Master's Thesis

jSTM: JavaScript Software Transactional Memory System

Kyoungju Sim (심 경 주)

Department of Creative IT Engineering

Pohang University of Science and Technology

2015

jSTM: JavaScript Software Transactional Memory System

jSTM: JavaScript Software Transactional Memory System

by

Kyoungju Sim

Department of Creative IT Engineering

Pohang University of Science and Technology

A thesis/dissertation submitted to the faculty of the Pohang University of Science and Technology in partial fulfillment of the requirements for the degree of Master of Science in the Creative IT Engineering

Pohang, Korea

12. 17. 2014

Approved by

Hanjun Kim (Signature)

Academic Advisor

jSTM: JavaScript Software Transactional Memory System

Kyoungju Sim

The undersigned have examined this thesis and hereby certify that it is worthy of acceptance for a master's degree from POSTECH

12/10/2014

Committee Chair Hanjun Kim (Seal)

Member Hwangjun Song (Seal)

Member Jangwoo Kim (Seal)

MCITE 심경주, Kyoungju Sim

20130810 jSTM: JavaScript Software Transactional Memory System,
자바스크립트 소프트웨어 트랜잭셔널 메모리 시스템
Department of Creative IT Engineering, 2015,
**p, Advisor: Hanjun Kim, Text in English

Abstract

This thesis proposes a JavaScript software transactional memory (jSTM) system only using features of HTML5. As web applications become widely used because of high portability, web applications become more complicated. To increase the processing speed of these applications, HTML5 supports web workers for JavaScript parallelization. However, the web worker is not perfectly suitable for parallelization because web workers do access the same memory address. In this reason, several JavaScript parallelization systems introduce transactional memory systems, but these systems need to install additional components. In contrast, with the jSTM, programmers can parallelize web applications easier than lock-based systems without installing additional components. This thesis implemented the prototype of jSTM system, and analyzed the overhead to improve the system.

Contents

I. Introduction

As web applications become widely used, portability across various platforms also becomes important. Moreover, programmers can implement more complicated application using only HyperText Markup Language (HTML), JavaScript, and Cascading Style Sheets. High performance is one of the main factors for using applications, but applications cannot guarantee the high performance as these become complicated. As a result, performance improvement for web applications becomes important. One of the possible performance optimization methods is parallelization because parallelized applications can utilize multi-core CPUs more effectively. For this reason, HTML5 supports a web worker [1] for the JavaScript parallelization.

The web worker is a JavaScript thread that runs in the background with a main HTML page. Because the main page can execute several workers, applications can use web workers for performing works in parallel. However, the web worker supports a limited parallelization because each web worker uses its own contents and does not share these contents with other workers.

Threads for executing the parallelized application need to share the memory because several threads can access same memory address at the same time, so the web worker is not perfectly suitable for parallelization. For this reason, the parallelization system needs to introduce a shared memory system such as a lock-based or a transactional memory mechanism. However, the lock-based system is difficult to use and error-prone, so parallelization systems introduce the transactional memory system for ensuring correctness of parallelized applications more easily than lock-based systems.

In addition, several JavaScript parallelization systems such as TigerQuoll [2] or ParaScript [3] introduce the transactional memory system. However, these systems need additional components for parallelizing such as Mozilla SpiderMonkey [4] for TigerQuoll or TraceMonkey [5] engine for ParaScript [3].

In this reason, these systems have a limited portability.

Meanwhile, previous JavaScript parallelization systems introduce various parallelization methods such as DOALL. These systems can increase the performance of applications but parallelizes loops limitedly if dependencies exist between loops. To solve this problem, some JavaScript parallelization systems use speculative parallelization methods. For example, ParaScript [3] utilizes the Spec-DOALL method for parallelizing applications speculatively with the DOALL method. These speculative parallelization systems can parallelize more kinds of applications than non-speculative methods because speculative methods can remove some instructions speculatively if these instructions have dependencies and low probability of execution.

This thesis proposes a software transactional memory system for JavaScript (jSTM) only using features of HTML5. This system aims that programmers can parallelize applications easily with less limitation unlike the current web worker. For this, this thesis implemented the JavaScript parallelization API using the transactional memory. After defining the unit of works and read & written variables using the API, applications can be parallelized.

Contributions of this paper are as follows:

(1) Introducing the transactional memory, programmers can easily parallelize applications speculatively without using a lock-based system.

(2) Without installing additional components, parallelization is available in most browsers.

The remainder of this paper consists of as follows. Section 2 describes the background for the JavaScript speculative parallelization and the transactional memory. Section 3 explains about the design of the system and what features need to be implemented. Section 4 shows the evaluation for the processing speed of some benchmarks using parallelization comparing with the

processing speed before parallelization. Section 5 compares our system with other parallelization systems, and finally, Section 6 concludes the paper.

II. Background

2.1. Parallelization

Controlling the frequency of a CPU core has a limited impact on increasing the processing speed because of the *memory wall*, the *ILP wall*, and the *power wall.* For this reason, CPU manufactures choose a multi-core design for improving the performance. For utilizing the multi-core system effectively, using multiple cores simultaneously is necessary. However, programmers previously did not implemented applications with a consideration for parallel programming. As a result, several parallelization methods are suggested such as DOALL for parallelizing these applications.

With the DOALL method, threads execute iterations independently and simultaneously. Figure 1 shows an example for DOALL parallelization. Because an instruction B uses the value of *nodeIdx* that is from an instruction A, the instruction B has data dependency on the instruction A (Figure 1b). Data dependency exists in the iteration, but any dependency between iterations does not exist. In this case, DOALL can parallelize loops. As a result, each CPU core executes iterations simultaneously (Figure 1d). With DOALL, CPU cores do not need to communicate with each other because CPU cores execute iterations independently, so the overhead for parallelization is low. However, DOALL cannot parallelize loops if dependencies exist between iterations.

To mitigate this limitation, speculative parallelization methods such as Spec-DOALL [6] are suggested. Using analyzed data (e.g., as profiling information), these methods remove dependencies speculatively for increasing the chance to parallelize.

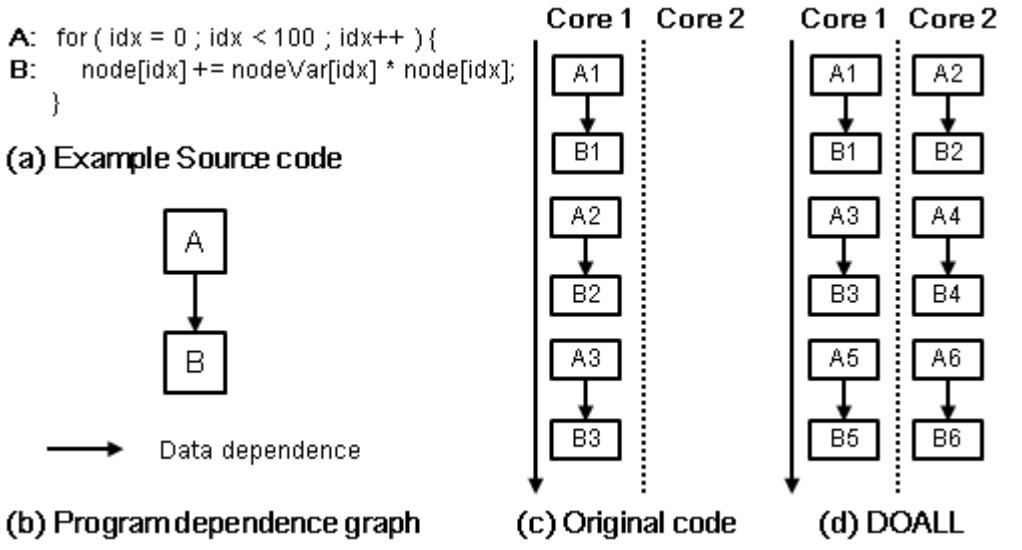


Figure 1. Example for DOALL. (a): Example source code, (b): Program dependence graph of (a), (c): Diagram of each instruction executed in CPUs before parallelization, and (d): Diagram of each instruction executed in CPUs after using DOALL method.

