CS703 Project Report

CriSPE: Critical Section Pre-Execution

Min Xu and Vinod Ganapathy
{xu, vg} @cs.wisc.edu

Abstract

Traditional methods for concurrency control use spin locks to guard access to critical sections. This can come at significant loss of performance, since only one thread can exist in the critical section at a given point in time. In this paper, we explore the possibilty of allowing for multiple threads to co-exist in the critical section at the same time. If a problem is detected, then we roll back to the beginning of the critical section and restart. Our preliminary results on some microbenchmarks are promising.

1 Introduction

One of Java's strong points is that it allows for multithreaded applications at the language level. Multithreaded programs give rise to race conditions when two threads of the program try to modify shared data. Shared objects are typically allocated in the heap, and when two or more threads try to gain access to the same object in the heap, some kind of synchronization is essential to control their access to the object. Synchronization is implemented in Java using the keyword `synchronized`. By declaring a method `synchronized`, the programmer can prevent multiple threads from invoking the method on the same object.

2 The Problem

In commercial applications, which tend to be highly multithreaded, there is bound to be contention amongst threads that try to modify shared data. One of the threads that contests for the lock wins, while the others wait for the "winner" thread to move out of the critical section before they can gain entry. As a result the threads are serialized, and hence the number of threads processed per unit time decreases, which has an impact on the throughput of the system.

One way of improving system throughput is to use a multiprocessor system. Here the threads can be scheduled to different processors as a result of which the execution times of the threads overlap, thus on the whole the throughput increases. However, even in this case, two threads which want to modify a shared data structure will contend for the critical section.

The most commonly used primitive for achieving synchronization is through the previously explained `synchronized` clause. To ensure the program correctness, common programmers use conservative locking. That means, the programmers simply put a lock on a big global data structure. So that all accesses to that data structure are synchronized and safe. Clearly, this ensures program correctness, but comes at the cost of performance. This causes considerable *unnecessary* synchronization if the critical section is read intensive, or consists mainly of computation that cannot be affected by the presence of other threads in the critical section.

Our idea is to speculatively execute the critical section. This means that if a thread seeks entry into the critical section, we allow it to do so irrespective of whether another thread was present or not. However, speculative execution could cause the threads to change memory in such a manner as would be irreproducible in an execution in which the threads went one after another in the critical section. Hence we also add the provision of rollback if a situation like this were to occur. Note that our concern here is not to boost the performance of a single thread of the application, it is to boost the throughput and achieve processing of more threads per unit time. Boosting performance of a single thread is a step in that direction, however our experiements are focused in getting the entire system to process more, even if it means slight degradation in performance of individual threads.

3 Target Workload

TPC-W is a transactional web e-commerce benchmark, which simulates the activities of a online book store. This workload is characterized by multiple on-line browser sessions, which will lead to multiple threads of execution. Transaction integrity has to be maintained, and hence accesses to the shared database tend to be controlled by synchronization. The performance of TPC-W is measured by the number of web interactions processed per second, which is the throughput of the workload. We used this workload in the initial phase of our experiments.

The IBM Jikes RVM is our research infrastructure. We modified and built it on a 18-way PowerPC server running AIX 4.3. To collect information about thread contention, we added profiling functionalities to the virtual machine to monitor the lock acquiring and releasing events that happen in the application or the RVM itself.

In our experiments, we used Tomcat, which is a lightweight web server, servlet container and Java Server Pages container. This was driven by emulated clients from TPC-W building kits from Mikko's group. Tomcat forwards these web requests to a servlet implementation of an online book store. Then, those servlets process the requests by talking with a database, PostgreSQL in our case, using JDBC. Tomcat is written in Java; Tomcat, and the TPC-W workload running on top of it are the subjects of of our investigation. The setup of the system is described in Figure 1.

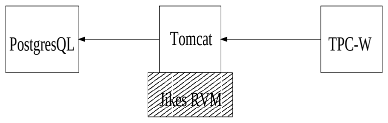


Figure 1: Experimental Setup

To collect performance data, we used the emulated clients to drive the system in the steady state for about 100 seconds. We then repeated the experiments 10 times for each configuration. Since we are using an 18-way multiprocessors system, to avoid contention during lack of system resources, we drove TPC-W with only 18 users. Thus, if a thread is put on sleep during acquiring the lock, it should be able to grab the lock as soon as the lock is freed. Spare CPUs should always to available for a ready thread. In other words, the system is operating in a lightly loaded mode, and not all CPUs are working all the time. Thread execution is mostly bounded by IO or lock contention, if any.

