Jsweet407

January 11, 2025

Contents

1 Overview

JSweet407 is a Java-to-JS transpiler built on top of Jsweet, enhanced with support for Java's concurrency features. It consists of two main components: an extended transpiler and a JavaScript multithreading runtime. Features supported include the `Thread` class, the `Runnable` interface, the `synchronized` keyword, and thread-related methods from standard libraries such as `join` and `sleep`.

Additionally, the design enables the implementation of Java standard library locking classes, such as `ReentrantLock`, `ReadWriteLock`, and `StampedLock`, which are provided as examples in the project. This section discusses JSweet407's design goals, architectural approach, and example code.

1.1 Design Goals

The primary goal of JSweet407 is to intercept cross-thread shared variable read and write operations and simulate shared memory access using message-passing mechanisms (`postMessage` and `sharedArrayBuffer`).

While JSweet407 does not analyze or classify Java's source-level concurrency patterns, it implements JavaScript Worker-based concurrency capabilities. Furthermore, the multithreading runtime system provides fundamental mechanisms for messaging, synchronization, and blocking, supporting higher-level constructs like `synchronized` and `lock`. The runtime optimizes message quantity based on the Java Memory Model to minimize communication overhead, synchronizing data only when necessary.

1.2 Key Design Concepts

1.2.1 Multithreading Model of the Runtime

JSweet407 compiles all child threads into JavaScript Workers. Each Worker maintains its local copy of the shared memory, while the main thread acts as the primary memory store and synchronizes data with Workers as needed.

Two strategies were considered for shared memory updates:

- **Push Updates**: A Worker performing a write operation pushes the changes to other Workers.
- **Pull Updates**: A Worker performing a read operation actively requests the latest data.

Due to JavaScript runtime's single-threaded asynchronous nature, implementing push updates poses challenges, such as maintaining lock and synchronization mechanisms. This requires full program asynchronization or CPS (Continuation-Passing Style) transformations. To address these challenges, JSweet407 adopts pull-based synchronization, where child threads request data updates and block until synchronization is complete.

1.2.2 Identifying Concurrent Classes

To focus on key features, JSweet407 recognizes classes that directly extend `Thread` or implement `Runnable` as concurrent classes. It does not consider deeper inheritance or implementation hierarchies.

1.2.3 Intercepting Cross-Thread Shared Variables

After identifying all concurrent classes in the first pass, JSweet407 analyzes the escape variables within them, marking these variables as cross-thread shared variables. Corresponding classes are flagged as cross-thread classes. During compilation, shared variable access is intercepted in each child thread using JavaScript's `Proxy` feature. The multithreading runtime updates and fetches data as required.

1.2.4 Messaging, Synchronization, and Blocking Mechanisms

JSweet407 utilizes `postMessage` for communication and `sharedArrayBuffer` with `Atomic` for synchronization and blocking operations. When a Worker requires data synchronization, it creates a buffer and sends it to the relevant data thread via `postMessage`, blocking itself using `Atomic.wait` until the required data is written to the buffer.

For `synchronized` blocks, handling locks relies on generating unique keys for lock objects. When multiple Workers attempt to acquire the same lock, they query the main thread to check the lock status and block themselves until notified by the main thread.

1.3 Example Code

As shown in Figure **??**, this is a multithreaded producer-consumer Java example using `synchronized` blocks with the static variable `lock` of the `Desk` class as the lock object. Figure **??** shows the corresponding compiled TypeScript code.

The `synchronized` blocks are transpiled into calls to `Comm.sync`, with exits like `break`, `return`, and `continue` translated to `Comm.unsync` calls to release the lock. Both `sync` and `unsync` take the lock object as an argument. The transpiler and runtime ensure that lock objects have unique multithreading keys, which are used for locking, unlocking, and thread scheduling.

Additionally, the runtime defines lock-related methods like `wait` and `notify` for each object.

1.4 Acceptance Process

To ensure proper setup, the required tools and dependencies, such as Python, JDK, and TSC, must be available in the command-line environment.

All test cases for JSweet407 are located in the `benchmark` folder in the project root directory. A `run.sh` bash script is provided to compile and test files. This script uses a pre-compiled JAR file in the `artifacts` directory to compile files in the `example` folder. Note that only one file can be processed at a time to avoid conflicts.

The intermediate results are saved in the `tsout` directory, and the final compiled JavaScript code is copied to `runtime/src/compiled.js`. Users can view the results by serving `runtime/src/test.html` via an HTTP server. The complete process is demonstrated in the provided acceptance video.

2 Specific Implementation

2.1 Runtime Design

2.1.1 Concurrent Class Initialization

The concurrent class `WebWorker` corresponds to the `Thread` class in Java, and is also the entry point for accessing a series of runtime APIs. If a class inherits from `WebWorker`, it will have the characteristics of a concurrent class. `WebWorker` will treat the `source` in that class as an executable function and pass it to an initialized Worker for execution. (Note that the Worker here refers to a real Web Worker, while our `WebWorker` class is just a wrapper around the Web Worker.)