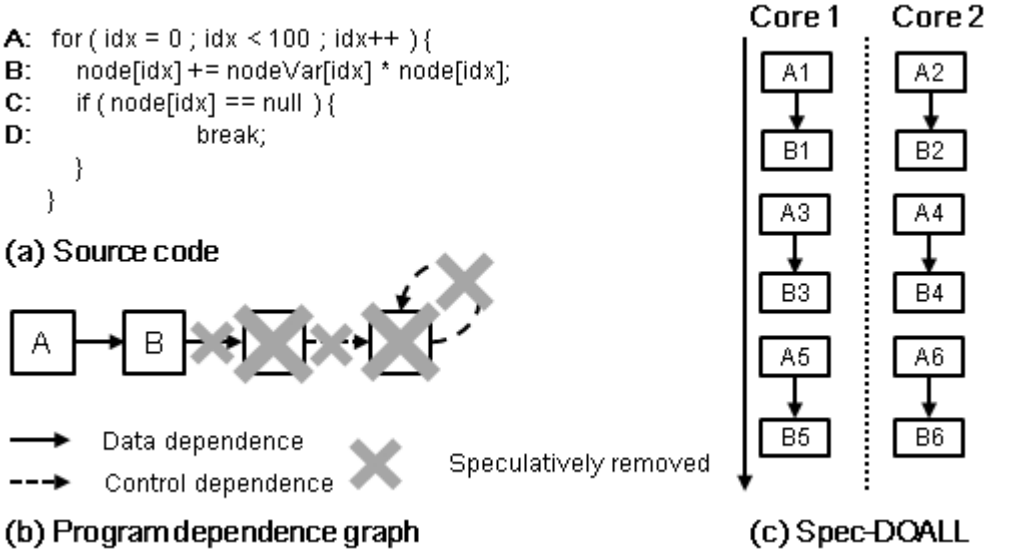


Figure 2. Example for Spec-DOALL. (a): Example source code, (b): Program dependence graph of (a), (c): Diagram of each instruction executed in CPUs after using Spec-DOALL.

Spec-DOALL removes dependencies between iterations if these dependencies have a little chance to occur. After removing dependencies, the Spec-DOALL method can parallelize loops using the DOALL method. Figure 2

shows an example for applying Spec-DOALL parallelization. Originally, DOALL cannot parallelize a loop in Figure 2a because of control dependency that exists on instruction D. However, Spec-DOALL can remove these dependencies like Figure 2b if instruction C and D have little chance of execution. After this removing process, Spec-DOALL parallelizes the loop like Figure 2c. If speculations are wrong in some iterations, the Spec-DOALL system recovers statuses before executing the misspeculated iteration and executes the original iteration. This recovering process causes the additional overhead.

2.2. Transactional memory system

Using parallelization, threads need to access the same memory address. To ensure the correct result of the parallelized program, synchronization between threads is necessary. For this reason, several synchronization methods are suggested such as using the lock-based synchronization. However, the lock-based synchronization is hard to use and error-prone method because programmers must use the lock with considering that errors such as deadlock do not occur. To solve this problem of the lock-based system, a transactional memory (TM) is introduced.

A transaction is a sequence of atomic instructions, so a thread executes all instructions from the transaction or does not execute them at all. After committing written values in a transaction, other transactions can use these changed values. On the other hand, when conflicts occur with other transactions, so that an instruction in the transaction accesses the wrong value of memory, other transactions must not use changed values by this transaction. In this case, the TM system discards all written values from the transaction that has conflicts. Using this mechanism, multiple threads can access the same memory address safely.

To manage TMs, the system selects between several mechanisms. The TM system needs to decide when the system checks conflicts between transactions and decide when the system applies changes of values from write operations. Section 2.2.1 and Section 2.2.2 describes mechanisms for each.

2.2.1. Eager conflict detection and Lazy conflict detection

The TM system detects conflicts for each memory access or at the end of each transaction. A mechanism for the former is the eager conflict detection and for the latter is the lazy conflict detection. With the eager conflict detection, the TM system checks conflicts for each memory access in a transaction because read and write operations from the transaction are visible to other transactions. On the other hand, with the lazy conflict detection, the TM system checks conflicts between a transaction and already committed transactions.

The eager conflict detection is not effective for the jSTM because the jSTM system utilizes the web worker and web workers do not share their own contents such as memory access records. In this reason, jSTM uses the lazy conflict detection.

2.2.2. Undo log and Redo log

To commit or to abort written values in transactions, the TM system uses an undo log or a redo log. Using the undo log, write operations in a transaction directly changes the value that stored in the memory. In addition, the TM system makes an undo log for recovering changed values when a conflict occurs.

On the other hands, write operations in a transaction do not directly changes the value when the system uses the redo log. Instead, the system records values and memory addresses that write operations aim to change.

Each transaction records access for read operations in a read set and for write operations in a write set. When comparing access records from the read and the write set, so that any conflict does not occur in the transaction, the system applies value changes using the redo log during the commit process. The jSTM system uses the redo log with the lazy conflict detection.

2.3. JavaScript Parallelization

To implementing the TM system, the commit unit and the recovering process for solving conflicts are necessary. Previously, several papers such as [7, 8, 9] implement the software transactional memory system (STM) using C or C++ language. However, C language and JavaScript language have difference structures, so the jSTM system needs to consider that. Table 1 shows the main difference between C and JavaScript language for implementing the TM system. First, JavaScript is the event-based language, so JavaScript is difficult to use synchronization basically because the event-listener operates asynchronously after the event occurs. Moreover, each worker does not share the same memory area. In this reason, the JavaScript-based system is difficult to introduce the lock-based STM like [7, 10] without modifying the JavaScript engine. Moreover, unlike that C language can access variables in the low-level with the memory address, JavaScript language can only access variables as the *object*. In this reason, the copying process for recovering uses the deep-copy mechanism in JavaScript. That is, the copying process copies all values of attributes for all objects. Section 3 describes details about how the jSTM solves these problems.

Table 1. The main difference between C and JavaScript for Spec-DSWP

	C	**JavaScript**
Synchronization	Available	Not available
The way to access memory	Using the *pointer*	Using the *object*

III. Design and Implementation

This thesis implemented the STM system and the speculative parallelization API using JavaScript language. This system aims to provide parallelization easily without any additional installing modules, so the jSTM system only uses features of HTML5. Meanwhile, as describing in Section 2.3, this system needs to apply the difference between C and JavaScript to introduce the STM system in JavaScript. Section 3.1 and 3.2 describes how this thesis implements the system considering these differences: Section 3.1 describes how this thesis implements the commit process without the lock, and Section 3.2 explains how this thesis implements the recovering process using the deep-copy mechanism. Section 3.3 describes the architecture of the parallelization system using the transactional memory and the mechanism for parallelizing with the API.

3.1. Committing process

The jSTM system introduces the separated validation and committing process. Because each worker thread (i.e., threads that execute parallelized loop) uses separated memory area, a worker thread cannot validate its records of memory accesses nor cannot commit for other worker threads to use changed values. For this reason, the jSTM system separates the committing process from worker threads. The committer gets speculative records from worker threads, and validates and commits records. As a result, all worker threads share memory states of the committer.