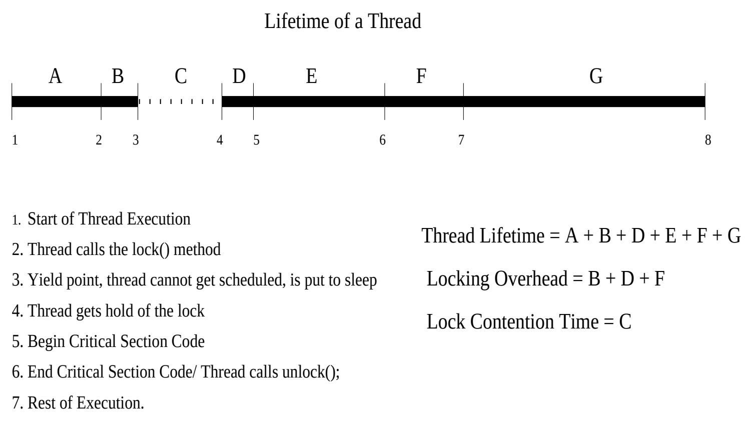
4 Limit Study

In this section, we performed a limit study on the impact of reducing lock contention. Using TPC-W as an example, we found potential for substantial performance gain.

4.1 Locks: overhead or contention?

Figure 2 shows the lifetime of a thread. Note that in this case we have shown the thread acquiring only *one* lock; and the thread is placed on the lock's entering queue to sleep. It successfully acquires the lock right after waking up. However, in general a thread can acquire more than one lock, or may acquire the same lock more than once, or it could be put on sleep more than once. This figure just serves to illustrate the metrics we have used in gathering profiling information.

In this figure, the total life time a thread is broken into three parts. The thread execution time consists of all CPU time this thread has spent. The locking overhead time consists of the CPU times



Lifetime of a Thread
A
B
C
D
E
F
G
1
2
3
4
5
6
7
8
1. Start of Thread Execution
2. Thread calls the lock() method
3. Yield point, thread cannot get scheduled, is put to sleep
4. Thread gets hold of the lock
5. Begin Critical Section Code
6. End Critical Section Code/ Thread calls unlock();
7. Rest of Execution.
Thread Lifetime = A + B + D + E + F + G
Locking Overhead = B + D + F
Lock Contention Time = C

Figure 2: Lifetime of a Thread

used to acquire and release the lock, not including thread sleeping time. Finally, the lock contention time simply counts the wall clock time spent inside the lock acquiring function when the thread is sleeping.

	% Execution	% Overhead	% Wait
With Profiling	22.51	1.13	76.36
No Profiling	21.38	1.20	77.42

In table 4.1, the total time breakdown of all threads from 10 runs is presented. The three columns are the three times presented above. Clearly, even in a lightly loaded system, i.e. 18 users using 18 CPUs, the lock contention is extremely high. It is also interesting that locking overhead is very low, since RVM implements a lightweight Java locking scheme called thin locks[9].

However, while collecting these data, we were also doing some other profiling. We were worried that other profiling taskes may alter the time breakdown. The second row of the table presents the time breakdown without the relatively more expensive profiling turned on. It shows a similar trend.

In conclusion, the results in the time breakdown show that lock contention is an important problem for commercial workloads.

