Digital Object Identifier: 10.1109/IPDPSW.2014.50

URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6969413

Deterministic Synchronization of Multi-Threaded Programs with Operational Transformation

Christopher Boelmann, Lorenz Schwittmann, Torben Weis
Distributed Systems Group
University of Duisburg-Essen
Duisburg, Germany
Email: firstname.lastname@uni-due.de

***Abstract*—Today's mainstream programming language concepts originate from a time when processes were executed in a single thread and the outcome of computation was deterministic. To deal with multi-threaded execution synchronization mechanisms have to be used to restrict parallel execution to a point where the program produces correct results for all possible interleaving executions. This is constantly leading to deadlocks and race conditions, i.e. undesired non-deterministic behavior.**

In this paper, we propose a new set of synchronization primitives, *Spawn* and *Merge*, that yield deterministic program execution for multi-threaded programs. This means that there are no race conditions when using this synchronization technique and deadlocks can be avoided right away. Concurrent access to data structures is resolved using *operational transformation*. Using two example scenarios we show how these synchronization primitives can be used and that they are equivalent to semaphores.

Furthermore, we evaluate our framework by implementing a network simulator. We show that despite a constant overhead, the performance is comparable to using standard synchronization primitives while yielding deterministic results.

***Keywords*-Deterministic Synchronization; Parallel programming models; Spawn and Merge; Operational Transformation**

I. Introduction

Multi-threaded programming has always been difficult using imperative programming languages, because different threads worked on the same set of data structures. The annotation of classes or methods with a `synchronized` keyword (Java), or a `lock` statement (C#) has reduced possible error sources, but has by no means solved the problems of multi-threaded programming. There is little tool support available to prove that such a program is free of race conditions. Extensive testing can only show that on today's hardware (i.e. with 2 to 8 CPU cores) a program is *most likely* fine. Thus, we believe that locking is a cumbersome extension to imperative programming languages that has been introduced to enable multi-threaded programming with old-style single-threaded programming languages.

In this paper we present our approach to write programs in a way that the result of a concurrent program is always deterministic, i.e. there is a guarantee that concurrent programs always deliver the exact same results, regardless of the number of cores they are executed on. Like Brocchino et. al. [1], who proposed the same *philosophical* approach, we call a parallel program deterministic, if every execution of the program produces the same results and there is no non-determinism caused by concurrent code execution[1]. Especially for enterprise applications results have to be reproducible and therefore determinism is important. But even for applications where determinism is not a requirement it has the potential to significantly simplify debugging: A bug will not appear only in *some* executions of a program. This also allows to achieve a higher confidence in unit tests since it is impossible to miss an erroneous schedule.

However, when user interfaces or I/O with remote computers is required, there is inherent non-determinism caused by the non-deterministic program input. Thus, we allow the programmer to explicitly introduce non-determinism where required. This is the exact opposite to the current approach where non-determinism is the default and the programmer has to work hard to introduce determinism where required.

To achieve determinism despite parallel program execution we devised two synchronization primitives *Spawn* and *Merge*. *Spawn* can be compared to the UNIX system call `fork`, i.e. it *spawns* a new task[2] that can be executed in parallel to its parent task. A newly spawned task gets a *copy* of the data it is supposed to work on. This copying may become expensive. However, clever copy-on-write techniques can be used to mitigate this problem to some degree. Once spawned, the tasks can execute in parallel using their local data copies without any synchronization, i.e. no performance is lost because of lock contention and race conditions are impossible. Depending on the amount of locking avoided, this will make up for the cost of initial/on-write copying of data. At this point we want to emphasize, that the optimization goal of our work is to build *correct*

[1]Non-determinism resulting from calling non-deterministic functions (e.g. `Random()`) is not tackled by our approach.

[2]We use the term *task* instead of thread, because a task encapsulates additional information, e.g. the data structure copies. There is not necessarily a one-to-one mapping between threads and tasks, since tasks may also be scheduled to be executed on a pool of threads.

concurrent programs and not to squeeze the last few percent of performance out of a multi-core CPU. However, in section III we show that despite a constant overhead the performance of our framework is comparable to a regular implementation using standard synchronization primitives.

Eventually, the parent waits for its children to finish and it *merges* their results with its own. In UNIX, there is no *Merge* counterpart to `fork`. This is because the operating system knows how to copy memory, but it cannot know how to merge the divergent copies created by the child processes, because this is application specific. When the tasks are finished, each task may have produced its own version of the data structure. To merge these together we use *operational transformation* [2] which means that each task has to record the operations applied to its data structures. For example, a list records operations of the form `ins(0,obj)` or `del(1)`. Merging takes the concurrent operations and *transforms* them into a *deterministic* non-concurrent sequence of operations. It takes some experience to fully understand how operational transformation based merging works. But in the end, we are able to write deterministic programs which are guaranteed to be free of internal deadlocks and free of race conditions as we will show in section IV. We believe our approach is especially useful in cases where each task primarily performs computations and/or network I/O but in the end changes only small parts of a shared data structure. This is the case for example in scalable web application, distributed enterprise software, and emulation of large networks.