Now, let's explain how `WebWorker` works in detail.

```
class WebWorker {
    worker = null;
    workerId = null;
    static workerCounter = 0;
    static nextId() {
        WebWorker.workerCounter += 1;
        return WebWorker.workerCounter;
    }
    init() {
        if (this.worker)
            return;
        // initialize worker
        this.worker = new Worker('./initWorker.js');
        this.worker.onmessage = onmessage;
        // assign the id to every worker
        this.workerId = WebWorker.nextId();
    }
    // 注册完成后，启动worker
    start() {
        this.init();
        this.worker.postMessage({ 'command': 'start', 'source': `(${this.source.
            toString()})()` });
    }
}
```

As shown, after calling the `init` method, `WebWorker` will automatically create a web worker. This worker is a template worker, meaning that all `WebWorker`s will create the same `initWorker`, which will listen for messages from the main thread. So when does each thread start running its own code? After calling the `start` method, the main thread will send the code that the worker needs to execute to the worker. Upon receiving the message, the worker will begin execution.

2.1.2 Message Communication

Since each worker's memory is isolated, but workers may need to share data, a message communication mechanism is required.

Before introducing the mechanism we are using, let's first discuss another method we tried.

We initially used a decentralized message-passing model, meaning each worker could directly communicate with another worker. Each worker has a copy of all shared variables, and when a worker modifies a shared variable, it notifies all other workers holding that shared variable to ensure data consistency across workers.

The problem with this method is that implementing locks becomes cumbersome, as consensus needs to be reached between all relevant workers to acquire and release the lock. This involves significant messaging overhead. Additionally, due to the latency in message communication, it is difficult to synchronize processes reliably.

The solution we currently adopt is to have the main thread act as the message hub. Specifically:

1. All messages from threads pass through the main thread.
2. The main thread manages all shared variables, and local copies in the worker are mainly used to optimize the frequency of message communications (as will be discussed later).

Message Format Messages have many attributes, usually including a `command` attribute. The `command` is an instruction for the receiver to take action. For example, the following message:

```
postMessage({ 'command': 'update', key, value * other properties * });
```

This is a data update instruction sent from a child thread to the main thread, indicating that the thread has changed the value of a shared variable and wishes for other threads to see the update. In addition to this command, there are many other commands, such as queries for shared variables, notifying the main thread to release a lock, etc., which will be introduced later.

Data Synchronization Timing When should these shared variables be synchronized? Here, we use JavaScript's **Proxy Objects**.

```
new Proxy(target, {
    get: function (_target, propKey) {
        // Request data from the main thread
        return _target[propKey];
    },
    set: function (_target, propKey, newValue) {
        // Send the latest data to the main thread
        return true;
    }
});
```

Each shared variable requires a Proxy object, and all operations on the shared variable must go through the Proxy object. When a `get` operation is performed on the shared variable, it is converted into a query operation to the main thread; when a `set` operation is performed, it initiates an update operation to the main thread.

Synchronized Messages In JavaScript, `postMessage` is an **asynchronous method**, meaning the thread continues executing after sending a message and does not wait for a reply. This is unacceptable in many cases, such as when a child thread queries data from the main thread, needs to wait for a reply from the main thread, and then proceed after obtaining the updated data. Moreover, the implementation of locks in the future will also require similar synchronization mechanisms.

The implementation of the synchronization mechanism is as follows:

```
static synchronizePostMessage(message) {
    const lock = createLock();
    postMessage({ ...message, lock });
    Atomics.wait(lock, 0, 0);
}
function createLock() {
    const lock = new Int32Array(new SharedArrayBuffer(8));
    return lock;
}
```

This requires the use of JavaScript's `SharedArrayBuffer` and `Atomics` operations. `SharedArrayBuffer` allocates a block of shared memory, and `Atomics` allows atomic operations on shared memory. They are usually used together. When a child thread sends a message to the main thread, it can attach a `SharedArrayBuffer` and wait for it to be modified.

```
function releaseLock(lock) {
    Atomics.store(lock, 0, 1);
    Atomics.notify(lock, 0);
}
```

When the main thread receives the `SharedArrayBuffer`, it writes to it after completing the operation. This allows the child thread to detect the modification, stop waiting, and continue execution.

This is the most basic synchronization method, which forms the basis for many subsequent synchronization methods.

2.1.3 Synchronization Methods

Synchronized In the example code from this section, we can see that code like **??** will be compiled into the form shown in **??**. `Comm` is a static class created when initializing the worker, providing methods like `sync`, `unsync`, etc. Calling a method in `Comm` sends a message to the main thread, which processes the message upon receiving it. When implementing synchronization, we define several data structures to assist in lock implementation, as follows.

```
if(lockHolders.get(key)){ //判断是否有线程持有该锁
    if(lockHolders.get(key).workerId !== data.workerId)     { //判断持有锁的线程
        是否为自己
        joinBlockQueue(key,data); //加入阻塞队列
    }else{ //实现可重入锁
        lockHolders.get(key).count += 1;
        releaseLock(data.lock); //线程拿到锁, 继续执行
    }
}else{  //当前无线程持有锁
    lockHolders.set(key,data); 设置锁的持有者为本线程
    releaseLock(data.lock);
}
```

