Parallel Transaction Processing in Functional Languages

Towards Practical Functional Databases

Lesley Wevers[1], Marieke Huisman[1], and Ander de Keijzer[2]

[1] University of Twente, The Netherlands
[2] Windesheim University of Applied Sciences, The Netherlands

Abstract. This paper shows how functional languages can be adapted for transaction processing, and discusses the implementation of a parallel runtime system for such functional transaction processing languages. We extend functional languages with current state variables and result state variables to allow the definition of transactions that can update, create and delete bindings in a state. Our runtime system features of a graph reducer, a transaction manager and a persistence module. Our graph reducer adapts template instantiation with anonymous templates, allowing lazy execution of ad-hoc transactions that can dynamically create and remove of bindings in the state. Additionally, we have developed a scheduler for graph reduction that guarantees low latency execution of transactions, where load balancing is performed by randomising the reduction order of threads. We have evaluated our prototype implementation via some practical experiments.

1 Introduction

Using functional languages for implementing, querying and manipulating databases provides many advantages over existing relational database technologies [8, 11]. List comprehensions provide a natural interface to a functional database, while being more general than the relational algebra used in current database technology. This allow a wide range of operations to be performed inside the database, providing a basis for constructing database applications entirely within a single system. Furthermore, transactions on a functional database can be executed in parallel without explicit management of hardware resources. Multiple transactions can be executed simultaneously by evaluating the state produced by transactions lazily, which also allows higher levels of concurrency than existing database technology. The purity of functional languages guarantees that transactions do not interfere, and laziness automatically minimises access to the underlying storage media. In this paper we discuss how functional languages can be adapted for transaction processing, and we discuss the implementation of a runtime system for such a language that allows ad-hoc transactions to be executed in parallel.

Transaction Processing Systems If a database system crashes with operations only partly executed, or if concurrent operations on the database interleave in their execution, the system could be left in an inconsistent state. To solve these issues, the concept of a *transaction* has been introduced [3]. A *transaction* provides guarantees about the execution of a collection of operations on a *state*. Most database systems guarantee that the *ACID properties* hold for transactions:

Atomicity: Either all operations in a transaction are executed, or none at all.
Consistency: Transactions preserve consistency of the state.
Isolation: The result of transactions executing in parallel is the same as the result for some sequential executions of the transaction.
Durability: Once a transaction has been committed, its effects must persist.

Formally, atomicity and isolation are defined in terms of serialisability and recoverability. A parallel execution of a set of transactions is *serialisable* if some sequential execution of these transactions produces the same final state as the parallel execution. A transaction is terminated by either committing the transaction, making its effects persistent, or aborting it and rolling back all changes. *Recoverability* states that, as long as a transaction t has not been committed, all transactions that have read changes by t cannot commit until t commits. If t chooses to abort, all transactions that have read changes by t must also abort.

A *transaction processing system* (TPS) is a system that manages a state through transactions. Ensuring that the ACID properties hold results in several challenges for TPS implementations. First, one would like to execute transactions in parallel in order to minimise response times, while at the same time avoiding inconsistent states that could be caused by interactions between transactions. Second, the TPS has to ensure correct behavior in case of system failure, as writes to persistent storage may only be partially complete at the time of failure.

Transaction Processing in Functional Languages Existing functional programming languages can be used to construct TPS's [11]. While this approach is very generic, and allows general purpose transaction processing systems to be built on top of the existing languages, it has some drawbacks which we discuss in the next paragraph. A modern example of this approach can be found in the acid-state [1] library for Haskell. In this approach the state of the system is of some user defined data type S, and transactions are functions of type $S \to (S \times T)$, taking the current state and producing a new state together with some observable result of type T. A *transaction manager* is in charge of executing transactions. For example, in Haskell a transaction manager can be implemented using an `IORef` to access and update the state atomically. It is easy to guarantee that the ACID properties hold in this model. Serialisability can trivially be satisfied by executing transactions sequentially. Recoverability can be satisfied by requiring that all transactions are *total* functions, i.e. they always produce a result. Consistency of a state `s` can be guaranteed by wrapping a transaction `f` by a function of the form `if g(f(s)) then f(s) else s`, where `g` validates that some consistency property holds for `f(s)`. To ensure durability of transactions,

the transaction manager can write every transaction to a journal before executing it, and in case the system crashes, re-execute the transactions in the journal. To ensure that the journal does not grow too large, the transaction manager can periodically create a snapshot of the state and empty the journal.

