Enabling Speculative Parallelization via Merge Semantics in STMs

Kaushik Ravichandran
Georgia Institute of Technology
kaushikr@gatch.edu

Santosh Pande
Georgia Institute of Technology
santosh@cc.gatech.edu

Abstract

STM (Software Transactional Memory) systems can be used to speculatively parallelize irregular applications such as those based on graphs and trees. While the transactional paradigm is suitable for such speculative parallelization, STMs do not deal with the semantics of speculation. In this paper we introduce *merge* semantics for speculations.

STMs optimistically execute code and monitor the execution for memory access conflicts. On detecting a conflict between a pair of transactions the STM performs a rollback on one of them, discarding all the work that has been done until that point. This wastage leads to performance and scalability issues when there are many transactional conflicts. The key insight of this paper is that it is sometimes possible to salvage partially completed work and *merge* it with the other transaction.

In this paper we motivate the need for the *merge* construct and develop the ideas behind it. We propose a simple API which can be used to define the *merge* operation. We discuss applications which would benefit from this construct. We implement the supporting framework and demonstrate its benefits on a Connected Components benchmark and a Minimum Spanning Tree benchmark. We report performance improvements of more than 75% when compared to a traditional STM parallelization of the Minimum Spanning Tree benchmark. Our framework also demonstrates excellent scalability when there are a large number of conflicts, a scenario where traditional STM systems do not scale well.

1. Introduction

Multi-core processors are ubiquitous but programming for these systems is notoriously difficult. With pitfalls like deadlocks, livelocks, scheduling issues and race conditions, it is easy to see why most programmers shy away from parallel programming. This has led to a scenario where parallelism available at the hardware level is quickly increasing but where parallel software models are inadequate. This makes it difficult for average programmers to harness and exploit the available parallelism. However, HPC applications, such as dense matrix applications, have seen great success in harnessing increasing amounts of parallelism. These applications have tremendous amounts of parallelism which can easily be exploited and are typically known as *regular* parallel applications.

Irregular parallel applications on the other hand are not so easy to parallelize. These applications typically rely heavily on pointer-based structures such as graphs and trees. An important characteristic of these applications is that the exact elements and therefore memory locations accessed are heavily data-dependent and can not be known until run-time. This cripples potential static analyses which are typically used to parallelize regular parallel applications. These irregular parallel applications, however, also benefit greatly from parallelization [24].

STM systems [29] can be used to speculatively parallelize irregular applications in limited circumstances. STMs provide an *atomic* construct that provides an illusion of atomicity to code executed within its scope. The programmer encompasses critical sections (called transactions) within these constructs. The STM optimistically executes transactions and monitors for conflicts. On detecting a conflict between a pair of transactions it will abort one of them and retry, while the other continues. We will refer to the transaction that will abort as the *aborting transaction* and the one that will continue as the *continuing transaction*.

STMs provide significant performance benefits by enabling parallelization of codes with limited programmer effort and are becoming a popular construct in parallel programming models. Languages like C++ [18], Java [8] and even newer languages like Chapel [30] provide STM interfaces.

Software transactions provide the programmer with atomicity and isolation properties to achieve serial consistency while writing parallel code. In theory, transactions can be used to execute highly speculative algorithms. However, overheads are so high that transactions are often not used for speculation at an algorithmic level.

Overheads of STMs Two dominant sources of overheads are:
1. Overhead due to logging All accesses to memory need to be monitored to detect conflicts.
2. Overhead due to rollbacks after a conflict

(a) The inherent cost of rollback There is an inherent cost of rollback that is incurred, typically due to having to restore the memory state.

(b) The cost of the lost work The work that was executed in the aborting transaction until the point of conflict needs to be thrown away and this leads to decreased parallel efficiency.

In this paper we target the inefficiencies due to 2.(b). One of the key insights of this paper is that in some applications work does not need to be thrown away when two transactions conflict but can rather be merged. In this paper we propose a merge construct to allow programmers to salvage partially completed work in an aborting transaction, merging the states of two conflicting transactions. This has the potential to dramatically reduce overheads due to rollbacks in STM systems thus enabling programmers to write and develop highly speculative parallel algorithms.

To build on the intuition behind algorithmic speculative parallelization and the merge construct let us consider the Connected Components Problem.

1.1 Connected Components Problem

The Connected Components Problem is to find all the connected components in an undirected graph. Each component in the graph is to be marked with a unique component number. After detecting each individual component the number of nodes and the nodes from that component should be printed. Figure 1 shows the pseudocode for this problem.

```
// Called by the main function.
void connected_components() {
  for (int i = 0; i < nodes.size(); ++i) {
      generate_component(i)
  }
}

// Use a DFS strategy to mark all the connected nodes.
void generate_component(int i) {
  stack<int> nodes_stack;
  set<int> marked_nodes;
  nodes_stack.push(i);
  while (!nodes_stack.empty()) {
    Node* node = nodes[nodes_stack.pop()];
    // Check if the node has already been marked.
    if (node->component != -1) {
      continue;
    }
    // Mark the node with the component number.
    node->component = i;
    marked_nodes.insert(node->id);
    for (each neighbor of node) {
      nodes_stack.push(neighbor);
    }
  }
  print("Component_number:", i, "size:",
       marked_nodes.size(), "nodes:", marked_nodes);
}
```

Figure 1. Pseudocode for connected components problem

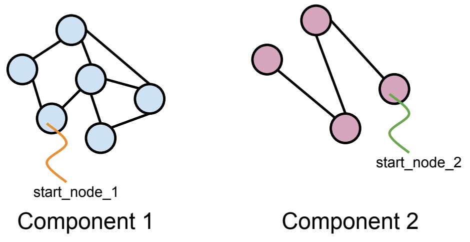
The code is fairly straight forward and employs a DFS strategy to detect each component. This algorithm cannot be parallelized in the traditional data parallel manner. Each iteration of the loop is dependent on the previous iteration and the execution is highly data dependent. This cripples any static parallelization techniques. There are specialized parallel DFS algorithms [2, 3, 17]. However, these parallel algorithms are typically complicated and not in the area of expertise of the programmer.