Figure 1: sync 逻辑

```
const blockQueues = new Map();
const lockHolders = new Map(); // Lock holders
const waitingQueues = new Map(); // Lock waiting queues
```

Here is the processing logic for the main thread.

sync(lock) When a process requests a lock, if another thread holds the lock, it is added to the blocking queue and execution is paused. Otherwise, the thread acquires the lock and continues execution, as shown in Figure 1.

unsync(lock) The process exits the lock and checks if the lock's reentry count is 0. If it is 0, the lock is allocated to other blocked processes. The code is shown in Figure 2.

wait() When a process calls the wait method, it enters the wait queue. Waiting is different from blocking; blocking occurs when a process fails to acquire the lock and is passively blocked, while waiting refers to a process that actively waits for a specific condition to occur. When a process calls wait, it releases its own lock so that other blocked threads can be scheduled. The code is shown in Figure 3.

notify() The process calls the notify() method to inform other threads waiting on the same lock that the waiting condition has been met and moves one of them from the waiting queue to the blocking queue, waiting to be scheduled. The code is shown in Figure 4.

notifyAll() Similar to the notify() method, the only difference is that it moves all threads in the wait queue for the same lock to the blocking queue.

ReentrantLock The ReentrantLock class is loaded at initialization to support code using the ReentrantLock lock. This class implements the following methods, and when the corresponding method is called, a message is sent to the main thread for the corresponding operation.

```
if (lockHolders.get(key)) {
    lockHolders.get(key).count -= 1;//持有锁的数量减1
    if (lockHolders.get(key).count === 0) {//判断是否释放锁
        lockHolders.delete(key);
        let dataList = blockQueues.get(key);//取出阻塞队列的第一个元素
        if(!dataList){
            return;
        }
        let first = dataList.shift();
        if(!first){
            return;
        }
        releaseLock(first.lock);
        lockHolders.set(key, first);
    }
}
```

Figure 2: unsync 逻辑

```
function wait(data) {
    let key = data.key;
    data.count = lockHolders.get(key).count; // 记录锁重入数
    joinWaitingQueue(key, data);
    lockHolders.delete(key);//释放锁
    dispatchLock(key); //调度其他进程
}
```

Figure 3: wait 逻辑

```
class ReentrantLock{
    lock(){...}
    unlock(){...}
    tryLock(){...}
    tryLock(time,TimeUnit){...}
    newCondition(){...}
}
```

The lock and unlock methods in ReentrantLock are similar to sync and unsync, and ReentrantLock does not provide notify() and wait() methods. The waiting and waking mechanism is implemented through the Condition class. ReentrantLock provides the newCondition() method to create a Condition. The Condition class provides the await(), signal(), and signalAll() methods to implement waiting and waking. Using Condition for waiting and waking is more flexible. A lock can create multiple Conditions, and the signal() and signalAll() methods in each Condition will only wake up the await() in the corresponding Condition.

The main thread processing logic is as follows:

```
function notify(data) {
    let key = data.key;
    if (waitingQueues.has(key)) {
        let set = waitingQueues.get(key);
        const d = set.shift();
        if (d) {
            joinBlockQueue(key, d);
        }
}
```

Figure 4: notify 逻辑

```
function tryLock(data){
    let key = data.key;
    let type =data.type;
    if(lockHolders.get(key)){ //判断是否有线程占有锁
        if(lockHolders.get(key).workerId !== data.workerId){ //判断占有锁的线程是
            否为本线程
            failReleaseLock(data.lock);//获取锁失败，返回false
        }else{//本线程已经占有锁，重入数加一
            lockHolders.get(key).count += 1;
            releaseLock(data.lock);
        }
    }else{//无线程占有锁
        lockHolders.set(key,data);
        releaseLock(data.lock);
}
```

Figure 5: tryLock 方法逻辑

await() To implement waiting and waking based on different conditions, the waiting queue in synchronized is replaced with a condition waiting queue, and the rest is similar to wait().

signal() Moves one thread from the condition waiting queue of the lock to the blocking queue, similar to notify().

signalAll() Moves all threads waiting for the same lock and condition from the waiting queue to the blocking queue, similar to notifyAll().

tryLock() The process attempts to acquire the lock. If the lock is held by another thread, it returns false; otherwise, it returns true. The code is shown in Figure 5.

tryLock(time, TimeUnit) Here, time is the time and TimeUnit is the time unit. The process tries to acquire the lock within the given time. If the attempt fails, it joins the blocking

```
tryLock(time, timeUnit) {
    let waitTime = time * timeUnit;
    const lock = createLock();
    postMessage({ 'command': 'sync', 'key': this.__key, 'lock': lock, "workerId":
        workerId });
    if (Atomics.wait(lock, 0, 0, waitTime) === "timed-out") {
        Atomics.store(lock, 0, 2); //如果超时，将锁标记为超时
        return false;
    }
    return true;
}
```

Figure 6: 带参数的 tryLock 子线程处理逻辑

```
function tryLock(data){
    let key = data.key;
    let type =data.type;
    if(lockHolders.get(key)){ //判断是否有线程占有锁
        if(lockHolders.get(key).workerId !== data.workerId){ //判断占有锁的线程是
            否为本线程
            joinBlockQueue(key,data);//加入阻塞队列
        }else{//本线程已经占有锁，重入数加一
            lockHolders.get(key).count += 1;
            releaseLock(data.lock);
        }
    }else{//无线程占有锁
        lockHolders.set(key,data);
        releaseLock(data.lock);
}
```

Figure 7: 带参数的 tryLock 主线程处理逻辑

queue, and if the timeout expires without acquiring the lock, it returns false. Otherwise, it returns true.