As has already been mentioned there are some drawbacks to using existing functional languages to construct TPS's. First, a major strength of database management systems is their ability to execute ad-hoc queries, e.g. for the purpose of generating reports, manually changing data in the database, or upgrading existing data to a new format. In the approach outlined above, transactions must be predefined at compile time. If a user wants to execute some *ad-hoc transaction*, i.e. a transaction that has not been predefined, the system has to be recompiled and restarted, which leads to downtime. Another problem with this approach is that the state of the system is limited to the available main memory. If larger states are to be used, parts of the state must be swapped to persistent memory when the available main memory is insufficient. This is difficult to implement transparently as part of a library in a functional language.

Transactional Functional Languages Our approach to allow ad-hoc transactions on a functional state is to construct an interactive environment that allows multiple users to perform operations on the state simultaneously through transactions. We call such a system a *transactional functional language* (TFL). A TFL is in a way similar to an interactive programming environment, such as GHCi and Hugs for Haskell, which allows execution of ad hoc expression within an environment, as well as creating new bindings in this environment. However, the main feature of a TFL is that it allows concurrent multi-user access through transactions, and it guarantees that the ACID properties hold for these transactions. The state of a TFL can be structured as a set of bindings that map names to expressions. Expressions can be functions in the form of λ-abstractions, or values such as associative maps to store data. Transactions can atomically evaluate an expression in the context of the current state, as well as change the bindings in the state.

In the literature, practical implementations of a TFL have already been given by Nikhil with the Agna system in 1991 [7], and McNally with the STAPLE system in 1993 [6]. However, while Agna supports parallel execution of individual transactions, both systems do not support the execution of multiple transactions in parallel. It seems no more work has been done on TFL's since these publications. However, we think that TFL's can provide many benefits over the technology that is used today, and are worth investigating in more detail.

Contributions This paper extends on previous work by investigating the implementation of a TFL that allows the execution of multiple transactions in parallel. We have defined constructs to extend a functional programming language to define functional transactions. Moreover, we have implemented a runtime system in Java that allows ad-hoc transactions to be executed in parallel.

To allow ad-hoc transactions to be interpreted, we have implemented a graph reducer based on template instantiation [9]. A special feature of our graph reducer is that it allows bindings to be created and removed dynamically. Parallel

execution of ad-hoc transactions complicates the managing of bindings, because previous transactions may still use bindings that have already been deleted by later transactions. We have solved this problem by adapting template instantiation so that templates are anonymous. This allows bindings to be deleted at the top level without removing the accompanying template, and allows a standard garbage collector to clean up templates that are not in use anymore.

A key requirement of transaction processing systems is that they respond quickly to transaction processing requests. Existing parallel functional language implementations commonly use work-stealing for dynamic scheduling of reduction tasks on a fixed set of reduction threads [10]. The problem with work-stealing in the context of transaction processing is that a reduction thread working on reducing the result of a certain transaction may steal a task from a reduction thread working on another transaction. The result of this is that the reduction of a certain result may not be finished before the reduction of another result is finished, delaying the response unnecessarily.

We have developed a scheduler for parallel graph reduction that solves this problem. In contrast to work stealing, in our graph reducer a reduction thread is certain to make progress on the transaction that it is scheduled to work on. We distribute work between reduction threads by randomising their reduction order, and reduction threads share results of their work to co-operate on tasks.

Stucture of this Paper The remainder of this paper is organised as follows. In Section 2 we define constructs to use existing functional languages for defining transactions on functional states. In Section 3 we provide a high level overview of a runtime system that can execute transactions in such a language atomically, and which ensures durability of the executed transactions. Section 4 explains how we adapt template instantiation to support anonymous templates. Section 5 presents our scheduler for low-latency parallel graph reduction, and Section 6 presents our experimental results. Finally, Section 7 present related work and conclusions.

2 Transactional Functional Languages

This section describes how a functional language can be used for transaction processing. We first discuss a model for a transactional functional language. Next, we discuss language constructs to adapt existing functional languages for transaction processing, and we show how to use such a language to implement, manipulate and query a simple database.

A Model for Transactional Functional Languages To define a TFL, we follow the model as used by Trinder [11], Nikhil [7] and McNally [6]. In a TFL, a *state* consists of a set of bindings similar to a traditional functional progam, but without the presence of a main binding. The runtime of a TFL accepts a stream of *transactions*, which are sequentially executed in the context of the *current state*, and which produce a *result state*. When the runtime of the TFL is started,

there is an *initial state* which is the current state of the first transaction in the stream. For each subsequent transaction, the current state is the result state of the previous transaction. In practice, when multiple transactions arrive at the system simultaneously, they can be merged non-deterministically into a stream of transactions. This model ensures that there is no interference between the execution of transactions.