1.2 Speculatively Parallelizing Graph Algorithms

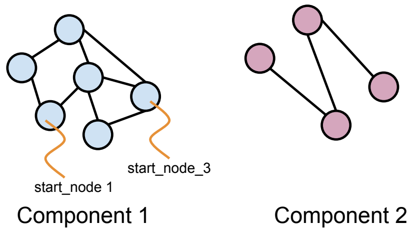
There is an extremely simple and intuitive method to speculatively parallelize some graph algorithms using STMs. The *basic idea is to speculatively launch multiple parallel threads in different parts of the graph, each executing the same code as the sequential version* as software transactions. These threads run normally until they conflict with another thread. On a conflict the STM system kicks in and aborts one of the transactions, thus leading to a correct parallelization.

To elaborate, consider graph applications which traverse the edges of a graph performing some arbitrary operation on the nodes (the connected components problem, is one such application). We can speculatively launch multiple threads at different parts of the graph (see Figure 2) each executing the same code as the sequential version. If it turns out that the threads were speculatively launched on disconnected parts of the graph (see Figure 2(a)), they will not conflict and this approach leads to a correct parallelized execution. However, if the speculatively launched threads operate on the same component they will eventually conflict (and our STM system performs a rollback on one of them).

This is a very powerful notion, since we have obtained a sort of data parallelism over an irregular data structure. This type of data parallelism is much easier for the programmer to reason about and deal with when compared to specialized parallel algorithms. As long as a sufficient number of speculative threads operate on disjoint parts of the graph we can attain a speedup. On the flip side, each time two threads conflict, the STM aborts one of them and its work is lost leading to a less efficient solution.



(a) Threads with non-conflicting start nodes



(b) Threads with conflicting start nodes

Figure 2. Threads with different start points

This type of speculative parallelization is susceptible to the typical scalability concerns of STM applications when there are a large number of conflicts. A measure of scalability is parallel efficiency, P_e, defined as:

$P_e = S/(p * T(p))$

Where S represents the wall clock time of a sequential execution, p is the number of processors and $T(p)$ is the wall clock time of an execution on p processors. As the number of conflicts increase, there is an increase in overhead due to both the loss of useful work and time spent in servicing these conflicts, consequently increasing $T(p)$.

Instead of discarding the work performed by the aborting transaction if we can *merge* it with the continuing transaction, we would get a significant reduction in the overhead due to conflicts, thereby decreasing $T(p)$ and consequently increasing P_e. This enables the execution of highly speculative algorithms efficiently without performance loss due to mis-speculation. We believe the *merge* construct is key in enabling speculative parallelization to achieve performance gains. Algorithms which have potential for speculative parallelization such as connected components are often used as kernels in a wide variety of areas such as video processing [21], image retrieval [12], traffic monitoring [5], object recognition in 3D images [9] and many more. The merge construct can not be applied to all STM based applications. We discuss the properties of the merge construct in Section 4.

1.3 Contributions

In this paper, we enable speculative parallelism via the merge construct in STM systems. Specifically we make the following contributions:

- We recognize the need for speculative parallelization and the role *merge* plays while developing the basic ideas behind it.
- We demonstrate the usefulness of merging in a connected components problem and discuss the subtleties involved.
- We propose an API that programmers can use to express the merge construct in transactional applications.
- We discuss the properties of the merge construct.
- We demonstrate our framework on a connected components benchmark and a minimum spanning tree benchmark, while reporting significant speedups and excellent scalability.

The remainder of this paper is organized as follows, in Section 2 we discuss the merge construct in detail. In Section 3 we discuss the API. We discuss the properties of the merge construct in Section 4. In Section 5 we discuss the benchmarks, experimental results and observations. We discuss related work in Section 6, then conclude and present future work.

2. Merge Construct

In this section we'll discuss the merge construct in detail using the connected components problem introduced in the previous section. Figure 1 shows the pseudocode for this problem. This problem is also amenable to the type of speculative parallelization discussed in the previous section, i.e. speculatively launching threads on different parts of the graph (as transactions) and relying on an STM to rollback on conflicts. If at least a few of the speculatively launched threads start in disjoint components we would have obtained some amount of parallelism. The underlying reason why no parallelism is obtained when two transactions conflict is because the work performed by the aborting transaction is thrown away. We will now discuss how to merge the work performed by the aborting transaction into the continuing transaction using the merge construct.

The merge construct consists of user defined MERGE and UPDATE functions and a set of APIs exposed by our framework: MAKE_AVAILABLE, SAFE_UPDATE_POINT and SNAPSHOT (optional).

MERGE The job of the MERGE function is to take the work from the aborting transaction and put it into the continuing transaction. A preliminary MERGE function for the connected components problem is shown in Figure 3 (we will refine this as we go along). The essence of the MERGE function is extremely simple. It simply takes the $nodes\_stack$ and the $marked\_nodes$ from the aborting transaction and adds it to the corresponding data structures in the continuing transaction.

```
merge(transaction t1, transaction t2) {
  t1.nodes_stack.add(t2.nodes_stack);
  t1.marked_nodes.add(t2.marked_nodes);
}
```

Figure 3. Preliminary pseudocode for the MERGE function. $t1$ = continuing transaction, $t2$ = aborting transaction.

Our framework automatically invokes the user defined MERGE function once on every conflict. The state of the aborting transaction is guaranteed to still be in tact during the execution of MERGE. MERGE is executed by the aborting thread before it terminates. Figure 4 shows the execution schedule of two conflicting transactions: $t1$ and $t2$.

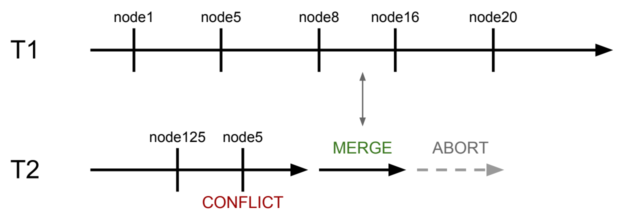


Figure 4. Transaction Schedule for T1 and T2

As you can imagine, all kinds of weird race conditions can arise with such a simple implementation. We have to deal with two main issues. Firstly, we need to ensure that the $nodes\_stack$ and $marked\_nodes$ in the aborting transaction ($t2$) are consistent when MERGE is executed (subsection: 1. Consistency). Secondly, MERGE needs to be able to access the $nodes\_stack$ and $marked\_nodes$ in the continuing transaction ($t1$) in a synchronized manner (subsection: 2. Synchronization).

1. Consistency To tackle the first issue let us define the state, DS, of the aborting transaction ($t2$) at any point in its execution. We define DS as the union of all the data structures from the aborting transaction used in the MERGE function. In our example:

DS = ($nodes\_stack$, $marked\_nodes$).