The paper is organized as follows: In section II we present *Spawn* and *Merge*, which enable deterministic execution of concurrent programs by utilizing operational transformation. Furthermore, we provide the reader with two examples for their usage: a simple server software and a simple simulation software. Afterwards, we evaluate our framework to show that performance is comparable to using conventional synchronization primitives. In section IV we show that the expressive power of our approach is equivalent to using semaphores and analyze it with respect to deadlocks. Finally, we summarize and compare related approaches for synchronizing parallel execution of concurrent programs and the problems they address.

II. Spawn and Merge

To illustrate and motivate our approach to thread synchronization, we need to recapitulate how parallelism has been approached before multi-threading became commonplace. On UNIX programmers had to launch multiple processes using `fork` to enable parallel execution of a program. Here it is important to emphasize the difference between *parallel* and *concurrent* execution. The execution of a program is concurrent, if multiple commands *may* be executed at the same time. Parallel program execution means that concurrent parts of a program *are* executed on multiple CPU cores at the same time. Each forked child process had its own dedicated memory which was initially a copy of the parent's memory. This made race conditions impossible due to concurrent memory access, since by default memory was not shared. This yielded deterministic program execution for each process.

In our approach an executing program consists of a set of tasks that form a task hierarchy much like processes in UNIX form a process hierarchy. A task is not completed unless all its children have completed and have been merged with their parent. A task can create a new child task using **Spawn**, which is somewhat comparable to `fork`, because the child task receives a copy of some of its parent's data. The difference between tasks and processes is that tasks are much more lightweight and are therefore cheap to create and to delete. When a child task has completed, the parent must eventually merge back its results with its own data set using **Merge**.

To achieve determinism, the merging happens in a deterministic order, i.e. the parent task (which is deterministic) states in which order it wants to merge its child tasks. In contrast, allowing the merge to happen on a first-to-complete basis (i.e. first child task to complete merges first) can be used to introduce non-determinism explicitly in the system. So far it should be clear that the merge step is essential to our approach. Merging results from two concurrent computations is complex and requires switching from a data structure centric view to an operation-centric view.

A. Operations

To motivate the operation-centric view we turn to an interesting problem of concurrency: how to formally describe the result of a concurrent execution. Imagine two functions f and g that modify a list. For this purpose they take a reference to a list as argument and modify it. Furthermore, a function h executes these functions in parallel and passes both functions a reference to list a as an argument, i.e. both functions can concurrently modify the list a. We write $f(a) \rightarrow a_f$ to describe that f modifies a which in the end becomes a_f, but only if no other task modified a concurrently.

$$f(a) \rightarrow a_f \tag{1}$$

$$g(a) \rightarrow a_g \tag{2}$$

$$h(a) := f(a)||g(a) \rightarrow a_h, \neg\exists x : x(a_f, a_g) \rightarrow a_h \tag{3}$$

Here, in equations (1) to (3), a, a_f, a_g, a_h are all lists. The difficulty is to deterministically express the result of $h(a)$. Due to the concurrent access to a and the resulting non-determinism, there is in general no deterministic function x that could express $h(a)$ in terms of the results that f and g would have produced on their own.

Our approach is to focus on the operations generated by f and g rather than looking at a_f and a_g. We assume that a function computes a list of operations that can be applied to its argument.

$$f(a) \rightarrow ops_f, ops_f(a) \rightarrow a_f \tag{4}$$
$$g(a) \rightarrow ops_g, ops_g(a) \rightarrow a_g \tag{5}$$
$$h(a) := f(a)||g(a) \rightarrow (ops_f, ops_g) \tag{6}$$
$$merge(ops_f, ops_g) \rightarrow ops_h \tag{7}$$
$$ops_h(a) \rightarrow a_h \tag{8}$$

Here the result of h is deterministic since it does not depend on any interleaving of f and g and it is simply a tuple consisting of the results of f and g. The $merge$ function serializes these operations using operational transformation. This produces a new sequence of operations ops_h that can be applied to a to obtain the result a_h of computing $h(a)$. The function $merge$ is a complex second order function, since it reasons about functions, but nevertheless we obtain a fully deterministic description of what it means to execute two functions in parallel. Note that in general $merge(x, y) \neq merge(y, x)$. Thus it matters in which order the results are merged.

This example exhibits the very basic idea of our approach. *Spawn* copies a so that $f(a)$ and $g(a)$ can be computed concurrently while *Merge* is a means to compute the merged result using second order logic.

B. Operational Transformation

To merge concurrent operations into a non-concurrent sequence of operations (i.e. to serialize the operations) we use a technique called operational transformation that has been first introduced by Ellis and Gibbs in 1989 [2]. Operational transformation has been extensively studied ([3], [4], [5]) and literature shows that a merge based on operational transformation can be realized for many different data structures, including strings, formatted text, lists and trees. In contrast to other serialization techniques, e.g. used by transactional databases, there are no aborts in operational transformation and merging always succeeds.

Operational transformation originated from research in the fields of computer-supported collaborative work (CSCW) and collaborative editing. The idea is that all participants (e.g. concurrent editors of a document) work on local sets of data and only operations performed on these data sets are exchanged. An operational transformation system can be divided into two layers, the transformation control algorithm and the transformation functions [2]. The transformation control algorithm decides, which transformation function has to be applied to which set of concurrent operations. The transformation functions on the other hand are used to transform concurrent operations. If the transformation function and the transformation control algorithm are correct, the local data converges against an equal state for every site, even though each site applies the operations in a different order.