4.2 Number of Owners of a lock

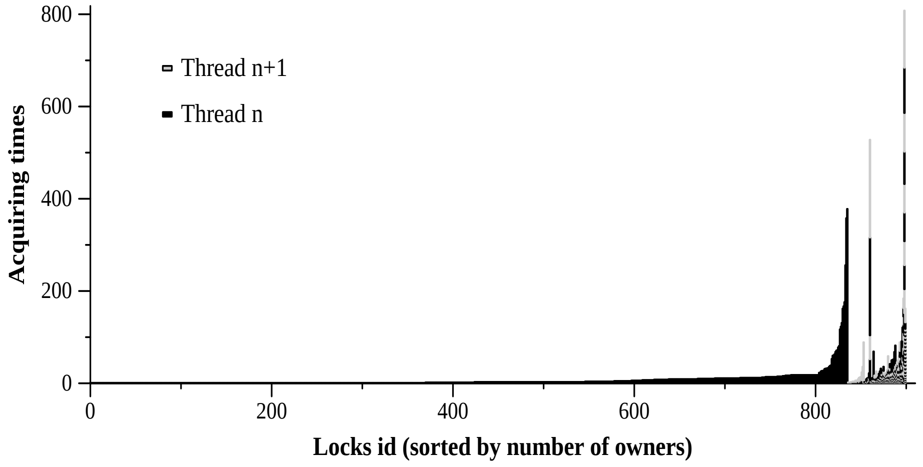


Figure 3: Breakdown of number of threads obtaining a lock, and showing the number of times each thread obtains a lock

Figure 3 shows on the X-axis locked objects, sorted by the number of threads that acquire the lock on them. The Y-axis denotes the number of times the thread acquires the lock on the object. We do not differentiate here between the cases where the thread acquires a nested lock on the object, and the case where the thread repeatedly locks and unlocks the same object. It can be seen that there are about 850 objects that are locked only by one thread during their lifetime. This graph serves to illustrate the potential for contention. The larger the number of threads acquiring a lock on an object, the larger the potential for contention. Only two colors are used to denote all different threads. The area of the first solid black spike is basically the overhead paid to lock unnecessarily. These locks are only locked by one thread. The remaining part of the graph, shows that one lock could be accquired by multiple threads for upto 800 times.

To gain a closer look at lock contention, we sampled the lock entering queue length and obtained the distribution in figure 4. In this figure, the left most bar shows that 11590 times when a thread is trying to acquire a lock, no other thread contends with it. And for about 200 times, two threads are racing for a lock, and so forth. This result shows that, on average, 2-10 threads are contending for a lock.

A very important implication to us is that a more relaxed concurency control algorithm could be very beneficial to the system throughput even in the if the latency of the critical section is slightly more.

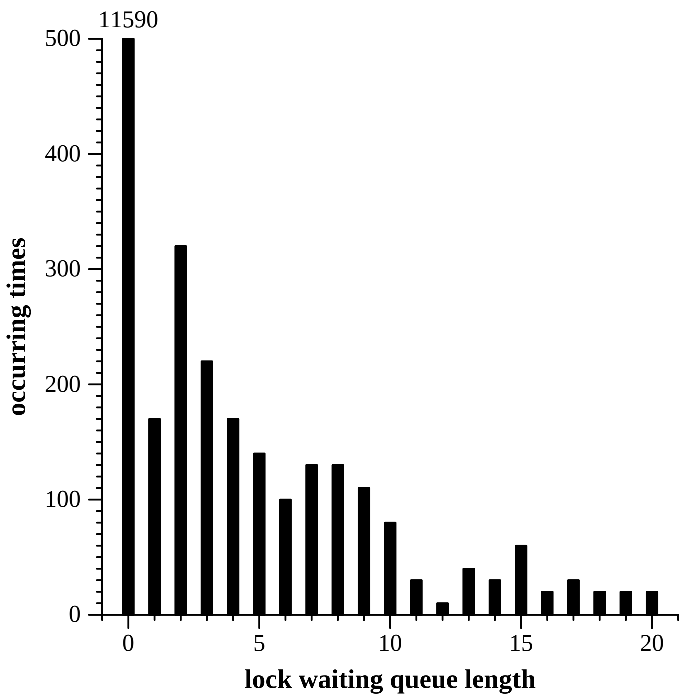


Figure 4: Entry queue length distribution

4.3 Program's hot locks

Having identified that locking contention is an important bottleneck in commercial workloads, such as TPC-W, we gatherred information on what those hot locks were:

In table 4.3, top 10 congested locks are presented. We also included the calling context. Clearly, the *getConnection* and *returnConnection* methods are the main bottlenecks to the system. Ironically, these methods are created as an optimization. The system maintains a central JDBC connection pool, and the overhead of openning an new JDBC connection can be avoided if using the connection pool. However, as indicated in the table, the pool itself becomes the main bottleneck.

