# CCTrans: A Java-to-JavaScript Translation with Concurrency Runtime

Wenzhang Yang[*]
yywwzz@mail.ustc.edu.cn
University of Science and Technology of China
Anhui, China

Yan Guo
guoyan@ustc.edu.cn
Suzhou Institute for Advanced Study University of Science and Technology of China
Jiangsu, China

Yinxing Xue[†][*]
yxxue@ustc.edu.cn
University of Science and Technology of China
Anhui, China

## ABSTRACT

Java is a widely used programming language with the most comprehensive libraries in the world. To leverage the vast resources of the Java ecosystem, several transpilers exist that facilitate the conversion of Java libraries to the more nascent programming language, JavaScript. Nonetheless, the transpilation of concurrent Java to JavaScript presents significant challenges, primarily due to the profound differences in memory model and concurrency model between the two languages. To bridge this gap, we develop a JavaScript concurrency runtime capable of supporting the shared memory model and synchronization mechanisms inherent to Java threads. To evaluate the effectiveness of our tool, we manually construct a concurrency Java dataset. Preliminary experimental findings indicate that our tool successfully transpiles concurrent Java to JavaScript using multiple workers, while maintaining identical behavior. The source code of our tool is available on: https://atomgit.com/openharmony_jsweet/06-jsweet407. The corresponding demonstration video can be found at: https://youtu.be/jB7sVUVWWTo.

## KEYWORDS

Transpiler, Java, JavaScript, Concurrency

**ACM Reference Format:**
Wenzhang Yang, Yan Guo, and Yinxing Xue. 2024. CCTrans: A Java-to-JavaScript Translation with Concurrency Runtime. In . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

As one of the most popular programming languages, Java boasts the largest ecosystem in the world [21]. To leverage the benefits of this extensive ecosystem, several transpilers have been developed to automatically convert Java code to JavaScript (JS) [2, 11, 15]. However, due to the fundamental differences in memory and concurrency models between Java and JS [7], significant challenges remain in Java-JS transpiler.

The basic concurrency components in Java and JS are Thread and Webworker [6], respectively. There are two primary challenges caused by differences between the concurrency models of threads and workers. The first **challenge**❶ is that each thread in Java shares memory, whereas each worker in JS communicates through message passing. Although modern JS supports `SharedArrayBuffer` [3] to share binary data between workers, recovering complicated JS object from a binary buffer remains challenging. The second **challenge**❷ concerns maintaining Java's synchronized execution semantics in JS's asynchronous execution environment. In Java, the execution flow can easily be paused to wait for signals, whereas in JS, this is impossible unless the flow is labeled as async. Unfortunately, the async label complicates the subsequent control flow and significantly increases the implementation complexity of Java-JS transpiler.

To tackle the challenges caused by the aforementioned differences, we propose our novel runtime, JCRuntime, and implement the corresponding transpiler, CCTrans, for concurrent Java. In JCRuntime, each JS worker maintains a copy of shared objects (SOs). Consequently, JCRuntime creates *get* and *set* proxies for all SOs, updating the latest values for each copy via messages whenever a *set* or *get* operation occurs. To simulate Java's synchronized mechanism, JCRuntime provides a series of blocked communication functions to update the value of SOs without requiring JS async annotations. To evaluate CCTrans, we manually construct a concurrent Java dataset and conduct experiments on it. Early experimental results indicate that our tool can successfully transpile concurrent Java to JS while preserving the same behaviors.

This paper is divided into four main sections. In Section 2, we provide background on the main concurrency features of Java and JS, including some examples of related work. In Section 3, we detail the design of JCRuntime and CCTrans, explaining their implementation and the limitations of our design. This section also presents our experimental results and discusses some interesting findings. In Section 4, we present the main conclusions and future work drawn from implementing and evaluating our approach.