To construct a result state, a transaction may change the bindings in the result state compared to the current state. More specifically, a transaction may do zero or more of: change the values of a binding to a given expression, create a new binding with a given expression as value, or delete a binding. Within a transaction, each binding may only be modified once, and the order in which this is done does not affect the result. In addition to updating the state, a transaction may produce an observable result by evaluating an expression in the context of the current state.

Expressions in transactions may contain variables that refer to values in the current state. When a transaction is executed, these variables are *bound* to the values in the current state to obtain closed expressions. Finally, the result expression of a transaction can be reduced to normal form to obtain the observable result of the transaction. Note that, semantically, it does not matter whether the expressions in the next state are reduced before subsequent transactions are executed. This is because expressions in the state are reduced on demand when reducing the results of subsequent transactions. In our implementation of a runtime for TFL's we use this property to execute transactions in parallel by evaluating states lazily.

Language Constructs for Transaction Processing Now we describe our constructs to extend existing functional languages so that they can be used to define functional transactions according to this model. We do this by distinguishing between names in the current state, and names in the result state. Syntactically, we can make a distinction between such names by writing *result state variables* primed, and writing *current state variables* normally. To create or update the binding x, we assign an expression E to variable x', where variables in E may refer to both the current and the result state. To simplify programming in this model, we can assigning expressions to variables in the current state to define a binding that is local to the transaction. Local bindings can be used within a transaction, but will not be available in successive transactions. The result expression of a transaction can be assigned to a special name, such as *result*. Finally, bindings can be removed by writing `remove` x, for some name x in the state.

We now illustrate how this approach can be used to set up and use a simple database of user names. For these examples we use Haskell syntax, augmented with current state and result state variables. However, we have to note that our current prototype language implementation uses a simpler (untyped) core language. The following transaction updates the state to include a variable *users*, which is initialised to the empty list, and a function *length* that can be used to compute the length of a list:

```
1 users' = Nil
2 length' [] = 0
3 length' (x:xs) = 1 + length' xs
```

Note that in the definition of *length′* we refer to *length′* to create a recursive function. If we would refer to *length* instead, we would refer to the value of *length* in the current state. The next transaction inserts a user into the database, and requests the size of the resulting database:

```
1 users' = "bob" : users
2 result = length users'
```

Note that in the definition of *users′* we refer to *users* in the current state; thus inserting a user into the existing database. The observable result of the transaction is defined as length of *users′*, which includes our newly inserted user "bob". Finally, the following transaction queries the database with a locally defined function *contains* that exists only for the duration of the transaction:

```
1 contains value [] = False
2 contains value (x:xs)
3   | value == x = True
4   | otherwise  = contains value xs
5 result = contains "bob" users
```

3 A Prototype Runtime System

This section provides an overview of a prototype runtime implementation for TFL's. After providing a general overview, we first discuss the role of graph reduction in our implementation. Next, we discuss the implementation of a transaction manager, and finally we discuss how to store states in persistent memory and how to ensure durability.

Overview Our system adopts a client-server architecture, where our implementation takes the role of the server. A client interacts with the server by sending a transaction to the server, to which the server responds with a reduced result expression. The server executes transactions in parallel by spawning an operating system thread for each request. The thread parses the request to obtain a set of updates to the bindings in the state, together with an optional result expression which is to be evaluated in the context of the current state. The thread then executes the transaction by invoking the transaction manager on te parsed transaction, which updates the state and returns an unreduced bound result expression. Finally, the thread invokes the graph reducer to reduce the result expression to normal form, which it then returns to the client.

Graph Reduction The graph reducer is a major component of our runtime system. The graph managed by the graph reducer contains the state of the system,

as well as the result expressions of the transactions. The graph reducer provides a procedure `whnf` to reduce nodes to weak head normal form (WHNF), which is used to reduce the result of transactions. Our graph reducer has two special features for transaction processing. First, it allows bindings to be added and removed dynamically. And second, it allows parallel graph reduction, while guaranteeing that transactions are executed with low latency. These features are covered in more detail in the next two sections.