In addition, the system uses an in-order commit to guarantee the correctness of the execution result. Figure 3 shows how transactions in worker threads and the committer work using the in-order commit. Each CPU core has an identifier (ID) and each transaction also has an ID, so CPU cores only execute transactions that have the same ID as theirs. For example,

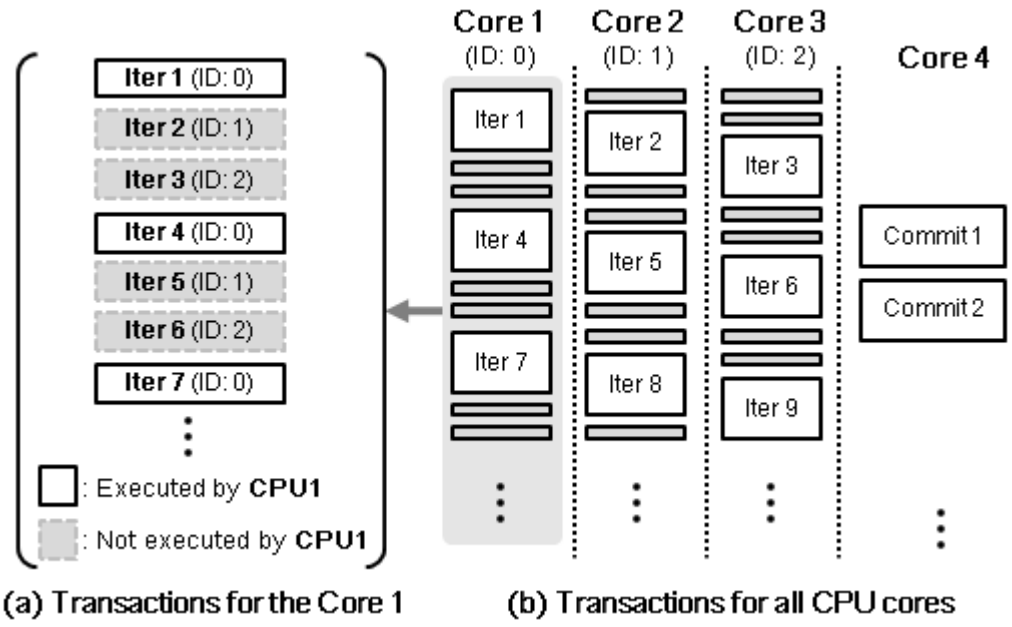


Figure 3 Example for the in-order commit process. (a): transactions that executed or not by CPU core 1, (b): figure for how workers and committer works; gray one is empty transaction. Each transaction from core 1-3 communicates with the core 4 for the committing process.

in Figure 3a, Core 1 that has an ID 0 only executes the transactions that have an ID 0. Otherwise, Core 1 does not execute instructions of other transactions, so Core 1 executes these transactions as empty transactions. After worker threads send records of memory access to the committer, the committer store these records. If the committer gets at least one memory-access record from each CPU cores and the committer did not validate and commit these records yet, the committer starts to validate these records. As the result, though all CPU cores execute each transaction independently, the committer validates and commits transactions as same order as the original sequential program.

3.2. Backup process using deep copy for global variables

The jSTM system uses the deep-copy mechanism for making checkpoints of global variables. JavaScript is the object-oriented language, so JavaScript accesses variables as the unit of object. Because JavaScript does not use a

pointer like C language, the jSTM system cannot access the memory in the low-level. For this reason, the jSTM system copies the name and all property of objects using the *Object.keys()* method.

The *Obejct.keys()* method returns properties of objects as an array of string values, so worker threads make a checkpoint of global variables utilizing this method for the recovering process. Figure 4 shows an algorithm for making checkpoints of global variables automatically. First, using the *Object.keys()* method, a worker thread gets all names of global variables from its memory area. In this case, the worker thread removes properties not defined by a programmer to reduce the overhead for copying (Figure 4a: line 2-3). Then, the *makeCheckPoints* function checks the type of each global variable. If a global variable is an object type, this function makes copies of all properties of objects using the *copyContents* function (Figure 4a: line 5-6). Otherwise, this function makes copies of value of the global variable (Figure 4a: line 7-8).

The copyContents function performs differently for array type objects and other objects. For arrays, the copyContents copies all contents of array (Figure 4b: line 3 7). Otherwise, the copyContents copies both a prototype and properties of objects (Figure 4b: line 8-12). To check and copy all properties of objects, the copyContents check recursively if type of each property is object.

When misspeculation occurs, the worker thread recovers all global objects defined by the programmer using copied values of objects. However, this method cannot make a copy of local variables because local variables are not accessible programmatically with this method. Instead, the programmer can make the copy of local variables using the parallelization API.

```
    // objectName: list of global variables in a worker thread.
    // globalObject: each variable name from objectName
    // valueOfGlobalObject: value of 'globalObject'
    // checkPoints: array for saving copies of global variables
1   void makeCheckPoints(){
2       objectName = Object.keys(this);
        // delete global variables not defined by users from the list
3       spliceObjectsNotDefinedByUsers(objectName);
4       for ( globalObject in objectName) {
            // copy the value of global object to the 'checkPoints'
5           if ( typeof globalObject == 'object' )
6               checkpoints[property] = copyContents(valueOfGlobalObject);
7           else
8               checkpoints[property] = valueOfGlobalObject
9       }
10  }
```

(a) Function for making checkpoints

```
    // returns copiedObject(the copy of gObject)
1   copiedObject copyContents( gObject ) {
2       if ( typeof gObject == 'object' ){
3           if ( typeof gObject == 'array' ) {
                // copy all contents of an array
4               for ( i = 0; i <value.length; i++ )
5                   copiedObject[i] = copyContents(i);
6               return copiedObject;
7           }
8           else {
                // copy all properties of an object
9               for ( property in gObejct )
10                  copiedObject[property] = copyContents(property);
11              return copiedObject;
12          }
13      }
14  }
```

(b) Deep-copy function for copying objects

Figure 4 Algorithms for making checkpoints. Function (a) uses the function (b) for copying all properties of objects.

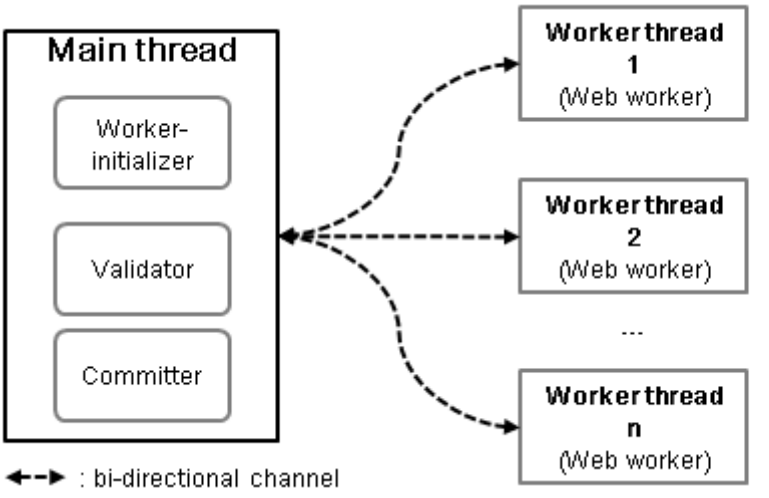


Figure 5. Architecture of the parallelization system

3.3. Implementation

3.3.1. The architecture of the parallelization system

Figure 5 shows the architecture of the parallelization system using jSTM. This system has two kinds of threads: the main thread and worker threads. The main thread executes the main HTML page and worker threads are web workers for parallelized works. The main thread and each worker thread are connected bi-directional, so the main thread sends variables for initializing at the beginning and the copy of variables after misspeculation occurs to worker threads. In addition, the main thread receives records of speculatively access for validating and copies of variables for preparing the recovering process from worker threads.