## 2 BACKGROUND AND RELATED WORK

This section briefly describe the background for this paper, covering the basic concurrent features of Java and JS. After that, we introduce some related work on the translation of concurrency models and the research topic of automatic translation from Java to JS.

[*]Also with Suzhou Institute for Advanced Study University of Science and Technology of China.
[†]Yinxing Xue is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*Conference'17, July 2017, Washington, DC, USA*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
 1 class Bar {
 2   public static Object
 3     lock = new Object();
 4 }
 5 class Foo extends Thread {
 6   public void run() {
 7     synchronized(Bar.lock)
 8     {
 9       doSomething();
10       Bar.lock.wait();
11       doSomething();
12       Bar.lock.notify();
13     }
14   }
15 }
```

(a) Concurrent Java

```
 1 // In main executor
 2 const worker =
 3   new Worker("worker.js");
 4 worker.postMessage("foo");
 5 onmessage = (e) => {
 6   console.log("received");
 7 };
 8
 9
10
11 // In worker.js
12 onmessage = (e) => {
13   console.log("received");
14   postMessage("bar");
15 };
```

(b) Concurrent JavaScript

Figure 1: Concurrent Features

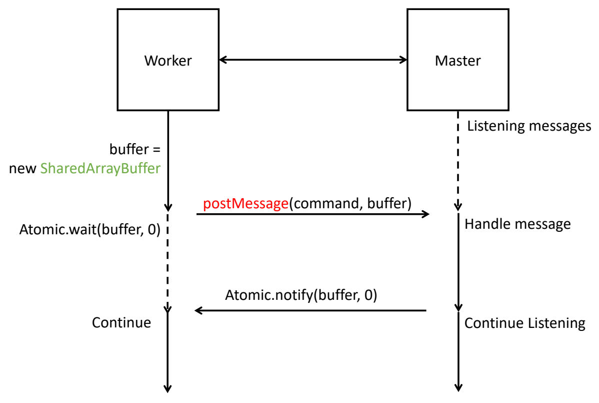

Figure 2: Concurrency Model in JCRuntime

## 2.1 Concurrency Features

As shown in Figure 1a, class Bar has a static object lock, and class Foo is a subclass of Thread that can create an instance executing as a thread. In Java, the primitive keyword *synchronized* is used to obtain a locker in every object. At line 7, each Foo instance uses *synchronized* to obtain the locker in the object Bar.lock. If an instance successfully acquires the lock, it executes the code of synchronized body from lines 9 to 12. Otherwise, the thread is blocked at line 7 until the lock is available. At line 10, the thread instance calls method *wait* to release the lock and switch its state to waiting until it is notified by another thread. At line 12, the thread instance calls notify to wake up a waiting thread, allowing it to attempt to acquire the lock and continue execution.

For JS, the only way its workers can communicate is via the asynchronous two-way communication channel that allows either worker to send messages to other. These messages are processed using a registered callback (onmessage). In Figure 1b, the JS main executor creates a worker and sends a message at line 4. Consequently, lines 5-7 show the registration of a listener to receive all messages from other workers. In the worker.js file, the worker also registers a listener to receive messages and sends a message back to the main executor using postMessage at line 14.

In summary, instances of Foo share the same memory space and can freely access any in-scope variables. Furthermore, Java threads switch their execution status between three states: blocked, waiting, and running. In contrast, workers in JavaScript do not share memory and only communicate through postMessage, which serializes and deserializes objects for messaging. Due to the specifications of JS, the listener will only be executed when the worker is in an idle state, that is, no more code to run.

## 2.2 Related Work

Over the past years, many researchers have focused on translating concurrent programs. Especially in the area of distributed systems, numerous works have attempted to convert shared memory systems to message-passing systems in C++. Attiya et al.[9] proposed an approach to simulate single-writer multiple-reader shared memory programs in a message-passing system. Davidson et al.[14] devised a dynamic analysis approach to identify concurrency semantics and translate them into a message-passing model. Additionally, there is extensive work based on Regular Section Descriptors to identify shared memory [12, 13].

However, few researchers have focused on the translation of concurrency code between different languages, such as Java and JS. JSweet [2] is a transpiler that converts Java to JS, whereas it ignores concurrency features. There are many other techniques aimed at synthesizing JS from Java bytecode [4, 15]. Among them, Leopoldseder et al. present a state-of-the-art approach to cross-compile Java bytecode to JS, but it still has limitations in supporting multithreading and synchronous APIs. The most related work is Doppio [17], which simulates blocking using asynchronous JS APIs and multithreading within a single main worker, as it lacked access to the modern Atomics API at the time. Any language implemented using Doppio must satisfy two properties to adopt its event segmentation. Additionally, there are some other JavaScript transpilation efforts, such as LLVM to JS [20], Racket to JS [19], and OCaml to JS [18].