Transaction Manager The transaction manager is concerned with executing transactions on the state. The *state* consists of a set of bindings as named pointers into the graph managed by the graph reducer. A transaction is executed by first binding free variables in the transaction to values in the state, and then updating the bindings in the state. The transaction manager is not concerned with the reduction of values in the state, as this is the concern of the graph reducer. In our implementation the transaction manager takes the form of a procedure that takes a transaction and returns an unreduced result expression that has been bound to the state.

The transaction manager may be invoked concurrently from multiple threads, so it has to ensure that concurrent operations on the state are executed correctly. One approach to ensure atomicity is to construct new states non-destructively. We keep a single pointer to the current state, and we update this pointer if a new state is available. This allows read transactions to proceed without additional synchronisation, because once a pointer to a state has been obtained, this state cannot change. Updating the state concurrently requires some synchronisation to avoid data races. One approach is to serialise updates through mutual exclusion. However, the main drawback of this approach is that the state can not be updated in parallel.

Another approach is to update the state optimistically: read the current state, construct a new state, and update the state pointer only if it is still the same as the state that was read initially. The last step can be implemented atomically using a compare-and-set operation. If updating the state fails, the operation can be retried until it succeeds. In this basic form, performing updates optimistically may lead to many failed updates when there is a high level of contention. This can be improved by computing new states lazily, as to minimise the duration between reading and updating the state pointer. This has the additional benefit that multiple updates can be processed in parallel. However, this also defers construction of the new state to read transactions, potentially leading to higher response times. In our prototype we have implemented optimistic updates, but we have not yet implement lazy construction of states.

Persistence While not the main focus of this paper, we shortly discuss the implementation of persistence in our runtime system. Our implementation of persistence consists of two parts: journaling transactions and snapshotting the state. Journaling is a standard method in databases for guaranteeing durability [3]. The idea of journaling is to write a transaction to a log file in persistent memory before it is executed. If the system crashes and starts up again, it recovers by

re-executing all transactions in the log to obtain the same state as prior to the crash. To guarantee durability to the user, the system must ensure that a transaction is stored in the journal before confirming the execution of the transaction to the client. Furthermore, transactions have to be journaled in the same order they are executed.

In theory, having an initial state and a journal starting in this initial state is enough to reconstruct the state at any point in time. However, in practice this is not really sufficient: the log grows beyond bounds as entries can never be removed, and moreover it is extremely inefficient to re-execute all transactions when a journal grows large. For this reason, we want to store reduced forms of states as a *checkpoint*, so that the system only has to recover from the last checkpoint. A simple method for constructing a checkpoint is *snapshotting*, where we serialise the state and write this to persistent memory. However, a complication here is that snapshotting a state containing suspended computations, while concurrently reducing the state, can lead to a loss of sharing in computation as well as data in the snapshot. A simple solution to solve this problem is to reduce the state to normal form before snapshotting it, which is also the approach that we have implemented in our prototype. A limitation of snapshotting is that it only supports states that fit in main memory. Supporting states larger than main memory remains future work.

4 Graph Reduction with Dynamic Bindings

A special feature of our graph reducer is that it allows bindings to be created and removed dynamically. This allows ad-hoc transactions to create and remove bindings, as well as allowing local bindings in transactions.

In a graph reducer based on template instantiation, for every binding a *template graph* is constructed, and free variables are resolved by looking up the template in a global map [9]. We face two problems when we want to add and remove bindings dynamically in a graph reducer based on template instantiation. In a system that executes transactions eagerly, we can simply remove deleted and local template graphs directly after executing the transaction. However, when delaying the execution of lazy transactions, these bindings may still be in use by previous transactions that have not finished executing. Another problem is that the names of template graphs may collide, e.g., when multiple transactions use the same names for local functions.

Our solution to solve both these problems is to resolve free variables to templates statically, that is, instead of referring to templates indirectly by their name, we make templates anonymous by pointing to them directly. The result of this is that we obtain a single graph that is both our reduction graph as well as containing our templates. When a transaction introduces a new binding, we create an anonymous template for its expression, and we refer to this template by its pointer. This has two advantages compared to the standard template instantiation approach. First, there is no possibility of name collisions. And second, a

standard garbage collector can be used to clean up anonymous templates when they are not used anymore.