Note, that we are only concerned about the semantic state of these data structures and not their actual in-memory representation. For example, let's say the $marked\_nodes$ set contains integers 1 and 2. Whether it is stored as (1, 2) or (2, 1) internally is not relevant to us and they are semantically equivalent. For our purposes, DS completely represents the state of the aborting transaction with respect to the merge construct. DS can either be a:

Valid State is defined as a state, in which all the data structures in DS are currently consistent with each other and are safe to use in the MERGE function.

Invalid State is defined as a state in which the data structures in DS are currently inconsistent with each other and are not safe to use in the MERGE function.

When MERGE is executed, if $t2$'s DS is in a valid state a consistent set of data structures are available. However, if DS is in an invalid state an inconsistent set of data structures will be available and can not be used by MERGE.

To get a better understanding of what valid states are let us first see how an invalid (inconsistent) state might arise in MERGE. Consider again the pseudocode from Figure 1. The only lines in the pseudocode which can potentially cause an STM conflict (and hence cause MERGE to be executed) are those with read/write operations on global memory. In particular, lines 16 and 20 read the component number of a node from global memory and write to it. Only at these two lines can a conflict potentially occur. Figure 5(a) shows this snippet again for convenience.

```
15 // Check if the node has already been marked.
16 if (node->component != -1) {
17   continue;
18 }
19 // Mark the node with the component number.
20 node->component = i;
21 marked_nodes.insert(node->id);
22 for (each neighbor of node) {
23   nodes_stack.push(neighbor);
24 }
```

(a) Original code

```
15 // Check if the node has already been marked.
16 if (node->component != -1) {
17   continue;
18 }
19 // Mark the node with the component number.
20 marked_nodes.insert(node->id);
21 node->component = i;
22 for (each neighbor of node) {
23   nodes_stack.push(neighbor);
24 }
```

(b) Modified code

Figure 5. Code snippets

Figure 5(b) shows the same code, but with lines 20 and 21 interchanged in position. While this alternate implementation is still a correct sequential implementation *and* a correct parallel STM implementation it can lead to an invalid state while executing MERGE. Let's see how. Potential STM conflicts can now occur on either line 16 or line 21 (Figure 5(b)). Let's say a conflict occurs on line 21 while processing $node_a$. $node_a$ would have already been added to the $marked\_nodes$ set, semantically indicating that the node has already been processed. However, since the conflict occurs on line 21 and its execution ceases at that point, $node_a$'s neighbors will not be inserted into $nodes\_stack$. Once MERGE (Figure 3) executes after the conflict on line 21 it will append the contents of $t2$'s data structures into $t1$'s. $t2$'s $marked\_nodes$ indicates that $node_a$ has been completely processed, while its $nodes\_stack$ indicates that $node_a$ has not been processed. Now, when the continuing transaction $t1$ continues executing its code, there is a possibility that all the nodes which are neighbors of $node_a$ get skipped since they were never added to the traversal stack $nodes\_stack$ in the aborting transaction. This is an incorrect execution due to an invalid state.

While this example is contrived and it is simple to see that in the natural expression of the transaction the states are in fact always valid during MERGE, it might be the case that for some transactional applications it is extremely difficult to write code such that state (DS) is valid during MERGE execution. For such situations we provide the programmer with a SNAPSHOT API (described shortly).

To determine if a given piece of code will result in a valid state or invalid state during the execution of MERGE the programmer needs to:

a. Identify conflict points Conflict points are those lines in the code which read/write to global memory. Depending on the STM system that is begin used these statements are typically wrapped in special STM API calls. Hence, this is normally an easy step.

b. Identify the possibility of an Invalid State The programmer needs to take each of the conflict points identified in the previous step and check if executing the MERGE function after that conflict point will lead to an invalid state in MERGE. In our example, this is straightforward, since all changes to structures in DS are made after all potential conflict points. No invalid states can arise. However, it might not be so simple to make this determination for all applications. To deal with these applications we provide the programmer with the SNAPSHOT API.

A call to the SNAPSHOT API can be inserted by the programmer at any point that the data structures will be consistent in the main transaction code. This API takes a checkpoint of the current state of the data structures and ensures that only this state is accessed in any subsequent invocation of MERGE, thereby alleviating any concerns of invalid states. It is typically extremely simple to determine where to make the SNAPSHOT call, normally at the beginning or end of the loop or function call. For example, in our example it could be placed as the first line in the loop (see line 8 in Figure 6). Note that the SNAPSHOT call is not required in our implementation since valid states are guaranteed but is shown here for completeness.

Now that we have guaranteed that the aborting transaction's DS is in a valid state, we shift our focus to the issue of synchronization with the continuing transaction in MERGE.

2. Synchronization Consider again the execution schedules of two conflicting transactions, $t1$ and $t2$ in Figure 4.

In Figure 4, once $t2$ encounters a conflict, it will execute MERGE. MERGE needs access to the private states of both $t1$ and $t2$. In the previous section we discussed how we can guarantee that $t2$'s data structures (DS) are in a valid state. However, while MERGE is executing, $t1$ will be executing simultaneously and its data structures will be undergoing modifications. To overcome this, we allow the programmer to provide merge specific data structures for $t1$ in the transaction and identify them using our API. Only these merge specific data structures should be used in the MERGE code. Our framework guarantees that these merge specific data structures will only be used in a synchronized manner (i.e. with appropriate locking).

In our example MERGE code (Figure 3) instead of directly inserting into $t1$'s $nodes\_stack$ and $marked\_nodes$ set, merge specific data structures should be used. See lines 34 - 36 in Figure 6 for the final MERGE code.

UPDATE Since MERGE inserts into merge specific data structures, the continuing transaction $t1$ needs to incorporate this information into its actual data structures. To enable this, the programmer provides the second user-defined function: UPDATE. This function simply takes the information from the merged data structures and incorporates it into the actual data structures (lines 26 - 32 in Figure 6).

Our framework provides a SAFE_UPDATE_POINT API which can be called periodically by the transaction. Our framework keeps track of when MERGEs are performed and uses these SAFE_UPDATE_POINT calls as opportunities to invoke the UPDATE function. The UPDATE function is executed by $t1$ itself.