Other methods involve sending specific messages in the child thread and processing them in the main thread, but this method differs as it requires additional handling in the child thread. The child thread must mark processes that have timed out to prevent them from being scheduled again. The child thread logic is shown in Figure 6, and the main thread logic is shown in Figure 7.

ReentrantReadWriteLock During initialization, the ReentrantReadWriteLock class is loaded, implementing readLock() and writeLock() methods that return read and write locks.

The readLock and writeLock classes contain the following methods, and the trigger mechanism is similar to that of ReentrantLock.

```
class writeLock{
    lock(){...}
```

```
if(type === 'Write'){ //新申请的锁是写锁
    if(lockHolders.get(key)){ //当前有线程持有锁
        if(lockHolders.get(key).workerId !== data.workerId || lockHolders.get(key
            ).type === 'Read'){ //如当前锁的持有者不是本线程或者，本线程已经持有读
            锁
            joinBlockQueue(key,data); //加入阻塞队列
        }else{
            lockHolders.get(key).count += 1;//重入数加一
            releaseLock(data.lock);
        }
    }else{//无线程持有锁
        lockHolders.set(key,data);
        releaseLock(data.lock);
    }
}else{  //新申请的是读锁
    if(lockHolders.get(key)){ //有线程持有锁
        if(lockHolders.get(key).workerId !== data.workerId && lockHolders.get(key
            ).type === 'Write')    {  //如果是其他线程的写锁
            joinBlockQueue(key,data); //加入阻塞队列
        }else{
            lockHolders.get(key).count += 1; //重入数加一
            releaseLock(data.lock);
        }
    }else{ //无线程持有锁
        lockHolders.set(key,data);
        releaseLock(data.lock);
    }
}
```

Figure 8: 读写锁判断逻辑

```
    unlock(){...}
    tryLock(){...}
    tryLock(time,TimeUnit){...}
    newCondition(){...}
}
```

lock() Since the read lock is shared, when a process holds the read lock, other processes can still acquire the read lock. Therefore, additional checks are needed when attempting to acquire the lock. The logic is shown in Figure 8.

unlock() When releasing a lock, if the lock being released is a write lock and the first request in the blocking queue is a read lock request, all read lock requests are released. The logic for this is shown in Figure 9.

```
//如果阻塞队列第一个是申请读锁，那么将所有读都释放
if(first.type === 'Read'){
    //从后向前遍历，释放所有读锁
    for(let i = dataList.length - 1; i >= 0; i--){
        if(dataList[i].type === 'Read'){
            releaseLock(dataList[i].lock);
            lockHolders.get(key).count += 1;
            dataList.splice(i,1);
        }
    }
}
```

Figure 9: 读写锁释放逻辑

StampedLock During initialization, the StampedLock class is loaded, which implements the following methods. The trigger mechanism is similar to the locks mentioned above, sending a message to the main thread for processing.

```
class StampedLock{
    readLock() {...}
    unlockRead(stamp) {...}
    writeLock() {..}
    unlockWrite(stamp) {...}
    tryReadLock(time, timeUnit) {...}
    tryWriteLock(time, timeUnit) {...}
    tryOptimisticRead() {...}
    validate(stamp) {...}
    tryConvertToWriteLock(stamp) {...}
    tryConvertToReadLock(stamp) {...}
}
```

In StampedLock, the state variable is used to represent the current lock version number and other lock information. The state is an integer, where the first 24 bits represent the version number, the high bit of the last 8 bits represents whether a write lock is held, and the lower 7 bits represent the number of read locks, so up to 127 read locks can be held.

readLock() If there is no write lock and the number of read locks is less than 127, the request succeeds and returns the state value. Otherwise, it returns 0, indicating failure.

writeLock() If there are no other locks, the request succeeds and returns the state value. Otherwise, it returns 0, indicating failure.

unlockRead(stamp) Releases the read lock based on the stamp value and updates the state.

unlockWrite(stamp) Releases the write lock based on the stamp value and updates the state.

tryReadLock() Attempts to acquire a read lock. If another process holds the write lock, it returns 0; otherwise, it returns the state.

tryWriteLock() Attempts to acquire a write lock. If another process holds a read or write lock, it returns 0; otherwise, it returns the state.

tryReadLock(time, TimeUnit) Attempts to acquire a read lock within the given time. If successful, it returns the state; if not, it returns 0.

tryWriteLock(time, TimeUnit) Attempts to acquire a write lock within the given time. If successful, it returns the state; if not, it returns 0.

tryOptimisticRead() Attempts to perform an optimistic read and returns a state value.