Implementing this scheme is not as trivial as it might seem. When we instantiate a template, we have to know where its boundary is so that instantiation terminates at that boundary. In traditional template reduction the boundary of a template is marked by free variable nodes. A complication with anonymous templates is that the free variable nodes have been resolved, so we no longer know where the boundary of the template is. To resolve this issue, we can resolve free variable nodes with special nodes to mark the end of the template scope [5]. Additionally, When reducing a redex, we have to know whether the function involved is a template that must be instantiated, or an instantiated graph that must be reduced. To mark the beginning of a template, we let templates start with a `Template` node. If a free variable is resolved to a non-constant template, we will always encounter this `Template` node at the start of the template. So, a `Template` node also indicates that we are at the boundary of the template that we are instantiating. Thus, end-of-scope nodes can be omitted for free variables that refer to non-constant templates by letting a `Template` node act as an end-of-scope node.

5 Low-Latency Parallel Graph Reduction

This section describes a scheduler that guarantees low response times for transactions. In contrast to traditional work-stealing schedulers, our scheduler guarantees that when a transaction is scheduled, progress is made on that transaction. In this section we first discuss our goals, next we discuss how we distribute work in our approach, after that we discuss result sharing between threads, and finally we analyse and compare our approach to work stealing and provide directions for further investigation.

Distributing Work The goal of a scheduler in parallel graph reduction is to distribute work among a set of reduction threads. We do not want idle threads as long as there is work to be done. Additionally, we want to avoid that multiple threads work on the same task, as this leads to cache contention.

The problem of work distribution can be modelled using task graphs. A *task graph* is a directed acyclic graph where nodes represent tasks, and edges represent dependencies between the tasks. In the context of graph reduction, redexes are tasks, and a redex a depends on another redex b if b is an argument of a and a is strict in b. The reduction of a redex may as a result create a new redex that has to be reduced. Furthermore, when a redex has been reduced, the redex is replaced by its result as to share results. To reduce a node, first the nodes that it depends on need to be reduced. The goal of graph reduction is to reduce the root node of a task graph.

In the context of TFL's, we have multiple root nodes corresponding to the result expression of transactions that are executed in parallel. As transactions operate on the same state, these roots share a common sub-graph between them.

Threads can see the results produced by other reduction threads, which allows them to cooperate on reducing the state. If multiple threads reduce the same part of the state unaware of each other, they may duplicate work. In work stealing, this is solved by marking tasks as being in progress, allowing threads to work on another task when they see that a task is in progress. If a thread is out of work because it is blocked, it may steal a task from another thread to work on that. The problem with this approach in our context is that a thread may work on a task that is not relevant to the transaction it has been scheduled to work on. In the next paragraph we discuss load balancing in our graph reducer, which solves this problem.

Randomisation Sometimes tasks depend on multiple other tasks. The order in which these dependent tasks are executed does not affect the result. In our graph reducer we attempt to distribute work among the threads by making each threads execute these dependent tasks in a different order. The intended effect of this approach is that the reduction threads are working in different parts of the graph, such that contention between threads is kept to a minimum. The idea of randomisation and result sharing is not new, and has already been applied successfully in the context of model checking for the parallel exploration of state-spaces [2], however to our knowledge this method has not yet been applied to graph reduction.

There are many strategies that can be used to determine the reduction order of a thread. In general there are two aspects to such a strategy: the way in which threads make different decisions with respect to each other, and the possible reduction orders that can be chosen. Theoretically, any permutation of the dependent tasks can be chosen as a reduction order. However, in our implementation, the only possible reduction orders that can be chosen is reducing dependent tasks from left to right, or from right to left. We have chosen this approach, because this requires less decisions to be made than deciding on a permutation. One strategy to make different decisions is to use a (pseudo-)random number generator for each thread. Alternatively, it is also possible to make threads communicate to actively avoid them from moving into the same part of the graph. A drawback of random number generators is the time it costs to compute random numbers. A drawback of communicating between threads is the associated communication overhead, but on the other hand it also tries to actively avoid collisions between threads.

In our implementation we have chosen to implement the communication approach, as this is relatively simple to implement. In each primitive function node we maintain a boolean flag indicating whether to reduce arguments from left to right, or from right to left. When a thread wants to reduce a node, it determines in which order to reduce based on the flag, and it flips the flag such that the next threads takes a different order. As an example, consider the evaluation of the addition node:

```
1 whnf(Add(left, right, left_to_right)) : Node {
2   left_to_right ← !left_to_right;
3   if(left_to_right) {
4     l ← whnf(left); r ← whnf(right);
5   } else {
6     r ← whnf(right); l ← whnf(left);
7   }
8   return Int(l.value + r.value);
9 }
```

In this example we have a data structure `Add` which has a left and right expression, and includes a field `left_to_right`. Every time a thread starts reducing this node, the `left_to_right` field is negated. Assuming a tree of additions, and assuming that no race conditions occur, the first thread will go `left` → `left` → .... The second will go `right` → `left` → ..., the third will go `left` → `right` → ..., the fourth `right` → `right` → ..., etc. Assuming that the computation is shaped as a balanced tree, this provides an approximately uniform distribution of the reduction threads.