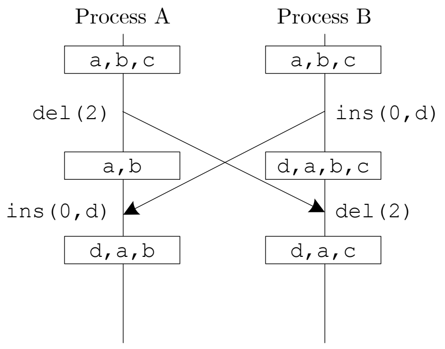

Figure 1. Without Operational Transformation

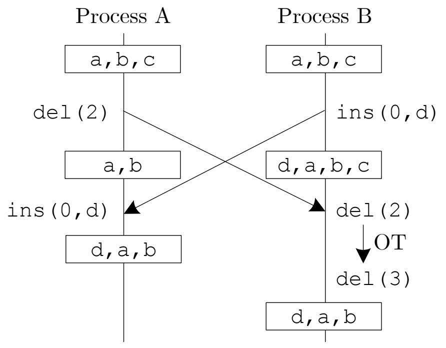

Figure 2. With Operational Transformation

For example we assume a list $[a, b, c]$ that can be modified by two concurrent processes A and B. Let us assume that process A intends to delete the element c at index 2 (the list index starts at 0), which results in an operation $op_A = del(2)$. At the same time process B inserts a new element d at the beginning of the list (i.e. at index 0), resulting in an operation $op_B = ins(0, d)$. Both processes directly apply their own operation (op_A and op_B) to their own local list. Later on they receive the operation sent by the other process. Applying the received operations without operational transformation would yield different lists for processes A and B (namely $[d, a, b]$ for process A and $[d, a, c]$ for process B) as shown in figure 1. This is because after process B inserted the element d, the index of the element that process A intended to delete shifted from 2 to 3. The operational transformation algorithm would thus modify the index of the delete operation in op_A, to preserve the intention of process A. If process B applies the transformed operation op_A with index 3, both processes result in the same list $[d, a, b]$ as shown in figure 2.

C. Spawn

Spawn creates a new child task and copies a set of data structures to work on. Thus we can think of spawning like calling a function where all arguments are passed by value (although an efficient implementation is more sophisticated). The spawned function is executed concurrently. Hence, *Spawn* immediately returns a handle to the new child task. *Spawn* accepts any type of parameter. However, it can only perform a *Merge* on data structures for which the compiler/runtime knows how to copy and merge them. Since defining operational transformation algorithms for data structures is a non-trivial task we intend to provide a set of commonly used mergeable data structures as a library, e.g. mergeable strings, lists and trees. Furthermore, programmers can use an interface to implement new mergeable data structures that work with our system. However, writing custom mergeable data structures should not be the common case. The following code snippet (for brevity reasons written in a GO-like pseudo-language) in listing 1 shows how *Spawn* and *Merge* are used:

```
func f(l List) {
  l.Append(5)
}
list := NewList(1,2,3)
t := Spawn(f,list)
list.Append(4)
MergeAllFromSet(t)
Print(list)
```

Listing 1. Usage of *Spawn* and *Merge*

The function `f` is to be executed by the child task. The parent task creates a list `list` with some values and then spawns a child task to execute `f` on a copy of `list` using `Spawn(f,list)`. Now `f` and the parent task concurrently add a new value to `list`. Since both work on their own copy, there is no need for locking and no race conditions appear. The function `MergeAllFromSet` performs operational transformation to merge the operations performed by both tasks into the final result $[1, 2, 3, 4, 5]$. Let us take a look at a comparable mutex-based implementation (written in GO) that is shown in listing 2.

The `mutex` variable is used to protect `list` while the `wait` variable is used to block execution until the function `f` completed. The `defer` statement used in function `f` makes sure that the locks are freed even if the `Append` function panics, for example because the maximum list size has been reached. It is obvious that the mutex-based version is much more verbose and it is not trivial to judge its correctness.

Furthermore, the mutex-based version is not even deterministic. When the GO scheduler uses only one thread for computations, the result is indeed always $[1, 2, 3, 4, 5]$, because the child is always executed after the parent is blocked acquiring a lock. When the scheduler has more than one thread at its disposal then the two `Append` functions could be executed concurrently. Most of the time the function `f` is last because it takes some time to launch a new go-routine via `go f(list)`. However, if we uncomment `DoSomething()` then the result can be $[1, 2, 3, 5, 4]$, depending on how much time the `DoSomething` function consumes. This does not only show the obvious fact that the mutex-based version is non-deterministic. It also highlights how difficult it is to catch these issues with testing. In contrast the *Spawn* and *Merge* solution is shorter and always deterministic.

```
var mutex sync.Mutex
var wait sync.Mutex
func f(l List) {
  mutex.Lock()
  defer mutex.Unlock()
  defer wait.Unlock()
  l.Append(5)
}
list := NewList(1,2,3)
wait.Lock()
go f(list)
// DoSomething()
mutex.Lock()
list.Append(4)
mutex.Unlock()
wait.Lock()
Print(list)
```