Let's recap by looking at the complete, final code sample in Figure 6.

```
   void generate_component(int component_number) {
 2 atomic {
     stack<int> nodes_stack, merged_nodes_stack;
 4   set<int> marked_nodes, merged_marked_nodes;
     MAKE_AVAILABLE(nodes_stack, marked_nodes,
          merged_nodes_stack, merged_marked_nodes,
          component_number);
 6   nodes_stack.push(i);
     while (!nodes_stack.empty()) {
 8     SNAPSHOT(nodes_stack, marked_nodes);   // OPTIONAL
       Node* node = nodes[nodes_stack.pop()];
10     // Check if the node has already been marked.
       if (node->component != -1) {
12       continue;
       }
14     // Mark the node with the component number.
       node->component = component_number;
16     marked_nodes.insert(node->id);
       for (each neighbor of node) {
18       nodes_stack.push(neighbor);
       }
20     SAFE_UPDATE_POINT();
     }
22   print("Component_number:", i, "size:",
          marked_nodes.size(), "nodes:", marked_nodes);
   }
24 }

26 UPDATE(transaction t) {
     t.nodes_stack.add(t.merged_nodes_stack);
28   t.marked_nodes.add(t.merged_marked_nodes);
     for (node in t.merged_marked_nodes) {
30     node.component = t.component_number;
     }
32 }

34 MERGE(transaction t1, transaction t2) {
     t1.merged_nodes_stack.add(t2.nodes_stack);
36   t1.merged_marked_nodes.add(t2.marked_nodes);
   }
```

Figure 6. Complete pseudocode for connected components with *merge*

Note that in the UPDATE function, by virtue of marking all the merged nodes, locks are acquired by the STM system on all of the newly added nodes. The MAKE_AVAILABLE API call allows the programmer access to any local variables inside the MERGE and UPDATE functions.

In the context of the connect components problem, while there is a significant performance improvement due to this kind of speculative parallelization technique using STMs, there is no performance improvement attained due to the merge construct. This is because the DFS algorithm is as quick as the merge function itself. We use this example to motivate our ideas. While the merge does not provide a performance benefit in this application, it is still applicable to other applications such as the Minimum Spanning Tree (MST) problem. This is discussed in more detail in Section 5.1. Similar modifications are made in the MST Problem and we demonstrate significant speedups due to *both* the speculative STM parallelization *and* the merge construct as well as sustained scalability at higher core counts. The key behind the improvement being the dramatic reduction in redundant work that needs to be performed due to the merge construct.

2.1 STM Requirements

Our framework requires the STM system to satisfy certain requirements. Some modifications need to be made to the underlying STM as well. Here are the requirements of the STM system:
1. Detect conflicts early The STM system must be capable of detecting conflicts and rolling back as soon as a conflict is detected. In a write through system locks are typically acquired during the write itself and hence conflicts will be detected early. In a write back system there are two popular locking schemes:

a. Commit Time Locking (CTL) postpones acquisition of the locks till commit time. Our model cannot use this kind of a locking scheme since conflicts need to be detected early, not during commit time. Detecting conflicts at commit time will negate any potential benefit of preventing the execution of redundant work.

b. Encounter Time Locking (ETL) In write back encounter time locking, though the writes to memory happen only during commit time, locks are acquired during the execution of the transaction itself, hence allowing conflicts to be detected early.
2. No-retry transactions The STM system must be capable of not retrying aborted transactions.

Most importantly we need either a write through or a write back ETL STM system to use our framework. We built our framework on top of TinySTM [14] and found it was straight-forward to make any additional changes. Another minor change we had to make is that TinySTM yields to the conflicting transaction before retrying, as a performance improvement technique. Since we do not retry after aborts we removed the option to yield.

3. Framework API

In this section we propose an API to support the merge construct and briefly discuss the implementation of our prototype. The five components of the API are summarized in Table 1.

MAKE_AVAILABLE This API can be called with an arbitrary number of arguments and each of the arguments becomes available for the programmer to use in the MERGE and UPDATE functions. For example, calling:

$MAKE\_AVAILABLE(value1,\ value2);$
will make $value1$ and $value2$ accessible inside the functions as $t1.value1$ and $t1.value2$.

MERGE function is the user defined function which merges information from the aborting transaction. It provides to the programmer the two conflicting transactions $t1$ and $t2$ as input.

$MERGE(transaction\, t1,\ transaction\, t2);$
$t1$ is the continuing transaction and $t2$ is the aborting transaction. Variables $t1$ and $t2$ provide access to the private members made accessible through the MAKE_AVAILABLE call (example: $t1.value$). This function is executed by the aborting thread.

SAFE_UPDATE_POINT can be called by the programmer to specify safe update point(s) in the transaction (single or multiple times). This is a simple function call:

$SAFE\_UPDATE\_POINT();$
These point(s) mark where in the transaction it is safe to incorporate the merge specific data structures into the main data structures. Our framework will call the UPDATE function only when a merge has actually been performed irrespective of the number of times this API gets called during execution.

UPDATE function is the user defined function that allows the programmer to incorporate the results from the merge specific data structure into the main data structures in a safe, synchronized manner.

$UPDATE(transaction\, t);$
Variables made available through MAKE_AVAILABLE are accessible inside the function as members of t (example: $t.value$). This function is executed by the continuing thread.

SNAPSHOT This is an optional API. As discussed previously (Section 2) it might be difficult for some applications to ensure that the aborting transaction is in a valid state during execution of the MERGE function. In such cases the programmer can simply insert a call to the SNAPSHOT API. Our framework will create a snapshot of the data structures specified as arguments. In all subsequent MERGE invocations, the snapshot created by the latest SNAPSHOT call will be used. This API can take an arbitrary number of arguments as input. For example:

$SNAPSHOT(value1,\ value2);$
This will create copies of $value1$ and $value2$. All calls to MERGE will use the latest snapshot values for $value1$ and $value2$ instead of their current values. Creating SNAPSHOTs will indeed increase the execution time since copies need to be made. However, the overall improvements we gain due to merging far outweigh this overhead (see Section 5).

Mutual Exclusion The MERGE and UPDATE functions are synchronized. Our framework uses a set of per transaction locks to ensure mutual exclusion and guarantee correctness. When MERGE is called our framework's internal locks on both transactions are automatically acquired (order is based on the transaction id which prevents deadlocks). When UPDATE is called our framework's internal lock on the continuing transaction is automatically acquired. This ensures that accesses to shared data members are safe.

A transaction is not allowed to abort when another transaction is MERGE-ing with it. This is ensured by acquiring the internal lock on the transaction before aborting. Further, if two transactions encounter conflicts at the same time, the order of lock acquisition determines order of MERGE-ing.