## 3 CCTRANS

In this section, we begin by introducing the implementation of CCTrans. Then, we discuss the limitations of current design. After that, we present the early experimental results to show the effectiveness of our tool.

## 3.1 Implementation

Since Java is a complex industrial-level programming language, crafting a Java-JS transpiler from scratch is challenging. We decided to extend JSweet [2], a Java to JS transpiler without supporting concurrent features, to demonstrate our design. In JSweet, each Java file is translated to a TypeScript [5] file and subsequently converted to pure JS by the TypeScript Compiler (TSC).

**Concurrency Model.** Figure 2 depicts the concurrency model of JCRuntime. To simulate the behaviors of shared memory, the master worker maintains the values of all shared objects (SOs) as a data center. When a *get* operation of SO occurs, the worker changes its state to sleep and sends a data query to the master to fetch the latest value. The worker wakes and continues execution once the

**Table 1: The number of messages. (CCTrans$_{unop}$ indicates the transpiler without optimization, while CCTrans$_{op}$ is the optimized transpiler.** ***average$_{op}$*** **and** ***average$_{unop}$*** **represent the average number of messages with and without optimization, respectively.)**

| | CCTrans$_{unop}$ | | | | | CCTrans$_{op}$ | | | | | *average$_{unop}$* | *average$_{op}$* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |
| 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 |

required value is fetched. The reason for workers to actively fetch a value lies in the fact that JS lacks preemption; Once an event begins execution, it will continue uninterrupted until it either finishes or is terminated by the browser. If value updates rely on passive message pushing, workers can read outdated and dirty values. Imagine this scenario: when a worker Foo is executing a long loop in a function, a variable bar's value is updated by other workers from 1 to 2, and the update message is pushed to worker Foo. Due to the lack of preemption, Foo never triggers the message receiving handler until it finishes the currently executing function. In this function execution, Foo still reads the value of bar as 1, which is an outdated and dirty value.

On the other hand, for *set* operations of SOs, the worker sends the updated value to the master worker without going to sleep. Since the master handles received messages sequentially in a message queue without parallelism, it ensures that value fetch requests occur after the corresponding value update requests.

**Runtime.** To simulate the behaviors of Java concurrent code, we provide substituted JS classes for Java standard classes. For example, JCRuntime contains a JS class Thread which starts a new worker and waits for signals, executing the run method once a start signal is received. We leverage the modern JS feature Atomics [1] to simulate the thread synchronization mechanism. Atomics provides two significant methods, *wait* and *notify*, to change the states of workers similar to Java threads. Based on Atomics, we devise substituted functions for Java concurrent features (keywords and standard class methods). Notably, we design the *sync* function of JCRuntime with the aim to replace the synchronized keyword. After calling sync, the worker sends a message to the main worker with a signal buffer and calls *Atomics.wait* to wait for the resume signal from the signal buffer. Furthermore, we assign each SO a unique ID across different workers to identify the copies' values. Consequently, the master maintains four queues with the IDs of workers and SOs to record blocked workers, lock holders, waiting lock workers, and conditionally waiting workers. With these queues, JCRuntime can simulate the status-switch behavior of Java threads in JS workers.

JCRuntime goes beyond basic implementations by leveraging the Java Memory Model (JMM) [8] to reduce the number of messages. In summary, the JMM is a weak memory model, which means that if there are no data races under Sequential Consistency (SC) [10], we can assume SC when reasoning about our program. Therefore, the optimized JCRuntime only gets and sets values for the master when obtaining and releasing a lock or when the value is marked as volatile.