Time	Locked Object	Calling Method	Calling Method's Calling Method
330.3	java.lang.Class	TPCW_Database.returnConnection	TPCW_Database.getRelated
295.9	java.lang.Class	TPCW_Database.getConnection	TPCW_Database.getRelated
70.98	java.lang.Class	TPCW_Database.getConnection	TPCW_Database.getBook
64.31	java.lang.Class	TPCW_Database.returnConnection	TPCW_Database.getBestSellers
41.01	java.lang.Class	TPCW_Database.returnConnection	TPCW_Database.getBook
26.08	java.lang.Class	TPCW_Database.getConnection	TPCW_Database.getName
22.33	java.lang.Object	VM_DynamicLinker$DL_Helper.compileMethod	VM_DynamicLinker.lazyMethodInvoker
14.50	java.lang.Class	TPCW_Database.getConnection	TPCW_Database.getNewProducts
14.22	java.lang.Class	TPCW_Database.returnConnection	TPCW_Database.getName
13.86	java.lang.Class	TPCW_Database.returnConnection	TPCW_Database.createEmptyCart

5 CriSPE

The previous section shows that the main bottleneck to achieving high throughput is the time that the threads spend contending for the locks. In this section, we present our algorithm for solving the problem. Our method is called CriSPE, short for Critical Section PreExecution. The rest of this section is organized as follows: section 5.1 describes the technique employed by CriSPE. We illustrate this technique using a small example in section 5.2. Section 5.3 discusses how we implemented the system in the Jikes Research Virtual Machine. We then give a proof in section 5.4 that our system works correctly. Section 5.5 gives results of our experiments with CriSPE. Finally, in section 5.6 we discuss the limitations as they exist in the current implementation of CriSPE.

5.1 CriSPE Overview

The purpose of the locks around a critical section is to serialize accesses to shared variables that occur within the critical section. To achieve high performance, we allow many threads to co-exist in the critical section. However, we must still preserve the serializability of the resulting schedule.

CriSPE attempts to make threads execute concurrently even in critical sections. The way this is achieved is by speculatively pre-executing the threads in the critical section, ignoring the locks that guard the critical section. No outputs from the critical section are committed until speculative execution is confirmed to be correct. Otherwise, a thread has to abandon all speculative outputs and start over. We confirm the speculative execution by double checking all inputs to the critical section at the commit time. If all the input values match the values that were originally read, we can be sure that speculative execution of this thread did not interfere with the execution of other threads.

We note that there is a problem with the simple scheme as discussed above, similar to the output-commit problem in fault tolerant literature: not all outputs from the critical section can always be buffered. Depending on what the speculative execution boundary is, one might not be able to defer sys-calls which can not be undone without help from outside the system. We will discuss this in more detail in section 5.6.

The flow of control in CriSPE is summarized in figure 5. The system can be thought of as accepting a critical section as input, and applying the transformations shown to the critical section. The output from CriSPE can be used in place of the critical section. Note that because of the speculative nature of CriSPE, the transformed critical section can co-exist in the system, and concurrent execution of CriSPE and non-CriSPE versions of the critical section is allowed.

The first step is to remove the locks that guard the critical section. The locks are not only removed at the top level locks but also at all levels. When threads execute concurrently in the critical section, all the concurrently executing threads have access to the shared variables, each of which could be potentially modified by any of these threads. In other words, we lose *serializability*. A schedule of thread execution is said to be *serializable* if there exists an equivalent schedule in which only a single thread has access to the critical section at any point in time, and the two schedules leave the memory

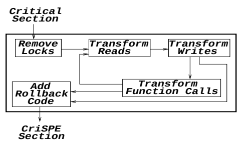


Figure 5: The CriSPE Transformation

in the same state on completion.

CriSPE ensures serializablity by making each thread buffer its writes to a local table, and allowing the thread to commit its writes to architectural state only after ensuring that serializability is not lost as a result of the operations performed by the thread. To do so, CriSPE employs steps 2, 3, 4 and 5 of the transformation shown in figure 5. Every critical section can be thought of as a collection of reads and writes. We are interested only in the reads and the writes of the shared variables. Step 2 of the transformation instructs each thread executing in the modified critical section to buffer its reads. In other words, when the thread reads its values from shared variables, it remember the values that it read from shared variables into a local table as the *read values*. The table consists of the variable name read, and the value associated with that variable. Only the initially read value is recorded in the table for each distinct shared variable.