The main thread consists of the worker initializer, the validator, and the committer. The worker initializer initializes worker threads, and the validator and the committer checks conflicts and commits speculative writes. Actually, the main thread executes the validator and the committer together if worker threads send values. Figure 6 shows an algorithm of validating and committing. First, the validator checks whether or not conflicts exist in each iteration:

```
1  void validateNcommit(){
2       specValueList = getTheSpecRecord();  // get records of speculative records
3       for ( value in specValueList ) {
4            if ( value is from write set) {
                  // save the value from write set temporarily before commit
5                 putToTempList( value );
6            }
7            else if ( value is from read set ) { // check the transaction read correctly
8                 if ( value == undefined ) { // get the committed value
9                      committedValue = getTheCommittedValue( nameOfValue );
10                     if ( value != committedValue ) {
11                          status = misspec;
12                          break;
13                     }
14                }
                  // comparing the read value and uncommitted value
15                else if ( value != getFromTempList( nameOfvalue ) ) {
16                     status = misspec;
17                     break;
18                }
19           }
20      }
        // commit speculative written values
21      if ( status != misspec )  commitValues( tempList );
        // discard speculative value and recover memory states
22      else {
23           discardAllWrites( tempList );
24           terminateAllWorkerThreads();
25           executeMTXSequentially();
26           restartNextStage();
27      }
28 }
```

Figure 6 Algorithm for the validator and the committer

If an iteration read speculatively a wrong value, the validator considers that conflict exists in this transaction. The comparison target can be a previous-committed value (Figure 6: line 7-14), or an uncommitted value from previous iterations (Figure 6: line 15-18). After validating, the committer applies changed values to the memory if conflict does not exist (Figure 6: line 21-23). If conflict exists in the iteration, the main thread discards all record of

speculative writes and terminates all of currently running worker threads. Then, the main thread executes the misspeculated iteration without speculation and reinitializes worker threads for restarting later iterations (Figure 6: line 23-26). These workers will execute the recovery process before executing later iterations.

3.3.2. API for parallelization

This thesis implemented the Parallelization API to the JavaScript library. Programmers can use this parallelization API by adding the implemented JavaScript library without any changes of the web browser: Programmers use the API for initialization and execution threads in the main thread and use for defining and using MTXs in worker threads. Table 2 shows components of the API that users can use for parallelization.

The overall execution flow of the main thread and worker threads is described as follows: When the main thread executes *executeTx* after executing *createChannels* for initializing, *executeTx* function triggers for the worker threads to start executing. After receiving data from the main thread, worker threads start to execute their transaction. First, worker threads call *txBeginInvocation* at the starting point of a loop. *txBeginInvocation* recovers the memory state if conflicts occurred from previous iterations. Then, worker threads call *txBegin* at the beginning of iteration. This function makes a copy of global variables for the recovering process. On the other hand, *txCopy* makes a copy of local variables if the programmer wants to backup. During the transaction, *txRead* records read operations for making the read set and *txWrite* records write operations for the write set. When worker threads call *txEnd* at the end of iteration, threads send their read & write records for validating and the copy of variables to the main thread. To reduce the communication overhead, *txEnd* sends the copy selectively when the value of variable changed in the transaction. At the end of the loop, *txEndInvocation*

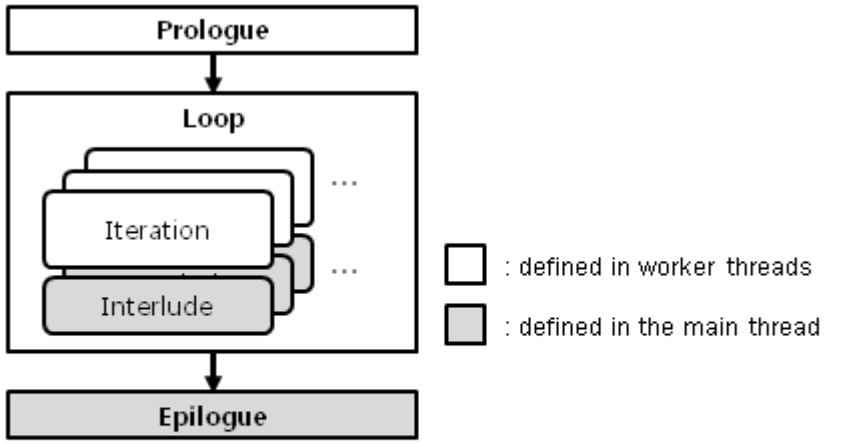
Table 2. API for speculative parallelization

For the main thread	
createChannels	This is for initializing worker threads. This function creates and initializes web workers.
executeTX	With given arguments, let worker threads start transactions.
For worker threads	
getThreadNum getMyThreadID	These functions are for executing iterations selectively. getThreadNum returns the number of worker threads, and getMyThreadID returns the ID of the worker thread.
txBeginInvocation	It indicates the starting point of a loop. If conflicts occurred in the worker thread previously, memory-state recovering function will be executed.
txBegin	It indicates the starting point of loop iteration. If conflicts did not occur previously, this function makes copy of global variables to the main thread for preparing the recovering process when conflict occurs later.
txWrite	This is for the write operation. This function saves names and values of written variables for the write set.
txRead	This is for the read operation. This function saves names and values of read variables for the read set.
txCopy	This function is for making copy of local variables. The recovering process will use copies after misspeculation occurs. If misspeculation occurred before, this function returns the copied value.
txEnd	It indicates the end point of loop iteration. This function sends copies of variables only if these are modified and send the records of speculatively access.
txEndInvocation	It indicates the end point of a loop

triggers for the main thread to stop validating. Figure 8 shows the example of speculative parallelization using these functions. This example parallelizes the source code of Figure 2a. Each transaction executes the iteration as if *node[idx]* has a specific value, so misspeculation occurs if some iteration reads empty *node[idx]*.

3.3.3. Execution model

Using the parallelization API, the parallelized loop consists of the prologue, the loop, and the epilogue. In addition, the loop consists of several iterations. Figure 7a shows the execution model of the parallelized loop. After the main thread triggers for the worker threads to start executing, each worker thread



(a) Execution model

```
createChannels(threadNum, 'testWorker.js', 'beforeGetInfo',
                                    'afterGetInfo', 'afterPII', 'seq_getInfo');
```

(b) Prologue, epilogue, and interlude function in the initializing function

Figure 7 (a): Execution model of parallelized loop for each worker thread, (b): example source code for initializing the worker thread from Figure 8.

executes the prologue function at first. After that, worker threads start to execute the loop. After executing iterations in the loop, the main thread executes the interlude function after committing if misspeculation does not exist. Finally, worker threads terminate after executing all iterations of the loop, the main thread executes the epilogue function.

The main thread has codes of interludes and the epilogue, and worker threads have codes of the prologue and the loop. When the programmer uses *createChannels* for initializing the worker thread, the programmer must specify functions for the prologue, interludes, and the epilogue like Figure 7b that is the part from the source code of Figure 8. For this example, the prologue function is *beforeGetInfo*, the interludes function is *afterGetInfo*, and the epilogue function is *afterPII* function.

```
...
// Initialize worker threads:
// - 1st argument: the number of worker threads,
// - 2nd: the name of JavaScript file,
// - 3rd: the name of the prologue function,
// - 4th: the name of the function executed after iterations
// - 5th: the name of the function executed after executing the loop,
// - 6th: the name of the non-speculative function executed if misspeculation occurred.
createChannels(threadNum, 'testWorker.js', 'beforeGetInfo',
                        'afterGetInfo', 'afterPII', 'seq_getInfo');
// execute worker threads with arguments
executeTX({node: node, nodeVar: nodeVar});
...
```

(a) Part of HTML source code for the main thread

```
function beforeGetInfo(arg)
{
    var node = arg.node;
    var nodeVar = arg.nodeVar;
    getInfo(node, nodeVar);
} // the prologue function: each worker thread executes this function firstly.
function getInfo(node, nodeVar)
{
    // get the number of threads and ID of a worker thread.
    var threadNum = getThreadNum();
    var threadID = getMyThreadID();
    txBeginInvocation(); // the starting point of the loop

    var endPt = txCopy("endPt", 100); // back local variables for preparing misspeculations
    for (var idx = txCopy("idx", 0); idx < endPt; idx++) {
        txBegin(); // the starting point of the iteration
        if( idx % size != threadID ) {  // for executing transactions selectively
            txEnd();  // : commit empty transaction
            continue;
        }

        txRead("node["+idx+"]", node[idx]); // make the record for the read set
        node[idx] = nodeVar[idx] * node[idx];
        txWrite("node["+idx+"]", node[idx]); // make the record for the write set
        txEnd(); // the end point of the iteration
    }
    txEndInvocation(); // the end point of the loop
}
```

(b) JavaScript source code for worker threads('testWorker.js')

Figure 8. Example of parallelization using the parallelization API. This example parallelizes the source code of Fig. 2a.