Sharing Results An important aspect of our graph reducer is that reduction results are shared between threads. Multiple threads may reduce a redex at the same time, which may lead to duplicate computations, as well as duplicate results [4]. While duplicate computations do not affect the final result of a computation in a pure functional program, duplicate results may lead to loss of sharing, resulting in a potential n-fold increase in computation time and memory usage, where n is the number of reduction threads.

A method to avoid duplicate computations, and thus to avoid duplicate results, is to ensure mutual exclusive access to a redex while it is being reduced. However, if a reduction thread blocks access to some node, all other reduction threads are unable to proceed beyond this node and may become idle. To make this approach work, the reduction of strict arguments required for the reduction of the redex must not be part of the mutual exclusive region, as to allow multiple threads to reduce these strict arguments in parallel. The problem with this approach is deciding what to do when a thread is blocked. The thread may wait for the result to become available, but the blocking thread may be pre-empted. Alternatively the thread could work on some other task, but this incurs overhead for finding another task to work on.

An alternative approach is to allow duplicate computations, and ensure that only one of the duplicate results is used to maintain sharing. An obvious drawback of this approach is that computational time may be wasted, but this only occurs if multiple threads are working in the same part of the graph. However, a major benefit of this approach is that the algorithm is wait-free, as each reduction thread can always make progress. Therefor we have chosen to implement this approach in our prototype.

We now discuss our implementation of result sharing. We share results through a special result sharing node that has a pointer to either an unreduced expression graph, or its reduced form. A result sharing node is inserted before every

redex in the graph. When a thread computes a reduced form of a shared node, it updates the sharing node to point to the new result. We ensure that sharing is maintained by checking if another thread has already computed a result before updating a sharing node. A minor complication is that for some nodes, we have implemented the `whnf` procedure so that it tail-recursively call itself in order to obtain the WHNF of intermediate results. In order to share these intermediate results in result sharing nodes, we use a procedure `reduce` that performs a reduction step towards WHNF, but which does not perform this tail recursive call. The implementation of `whnf` for the result sharing node is as follows:

```
1  whnf(Sharing(shared)) : Node {
2    local ← shared;
3    reduced ← reduce(local);
4    while(local ≠ reduced) {
5      if(compareAndSet(shared, local, reduced)) {
6        local ← reduced;
7      } else {
8        local ← shared;
9      }
10     reduced ← reduce(local);
11   }
12   return local;
13 }
```

This procedure first fetches the current shared node from the sharing node, and reduces it using the `reduce` procedure. If `local` is equal to `reduced` then `local` is already in WHNF, and we are done. In the other case, we have to update the `shared` variable of the result sharing node. We use `compareAndSet` (CAS) to atomically update the node, where we compare the value of `shared` with `local`. If the CAS operation fails, then `shared` $\neq$ `local` meaning that some other thread has already updated the sharing node with a result. In this case we continue reduction with that result by setting `local` $\leftarrow$ `shared`. In case the CAS operation succeeds, we can continue reduction with our own result by setting `local` $\leftarrow$ `reduced`. This algorithm guarantees that the final result is shared. Also, this algorithm is guaranteed to terminate if reduction of the shared node terminates, because in every loop either the current thread reduces the shared node by one step, or another thread reduces the shared node by at least one step and the current thread uses that result.

Analysis and Future Work An interesting property of this method of work distribution is that a reduction thread is never blocked. In the context of transaction processing, this means that a reduction thread assigned to a transaction always makes progress on reducing that transaction. This is in contrast to traditional work stealing approach, where a reduction thread may steal work belonging to another transaction, thereby not making progress on the transaction it was assigned to. It may be possible to adapt work stealing to provide similar guarantees, but this remains future work.

A drawback of randomisation and result sharing is that if multiple threads are used for a sequential task, there will be high levels of cache contention between the threads, leading to a lower performance than using a single reduction thread. Additionally, in the work-stealing approach, a thread may jump straight to where work is to be done, whereas this is not possible in our current implementation.