validate(stamp) Checks whether the version has changed (i.e., if any thread has requested a write lock). If the version has changed, it returns 0; otherwise, it returns 1.

tryConvertToWriteLock(stamp) Attempts to convert a read lock to a write lock. If it fails, it returns 0; otherwise, it returns the state.

tryConvertToReadLock(stamp) Attempts to convert a write lock to a read lock. If it fails, it returns 0; otherwise, it returns the state.

sleep The implementation of sleep is simple: it only needs to record the start time, calculate the end time, and then loop to check if the current time has exceeded the end time. This method avoids using await.

join Since the main thread cannot be blocked, we only implement join between child threads. Each thread, when finished, sends a message to the main thread, which marks the thread as finished. When a thread joins another, it sends a join command along with the id of the thread it wants to join. The main thread checks if the thread has finished. If it has, the thread continues execution; otherwise, it waits.

Example usage:

```
class Scope {
    public static Thread t1;
    public static Object lock = new Object();
}
// In another thread, call:
Scope.t1.join();
```

2.1.4 Message Optimization

The buildProxy function is used to create proxy objects that can intercept and process the properties of target objects. In this implementation, the proxy object is mainly used to access the properties of the class and provides optimization features. The optimization pattern for the buildProxy part of the code is as follows:

```
//initWorker.js
//map存储所有改变了的key value
let changedObjects = new Map();
//map存储所有的主线程中的key value
let mainObject = new Map();
let buildProxy = () => {
    return new Proxy(target, {
        get: function (_target, propKey) {
            let key = className + '.' + propKey;
            // 如果是数组, 递归创建代理
            if (Array.isArray(_target[propKey])) {
                return buildProxy(_target[propKey], key);
            }
            if (mainObject.has(key)) {
                return mainObject.get(key);
            }
            return _target[propKey];
        },
        set: function (_target, propKey, newValue) {
            ...
            _target[propKey] = newValue;
            mainObject.set(key, newValue);
            changedObjects.set(key, newValue);
            return true;
        }
    });
}
```

Optimization Approach According to the Java memory model, the program copies data from the main memory to the CPU cache during runtime. Accessing data from the cache is much faster than directly accessing it from the main memory. Based on this, our optimization approach is to temporarily store the shared data from the main thread in the child thread, thus reducing the number of communication calls and improving performance.

However, this creates a new problem: the data consistency between the main memory and the cache. To solve this, we refer to the happens-before principle in the Java Memory Model (JMM), which states that:

- Unlock operations occur before lock operations.
- Writes to volatile variables happen before subsequent reads.

- The previous instruction within a thread happens before the next instruction.

In our implementation, we also try to follow these principles as much as possible. For example, for write operations on volatile variables, we synchronize updates to the main thread. For read operations on volatile variables, we directly fetch data from the main thread, thereby simulating the behavior of volatile variables in Java. The specific implementation is as follows:

To ensure data synchronization, we need to retrieve the latest value from the main thread during a get operation, and update the main thread during a set operation. However, each get and set operation requires sending messages, so we optimize to reduce the number of messages. After entering the synchronized code block, i.e., in the Comm.sync function, we call the batch_query function to send a message to the main thread to retrieve all the data stored in the main thread. After that, during get operations, no message needs to be sent to the main thread. During set operations, modified data is recorded in a Map, and before exiting the synchronized code block, i.e., in the Comm.unsync function, the batch_update function is called to update the data in the Map to the main thread through a single message. This way, we can reduce the number of messages to two. The code for requesting data is as follows:

```
//initWorker.js
static batch_query() {
    const buffer = new SharedArrayBuffer(BUF_SIZE * 2);
    const arr = new Int32Array(buffer);
    let _objects = this.synchronizePostMessageWithData(arr);
    mainObject = new Map([...mainObject, ..._objects]);
}
static synchronizePostMessageWithData(arr) { // Int32Array
    postMessage({ 'command': 'batch_query', 'arr': arr });
    Atomics.wait(arr, 0, 0);
    let str = '';
    str += deserialize2MapStr(arr);
    let flag = Atomics.load(arr, 0);
    while (flag !== -1) {
        // 如果还有后续的数据传送，要继续接收
    }
    let objects = deserialize2Map(str);
    return objects;
}
```

The code for updating data is as follows:

```
//initWorker.js
static batch_update(changedObjs) {
    if (!changedObjs || changedObjs.size === 0) return;
    this.synchronizePostMessage({ 'command': 'batch_update', 'obj': changedObjs
       });
    changedObjs.clear();
}
static synchronizePostMessage(message) {
    const lock = createLock();
    postMessage({ ...message, lock });
    Atomics.wait(lock, 0, 0);
}
```

Data Transmission Next, we will explain how data is transmitted between the main thread and worker threads in the optimized model.

To retrieve data from the main thread, as shown in the code for requesting data, we create shared memory and send it to the main thread, using Atomic.wait to implement blocking. After receiving the message, the main thread serializes the Map that holds the data into a JSON string and writes it into shared memory one or multiple times (if shared memory is large enough, or multiple times if the memory is not large enough). The first byte is set as a flag: -1 means no further data, and 0 means more data follows. The worker thread reads the data one or more times and performs JSON deserialization to convert the string back into a Map.