Table 3. Ratio of execution time for time-consuming function and ideal speedup for parallelized benchmarks. Ideal speedup is calculated using Amdahl's law.

benchmark	Description[1]	Ratio [%]	Ideal speedup	
			4 threads	8 threads
2mm	Multiplication two arrays	99.84	3.98	7.91
3mm	Multiplication three arrays	99.84	3.98	7.91
covariance	Computing covariance	99.91	3.99	7.95
doitgen	Multi-resolution analysis kernel	99.61	3.95	7.79
dynprog	Dynamic programming	99.99	4.00	8.00
gemm	Matrix multiplication	99.91	3.99	7.95

IV. Evaluation

This thesis implemented the prototype of the jSTM system. To evaluate the performance of the jSTM system, evaluated and compared execution times of between the original and the parallelized source code of benchmarks. Moreover, to improve the jSTM system, this thesis also analyzed the overhead of the system for each benchmark.

For evaluation, this thesis utilized the polybench benchmark set [11]. Originally, the polybench benchmark set has total 30 benchmarks, but this thesis parallelized 6 benchmarks using DOALL method with the parallelization API[2]. This thesis used 16 GB RAM and 3.40GHz Intel® Core™ i7-4770 machine that has 8 cores. In addition, this thesis executed benchmarks using Google Chrome 39.0 version.

4.1. Evaluation results

This thesis parallelized 6 benchmarks in Table 3 and executed using 4 threads and 8 threads for evaluation. Figure 9 shows speedup of parallelized benchmarks using 4 threads (Figure 9a) and 8 threads (Figure 9b). Multiple threads could execute parallelized loop simultaneously using the

[1] Description is from [11].

[2] Because the original source codes are implemented using the C language, this thesis ported source codes to the JavaScript version.

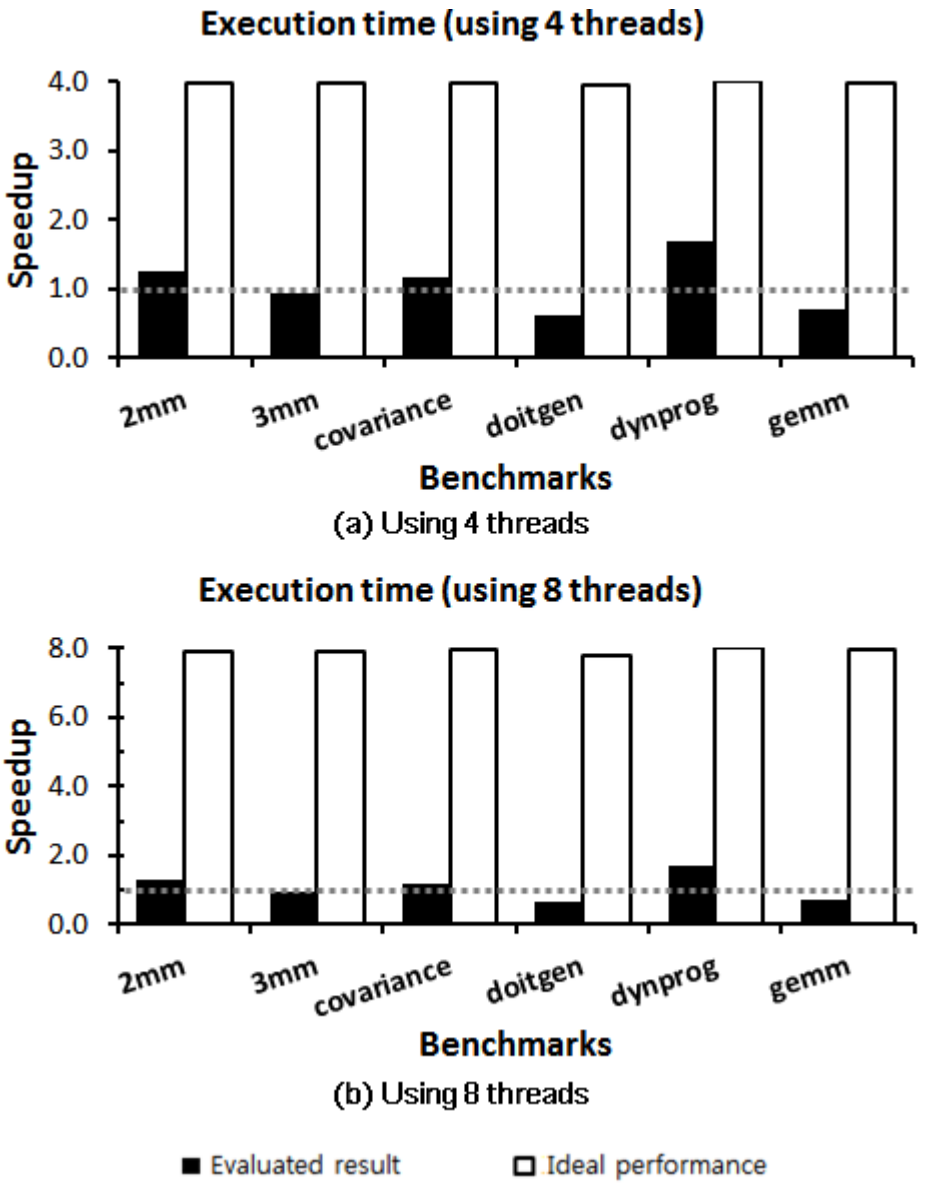


Figure 9. Speedup of parallelized benchmarks using 4 threads (a) and 8 threads (b)

parallelization API, but speedup of each benchmark was less than the ideal speedup because of the overhead of the prototype jSTM system: The average speedup using 4 threads was 1.07 and the average speedup with 8 threads was 1.17. Especially, the overhead sharply increased when using 8 threads.

4.2. Overhead analysis

Because the overhead of the prototype system was bigger than ideal execution time, so this thesis evaluated and analyzed the overhead to improve the system. When the programmer uses the DOALL method, main factors of the overhead are as follows:

- **Initialization overhead**: Initialization overhead occurs when the main thread initializes worker threads and sends arguments to them. This overhead contains the communication overhead.
- **Overhead of *txWrite***: *txWrite* saves all record of memory access for writing. The overhead becomes large if many writing operations exist in a transaction.
- **Overhead of *txEnd***: *txEnd* sends the record of write operations, so this function includes the communication overhead. If a size of record becomes larger, the overhead of this function also becomes larger.
- **Overhead of *txEndInvocation***: *txEndInvocation* notifies that a worker thread finished executing the parallelized loop to the main thread. This overhead also contains the communication overhead.
- **Overhead of the committing process**: The committer receives records of write operations from workers and applies them to the main memory. This overhead becomes larger when the size of records becomes larger.
- **Overhead of synchronization**: Because the main thread does not send arguments to all worker threads simultaneously, worker threads do not start to execute instructions at the same time. In this reason, the main thread needs to wait for all worker threads to finish executing instructions.