A direction for the improvement of our implementation is that a threads can see that another thread has already performed the work when the CAS operation fails. In this case, a thread can choose to work on some other task instead. This can for example be implemented by restarting a computation from the root of the graph so that the thread can move into a different branch of the tree. Preferably a thread should become idle if there is no more work to be done, but this is problematic to implement in this approach. Other directions for improvement are to investigate other methods for determining the reduction order of a thread, investigating an implementation that avoids duplicate computations instead of repairing sharing, and to investigate methods to reduce communication overhead by performing more work between result sharing.

6 Evaluation

This section presents some experiments that we have performed on our prototype implementation. Given the limitations of our current implementation, we have performed the experiments with artificial, and relatively small benchmarks[1].

Parallel Graph Reduction In our first set of benchmarks we measure the relative speedup of our parallel graph reducer. We have implemented two parallel algorithms: *nfib*, which naively computes the nth Fibonacci number, and *treesize*, which computes the size of a binary tree. For both algorithm, we have implemented a variant in our prototype language, as well as a native function implemented in Java to simulate the performance of a compiled language.

We measured the relative speedup by executing the algorithm with 1 up to 48 threads, and dividing the measured execution time with the single-threaded execution time. We noticed increasing amounts of variation in execution times with an increasing number of threads. The results shown in Figure 1 are the median of several measurements. The dashed line shows the ideal relative speedup assuming linear speedup as the number of threads increases. The vertical lines show the boundaries of the NUMA nodes of our testing system. We see that the relative speedup for both algorithms is nearly ideal when all threads are able to run on a single NUMA node. When more than one NUMA node is used, the increase in speedup drops for the treesize benchmark. We suspect that this happens because the threads have to access memory on another NUMA node when scaling beyond one node, which takes longer than accessing memory locally.

[1] All experiments are run on quad AMD Opteron 6168 system with 48 cores divided over 4 processors, using scientific linux, running Oracle HotSpot JVM version 1.7.0 build 147.

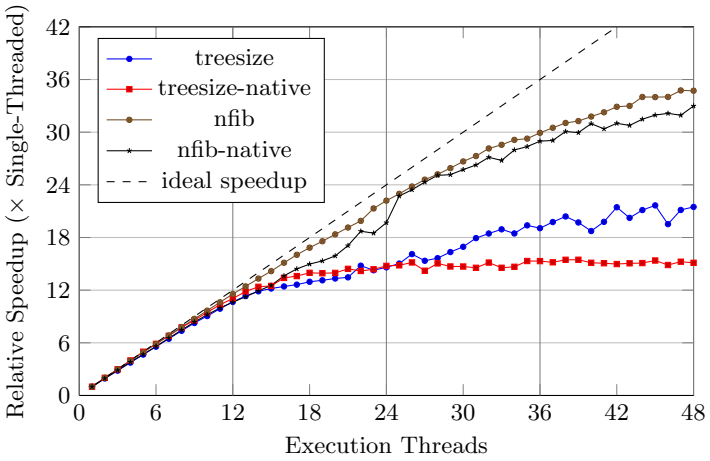

Fig. 1. Relative speedup of our parallel graph reducer.

Next, we measured the overhead of our parallel graph reducer compared to a serial version of our graph reducer that has randomisation and result sharing disabled. Table 1 shows the concrete execution times, as well as the overhead obtained by dividing the execution time of the parallel graph reducer by that of the serial graph reducer. In some cases the overhead is quite high. We think that there is a lot to be gained in this aspect by sharing results less often. However, this remains future work.

Transaction Throughput In our next set of benchmarks, we investigate the transaction throughput of our prototype. The benchmarks have been run on a state that is initialised with an associative map, implemented as an unbalanced binary search tree that maps the keys $0, \ldots, 100,000$ to the initial value zero. The map is initialised by randomly inserting the elements, as to obtain a random balancing. We measure the transaction throughput by issuing read and update transactions. An update transaction increments a random value, and a read transaction reads a random value. In these benchmarks we do not write updates to the journal.

We found that reads scale very well, from 36,000 transactions per second with one thread up to 1,109,000 transactions per second with 48 threads. However, updates scale worse, from 15,857 transactions per second using one thread, to 93,139 transactions per seconds with eight threads, and back to 49,714 trans-

	treesize	**treesize-native**	**nfib**	**nfib-native**
Serial	2666 ms	819 ms	3294 ms	626 ms
Parallel	3243 ms	1291 ms	4162 ms	819 ms
Overhead	21.6%	57.6%	26.4%	30.1%

Table 1. Running time of parallel reduction compared to serial reduction.

actions per second using all 48 threads. We found that synchronisation is the bottleneck here due to multiple NUMA nodes contending for a single lock on the state. This has been validated by running the benchmark with a limited number of cores, where no slowdown is observed after the peak performance has been reached. A NUMA aware lock might solve this problem, but we could not test this, as we did not have an implementation of this available in Java.