Listing 2. Mutex-based example

D. Merge

The *Merge* function utilizes logic implemented by the data structures to merge the results of concurrent execution via operational transformation. *Merge* comes in four different flavors:

- *MergeAll* waits for all child tasks to complete and merges them in a deterministic order, i.e. in the order of their creation
- *MergeAllFromSet* waits for all child tasks passed as argument to this variadic function and merges them deterministically in the order of the argument list
- *MergeAny* waits for the first child task to complete and merges it which introduces non-determinism
- *MergeAnyFromSet* waits for the first task of a given set of children to complete and merges it (also introducing non-determinism)

MergeAny is useful when implementing distributed systems which have non-determinism because of network delay or non-deterministic clients. For example a server can spawn a new task to handle incoming requests. Using *MergeAny* it will merge completed tasks on a first-completed-first-merged basis. *MergeAnyFromSet* waits for any child in a set. So *MergeAny* is just a special case of *MergeAnyFromSet*. Whenever a task that still has running child tasks finishes

MergeAll is called implicitly. The *MergeAllFromSet* function is useful when a task has a set of child tasks running and wants to wait and merge a subset of them.

In addition, a *condition function* can be passed to *Merge* functions to validate post-conditions. This way a task can modify its data structure copies and check the conditions on the computed results before merging. Whenever the validation fails, the *Merge* will not be performed (i.e. any changes of the child task will be omitted). This is essentially a rollback mechanism realized by the runtime system. The concept is close to the idea of transactional memory [6], where code regions are grouped into transactions, that will either be executed completely or not at all. However, when two threads using transactional memory write the same cache line, at least one transaction is rolled back. In our framework in contrast there is no rolling back if two tasks write the same data at the same time due to our use of operational transformation.

E. Sync and Clone

In this chapter we introduce two new constructs that do not extend the expressiveness of our synchronization system, but make the code more readable.

One problem of the *Spawn* and *Merge* approach is that tasks only merge after completion. This renders the creation of long running tasks (e.g. tasks handling a TCP connection) that influence data structures of the parent inconvenient to achieve. However, this is often a desired behavior, e.g. that merging should happen whenever a request is handled and not when the TCP connection has been closed. An ugly solution is for the child task to complete whenever a part of its responsibility is completed (e.g. handle one incoming TCP request) and merge with its parent. The parent task must then check after merging whether the task has to fulfill further duties, e.g. if the TCP-connection is still open, and spawn a new child task to handle this.

To simplify such scenarios, we offer the *Sync* function. When a child calls `Sync()` it blocks and waits until the parent wishes to merge with it. After the parent finished merging, it creates a new fresh copy of the parent's data structures and continues execution. Thus calling *Sync* is equivalent to completing the task and spawning a new one right after *Merge* is completed. *Sync* does therefore not extend the expressiveness of our system but it yields more readable code. Furthermore, it allows a child task to handle the case that a user defined *Merge* with its parent fails (as seen in listing 3).

Another practical problem is that it is implementation-wise difficult for a task to execute two blocking operations at once, e.g. `MergeAny` and `socket.Accept`. To execute both operations, the root task can spawn a new child task that executes the blocking `socket.Accept` function, while the root task itself blocks by calling `MergeAny` in an infinite loop. However, when the the child task accepting incoming connections spawns new children of its own, these are *grandchildren* of the root task and hence the root task cannot merge with them. The ugly solution is that the child task, that is accepting connections, completes when it has accepted a new incoming TCP-connection. The root task notices this when it merges with this child and can now spawn a new child to handle the established connection and spawn a second child to accept further connections.

Since this is a recurring pattern in practice when dealing with blocking I/O, we allow a child task to clone itself using the *Clone* function. Thus, it creates a new sibling task to e.g. handle a new TCP-connection. This way the root task can merge with the cloned task using `MergeAny` and the problem is solved, as shown in the server software example following the next section.

F. Abort

A task can willingly or unwillingly fail to complete. Exceptions within a task can be caught and error flags in the task can be set to notify the parent task of the error. Exceptions can occur if they are thrown by the task to abort itself or by the runtime if e.g. an index is out of range. When the parent merges an aborted task no data is changed. The copies on which the aborted task worked are simply dismissed. Furthermore, a parent can mark a task as being externally aborted. This is useful because a parent must merge with all its children eventually and this flag allows a parent to state that it does not want to accept the changes computed by this child task anymore. *Abort* does not force the task to stop working immediately, since most operating systems do not handle a forceful thread termination gracefully.

G. Example 1: Server Software

To further demonstrate the usage and simplicity of *Spawn* and *Merge* we show how to realize two example scenarios in the following sections. First, listing 3 shows code to realize a simple server software. When implementing a TCP-based server, the straightforward approach is to create a set of global data structures in the root task and then wait for incoming TCP connections. For each new TCP connection, the root task spawns a new child task that handles the connection. The child task completes when the connection is closed. Furthermore, the root task merges children non-deterministically using `MergeAny`.