To update the modified data to the main thread, as shown in the code for updating data, we send the Map as part of the message content and block after sending. After receiving the message, the main thread retrieves the Map to update the data and wakes up the worker thread after the update.

It should be noted that retrieving data from the main thread cannot be done using message-based data transmission in the same way as updating data. This is because after sending the message to request data, the worker thread needs to block, and during the blocking period, it cannot receive messages sent by the main thread. Therefore, we use shared memory and resolve the issue of shared memory size limitations. If shared memory is insufficient, the main thread writes the JSON serialized string multiple times into the shared memory, setting the first byte as a flag. -1 indicates no subsequent data, and 0 indicates that more data follows. The worker also processes this accordingly. The logic for responding with data is as follows:

```
//WebWorker.js
 function responseWithData(arr) {
    const JSONStr = JSON.stringify(Array.from(objects));
    const size = JSONStr.length;
    const batch = Math.ceil(size / (BUF_SIZE - 4));
    for (let i = 0; i < batch; i++) {
        //序列化
        populateArray(arr, JSONStr, i * (BUF_SIZE - 4), BUF_SIZE - 4);
        if (i === batch - 1) {
            Atomics.store(arr, 0, -1);
            Atomics.notify(arr, 0);
        } else {
            Atomics.store(arr, 0, size);
            Atomics.notify(arr, 0);
            while (Atomics.load(arr, 0) !== 0) { }
        }
    }
}
```

2.2 Compiler Extension

The original JSweet compiler does not support the recognition and translation of Java multithreading features. To address this issue, we have extended the compiler to allow JSweet to recognize concurrency-related features in Java and translate them into executable TypeScript files. The tool now supports the recognition of concurrent classes, the `synchronized` keyword, and the `volatile` keyword.

2.2.1 Concurrency Class Handling

In our compiler extension work, we first introduced support for recognizing concurrency classes. This support mainly involves three areas: adding the recognition feature for concurrent classes in the `Util.java` class, creating the `CvsAnalyzer.java` class to analyze and store shared variables that may be used for communication between different threads, and updating the `Java2TypeScriptTranslator.java` class to handle the translation of these classes and methods. The following sections will detail the improvements and implementation details in each area.

Util Class Update. In the `Util.java` class, we added a method `is_parallel(ClassTree clzTree)`. This method is used to determine whether a class has concurrency characteristics. Specifically, a class is considered a concurrency class if it either extends the `Thread` class or implements the `Runnable` interface. During implementation, we used syntax tree methods to obtain the inheritance and implementation relationships of classes and then check whether a class meets the conditions of being a concurrency class. This functionality provides the foundational support for subsequent multithreading analysis and processing.

New `CvsAnalyzer` Class. To further enhance the analysis and recognition of multithreading-related variables, we created a class named `CvsAnalyzer.java`. This class is an important component of our compiler, used to analyze cross-variable references in Java source code. These cross-variables include static variables and `volatile` variables, which may be used in a multithreaded environment and therefore need to be accurately identified and handled. In the implementation, we used syntax tree traversal and analysis to identify and record variables in each class that might be involved in multithreading. These variables are stored in the `capture_cvs` list for subsequent inter-thread communication.

In the implementation of `CvsAnalyzer`, we used a stack data structure to manage class scopes. By dynamically entering and exiting scopes, we were able to effectively track variable reference relationships during syntax tree traversal. The core of the class consists of two parts: scope management and cross-variable analysis. The scope management part has two key variables: `clzScope` and `currentOutScope`, which are used to record the current class's scope and its related outer classes. The cross-variable analysis part also has two key variables: `cvsScope` and `volatileCvsScope`, which store the mapping relationships between a class and its cross-variables and `volatile` variables.

Additionally, we overrode the methods `visitClass`, `visitMethod`, `visitVariable`, `visitIdentifier`, and `visitMemberSelect` to analyze the structure of each class according to our specific needs and accurately record the static and `volatile` variables. When recognizing variables, we also considered the issue of keywords in Java to avoid mistaking keywords for variable references. To improve recognition accuracy, we included references to static variables and external class member variables in the analysis scope to ensure that static variables in non-thread classes referenced by thread classes are not overlooked. This allows us to capture cross-variable references more comprehensively, providing accurate data for subsequent processing.

Java2TypeScriptTranslator Class Update. In the `Java2TypeScriptTranslator.java` class, we extended the translation of classes and methods to handle multithreading-related classes accordingly. Specifically, when a class is identified as a concurrency class, we add cross-variables and `volatile` variables during the translation process to facilitate inter-thread communication and handling in the generated TypeScript code.

2.2.2 synchronized

The `synchronized` keyword in Java ensures that only one thread can operate on a shared resource at any given time, thus achieving thread safety for shared resources in a multithreaded environment. `synchronized` is a reentrant, unfair, exclusive, pessimistic lock. The locks created using `synchronized` are divided into object locks and class locks.

