a new Operational Transformation algorithm for use in Spawn & Merge. The algorithm properties are settled between the properties of TP1 and TP2 algorithms, since it is capable of performing out-of-order merges (TP2) while removing the necessity for version vectors. To achieve this the algorithm leverages Spawn & Merge specific properties. First, the logical merge order for tasks is always known from program code and thus is deterministic. Second, tasks do not need to be able to merge with any other task, but only with its direct child tasks or its parent.

The analysis of our system has shown that distributed Spawn & Merge scales with a reasonable static overhead

depending on the application type and its characteristics. We measured the overhead to be 40% for low workloads and 28% for high workloads. The dynamic scheduling and the out-of-order merge achieve reasonable performance gains of up to 25% depending on application characteristics, since not every application will benefit from these mechanisms to the same amount. Furthermore, we evaluated the cost of adding determinism to a distributed system by measuring the ratio of Operational Transformation at runtime for several scenarios. For the results with he highest runtime only 5.9% of the time was used for Operational Transformation.

The entire approach is only useful when a programmer can conveniently implement his application logic in our paradigm and if all the heavy lifting is hidden from him. We developed a C++ library which takes care of spawning tasks and merging their output out-of-order while retaining deterministic results. For future work, we will extend our library with more generic data types and release it to the community. Furthermore, we plan to increase the performance of large data types.

REFERENCES

[1] N. Hunt, T. Bergan, L. Ceze, and S. D. Gribble, "Ddos: Taming nondeterminism in distributed systems," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 499–508. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451170

[2] C. Boelmann, L. Schwittmann, and T. Weis, "Deterministic synchronization of multi-threaded programs with operational transformation," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, May 2014, pp. 381–390.

[3] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '89. New York, NY, USA: ACM, 1989, pp. 399–407. [Online]. Available: http://doi.acm.org/10.1145/67544.66963

[4] A. Belbachir, J.-C. Smal, J.-M. Blosseville, and D. Gruyer, "Simulation-driven validation of advanced driving-assistance systems," *Procedia - Social and Behavioral Sciences*, vol. 48, pp. 1205 – 1214, 2012, transport Research Arena 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877042812028327

[5] K. Nagel and M. Rickert, "Parallel implementation of the transims micro-simulation," *Parallel Computing*, vol. 27, pp. 200–1, 2001.

[6] T. Weis and A. Wacker, "Federating websites with the google wave protocol," *IEEE Internet Computing*, vol. 15, no. 3, pp. 51–58, 2011.

[7] C. Sun and C. Ellis, "Operational transformation in real-time group editors: issues, algorithms, and achievements," in *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. ACM, 1998, pp. 59–68.

[8] G. Oster, P. Molli, P. Urso, and A. Imine, "Tombstone transformation functions for ensuring consistency in collaborative editing systems," in *Collaborative Computing: Networking, Applications and Worksharing, 2006. CollaborateCom 2006. International Conference on*, Nov 2006, pp. 1–10.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, pp. 789–828, Sep. 1996. [Online]. Available: http://dx.doi.org/10.1016/0167-8191(96)00024-5

[10] J.-D. Choi and H. Srinivasan, "Deterministic replay of java multithreaded applications," in *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, ser. SPDT '98. New York, NY, USA: ACM, 1998, pp. 48–59. [Online]. Available: http://doi.acm.org/10.1145/281035.281041

[11] S. Ren, L. Tan, C. Li, Z. Xiao, and W. Song, "Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016, pp. 551–564. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/ren

[12] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "Coredet: A compiler and runtime system for deterministic multithreaded execution," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 53–64. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736029

[13] T. Liu, C. Curtsinger, and E. D. Berger, "Dthreads: Efficient deterministic multithreading," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 327–336. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043587

[14] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: Efficient deterministic multithreading in software," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 97–108. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508256

[15] A. P. Black, K. B. Bruce, M. Homer, and J. Noble, "Grace: The absence of (inessential) difficulty," in *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2012. New York, NY, USA: ACM, 2012, pp. 85–98. [Online]. Available: http://doi.acm.org/10.1145/2384592.2384601

[16] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble, "Deterministic process groups in dos," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924956

[17] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A type and effect system for deterministic parallel java," *SIGPLAN Not.*, vol. 44, no. 10, pp. 97–116, Oct. 2009. [Online]. Available: http://doi.acm.org/10.1145/1639949.1640097

[18] J. Devietti, L. Ceze, and D. Grossman, "The Case For Merging Execution- and Language-level Determinism with MELD," in *Workshop on Determinism and Correctness in Parallel Programming w/ International Conference on Architectural Support for Programming Languages and Operating Systems (WoDet w/ ASPLOS)*, 3 2012. [Online]. Available: http://sampa.cs.washington.edu/papers/wodet3-paper3.pdf

[19] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşırlar, "Concurrent collections," *Sci. Program.*, vol. 18, no. 3-4, pp. 203–217, Aug. 2010. [Online]. Available: http://dx.doi.org/10.1155/2010/521797

[20] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and other cilk++ hyperobjects," in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 79–90. [Online]. Available: http://doi.acm.org/10.1145/1583991.1584017

[21] D. Coutts and A. Löh, "Deterministic parallel programming with haskell," *Computing in Science & Engineering*, vol. 14, no. 6, pp. 36–43, 2012.

[22] A. Aviram, S.-C. Weng, S. Hu, and B. Ford, "Efficient system-enforced deterministic parallelism," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924957

[23] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 12–25. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993501

[24] S. Burckhardt, A. Baldassin, and D. Leijen, "Concurrent programming with revisions and isolation types," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 691–707. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869515