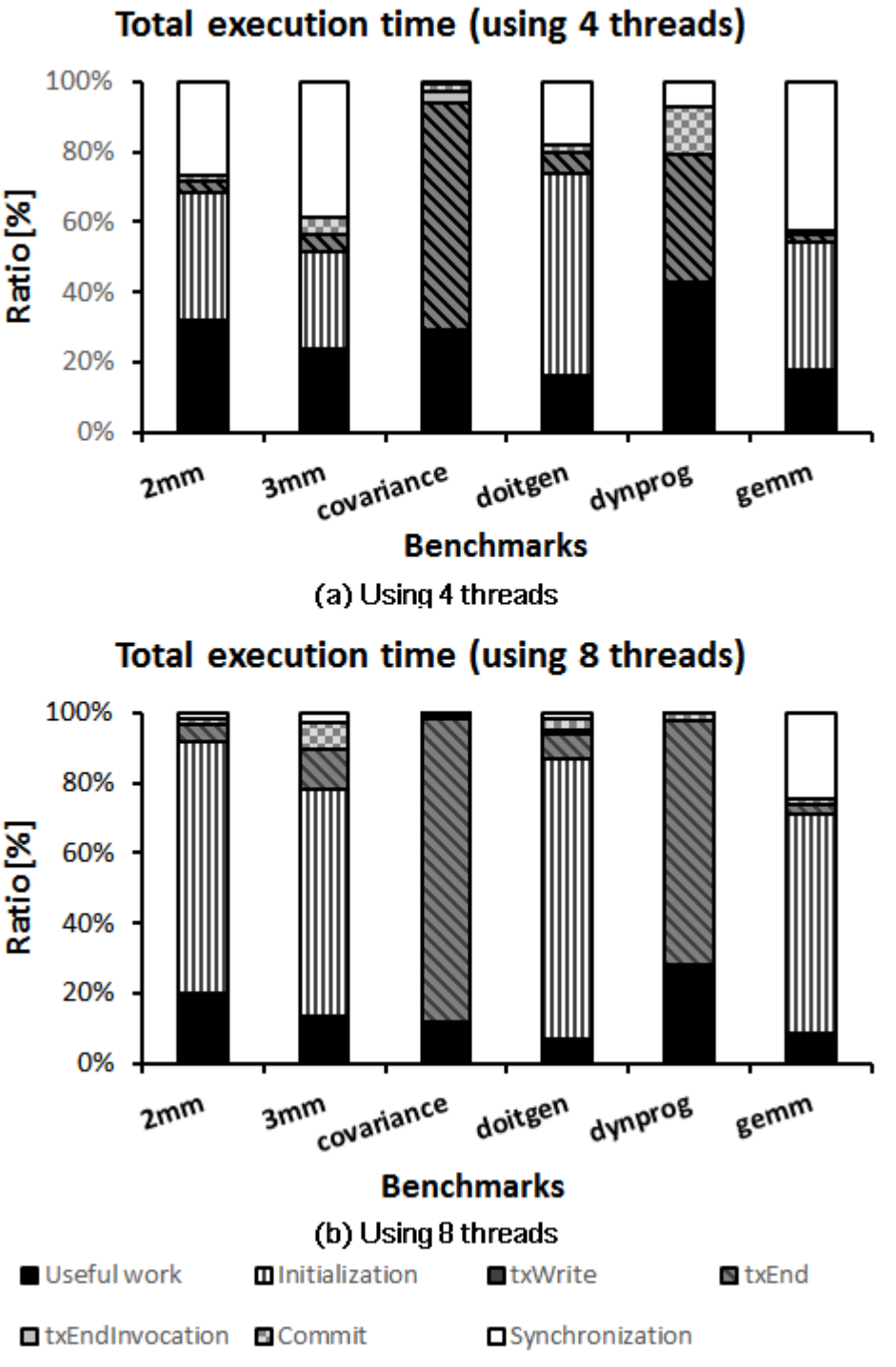


Figure 10. Ratio of overhead for each parallelized benchmark using 4 threads (a) and 8 threads (b).

Table 4. Size of sending data for each worker thread and ratio of communication overhead to initialization overhead.

Benchmarks	Data size [Byte]	Ratio [%]	
		4 threads	8 threads
2mm	32000048	95.93	99.52
3mm	32000064	98.52	99.04
covariance	64000040	98.35	99.99
doitgen	216002724	99.99	99.99
dynprog	160016	96.03	99.99
gemm	96000040	99.88	98.93

Figure 10 shows the ratio of each main factor for parallelized benchmarks. Especially, the overhead for initialization and the overhead of txEnd occupy high proportion, so this thesis analyzes and describes about these two overheads.

4.2.1. The initialization overhead

The initialization overhead that is from *createChannels* and *executeTX* function mainly affected the overall performance in the case of *2mm*, *3mm*, *doitgen*, and *gemm* benchmarks. Because these benchmarks used the large array objects for calculating as Table 4 shows, executeTX function needed to send the big data. In this case, the initialization overhead became bigger when the size of array objects became bigger. For example, the *executeTX* function at the line 13 in Figure 11 sent arguments and *A*, *C4* and *sum* were array-type variables. Especially, *A* and *sum* were three-dimensional arrays, so the overhead for communicating between the main thread and each worker thread increased as the size of arrays became bigger.

Moreover, because the jSTM system uses one main thread, the main thread sends arguments to only one worker thread at once. In this reason, the number of threads also strongly influenced the initialization overhead: the overhead when using 8 threads was larger than using 4 threads for the most case of benchmarks.

```
    // nr(=arg_r), nq(=arg_q), np(=arg_p): argument for deciding array size
    // array 'A[nr][nq][np]', 'C4[np][np]' are arrays used for calculation.
    // array 'sum[nr][nq][np]' is array used for saving result
    // init_array: function for initializing arrays
    // send nr, nq, np and arrays(A, C4, sum) to worker threads
1   function executor (arg_r, arg_q, arg_p, threadNum) {
2     nr = arg_r;
3     nq = arg_q;
4     np = arg_p;
5     init_array (nr, nq, np, A, C4);
6
7     for (var r = 0; r < nr; r++) {
8       sum[r] = [];
9       for ( var q = 0; q < nq; q++ ) sum[r][q] = [];
10    }
11    createChannels( threadNum, 'doitgen_pll_worker.js', 'kernel',
12                    'interlude', 'printExecTime' );
13    executeTX( {nr: nr, nq: nq, np: np, A: A, C4: C4, sum: sum} );
14  }
```

Figure 11. Part of source code for initializing and executing worker threads of *doitgen*

4.2.2. The overhead of txEnd

The overhead of *txEnd* function mainly affected in the case of *covariance* and *dynprog* benchmarks. Especially, as Table 5 shows, the overhead of *txEnd* function increased when the total size of the record of write operations became bigger.

In the case of *covariance*, the size of the record for write operations affected the overhead of *txEnd*. Figure 12 shows each iteration in parallelized loops from the *kernel_covariance* function and the *kernel_covariance1* function. In these functions, *txWrite* functions (Figure 12: line 17, 21 / line 42, 45) made the record of write operations. In this case, the size of records became bigger as the number of iterations in the *kernel_covariance* function and the *kernel_covaraince1* function increased. As the result, txEnd function must send the large size of records, so the communication overhead also increased.

Table 5. Data size per each transaction and the number of transactions for each worker threads. This thesis divided the time-consuming function of *3mm* and *covariance* into two parts and parallelized them. In the case of *covariance*, the data size and the number of transactions are different between two parts, so this thesis writes both data.

Benchmarks	Data size [Byte]	Number of transactions	
		4 threads	8 threads
2mm	8000	4000	8000
3mm	8000	12000	24000
covariance	16008	8000	16000
	16n (1≤ n ≤2000, n: iterator)	8000	16000
doitgen	2400	1200	2400
dynprog	8	400000	800000
gemm	16000	8000	16000

In addition, in the case of *dynprog*, because many transactions from several threads sent records of write operations but the main thread could execute the commit process for the record from one transaction at once. In this reason, the proportion of overhead of *txEnd* of *dynprog* was high though data size of iterations was relatively small comparing to other benchmarks.

As a result, communication overhead mainly affected the overall performance in the case of the initialization overhead and the overhead of *txEnd*. In this reason, a method for decreasing the communication overhead with other threads is necessary.

```
    // calculate the mean of each row of data array
    //        and subtracts this value to each value of data array
1   function kernel_covariance (m, n, float_n, data) {
...     ...
13          mean[j] = 0.0;
14          for ( var i = 0; i < n; i++ )
15              mean[j] += data[i][j];
16          mean[j] /= float_n;
17          txWrite( "mean["+j+"]", mean[j] );
18
19          for ( var i = 0; i < n; i++ ) {
20              data[i][j] -= mean[j];
21              txWrite( "data["+i+"]["+j+"]", data[i][j] );
22          }
...     ...
26  }
27  // calculate the covariance of the data array
28  function kernel_covariance1 (m, n, data, symmat) {
...     ...
39          for ( var j2 = j1; j2 < m; j2++ ) {
40              symmat[j1][j2] = 0.0;
41              for ( var i = 0; i < n; i++ )
42                  symmat[j1][j2] += data[i][j1] * data[i][j2];
43
44              txWrite( "symmat["+j1+"]["+j2+"]", symmat[j1][j2] );
45              symmat[j2][j1] = symmat[j1][j2];
46              txWrite( "symmat["+j2+"]["+j1+"]", symmat[j2][j1] );
47          }
...     ...
51  }
```

Figure 12. Part of source code for worker threads of covariance.