Object locks include synchronized block locks (with a user-defined lock object) and method locks (with the default lock object being `this`, the current instance object). The handling of synchronized block locks and the methods for locking and unlocking have been mentioned earlier. We use the custom `Comm.sync()` and `Comm.unsync()` instructions to lock and unlock code blocks, as shown in Figures 1 and 2. When we encounter a `synchronized` keyword, we perform the locking, but we must ensure that the unlocking occurs at the correct places, such as at the end of code blocks or before returning from functions, to avoid deadlocks.

For method locks, we typically use the `synchronized` keyword to modify a regular method. We first check if the method is marked with `synchronized`, and we treat this as locking the entire method’ s code block. As mentioned before, we use `Comm.sync(this)` to lock the method, and since the entire method is locked, we unlock it before returning using `Comm.unsync(this)`. To handle identical locks, we rely on each lock object’s key. Since `this` does not have a key, we assign a random key to `this` during runtime, ensuring uniqueness for each instance.

Class locks occur when `synchronized` modifies static methods or when the lock object is the `Class` object. For static methods marked with `synchronized`, we check if both `static` and `synchronized` appear in the method’s modifiers and then apply a lock using the `Class` object. In Java, this is implemented as `ClassName.class`. Here, we use `Comm.sync()` with `ClassName.class` inside the parentheses to achieve this. During compilation, we add a class variable, set its key, and initialize it with the class name. This ensures that locks and unlocks are applied correctly for different classes.

2.2.3 volatile

The `volatile` keyword in Java differs from `synchronized` in that it provides a lightweight synchronization mechanism, ensuring visibility and prohibiting instruction reordering, but it does not guarantee atomicity or mutual exclusion. In our implementation, to support `volatile` variables, we store `volatile` variables used in Java in a corresponding `__captured_volatile_cvs`

```
class TestThread extends Thread {
    @Override
    public void run() {
        int i = 1;
        while (!Data.stop) {
            i++;
        }
        System.out.println("Thread1 stop: i=" + i);
    }
}

class TestThread2 extends Thread {
    @Override
    public void run() {
        Data.stop = true;
        System.out.println("Thread2 update stop to: " +
             Data.stop);
    }
}
```

(a) Volatile 可见性测试核心代码

```
public class VolatileFunc implements Runnable {

    @Override
    public void run() {
        addnum();
    }

    public void addnum() {
        int i = 0;
        System.out.println("begin");
        while(i != 1000){
            mData.m = mData.m + 1;
            i = i + 1;
        }
        System.out.println(mData.m);
        System.out.println("end");
    }

    public static void main(String[] args) {
        Runnable Add = new VolatileFunc();
        Thread t1 = new Thread(Add);
        Thread t2 = new Thread(Add);
        t1.start();
        t2.start();
    }
}
```

(b) Volatile 原子性测试核心代码

dictionary in the TypeScript class. This dictionary stores all `volatile` variables and their corresponding values. The following example illustrates how we support `volatile` variables.

Example Figure 10a demonstrates the use case for `volatile` variables. In this example, if `Data.stop` is not marked as `volatile`, the `testThread` will get its value from its own cache, failing to detect other threads' modifications and not exiting the loop as expected. However, if `volatile` is used, `testThread` will get the updated value directly from the main memory and exit the loop as intended. Since JavaScript cannot directly access main memory, we ensure `volatile` variable visibility by immediately updating the data center to guarantee correct execution in a multithreaded environment.

Example Figure 10b demonstrates the difference between `volatile` and `synchronized`. Since `volatile` does not provide atomicity, when multiple threads perform the operation `mData.m = mData.m + 1`, the result will be unstable, and the final output will be a value between 1000 and 2000. During translation, we preserve this characteristic of `volatile`, so the generated JavaScript will always output a value between 1000 and 2000.

In practice, for each multithreaded object, we generate a proxy. When the multithreaded object accesses a variable, the proxy automatically checks if the variable is `volatile`. If it is, the proxy performs the corresponding `get` and `set` operations. For the `get` operation, the proxy immediately reads the latest value from the data center and returns it to the thread. For the `set` operation, the proxy writes the new value to the data center, ensuring other threads can immediately see the update. This guarantees the visibility of `volatile` variables and ensures data synchronization and consistency in a multithreaded environment.