In this example, a child task is created that accepts incoming TCP connections in the `accept` function. When `accept` has been spawned, the `data` variable has its initial value. When `conn` is spawned for a new connection, it is spawned as a *clone* of the accept task. Thus, it inherits the same initial value of `data` from its sibling. Since `data` will most likely be outdated, the `conn` task first retrieves an up to date version of `data` from its parent by calling `Sync()`. Afterwards the `conn` task will process incoming

```
func accept(data) {
  for {
    socket := tcp.accept()
    Clone(conn, socket, data)
  }
}
func conn(socket, data) {
  defer socket.Close()
  Sync()
  for {
    req, err := read(socket)
    if (err != nil) {
      return
    }
    req.doWork(data)
    err := Sync()
    if (err != nil) {
      write(socket, err)
      panic(err)
    }
  }
}
data := ...
Spawn(accept, data)
for {
  MergeAny()
}
```

Listing 3. Server Software realized using *Spawn* and *Merge*

requests and tries to merge with its parent via `Sync()`. If this fails, it sends an error message on the socket, closes it and aborts.

H. Example 2: Simulation Software

As a second example we chose a network simulation software, that has a special requirement for correct and deterministic results. In this simplified scenario a network of individual hosts, that communicate by message passing, is simulated. Each host receives a message, calculates the next recipient, and forwards the message accordingly. This simulation is inherently prone to race conditions when using common synchronization primitives: if two hosts send a message to the same recipient the order of processing is timing dependent. The basic code using *Spawn* and *Merge* to simulate a network of n hosts is shown in listing 4.

In this example a task is spawned for each simulated host. The hosts receive an identifier as well as a copy of the global `messageQueues` that is initialized with some starting messages.

Each `host` function is executed in parallel and contains a loop that determines the host behavior. First, the host updates its data structures against the parent data structures using `Sync` to ensure that his data is up to date. If its input message queue is not empty it processes the received message (i.e. `m_received`) to determine the new message to send and the destination host (i.e. `m_send` and `destination`). Finally, the message to send is stored in the message queue

```
func host(hostID, queues){
  for {
    Sync()
    if (queues[hostID].isEmpty()){
      continue
    }
    m_received := queues[hostID].pop()
    m_send, destination :=
      processMessage(m_received)
    queues[destination].append(m_send)
  }
}

messageQueues := MergeableQueue[n]
InitMessages(messageQueues)
for (i := 0; i < n; ++i){
  Spawn(host, i, messageQueues)
}
for {
  MergeAll()
}
```

Listing 4. Simulation Software realized using *Spawn* and *Merge*

of the destination host. The next loop iteration starts with `Sync`, which causes the own changes to be merged into the parents data structures. This also updates the local data structures.

This example shows how to realize deterministic and correct parallel programs using *Spawn* and *Merge*. The determinism is enforced by the use of `MergeAll` that deterministically merges all tasks in the order of their creation. The use of operational transformation when merging furthermore serializes the concurrent operations of tasks on the data structures.

III. Evaluation

In this section we evaluate an implementation of the example scenario described in section II-H, using a proof of concept implementation of our framework. To create some unpredictable processing load on hosts the destination address is derived from the message payload using cryptographic operations (i.e. *SHA-1* hashing).

To be able to compare the performance of our framework we also created a implementation using threads and conventional synchronization primitives. In this implementation each host is represented by a thread with an incoming queue. The thread performs a blocking read on its queue until a message is received. As soon as a message was read the thread performs the *SHA-1* calculations and pushes the message in the incoming queue of the destination host. Both implementations were created using C++11 and GCC version 4.8.1 and measured on an i7-3520M.

The load l on each host is controlled by adjusting the number of cryptographic hash operations per message (i.e. *hash iterations*). If l is low, a larger fraction of the simulation will be spent on queue operations. Thus, we measured the

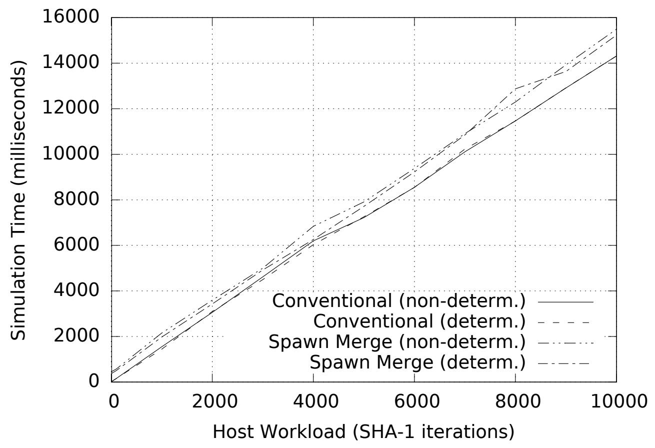

Figure 3. Evaluation results

performances of four test setups depending on the host workload l.

All test setups used a base of 20 simulated hosts. Initially there are 100 messages distributed over the simulated network, each with a time to live (TTL) of 100 hops. The first two test setups are conventional implementations, one with a non-deterministic simulation (i.e. calculating the destination host as stated introducing non-determinism at the destination host's incoming queue) and one with a deterministic simulation. The determinism here is achieved by sending messages only to the node with the next higher id, since the concurrency caused by sending two messages to the same host is no longer present. The other two setups used our *Spawn* and *Merge* framework, simulating the same behavior as the conventional implementations. However, note that using *Spawn* and *Merge* also the 'non-deterministic' test setup becomes deterministic and yields the same results in every run.