V. Related work

Because using the lock for parallelization is difficult and error-prone for programmers, previous researches implements TM systems using various mechanisms. Especially, this thesis introduces several STM systems: McRT-STM system and TL2 algorithm.

A Multi-core runtime software transactional memory system (McRT-STM) [7] is the lock-based STM system that operates on an experimental runtime system. The McRT-STM system implements transactions by using the strict two-phase locking protocol instead of the non-blocking protocol. With evaluating the performance of several alternatives of STM designs, this system selects to use the read-versioning mechanism with writer lock and the undo-log mechanism. In addition, this STM supports both the per-object and the per-cache-line conflict detection, but this paper shows the performance of benchmarks using the cache-line based conflict detection mechanism for focusing to the high performance. Actually, when using benchmarks, the McRT-STM system is faster than lock-based systems. However, the McRT-STM is not faster evidently than the lock-based system when using the real application.

A Transactional locking II (TL2) algorithm [10] is a STM algorithm which uses a global version clock. Write transactions increase a value of the global version clock for managing a version number, and transactions read this value for validating a read-set. The TL2 algorithm supports the write-lock per object for C or C++ and per stripe for Java, but an efficient memory management mechanism does not exist for C and C++ in [10]. In this reason, this paper implements the TL2 algorithm using the Java language. With adding the TL2 algorithm manually, performance of a parallelized benchmark is faster than using a single mutex lock. However, this paper only evaluates performance of the red-black tree benchmark, so other types of benchmarks can have

different aspects of performance with evaluated data.

To increase the speed of JavaScript applications, several researches developed JavaScript parallelization systems. For considering that multiple threads access the same memory address, these researches introduce various methods. For example, this thesis explains about DOHA that uses data-communication API and River Trail that supports immutable access.

DOHA [12] is the JavaScript parallelization system that utilizes the web worker. This system consists of event-loop and MultiProc that manages states and schedules events for load-balancing. Because the original web worker does not use share memory, DOHA uses a publish-subscribe based communication API and RPC events to share states. However, with this publish-subscribe API, workers use copied states and all of these copies need to be synchronized when one of workers updates copied states. As a result, the communication overhead is high because all of workers must update shared states when the public-subscribe layer sends the message for updating states.

River Trail [13] supports a programming model and a data-parallel API with a newly defined data type for JavaScript parallelization. This system cutilizes GPUs for parallelization, so parallelized applications gains high performance. For access the same memory address between multiple threads, River Trail introduces the immutable access. That is, child threads cannot change global states of a parent thread and the parent thread can change local states after all child threads finish their works. Meanwhile, the River Trail system uses the modified SpiderMonkey [4] that is a JavaScript engine in FireFox. This system also uses a compiler for porting JavaScript language to OpenCL [14], so GPUs execute the parallelized application using the OpenCL binding for FireFox. In [13], the River Trail system is implemented only in FireFox, so the evaluation process also utilizes FireFox. That is, this system can operates in

only limited environment.

In addition, some JavaScript parallelization system utilizes the transactional memory system to support memory access for multiple threads. For example, TigerQuoll and ParaScript utilize the STM system.

TigerQuoll [2] consists of the event-based API and the runtime system for JavaScript parallelization. This system provides the mutable shared memory space, so workers can communicate with each other. TigerQuoll adopts the transactional memory system for parallelization system. Similar to the TL2 STM, TigerQuoll uses the lock-based transactional memory that uses the global versioning clock, but differently from the TL2 STM, TigerQuoll provides the write-lock per field. Using the version number, TigerQuoll validates the read set and the write set.

ParaScript [3] supports automatic speculative DOALL. For supporting speculative parallelization automatically, ParaScript firstly selects hot loop, and then, the parallel-code generator generates parallelized bytecodes. ParaScript utilizes the STM system for parallelization, but not fully utilize the STM system for reducing the overhead. Instead, ParaScript checks conflicts of object arrays using the reference counting mechanism and the range-based checking mechanism for each memory access. If the system detects a conflict, the recovering process recovers a stack pointer and a frame pointer to start execution at checkpointed location.

Unlike DOHA and River Trail, TigerQuoll and ParaScript utilizes the transactional memory system, so this system can access the same memory address more effectively than DOHA or River Trail. However, both TigerQuoll and ParaScript need additional modified JavaScript engine for using the TM system. To increase portability, jSTM implements the STM system only using HTML5 features such as the web worker.

VI. Conclusion

This thesis designed the software transactional memory (STM) system and the parallelization API to make programmers parallelize applications easier. Especially, this thesis aims to increase portability, so the system only utilized features of HTML5. This thesis implemented the prototype of STM system and the parallelization API as JavaScript libraries, so programmers can parallelize applications with these libraries in most browsers. However, because of the communication overhead, the prototype of the system had the large overhead. In this reason, this system needs the method for reducing communication overheads additionally as future work.

요 약 문

다양한 환경에서 실행 가능하다는 장점 때문에, 웹 어플리케이션의 사용량은 점차 늘고 있으며, 그와 동시에 크고 복잡한 규모의 웹 어플리케이션의 수도 늘어나고 있다. 그런데, 프로그램이 복잡해지면 복잡해질수록 구동 속도는 점점 느려지기 때문에 이들을 빠른 속도로 구동하기 위해서 여러 가지 방법을 사용할 수 있으며, 병렬화도 그 방법들 중 하나이다.

병렬화를 사용하면 여러 개의 CPU 코어를 동시에 활용할 수 있어 프로그램의 구동 속도를 높일 수 있다는 장점이 있다. 그렇기 때문에 HTML5 에서도 자바스크립트 파일을 병렬화하여 실행할 수 있도록 웹 워커(Web worker)를 지원하고 있으나, 웹 워커는 서로 독립적인 메모리 영역을 사용하고 있기 때문에 워커들 간 메모리 값을 공유하는 것이 비효율적이라는 문제점이 존재한다.

이러한 문제를 해결하기 위해 이전에 자바스크립트 엔진 내부를 변형하여 자바스크립트 코드를 병렬화할 수 있도록 하는 병렬화 시스템들이 제시되었다. 스레드(thread) 간 메모리 공유를 꾀하는 방법들 중 하나로 내부적으로 락(lock)을 사용할 수 있도록 구현하는 방법이 있는데, 이는 프로그래머가 프로그램을 병렬화하면서 데드락(deadlock)과 같은 예외 사항들을 일일이 신경 써야 하기 때문에 사용하기 힘들다는

단점이 존재한다. 이 문제를 해결하기 위해 트랜잭셔널 메모리(transactional memory)를 사용한 병렬화 기법들이 제시되었다. 하지만, 이들은 자바스크립트 엔진을 직접 변형하는 방식이기 때문에 해당 엔진이 설치된 웹 브라우저 내에서만 한정적으로 작동 가능하다는 문제점이 존재한다.

본 논문에서는 웹 어플리케이션이 높은 호환성을 가지고 있다는 이점을 최대한 활용하기 위해 웹 워커 및 HTML5 에서 제공하는 기능들만 사용하여 자바스크립트에서 사용 가능한 소프트웨어 트랜잭셔널 메모리 시스템(jSTM)을 구현하였고, 이를 사용한 병렬화 시스템을 구현하였다.