Step 3 of the transformation modifies the writes in the critical section. The thread is instructed to write to the local table instead of global shared variables. The value is stored in the local table as the *written values*. Future reads of this variable will get their values from these *written values*. Only the last written value is recorded. These transformations ensure that the execution of the thread is effectively *sand-boxed* by the local table, and any changes made by this thread to the shared variables are not visible to other threads.

Step 4 repeats steps 2 and 3 for each of the methods called within the critical section. However, care must be taken while doing this transformation: this method may be called by other parts of the code as well, which do not fall within the critical section under consideration. Thus, the original copy of the function must be left as is for access by other parts of the code. Instead a copy of this function must be made, and the transformation defined by steps 2 and 3 must be applied to this copy. This transformation must also make sure that the calls inside the critical section to this function are now converted into calls to the copy of this function (which has undergone the transformation). Note that this step is a recursive step, and this transformation must be done till we encounter a function that calls no more functions.

Upon reaching the end of the critical section, the thread must check if the values that it read, which are saved in the thread local table match with the values that exist in memory. If the values match, then it means that the architectural state was not corrupted by other threads while this thread was in execution, and hence this thread can commit its values to architectural state without losing serializability. If the values did not match, it means that committing the values of this thread to architectural state can cause serializability to be lost. Hence, we discard the current execution of the thread by flushing the local tables of this thread, and restart execution of the thread at the beginning of the critical section. The entire operation of checking with memory, and committing must be done withing a critical section. The whole operation is captured in step 5 of our transformation.

Section 5.4 gives a detailed proof that the transformation produced by CriSPE always produces a serializable schedule.

5.2 An Example

We demonstrate the transformation carried out by CriSPE on a small example shown below. We note that it is difficult to show the transformation at the source code level, since the transformations are carried out at the bytecode level as shown in detail in section 5.3. Hence, we first present the source code, and then give a rough indication of how it would look like in bytecode. We then show the transformation that we apply on the bytecode.

```
class Obj{
  public static int counter;
  public static void main(String args[]){
    Obj o = new Obj();
    o.start();
  }
  public static void run(){
    o.update();
  }
  public synchronized void update(){
    counter = counter + 1;
    decrement_counter();
  }
  private void decrement_counter(){
    counter = counter - 1;
  }
}
```

The important points to be noted here are that the synchronized `update` function accesses and modifies a shared variable. Moreover the function `decrement_counter` called from the `update` function is not synchronized.

CriSPE works as follows:

1. It first removes the synchronization from the `update` function.
2. Next it finds all the functions that are called within the update function (in this case `decrement_counter`, creates a copy of these functions, and transforms these functions so as to buffer their reads and writes, as explained in the next two steps. This is a recursive procedure and will terminate when we encounter the last leaf procedure.
3. All read instructions are converted into a function call. This function reads the value, and also stores the value read in a buffer which is local to the calling thread.
4. All write instructions are converted into function calls in a similar fashion.

The original program and the program after the transformation broken down into bytecode are shown in the table. We have not written the actual bytecode here for ease of explanation, the actual bytecode and the issues involved in the translation of some classes of bytecodes are in the section 5.3

```
public synchronized update(){
  read counter
  add counter
  write counter
  call decrement_counter()
}

public decrement_counter(){
  read counter
  sub counter
  write counter
}
```

```
public update(){
  label: crispe_read(counter)
   add counter
   crispe_write(counter)
  call crispe_decrement_counter()
   if (crispe_commit())
    copy buffers to memory and return
   else
    flush buffers and start at <label>
 }

 public decrement_counter(){
  read counter
  sub counter
  write counter
}

public crispe_decrement_counter(){
  call crispe_read(counter)
  add counter
  call crispe_write(counter)
}

static int crispe_read(Object i){
  read i's value (from memory or local buffer)
  buffer i's value
  return i's value
}

static void crispe_write(Object i, int value){
  write i's value to local buffer
}

static boolean crispe_commit(){
  if (values in buffer=values in memory)
    return true
  else
    return false
}
```

5.3 Implementation Details

Dynamically Modifying the Java Class structure

The first thing we encountered in implementing CriSPE was to dynamically change a given Java class's structure inside the RVM. To do this, we implemented two new functions:

1. VM_Method.createCrispeVersion(): This function modifies the top level synchronized method to make it a CriSPE method as explained in section 5.1.