Our prototype assumes only one type of transaction in the code. To deal with multiple non-composable transactions we can name each distinct transaction with a simple type and use that to type the UPDATE and MERGE functions.

4. Properties

In this section we discuss the properties of the merge construct considering applications which have been speculatively parallelized using the technique discussed in Section 1.2. For the merge construct to operate as expected it must be *Correct* and *Efficient*.

Correctness The merge construct should be correct. That is to say that after merge is performed the application must be capable of continuing to execute normally. The result obtained after performing a merge should be the same as that, that would have been obtained without a merge. Formally, let the transaction T be operating on the graph G and its result represented by $T(G)$. Consider two instances of T: T_1 and T_2 operating on two disjoint sections of the graph G_1 and G_2. G_1 and G_2 evolve with time, as the transactions execute. If on continued execution of T_1 and T_2 the disjoint sections G_1 and G_2 intersect the MERGE function M is invoked.

API	Required	Summary
MAKE_AVAILABLE	Yes	Makes arguments available in MERGE and UPDATE functions.
MERGE	Yes	User defined function called by aborting transaction to merge information.
SAFE_UPDATE_POINT	Yes	Indicates point(s) in code where it's safe to invoke UPDATE.
UPDATE	Yes	User defined function called by continuing transaction to updates data structures.
SNAPSHOT	No	Creates copy of arguments for use in MERGE to ensure a valid state.

Table 1. Summary of API

M is invoked with the arguments (T_1, T_2) where T_2 is the aborting transaction and T_1 is the continuing transaction. Let the operation defined by M be $\odot$. To be correct the following property must be satisfied:

$$T(G_1 \cup G_2) = T_1(G_1) \odot T_2(G_2)$$

Where $=$ is a semantic equivalence and $G_1 \cup G_2$ is simple union of the graphs. This guarantees that the execution result after the merge is the same as that if there had been no merge.

Efficiency In addition to being *correct* merge must also be *efficient*. To obtain a performance improvement over an execution without merge, $\odot$ must be more efficient than running T itself. If T_2 is being aborted and it has operated on G_2 so far then:

$$t_e(T_1(G_1) \odot T_2(G_2)) < t_e(T(G_2))$$

Where $t_e(x)$ represents the execution time of x. In other words, it must be quicker to re-use the discarded work than to re-execute it in transaction T_1.

5. Experimental Evaluation

In this section we demonstrate the benefits of our approach through the Connected Components benchmark as well as the Minimum Spanning Tree benchmark.

All experiments were performed on a dual quad-core Intel Xeon E5540 (2.53GHz) machine running Ubuntu 10.10 using up to 8 concurrent threads. The benchmarks were compiled using GCC 4.4.5 (Ubuntu/Linaro 4.4.4-14ubuntu5) with the O3 flag set. OpenMP was used to parallelize the code. We used TinySTM 1.0.0 to protect the transactions. All results were obtained by averaging the results of 5 executions. TinySTM was configured with write back ETL (Section 2.1). Our results compare:

- Serial execution without any parallel overheads (No STM overhead)
- Parallel execution using STMs
- Parallel execution using STMs and *merge*

Datasets The input datasets for both the benchmarks were randomly generated undirected graphs. We parametrize the graph generation process using T, N, and X. Each graph is a forest containing T base trees. Each node in a tree has a random number of neighbors, selected uniformly from [0, N). If 0 is selected, it has no neighbors and the number of trees in the graph hence increases by 1. There are a total of X nodes in the graph which are evenly divided between the number of base trees.

5.1 Minimum Spanning Tree

Given a weighted undirected graph, the minimum spanning tree algorithm computes a tree from the graph whose weight is less than or equal to the weight of every other spanning tree for the graph. More generally, any undirected graph has a minimum spanning forest which is a union of the disjoint minimum spanning trees.

We use the same technique as described in Section 1.2 to parallelize the Minimum Spanning Tree benchmark. First with STMs to achieve parallelization over the serial execution and then adding the merge construct to provide even better performance. In this benchmark transaction durations are much larger than that of the connected components benchmark and the merge construct almost always provides consistent performance improvement.

```
void prims(int i) {
atomic{
  int tree_number = i;
  set<Pair*> tree_edges, merged_tree_edges;
  set<int> marked_nodes, merged_marked_nodes;
  MAKE_AVAILABLE(tree_edges, marked_nodes,
       merged_tree_edges, merged_marked_nodes,
       tree_number);
  int next_node = i;
  while (true) {
    SNAPSHOT(tree_edges, marked_nodes);   // OPTIONAL
    Node* node = nodes[next_node];
    if (node->component != -1) {
      return;
    }
    node->tree_number = tree_number;
    marked_nodes.insert(node->id);
    Pair* edge = get_next_edge(&marked_nodes);
    if (edge == NULL) {
      break;
    }
    next_node = edge->node2->id;
    tree_edges.insert(edge);
    SAFE_UPDATE_POINT();
  }
  print("Tree_number:", tree_number, "size:",
       marked_nodes.size(), "nodes:", marked_nodes);
}
}

UPDATE(transaction t) {
  t.tree_edges.add(t.merged_tree_edges);
  t.marked_nodes.add(t.merged_marked_nodes);
  for (node in t.merged_marked_nodes) {
    node.tree_number = t.tree_number;
  }
}

MERGE(transaction t1, transaction t2) {
  t1.merged_tree_edges.add(t2.tree_edges);
  t1.merged_marked_nodes.add(t2.marked_nodes);
}
```

Figure 7. Pseudocode for Prim's with merge

The benchmark is implemented using Prim's algorithm. The benchmark does not use the fastest MST algorithm available but uses a simple algorithm that a novice programmer might use. Figure 7 gives the pseudocode including the complete merge construct. The *get_next_edge* function does a simple linear scan over the *marked_nodes* set and looks for an adjacent node which has not yet been marked. We also demonstrate the performance impacts of the SNAPSHOT feature for this benchmark. The natural expression of the code as shown in Figure 7 does not need SNAPSHOT to behave correctly but we use it to study its effects on performance.

The datasets used for the Minimum Spanning Tree benchmark are (methodology was described at the beginning of Section 5).

- DS1 (6000 nodes): X = 6000, T = 6, N = 6.
- DS2 (9000 nodes): X = 9000, T = 5, N = 6.
- DS3 (12000 nodes): X = 12000, T = 4, N = 8.
- DS4 (16000 nodes): X = 16000, T = 3, N = 8.