Memory Usage While experimenting with our prototype, we found that a large amount of redexes may build up in the state due to lazily evaluated updates. In practice, this can lead to a stack overflow when reading after many updates. Some redexes may also never be evaluated at all. An example of the latter is that a `map` function applied to a binary search tree may leave redexes applied to `Leaf` nodes, which may not be accessible for read operations.

A partial solution to solve these problems is to reduce the state to normal form after each transaction. In order to implement this efficiently, we need to keep track of the parts of the state have already been reduced to normal form, because we want to avoid reducing parts of the state that are already in normal form after each transaction. This is implemented in our prototype by maintaining a flag on data nodes to indicates whether it is in normal form. With this in place, it is still possible for redexes to build up in the state faster than that the state can be reduced. We do not know if this is a problem in practice, and resolving this problem remains future work.

7 Conclusions

We have described a general approach to adapt functional languages for describing functional transactions, and we have shown how such a language can be used to construct a simple database systems and perform transactions on this database. Additionally, we have described the implementation of a runtime system for such a language that allows parallel execution of transactions through lazy evaluation of states. Our runtime system consists of a transaction manager that binds expressions to the state concurrently, and a mechanism for ensuring durability of the effects of transactions and persisting an in-memory state to disk. A major feature of our runtime system is the graph reducer that allows bindings to be created and removed dynamically. This allows ad-hoc transactions to be executed lazily, while allowing local bindings to be defined. Furthermore, the graph reducer features a scheduler that guarantees low latency execution of transactions. Work is distributed between threads by randomising their reduction order, and threads cooperate by sharing results.

Experiments with our runtime implementation show promising results, and we can conclude that using functional languages for parallel transaction processing is possible. However, we have also identified some issues that remain to be solved. While the model of persistent functional languages work in theory, issues arise in practice when there is a finite amount of memory, as lazy evaluation may lead to many of unevaluated thunks in the state.

Future Work The work described in this paper is only the first step of a much larger project. There are many ways in which this work can (and will) be continued. First, a concurrent functional balanced search tree is required for the efficient implementation of indices in functional databases. We plan to develop theory for concurrency in functional transactions, and investigate the development of algorithms in this setting. Moreover, we want to investigate the possibility of automatically optimising expressions to minimise blocking between transactions. Additionally, we will investigate other methods for introducing concurrency in a functional setting. One idea is to split a transaction into multiple transactions to allow for optimistic concurrency control: the first transaction performs a non-blocking optimistic computation, while a second transaction updates the state using the result of the first transaction only if the state has not been changed.

We will also further develop our runtime implementation. Currently it only supports untyped languages. We will explore how TFL's can be typed correctly and how type-checking can be performed efficiently in the runtime. Additionally, we want to investigate methods to cope with limited amounts of memory, and we want to investigate swapping parts of the state to persistent memory to allow states that are larger than main memory.

Acknowledgements We would like to thank Stefan Blom, Jaco van de Pol and Arjan Boeijink for providing valuable advice and feedback on this paper.

References

1. acid-state manual: http://happstack.com/docs/crashcourse/acidstate.html, August 2012.
2. M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07. IEEE Computer Society, 2007.
3. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.
4. Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*. ACM Press, September 2005.
5. Dimitri Hendriks and Vincent van Oostrom. Adbmal. In *Proceedings CADE-19*, Lecture Notes in Artificial Intelligence. Springer, 2003.
6. D. McNally. *Models for Persistence in Lazy Functional Programming Systems*. PhD thesis, University of St Andrews, 1993.
7. R. S. Nikhil and M. L. Heytens. Exploiting parallelism in the implementation of Agna, a persistent programming system. In *Proceedings of the Seventh International Conference on Data Engineering*. IEEE Computer Society, 1991.
8. R.S. Nikhil. Functional databases, functional languages. In *Proceedings of the First Workshop on Persistent Objects*. Springer-Verlag, 1985.
9. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., 1987.
10. Michael Alan Rainey. *Effective scheduling techniques for high-level parallel programming languages*. PhD thesis, University of Chicago, 2010.
11. P. Trinder. *A Functional Database*. PhD thesis, University of Oxford, 1989.