**Transpiler.** With JCRuntime, CCTrans transpiles concurrent Java in two primary stages. In the first stage, CCTrans identifies which objects are SOs. We utilize the symbol table and consider all public fields of Java threads as potential SOs. Consequently, CCTrans inserts proxy function calls for the classes containing SOs to hijack the *get* and *set* operations. In the second stage, CCTrans translates the concurrent keyword and standard library methods. Notably, the *synchronized* keyword is translated to JCRuntime functions *sync* and *unsync*. Since a thread releases the obtained lock after its control flow exits the Java synchronized block, CCTrans must explicitly insert an unsync function call to release the obtained lock before each control flow exit point. For example, the return statement and exceptions can directly exit the synchronized block while releasing the lock. Therefore, CCTrans analyzes and produces the control flow graph of the code, and inserts the unsync function call into the exit basic block to release the lock.

### 3.2 Limitation

We identify two primary limitations of our JCRuntime and CCTrans. Regarding JCRuntime, it currently struggles to manage complex objects within our simulation framework. Complex objects often contain intricate references to other objects or self-references, which renders them unserializable through message passing. Addressing this challenge goes beyond the scope of this paper. As for translation, its insertion of *unsync* calls during inter-procedural control flow analysis is not unsound. There are instances where the control flow terminates prematurely, leading to locks not being released from deeper levels of the call stack. In contrast, Java's release operation is handled at the bytecode level, ensuring that all control flow details are meticulously captured - a level of comprehensiveness that our current approach does not achieve.

### 3.3 Evaluation

We construct 21 Java files with concurrent features as our dataset and conduct experiments to evaluate our tool's effectiveness. Each experiment is repeated five times, and the passed messages are recorded in the master worker, as it acts as the data center. For the compiled JS files, we set up a running environment in Chrome version 125.0.6422.176 (Official Build) (x86_64). We manually check the compilation correctness by comparing the output results with those from OpenJDK 22.0.1.

The experimental results show that we successfully compile all the concurrent Java files while preserving their behaviors. Furthermore, the optimized transpiler (CCTrans$_{op}$) produces better JS code that performs with fewer passing messages. Specifically, for the NoSyncClass, CCTrans$_{op}$ sends only 2 messages, while CCTrans$_{unop}$ produces 40,008 messages. Since there are no synchronized keywords or volatile variables in the NoSyncClass file, the workers in the optimized JS never fetches or updates local memory to the master. In contrast, since the SO is annotated as volatile, each operation has to update the results to the master via messages, resulting in a similar number of messages for both CCTrans$_{op}$ and CCTrans$_{unop}$.

## 4 CONCLUSION AND FUTURE WORK

Several Java transpilers aim to leverage the Java ecosystem to JS. Current solutions partially address this problem but often circumvent significant performance-related features such as concurrency. Since JS lacks a concurrency infrastructure, translating two different concurrent models remains challenges.

In this paper, we attempt to provide the same multithreading semantics of Java for JS, eliminating the significant differences between their concurrency models. Specifically, we propose a novel concurrency model for JS and implement it to simulate the thread status and synchronous APIs of Java. Furthermore, we reduce the total number of passed messages based on the Java Memory Model. Finally, we implement a transpiler with our runtime and conduct early experiments on it. The experimental results show that our tool can successfully transpile the Java code in our dataset and that the optimization effectively reduces the number of passing messages.