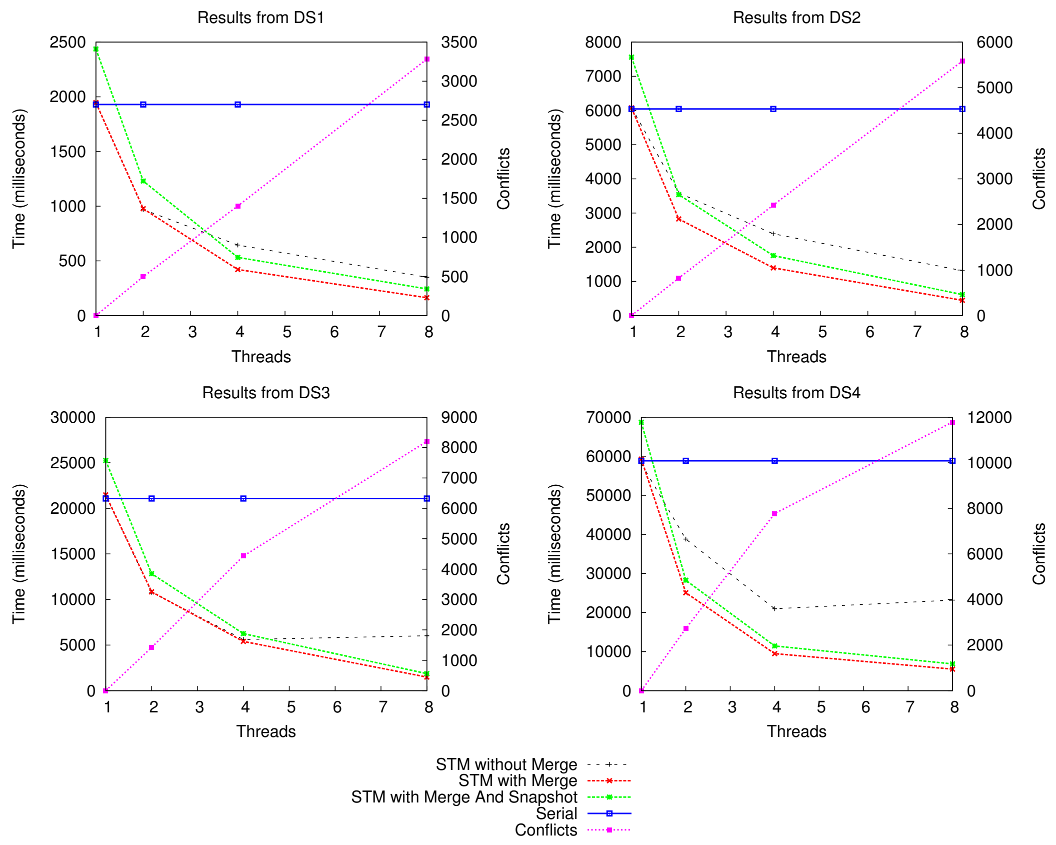


Figure 8. Results of the Minimum Spanning Tree benchmark

DS1 is a smaller dataset with lower contention. DS2 and DS3 are datasets with larger sizes and increasing contention. DS4 is the largest dataset with the most contention. The results we obtained are reported in Figure 8.

Observations and Discussion Parallelizing the application using the simple STM parallelization technique (Section 1.2) provides good speedups. In all the datasets: DS1, DS2, DS3 and DS4 all scenarios with more than 1 thread run faster than the serial version. In the STM implementation (without merge), as we increase the number of threads, execution time decreases as expected. However, in the case of DS3 and DS4 with 8 threads there is a slow down when compared to 4 threads. This is due to higher conflict overheads with an increased number of threads. In fact, this starkly depicts one of the main criticisms of STMs, that they do not scale very well with increasing contention. The merge construct is able to alleviate this issue completely and demonstrates sustained scalability with an increasing number of threads.

We report the results of two implementations of the MST benchmark using merge: one without SNAPSHOT-ing and one with SNAPSHOT-ing. Recall that for our benchmark the SNAPSHOT is an optional call. As expected the version without SNAPSHOTs performs better. This is simply due to lower overhead of not having to constantly create safe snapshots. At lower thread counts, the version without SNAPSHOT consistently performs better than the simple STM parallelization. And very importantly it also scales excellently as the number of threads increase. This can be attributed to the fact that the conflict overhead is reduced significantly as there is minimal wasted work when a transaction is aborted, as much of it can be re-used. The version with SNAPSHOT-ing performs slightly worse than the version without SNAPSHOT-ing but at higher thread counts even this version performs significantly better than the simple STM parallelization. The speedups are significant. For DS1 at 8 threads, the STM parallelization with merge is more than 50% faster than the simple STM parallelization and more than 90% faster than the serial version. The parallel efficiency P_e is surprisingly 1.48 (we will explain the super linear speedup in the next paragraph). For DS4 at 8 threads, the STM parallelization with merge is more than 75% faster than the simple STM parallelization and more than 90% faster than the serial version. The parallel efficiency P_e is 1.33. Again, a super linear speedup when compared to the linear version. DS2 and DS3 also exhibit very similar characteristics.

The super linear speedups are due to the fact that the get_next_edge function does a simple scan over the $marked\_nodes$ set, looking for other unmarked nodes. This simple lookup results in a $O(n^2)$ scan over the $marked\_nodes$ set. On parallelization, the $marked\_nodes$ set actually gets divided between the many competing threads. Therefore the algorithm runs much more quickly due to smaller set sizes. This results in the super linear speedups that we observe. The super linear speedup does not appear in the standard STM parallelization (without merge) as this advantage of splitting the $marked\_nodes$ set between threads can never be materialized as there is no functionality to merge them.

5.2 Connected Components

As discussed in Section 2, the speculative parallelization approach provides performance improvements for this benchmark while the merge construct does not. This is simply because the merge construct is as expensive as the DFS search itself. However, if in addition to a simple DFS, if the application needed to perform some arbitrary processing at each node, then the merge construct would indeed be beneficial.

We still present our results with the Connected Components benchmark to demonstrate the properties of the merge construct, especially with varying transaction durations. To better understand how the merge construct performs with transactions of varying duration (perhaps due to extra processing which needs to be performed at each node of the graph) we simulate transactions of varying duration in the benchmark and report these results in Figure 9(a) and Figure 9(b).