트랜잭셔널 메모리 시스템은 여러 개의 명령어들로 이루어진 트랜잭션을 단위로 하여 공유 메모리를 접근하는 방식인데, 시스템을 구현하기 위해 각 트랜잭션에서 쓰기 명령어를 사용하여 변경된 메모리 값을 적용시키는 커밋(commit) 하는 과정과, 이전에 커밋된 값과 비교하여 트랜잭션들이 공유 메모리에 접근하여 정상적인 값을 읽었는지 확인하는 과정을 구현하여야 한다. 또한, 트랜잭션에서 잘못된 값을 읽은 경우 공유 메모리 상태를 해당 트랜잭션이 실행되기 전의 상태로 복구해야 하기 때문에 복구 지점을 만드는 과정도 필요하다.

하지만 자바스크립트의 언어적 구조 때문에 자바스크립트 엔진을 변경하지 않은 상태에서 트랜잭셔널 메모리를 구현할 때 고려해야 할 점이 존재한다. 우선, 웹 워커들은 각자 고유의 메모리 영역을 사용하고 있기 때문에 스레드들 내의 트랜잭션이 커밋한 메모리 값들이 메모리에 직접적으로 반영될 수 없다. 또한, 자바스크립트에서는 메모리 주소를 직접 사용하여 해당 메모리를 접근하는 것이 불가능하기 때문에 다른 방법을 사용하여 메모리 복구 지점을 만들어야 한다. 이를 해결하기 위해 트랜잭션의 명령어를 수행하는 부분과 트랜잭션 내에서 변경된 값을

커밋하는 스레드를 분리하였으며, 복구 지점을 만들기 위해 딥 카피(deep-copy) 방식을 사용하여 각각의 스레드들 내에 정의된 오브젝트(object)들의 모든 프로퍼티(property) 값을 복사하는 방식을 사용하여 메모리 복구 지점을 생성하였다.

이렇게 구현된 트랜잭셔널 메모리 시스템을 사용하여, 반복문 내의 각 이터레이션(iteration)들이 독립적으로 실행될 수 있는 경우에 적용 가능한 병렬화 기법인 DOALL 기법과 예측적 기법을 사용하여 이터레이션 간 의존성을 제거한 경우 각 이터레이션들이 독립적으로 실행 가능할 때 적용 가능한 Spec-DOALL 기법을 사용할 수 있도록 병렬화 API 를 구현하였으며, 구현한 시스템을 사용하여 실제로 자바스크립트 프로그램들을 병렬화하여 시스템의 성능을 측정하였다. 하지만, 시스템 자체의 부하 때문에 병렬화된 프로그램들은 이상적인 경우와 비교했을 때 훨씬 낮은 성능을 냈으며, 따라서 시스템의 성능을 향상시키기 위해 시스템 부하를 분석하였다. 시스템의 부하는 각 워커를 동작시키기 위한 동신 부하와 워커 내 트랜잭션들이 커밋하기 위해 메모리값 변경 기록을 전송하는 통신 부하가 높은 비율을 차지했으며, 따라서 이를 줄일 수 있는 방법을 추가적으로 구현하는 것이 필요하다.

Reference

[1] "Web Workers," W3C, [Online]. Available: http://www.w3.org/TR/workers/.

[2] D. Bonetta, W. Binder and C. Pautasso, "TigerQuoll: parallel event-based JavaScript," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '13)*, 2013.

[3] M. Mehrara, P.-C. Hsu, M. Samadi and S. Mahlke, "Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*, 2011.

[4] "SpiderMonkey," Mozilla Developer Network, [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[5] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang and M. Franz, "Trace-based just-in-time type specialization for dynamic languages," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI '09)*, 2009.

[6] L. Rauchwerger and D. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *Parallel and Distributed Systems, IEEE Transactions on,* vol. 10, no. 2, pp. 160-180, 1999.

[7] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh and B. Hertzberg, "McRT-STM: a high performance software transactional memory system for a multi-core runtime," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '06)*, 2006.

[8] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal and X. Tian, "Design and implementation of transactional constructs for C/C++," in *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA '08)*, 2008.

[9] V. J. Marathe and M. Moir, "Toward High Performance Nonblocking Software Transactional Memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*

(PPoPP '08), 2008.

[10] D. Dice, O. Shalev and N. Shavit, "Transactional locking II," *Distributed Computing,* pp. 194-208, 2006.

[11] L.-N. Pouchet, "PolyBench/C," [Online]. Available: http://web.cse.ohio-state.edu/~pouchet/software/polybench/.

[12] A. Erbad, N. C. Hutchinson and C. Krasic, "DOHA: scalable real-time web applications through adaptive concurrent execution," in *Proceedings of the 21st international conference on World Wide Web (WWW '12)*, 2012.

[13] S. Herhut, R. L. Hudson, T. Shpeisman and J. Sreeram, "River trail: a path to parallelism in JavaScript," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications (OOPSLA '13)*, 2013.

[14] "OpenCL - The open standard for parallel programming of heterogeneous systems," Khronos Group, [Online]. Available: https://www.khronos.org/opencl/.

Acknowledgements

감사의 글

많은 분들의 도움이 있었기에 여기까지 올 수 있었습니다.

가끔은 앞이 보이지 않아 답을 찾아 헤매기도 했고, 시행착오도 많이 겪었습니다. 이렇게 부족한 저를 믿고 이끌어주셨으며, 제가 발전할 수 있도록 조언과 도움 주신 김한준 교수님께 먼저 감사의 인사를 드리고 싶습니다. 교수님께서 주셨던 많은 조언들 덕분에 깜깜하고 막연했던 눈앞이 밝아지는 것을 느낄 수 있었습니다. 많이 부족했던 저지만, 앞으로 더욱 발전된 모습을 보여드려서 부끄럽지 않은 모습을 보여드리고 싶습니다.

지금까지 많은 것들을 겪고, 배울 수 있었지만, 그 만큼 어려운 일들도 겪었습니다. 그런 어려운 일들이 있을 때 선뜻 도움 주셨던 선배님들께 감사의 인사를 드리고 싶습니다. (여기의 '선배님'은 학과 선배님들만을 이야기하는 것이 아닌, 인생의 지혜를 주신 선배님들이나 선생님들, 교수님들 등 모든 '인생 선배님'들을 포괄하고 있는 의미입니다)

랩 로테이션 기간 중에는 김재준 교수님과 박찬익 교수님 덕분에, 연구실 생활을 하면서 전공 지식 외에도 다양한 지식을 습득할 수 있었기 때문에 감사의 인사를 드리고 싶습니다. 대학원 생활을 하면서 동고동락한 컴파일러 연구실의 모든 사람들, 그 중에서도 특히 제가 곤란한 일을 겪었을 때 자기 일처럼 도와줬던 현준이나 광무에게 감사의 인사를 전하고 싶습니다. 그리고, 바쁜 시간 쪼개셔서 논문 심사해주시고 논문 및 발표에 대해 조언 주셨던 김장우 교수님, 송황준 교수님께 감사 드립니다.

마지막으로 제가 이 자리에 있을 수 있도록 이 세상에 절 낳아주시고 길러주셨으며 제가 성장할 수 있도록 인생의 조언을 아끼지 않으셨던, 사랑하는 부모님께 이 자리를 빌어 큰 감사의 인사를 드립니다.

감사드리고 싶은 분들이 참 많습니다. 하지만 평소에는 쑥스러워서 표현 못 했던 말들이라서, 어떻게 이야기해야 이 마음을 제대로 표현할 수 있을지 모르겠습니다. 사실, 여기에는 그 마음을 온전히 표현하지 못한 것 같지만 항상 감사한 마음을 품고 있었으며, 잊지 않고 있습니다. 그래서 이 자리를 빌어서 감사의 인사를 드립니다.

감사합니다.

Curriculum Vitae

Name : Kyoungju Sim

Education

(2009~2012) B.S. in Computer Science and Engineering, Kyungpook National University

(2013~2014) M.S. in Creative IT Engineering, Pohang University of Science and Technology

Experience

(2013.10~2014.12) 차세대 Mobile Cloud Infra 기반 기술 연구, 삼성전자