We run every test setup several times with up to 10000 hash iterations for the host workload l. The average measurement results are shown in Figure 3. The execution time of the conventional implementations rises proportionally with the number of hash operations l with negligible performance differences for deterministic and non-deterministic simulations. The execution time using *Spawn* and *Merge* rises linearly alongside the conventional implementation performance, while the deterministic simulation is slightly faster than the 'non-deterministic' version. This small performance gap (of about 1% to 4%) results from multiple messages sent to one host, that are processed in consecutive simulation cycles.

However, there is a constant overhead of about 400 milliseconds per run. This is because on *Spawn* the initial data structures have to be copied for every spawned task (i.e. 20 tasks with 20 queues each which have to be copied). The measurement results show that the overall effect of the overhead decreases with increasing host workload l. Namely, while for 1000 iterations the overhead is about 38% (since the conventional approach finishes in about 1.5 seconds the overhead of about 400 milliseconds is comparably high) it decreases to about 7% for 10000 hash operations. Note however, that we used an unoptimized proof of concept implementation of our framework. The overhead may be further decreased by using sophisticated copy-on-write mechanisms, faster merging algorithms and other optimization techniques. Furthermore, the effect of lock contention avoided using *Spawn* and *Merge* cannot be derived from the measurements of the unoptimized framework, yet.

IV. Analysis

In this section we show that for synchronization the *Spawn* and *Merge* primitives have equivalent expressive power to the usage of semaphores, that have been introduced by Dijkstra in 1965 [7]. This way we prove that the same concurrent execution, as in a program using semaphores, can be achieved using our system. Furthermore, we analyze our proposed system to show that *Spawn* and *Merge* based systems are by nature deadlock-free.

A. Equivalence to Semaphores

We assume that a semaphore-based system does not concurrently access the same memory bytes without locking them via a semaphore. This is reasonable, since everything else is most likely a bug anyway that induces the kind of non-determinism that we try to avoid in the first place. Furthermore, we assume that the semaphore-based system does not end up in a deadlock.

To prove that *Spawn* and *Merge* are equivalent to semaphores we will model a semaphore using only *Spawn* and *Merge*. We model a semaphore as a list of integers L. The first element is the value of the semaphore, all following numbers are the IDs of tasks waiting at this semaphore. When a child task wants to acquire the semaphore it appends itself to the list L and calls `Sync()` and then `Sync()` again. The first `Sync` wakes up the parent task that is merging with any child of a set S of children, using `MergeAnyFromSet(S)`. This set S covers all children initially. To match the semaphore-based system, we create one child for any thread used by the semaphore-based system. When `MergeAnyFromSet(S)` returns, the parent checks L to see if it can grant a waiting child task access to the semaphore. If the semaphore value is already zero, then the child is removed from S. Hence, the parent task does not merge with the waiting child task anymore and therefore the child is blocked in the second `Sync` call. Otherwise, the semaphore value is decreased, the waiting child is removed from L and added to S. Hence, the second `Sync` will proceed because the parent merges with this child again. Now the child has successfully acquired the semaphore.

To release a semaphore, a child adds its negative task ID to L and calls `Sync()`. This wakes up the parent. When the parent task returns from `MergeAnyFromSet(S)` it checks for these negative task IDs. It removes them from L and increases the semaphore value by one for each removed negative ID. Then it checks whether other waiting child tasks can now get access to the semaphore.

While this procedure is inefficient and cumbersome, it shows that we can achieve the same parallel execution that a semaphore-based system can realize. Due to the use of `MergeAnyFromSet` we allowed non-determinism to creep into our system, but this is necessary because semaphore-based systems are non-deterministic, too. A programmer should always use the deterministic *Merge* functions by default and resort to the non-deterministic ones if he explicitly wants to have non-deterministic behavior, which is usually the case in server-software or interactive applications.

B. Deadlocks

Using only *Spawn* and *Merge* it is impossible to create a deadlock. A deadlock requires four conditions to hold [8], one of them being a *cyclic dependency*. However, the task hierarchy has a tree shape and the only possible waiting is between a parent and a child task via *Merge* and *Sync*. Thus, the only cyclic dependency possible is that a child waits for its parent (only possible with *Sync*) and the parent waits for the child (only possible with *Merge*). But in this case the merge will happen and both are unblocked. Hence, cyclic waiting conditions are impossible.

Following the equivalence to semaphore-based systems shown in the previous section it is interesting to see how the *Spawn* and *Merge* based system simulates a deadlocked semaphore system. If all tasks are blocked, the set S (held by the parent) is empty, because by definition all blocked children are removed from S. Thus, the parent task will iterate in an infinite loop calling `MergeAnyFromSet(S)` where S is the empty set. This means that `MergeAnyFromSet` will never block, because there is nothing it could wait for. This reveals (to no surprise) that a *Spawn* and *Merge* based system can still livelock, but this is true for all programming languages that allow infinite loops and is nothing that could be prevented by a synchronization system. However, deadlocks are impossible to realize with *Spawn* and *Merge*.