The pseudocode for the Connected Components benchmark is given in Figure 6.

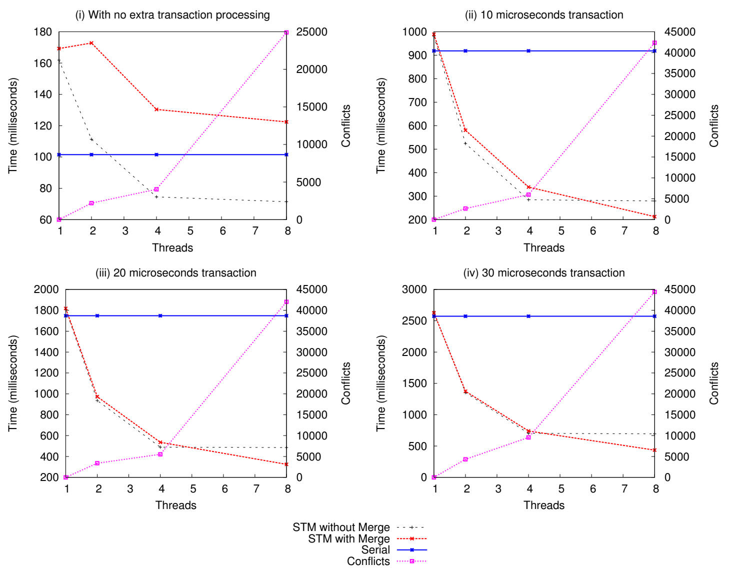
We used two datasets DS1 and DS2 to generate the graphs. The exact parameters used to generate DS1 and DS2 are:

- DS1 (80000 nodes): $X = 80000$, $T = 4$, $N = 8$.
- DS2 (500000 nodes): $X = 500000$, $T = 2$, $N = 6$.

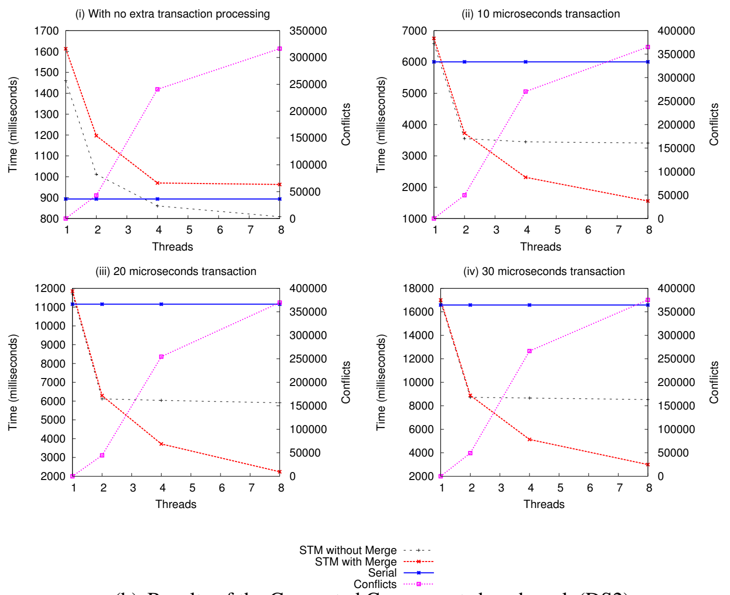
Observations and Discussion Figure 9(a) and 9(b) follow the same pattern. Each shows the execution time of the connected components application with an increasing amount of transaction processing time inserted into it. DS1 is a smaller dataset containing 80000 nodes (with lower number of conflicts) while DS2 is bigger at 500000 nodes (with higher amount of conflicts).

In Figure 9(a)(i) the parallel STM implementation without merge scales as expected and it runs quicker than the serial implementation when there are more than 2 threads. This demonstrates that this method of parallelization is effective, yet simple to achieve. As discussed, the merge construct does not provide any performance benefits in this case simply because the merge is as expensive the original DFS. However, if we were to consider any application which also performed some processing at each node, performance gains are obtained. Figures 9(a)(ii), 9(a)(iii) and 9(a)(iv) show the performance gains when we introduced processing at each node (in the form of a busy loop) for 10, 20 and 30 microseconds. Even with such small transaction durations, we see sizable speedups. At 8 threads we get performance gains of 23%, 33% and 37%. The performance gains increase as we increase the duration of the transaction. The same pattern is observed in Figure 9(b).

We also observe higher performance benefits of using merge when there are more conflicts. This is expected as this causes a higher number of aborted transactions and merge removes much of the penalty of a transaction aborting. Comparing Figure 9(a) and 9(b) we see that with the increased contention of DS2 we get much better performance gains using merge.



(a) Results of the Connected Components benchmark (DS1)



(b) Results of the Connected Components benchmark (DS2)

Figure 9. Results of the Connected Components benchmark

6. Related work

TM systems have been widely recognized as an effective and simple method of parallelization. The different types of TM systems: STMs [8, 13–15], HTMs (Hardware TMs) [7, 26, 31] and HyTMs (Hybrid TMs) [11, 27, 28] provide varying degrees of flexibility, programmability, scalability and hardware support.

Overhead reduction in STMs is an active area of research. [1] tries to reduce overheads through static analysis. [19] deals with conflicts at an abstract data type (ADT) level. [10] tries to decrease the number of conflicts by predicting data access patterns. We on the other hand, try to decrease the severity of conflicts. Many other techniques have been developed (see [25] for a recent list). Our work is orthogonal to the other work and they can benefit from our approach as well.

Much work has also gone into creating new parallel algorithms for problems such as the minimum spanning tree [4, 22], the connected components problem [16, 20] and DFS [17]. However our approach maintains its simplicity and generality making it available to any programmer irrespective of background.

[6] proposes a mechanism to run ”twilight” code at end of a transaction before it commits/aborts. While this mechanism allows the programmer to correct errors before committing, it does not prevent the execution of redundant work and re-use of work between threads which is the main source of speedups in our framework.