Table 1: Message Number

	Before Optimization					**After Optimization**					**Average Before**	**Average After**
Actor	183	212	184	183	183	69	83	70	69	69	189	72
Alter	60	57	60	57	60	169	169	170	169	170	169.4	58.8
Array	1866	1563	1563	1565	1865	20	20	20	20	20	1684.4	20
BankAccount	185	182	181	181	182	75	75	75	75	75	182.2	75
Bicycle	378	378	378	378	378	141	138	141	141	141	378	140.4
CookAndCustomer	576	295	295	575	295	108	108	108	229	222	407.2	155
CookAndCustomerWithSleep	358	356	356	357	357	150	150	150	150	150	356.8	150
CookAndCustomers	3482	3426	3425	3411	3425	1333	1301	1301	1308	1301	3433.8	1308.8
Join	26	25	25	33	25	11	11	11	14	11	26.8	11.6
MultiProducer	603	603	502	581	591	246	252	248	248	246	576	248
NoSyncClass	40008	40008	40008	40008	40008	2	2	2	2	2	40008	2
NoSyncFunc	4012	4012	4012	4012	4012	10	10	10	10	10	4012	10
OnlySyncFunc	40012	40012	40012	40012	40012	10	10	10	10	10	40012	10
ParkingLot	311	311	311	311	311	107	107	107	107	107	311	107
ProducerAndAssmbler	466	463	325	442	463	177	177	169	169	119	431.8	162.2
ProducerAndComsumer	29030	29017	29027	29021	29017	10004	10004	10002	10006	10004	29022.4	10004
ReadWriteLock	-	-	-	-	-	870	873	870	870	870	-	870.6
StampedLock	-	-	-	-	-	72	96	72	75	75	-	78
SyncClass	40010	40010	40010	40010	40010	10	10	10	10	10	40010	10
SyncFunc	40012	40012	40012	40012	40012	10	10	10	10	10	40012	10
TestRunnable	1	1	1	1	1	1	1	1	1	1	1	1
VolatileFunc	4006	4006	4006	4006	4006	4004	4004	4004	4004	4004	4006	4004
VolatileTest	1320	3959	3301	3672	3141	3669	3683	3322	2876	3457	3078.6	3401.4

3 Evaluation Results of Message Optimization

To validate the message optimization results of JSweet407, we conducted ten experiments for each test file in the project benchmark. Five experiments were performed before optimization, and five after optimization. Table 1 records the number of message packets transmitted after running all tests in the project benchmark. From the overall data performance, our optimization scheme based on the Java Memory Model shows significant effectiveness and performs well for most programs with correct locking mechanisms.

The best optimization result was achieved with the NoSyncClass case, reducing the number of messages from 40,008 to 2. The reason is that in this example, a shared variable was accessed tens of thousands of times under a lock, while the lock operation itself only occurred twice. Therefore, in the optimized code, only two data synchronizations and two message transmissions are performed.

Our optimization did not affect the VolatileFunc and VolatileTest cases, as these programs use the `volatile` keyword. Shared variables marked with this keyword update data to the main thread with every write operation, resulting in no significant changes before and after optimization.

Additionally, the ReadWriteLock and StampedLock examples had bugs before optimization, so the pre-optimization data is unavailable.

4 Features Pending Support

We have made some simplifications in the implementation of JSweet407, and it is currently not suitable for real multi-threaded Java translation. This section will detail the simplified

```
class F {
    public int i;
    public static int j;
}
public class A {
    public static void main(String[] args) throws InterruptedException {
        T t1 = new T();
        T t2 = new T();
        F f = new F();
        t1.f = f;
        t2.f = f;
        System.out.println(f.i);

    }
}
class T extends Thread {
    public F f;
    public int j;
    public void run() {
        while (j < 5) {
            synchronized(f) {
                f.i++;
                System.out.println(f.i);
            }
            j++;
        }
    }
}
```

Figure 11: 跨线程共享变量

features and corresponding follow-up solutions.

4.1 Complete Cross-Thread Variables

As shown in Figure 11, JSweet407 currently only supports correctly reading and writing static cross-thread variables, such as the `public static int j` in class `F`.

However, in the `main` function, the `f` member variable in the two thread classes `t1` and `t2` both point to the same object. Therefore, in `t1` and `t2`, member variables can be directly read and written. JSweet407 does not currently handle this scenario. A potential solution is to refer to the proxy approach for cross-thread classes and design a proxy for concurrent classes that sends corresponding messages to subthreads when read or write operations occur.

4.2 Separation of Main Thread Logic Code and Data Processing/Scheduling Code

In the current runtime implementation of JSweet407, the main thread is not only responsible for executing the logic code of the main thread portion of the Java source code, but also undertakes tasks such as data center management and subthread scheduling and locking.

These tasks, such as data center management and subthread scheduling/locking, are triggered by message reception events, which can conflict with the logic of the main thread's code. For example, if the main thread is handling a highly time-consuming task (e.g., `while(true)`), it cannot respond to message requests in the event loop. In this situation, communication with subthreads will be entirely blocked until the time-consuming task finishes, after which the main thread processes the messages in the event queue.

On the other hand, JavaScript's main thread environment does not allow blocking operations, making all thread collaboration mechanisms difficult to implement. For instance, when the main thread performs a `join` operation on a subthread, the compiled code has two options: 1. Directly block at the `join` point. 2. Transform the subsequent code into an asynchronous form and execute it via a callback after the `join` succeeds. Evidently, the first option is simpler and more aligned with Java's original semantics. However, due to the non-blocking nature of JavaScript's main thread, this behavior cannot be directly supported.

The solutions to the above two challenges are relatively straightforward: compile all the logical code of the original Java main thread into a subthread, leaving the main thread responsible only for data communication and locking/scheduling tasks. After separation, the main thread will not perform any time-consuming computational tasks. Moreover, the `join` code, now in the subthread, can conveniently execute blocking operations.

4.3 Public Method Invocation on Parallel Objects

JSweet407 currently does not support public method invocations on subthreads because method calls involve data transmission between the subthreads and the main thread. As mentioned earlier, the issue of data exchange between the main thread and subthreads is not yet supported.