V. Related Work

Multi-threaded programming introduced many new difficulties which primarily relate to synchronization of concurrent processes, achieving of deterministic results and a consistent view on data structures. In this section we describe how these problems are addressed by different approaches that have been proposed to cope with these issues.

Deterministic Parallel Programming: The general idea that parallel programming should be deterministic by default and that non-determinism should only be introduced if explicitly desired is not new has already been proposed by Brocchino et al. [1] and realized in the *Deterministic Parallel Java (DPJ)* project [9]. In DPJ the heap is partitioned into hierarchical regions. These regions are used to differentiate accesses to objects or object parts of the same object. Furthermore, an effect system is created by annotating each task with the regions that are accessed within the task. Based on these regions combined with the effect system a type checker is used to identify overlapping memory operations between concurrent tasks that may introduce nondeterminism. Thus nondeterminism cannot be introduced by accident.

Even though our *Spawn* and *Merge* framework follows the same philosophical approach as DPJ the way determinism is achieved differs completely. In DPJ conflicting memory accesses are exposed by the compiler. However, this requires the effect system that is built from the code annotations to be correct. In contrast our approach uses operation transformation to deterministically resolve conflicting writes and does not depend on additional annotations.

Parallel Programming Runtime Systems: Cilk [10] is a runtime system for multithreaded parallel programming. A programmer writing a *Cilk* program has to expose parallel executable code by spawning *Cilk procedures* using a *spawn* keyword. Spawned procedures may be executed by the runtime in parallel and return their computation results to their parent. The *sync* keyword introduces a barrier method stating that all spawned *Cilk procedures* have to return before execution may proceed.

Even though our approach looks syntactically similar to the *Cilk* approach due to the constructs *spawn* and *sync*, the constructs have different meanings. *Cilk* assumes that spawned procedures avoid conflicting data access. However, there is no guarantee for determinism concerning the results of spawned procedures. In contrast, using *Spawn* and *Merge* a programmer may explicitly choose determinism or non-determinism by choosing one of the *Merge* functions. Furthermore, our approach allows waiting for specific tasks if desired instead of always blocking until all spawned tasks completed. Finally, in our approach the *Sync* function enables programmers to propagate intermediate results to the parent task to update the parent data structures while the task is still running.

Parallel Programming Frameworks: Parallel programming frameworks provide an API to ease parallel programming. Thereby they can expand general purpose programming languages. The *OpenMP* API [11] for example provides developers with compiler directives to specify which code regions can be executed in parallel. This way developers can e.g. declare a loop as feasible for parallel execution while the runtime system will schedule single loop iterations to different threads. However, OpenMP does not guarantee determinism and the compiler cannot check whether e.g. a parallel loop execution is really safe.

Locking Primitives: In general purpose programming languages several techniques for handling parallel execution of multi-threaded programs have become commonplace. These techniques include semaphores, mutexes, and monitors. They all share the same shortcoming: the first thread arriving at a synchronization point gets access to the data first. This introduces *non-determinism* into multi-threaded programs, because even on the same hardware it is not deterministic which thread will arrive at a synchronization point first since it depends on many circumstances like scheduling of threads, I/O performance or interrupts. Furthermore, these constructs are negative by design, i.e. they state what *must not happen*, for example there must not be two threads active in a monitor [12] at the same time. This means that multi-threaded programs yielding deterministic results in every run are hard to achieve using these simple synchronization primitives.

In contrast, our approach is to use positive constructs for synchronization. Thus, a program specifies in which order results of spawned tasks are merged into the parent's data structures, which results in a deterministic execution of a concurrent program. In fact, we provide both solutions. It is up the programmer to make an explicit choice for non-determinism, if desired.

Message Passing: The idea of processes that communicate and synchronize by passing messages dates back to the *'Communicating Sequential Processes'* paper of Hoare [13]. An example is the *Message Passing Interface* (MPI) [14]. MPI provides an API for exchanging messages between processes that may run on different computers and thus is also applicable to distributed systems. Another example is the programming language GO that reintroduced message passing as a primary synchronization concept. In GO tasks (called *go-routines*) communicate and synchronize by message passing via channels. However, a system using message passing may also contain races introducing non-determinism.

Process Calculi: Process calculi deal with modeling and reasoning about concurrent systems, i.e. they can be used to describe and analyze their properties [15]. They provide a high-level description of concurrent processes, especially in terms of communication and synchronization, using only a small number of primitives and operators. Furthermore, process calculi allow the manipulation and analysis of a concurrent system description by using algebraic-laws to show e.g. equivalences between processes using equational reasoning.

In contrast to process calculi our approach does not try to verify the correctness of a concurrent program using a theoretical model for describing process communications. The operational transformation technique used in our proposed approach ensures the correctness of the executed program in terms of determinism and the absence of deadlocks and race conditions by design.

Coordination Languages: Coordination languages are used to specify protocols on how concurrent processes may communicate and interact, i.e. on how to coordinate the processes [16]. The coordination language *Linda* [17] for example allows to extend a sequential language with several primitives for inter-process communication that operate on *tuples*, stored in a tuplespace in a logically global associative memory. The idea is to treat the process coordination as an orthogonal activity to process computation. This allows coupling of processes on a data-level using the tuplespace. However, coordination languages like *Linda* do not ensure deterministic program execution by default.