[23] discusses a parallel speculative algorithm for Minimum Spanning Tree (MST); the main focus of the paper is on developing a parallel algorithm for MST that uses data merging to improve execution. Although the paper attempts to leverage the idea of merge, it does not concern itself with how to extend transactional memory to support the merge construct as a generalized abstraction for supporting and promoting aggressive speculation. More-

over, their implementation of the MST falls short on performance, even though it's scalable, they are unable to demonstrate speedups over a serial implementation. The main contribution of our paper in contrast is to propose APIs which programmers can use to leverage partially completed transactions and merge the results so that partially completed work is not wasted. We also study the properties of the merge construct and present a detailed empirical evaluation. One of the biggest impediments to the use of transactional memory in highly speculative computation is the high overhead of rollbacks and restarts. Such high overheads dissuade algorithm designers from speculating aggressively. This paper proposes the critical merge construct to avoid such costly rollbacks and restarts thus promoting the design of highly speculative algorithms.

Conclusion and future work

We motivated the need for the merge construct and used the connected components problem to explain how it can benefit graph applications that are speculatively parallelized. We address the biggest weaknesses of these types of applications: discarded work and poor scalability. We took an in-depth look at the performance implications using the connected components benchmark and demonstrate very significant speedups in the minimum spanning tree benchmark with sustained scalability. We believe the merge construct provides a simple yet effective mechanism to improve the performance of some parallel applications which use STMs.

We plan to explore more applications of our construct and extensions of the API. We are also looking at ways to minimize programmer effort by removing the necessity of one of the user defined functions. In particular, if we could allow both transactions to merge only at SAFE_UPDATE_POINTS we can remove the requirement for one of the functions.

Acknowledgment

The authors would like to thank the anonymous reviewers for their comments. The authors also gratefully acknowledge the support of NSF grants CCF-1018544 and CCF-0916962.

References

[1] Yehuda Afek, Guy Korland, and Arie Zilberstein. Lowering stm overhead with static analysis. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pages 31–45, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] A. Aggarwal and R. Anderson. A random nc algorithm for depth first search. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 325–334, New York, NY, USA, 1987. ACM.

[3] A. Aggarwal, R. J. Anderson, and M.-Y. Kao. Parallel depth-first search in general directed graphs. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, STOC '89, pages 297–308, New York, NY, USA, 1989. ACM.

[4] D.A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 39, april 2004.

[5] Alessandro Bevilacqua, Alessandro Lanza, Giorgio Baccarani, and Riccardo Rovatti. A single-scan algorithm for connected components labelling in a traffic monitoring application. In *Proceedings of the 13th Scandinavian conference on Image analysis*, SCIA'03, pages 677–684, Berlin, Heidelberg, 2003. Springer-Verlag.

[6] Annette Bieniusa, Arie Middelkoop, and Peter Thiemann. Brief announcement: actions in the twilight - concurrent irrevocable transactions and inconsistency repair. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 71–72, New York, NY, USA, 2010. ACM.

[7] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 127–138, Washington, DC, USA, 2008. IEEE Computer Society.

[8] Evgueni Brevnov, Yuri Dolgov, Boris Kuznetsov, Dmitry Yershov, Vyacheslav Shakin, Dong-Yuan Chen, Vijay Menon, and Suresh Srinivas. Practical experiences with java software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 287–288, New York, NY, USA, 2008. ACM.

[9] Juan F. Carrillo, Maciej Orkisz, and Marcela Hernández Hoyos. Extraction of 3d vascular tree skeletons based on the analysis of connected components evolution. In *Proceedings of the 11th international conference on Computer Analysis of Images and Patterns*, CAIP'05, pages 604–611, Berlin, Heidelberg, 2005. Springer-Verlag.

[10] Romain Cledat, Kaushik Ravichandran, and Santosh Pande. Leveraging data-structure semantics for efficient algorithmic parallelism. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 28:1–28:10, New York, NY, USA, 2011. ACM.

[11] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[12] S. M. Renuka Devi and Chakravarthy Bhagvati. Connected component in feature space to capture high level semantics in cbir. In *Proceedings of the Fourth Annual ACM Bangalore Conference*, COMPUTE '11, pages 5:1–5:6, New York, NY, USA, 2011. ACM.

[13] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. *Distributed Computing*, 4167:194–208, 2006.

[14] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, New York, NY, USA, 2008. ACM.

[15] Sérgio Miguel Fernandes and João Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 179–188, New York, NY, USA, 2011. ACM.

[16] Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. On identifying strongly connected components in parallel. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, pages 505–511, London, UK, UK, 2000. Springer-Verlag.

[17] Jon Freeman. Parallel algorithms for depth-first search. Technical Report (CIS) MS-CIS-91-71, University of Pennsylvania, University of Pennsylvania, Philadelphia, PA 19104-6389, October 1991.

[18] Justin E. Gottschlich and Daniel A. Connors. DracoSTM: A practical C++ approach to software transactional memroy. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD). In conjunction with OOPSLA.* Oct 2007.

[19] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[20] D.S. Hirschberg. A parallel graph algorithm for finding connected components. Technical Report TR7518, Rice University, ECE, October 1975.

[21] U. L. Jau and C. S. Teh. Real-time object-based video segmentation using colour segmentation and connected component labeling. In *Proceedings of the 1st International Visual Informatics Conference on Visual Informatics: Bridging Research and Practice*, IVIC '09, pages 110–121, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] Donald B. Johnson and Panagiotis Metaxas. A parallel algorithm for computing minimum spanning trees. In *Proceedings of the fourth*

annual ACM symposium on Parallel algorithms and architectures, SPAA '92, pages 363–372, New York, NY, USA, 1992. ACM.

[23] Seunghwa Kang and David A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 15–24, New York, NY, USA, 2009. ACM.

[24] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI '07*, pages 211–222, 2007.

[25] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.

[26] Marc Lupon, Grigorios Magklis, and Antonio Gonzalez. A dynamically adaptable hardware transactional memory. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 27–38, Washington, DC, USA, 2010. IEEE Computer Society.

[27] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 69–80, New York, NY, USA, 2007. ACM.

[28] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing hybrid transactional memory: the importance of nonspeculative operations. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 53–64, New York, NY, USA, 2011. ACM.

[29] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

[30] Jeffrey Vetter Srinivas Sridharan, Bradford L. Chamberlain, Peter Kogge, and Steve Deitz. A scalable implementation of language-based software transactional memory for distributed memory systems. Technical Report Series FTGTR-2011-02, Oak Ridge National Lab, Oak Ridge, TN: Future Technologies Group, Oak Ridge National Lab, May 2011.

[31] Rubén Titos-Gil, Anurag Negi, Manuel E. Acacio, José M. García, and Per Stenstrom. Zebra: a data-centric, hybrid-policy hardware transactional memory design. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 53–62, New York, NY, USA, 2011. ACM.