In the future, we will leverage large language models [16] to enhance our transpiler and improve the precision of inserting the unsync function calls. Additionally, we aim to tackle the challenge of serializing and deserializing objects with references by utilizing the rewriting capabilities of large language models. After that, we will conduct large-scale experiments to transpile popular concurrent Java libraries to validate the effectiveness of our tool.

## REFERENCES

[1] 2023. Atomics. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Atomics

[2] 2023. JSweet. https://www.jsweet.org/

[3] 2023. SharedArrayBuffer. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer

[4] 2023. TEA VM. http://teavm.org/

[5] 2023. Typescript. https://www.typescriptlang.org/

[6] 2023. Webworker. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

[7] Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA.

[8] David Aspinall and Jaroslav Ševčík. 2007. Java memory model examples: Good, bad and ugly. In *Proceedings of Verification and Analysis of Multi-Threaded Java-Like Programs (VAMP 2007).*

[9] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *J. ACM* 42, 1 (1995), 124–142. https://doi.org/10.1145/200836.200869

[10] Hagit Attiya and Jennifer L Welch. 1994. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)* 12, 2 (1994), 91–122.

[11] Andrés Bastidas Fuertes, María Pérez, and Jaime Meza Hormaza. 2023. Transpilers: A Systematic Mapping Review of Their Usage in Research and Industry. *Applied Sciences* 13, 6 (2023). https://doi.org/10.3390/app13063667

[12] Ayon Basumallik and Rudolf Eigenmann. 2005. Towards automatic translation of OpenMP to MPI. In *Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, Cambridge, Massachusetts, USA, June 20-22, 2005*, Arvind and Larry Rudolph (Eds.). ACM, 189–198. https://doi.org/10.1145/1088149.1088174

[13] Okwan Kwon, Fahed Jubair, Seung-Jai Min, Hansang Bae, Rudolf Eigenmann, and Samuel P Midkiff. 2013. Automatic scaling of OpenMP beyond shared memory. In *Languages and Compilers for Parallel Computing: 24th International Workshop, LCPC 2011, Fort Collins, CO, USA, September 8-10, 2011. Revised Selected Papers 24.* Springer, 1–15.

[14] Hsien-Hsin Lee and Edward S Davidson. 1995. *Automatic Parallel Program Conversion from Shared-Memory to Message-Passing.* University of Michigan, Computer Science and Engineering Division ....

[15] David Leopoldseder, Lukas Stadler, Christian Wimmer, and Hanspeter Mössenböck. 2015. Java-to-JavaScript translation via structured control flow reconstruction of compiler IR. In *Proceedings of the 11th Symposium on Dynamic Languages* (Pittsburgh, PA, USA) *(DLS 2015).* Association for Computing Machinery, New York, NY, USA, 91–103. https://doi.org/10.1145/2816707.2816715

[16] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24).* Association for Computing Machinery, New York, NY, USA, Article 82, 13 pages. https://doi.org/10.1145/3597503.3639226

[17] John Vilk and Emery D. Berger. 2014. Doppio: breaking the browser language barrier. *SIGPLAN Not.* 49, 6 (jun 2014), 508–518. https://doi.org/10.1145/2666356.2594293

[18] Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience* 44, 8 (2014), 951–972.

[19] Danny Yoo and Shriram Krishnamurthi. 2013. Whalesong: running racket in the browser. In *Proceedings of the 9th Symposium on Dynamic Languages* (Indianapolis, Indiana, USA) *(DLS '13).* Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/2508168.2508172

[20] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (Portland, Oregon, USA) *(OOPSLA '11).* Association for Computing Machinery, New York, NY, USA, 301–312. https://doi.org/10.1145/2048147.2048224

[21] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. 2023. Software Composition Analysis for Vulnerability Detection: An Empirical Study on Java Projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023).* Association for Computing Machinery, New York, NY, USA, 960–972. https://doi.org/10.1145/3611643.3616299