Path Expressions [18] use a syntax that vaguely resembles the idea of regular expressions. Where regular expressions define the set of all allowed character sequences, path expressions describe the set of all allowed parallel invocations of a set of functions. While powerful, path expressions and regular expressions suffer from the same problem: they are very concise, but for real-life problems which are more complex than a typical consumer/producer example, they become hardly readable. Furthermore, path expressions do not deliver determinism. Just like locking they restrict parallelism on function execution, but usually specify a large set of possible actual execution paths which in turn could each yield different results depending on the timing of threads.

VI. Conclusion and Further Work

In this paper we presented *Spawn* and *Merge*, two synchronization primitives that yield deterministic program execution of multi-threaded programs. While programs using *Spawn* and *Merge* are deterministic by default it is still possible to introduce non-determinism into a program if desired. The *Spawn* function is used to create a new *task* that works on a local data copy to avoid the need for data structure locking. It is combined with the *Merge* function, a deterministic function to merge the results of the concurrent task execution back into the data structures of the parent task. The *Merge* function utilizes operational transformation to deterministically serialize the concurrent operations performed by tasks.

We demonstrated the ease of use of *Spawn* and *Merge* by realizing a simple non-deterministic server software and a simple simulation software that always yields correct results. Furthermore, we evaluated an unoptimized proof of concept implementation of *Spawn* and *Merge* to show that the performance is comparable to using conventional synchronization primitives (despite a constant overhead of about 400 milliseconds per run, whose effect decreases with increased task workload) while yielding deterministic results by default. We finally proved that a *Spawn* and *Merge* based system is equivalent to a system using semaphores for synchronization in terms of concurrent program execution. Furthermore, we have shown that systems using *Spawn* and *Merge* are deadlock free. This way our technique enables an easier programming of multi-threaded programs that can be run at different kinds of many-core systems, like e.g. future CPUs, the Parallella Board [19] or even supercomputers with thousands of CPUs while always yielding the same result.

Next we will optimize our *Spawn* and *Merge* framework using techniques like copy-on-write and more efficient merge functions to decrease the overhead of using *Spawn* and *Merge* compared to implementations using conventional synchronization primitives. We will use these optimizations to reason about the generality and scalability of our approach for further interesting use cases like scientific computing. Furthermore, we plan to apply the concept of *Spawn* and *Merge* to distributed computing by using MPI.

References

[1] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir, "Parallel programming must be deterministic by default," in *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 4–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855591.1855595

[2] 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

[3] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 5, no. 1, pp. 63–108, Mar. 1998. [Online]. Available: http://doi.acm.org/10.1145/274444.274447

[4] C. lavinia Ignat and M. C. Norrie, "Customizable collaborative editor relying on treeopt algorithm," in *In Proc. of the European Conf. of Computer-supported Cooperative Work*. Kluwer Academic Publishers, 2003, pp. 315–334.

[5] A. H. Davis, C. Sun, and J. Lu, "Generalizing operational transformation to the standard general markup language," in *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, ser. CSCW '02. New York, NY, USA: ACM, 2002, pp. 58–67. [Online]. Available: http://doi.acm.org/10.1145/587078.587088

[6] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of the 20th annual international symposium on computer architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/165123.165164

[7] E. W. Dijkstra, "Cooperating sequential processes, technical report ewd-123," Tech. Rep., 1965.

[8] E. G. Coffman, M. Elphick, and A. Shoshani, "System deadlocks," *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, Jun. 1971. [Online]. Available: http://doi.acm.org/10.1145/356586.356588

[9] University of Illinois at Urbana-Champaign, "Deterministic Parallel Java (DPJ) project," http://dpj.cs.uiuc.edu/DPJ/Home.html, 2014. [Online]. Available: http://dpj.cs.uiuc.edu/DPJ/Home.html

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, Aug. 1995. [Online]. Available: http://doi.acm.org/10.1145/209937.209958

[11] OpenMP Architecture Review Board, "OpenMP Application Program Interface," http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2014. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[12] C. A. R. Hoare, "Monitors: an operating system structuring concept," *Commun. ACM*, vol. 17, no. 10, pp. 549–557, Oct. 1974. [Online]. Available: http://doi.acm.org/10.1145/355620.361161

[13] ——, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359585

[14] "Document for a standard message-passing interface," Knoxville, TN, USA, Tech. Rep., 1993.

[15] B. C. Pierce, "Foundational calculi for programming languages," in *in the CRC Handbook of Computer Science and Engineering. Available electronically*, 1995.

[16] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, no. 2, pp. 97–107, Feb. 1992. [Online]. Available: http://doi.acm.org/10.1145/129630.129635

[17] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and friends," *Computer*, vol. 19, no. 8, pp. 26–34, Aug. 1986. [Online]. Available: http://dx.doi.org/10.1109/MC.1986.1663305

[18] R. H. Campbell and A. N. Habermann, "The specification of process synchronization by path expressions," in *Operating Systems, Proceedings of an International Symposium*. London, UK, UK: Springer-Verlag, 1974, pp. 89–102. [Online]. Available: http://dl.acm.org/citation.cfm?id=647641.733391

[19] Adapteva, "Parallella Board," http://www.parallella.org, 2014. [Online]. Available: http://www.parallella.org