2. VM_Method.duplicateCrispeVersion(): This function makes a clone of the given method and the new function name has prefix "cripse_".

As mentioned briefly before, the top level of the CriSPE function is treated differently than the rest of the callee functions that are called from the top level function. Both of these functions undergo a bytecode tranformation in which read and writes are converted to calls to buffering functions, and function calls are translated into calls to the corresponding CriSPE version. However, the code to commit the critical section and roll back if an inconsistency is discovered, has to be inserted only in the top level. The top level function is not cloned; instead it is modified so that subsequent calls to the top level function use the new versions.

However, blindly adding a new method to a class will not work. One has to keep the internal RVM data structures consistent. Upon adding a new method to a class, we had to update the function dispatch table of the class and update the global method lookup dictionary. Also, the class's constant pool had to be updated in order to call the new CriSPE version functions.

Bytecode Transformation

As mentioned in an earlier section, we buffer all the values that we read from memory, and write to a thread local table instead of writing to memory. The way this is achieved is by intercepting the bytecode stream, and modifying it so that the desired functionality is achieved.

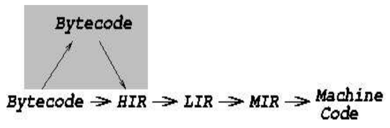


Figure 6: Transformation in Jikes

Figure 6 shows the way a method is compiled by Jikes. The Bytecode is converted to HIR, which then gets converted to LIR. LIR is converted to MIR, which is then compiled into machine code. At each stage, some optimization is done. The transformation employed by CriSPE fits in as shown by the darkened portion of the figure. We intercept the bytecode of the method, rewrite it as described in this section, and feed it to the module that converts it to HIR. The rest of the transformations done by Jikes are left untouched.

A study of the Java Bytecode reveals that variables local to a method and global variables are handled differently. We are not interested in the reads and writes of local variables since they are not visible to other threads executing in the critical section, and hence buffering the reads and writes of these variables is not necessary. Global variables are read using the `getstatic` instruction, while they are written into using the `putstatic` instruction. The two bytes following a `getstatic` or `putstatic` instruction are used to construct an index into the constant pool of the class to which this thread belongs. The value at this index of the constant pool contains a reference to the desired variable. To buffer the value of this variable, CriSPE first removes the `getstatic` or `putstatic` instruction. In its place, it inserts a call to a function, using an `invokestatic` instruction, which not only reads the value, but also writes the value to a local table.

Handling Array reads and writes

Arrays need to be handled differently. Consider the following example

```
class Obj{
   int[] A = new int[10];
   ...
   method1(){
      int[] B = A;
      read B[1]
    }
    ...
}
```

Array A is first read into a local variable, then an `iaload` instruction is used to access it. Not like other local variables, array local variables can be aliasing with global variable. We conservatively handled all local array access speculative by buffering their accesses. Strictly speaking it is not necessary for pure local arrays. Handling array more precisely is considered to be one of the future work.

It turns out that there are 8 instructions each to handle loads an stores to different types of arrays. We handled each of these separately. The load and store instructions we converted to calls in a fashion similar to the way `getstatic` and `putstatic` are handled.

Intercepting Function Calls

Calls to functions need to be intercepted and modified as well. This is because the functions that are invoked within the critical section may tend to modify shared variables. Hence care must be taken to ensure that the reads and writes performed within these functions are buffered as well. There are 4 bytecode instructions pertaining to calls `invokestatic`, `invokevirtual`, `invokeinterface` and `invokespecial`.

To handle calls, we need an extra pass over the bytecode stream. The first pass scans the bytecode to determine if there are any calls to functions within the bytecode stream, and gathers the names of all those functions are called within the stream. These methods then undergo the CriSPE transformation, following which calls to these functions in the critical section are replaced with calls to the transformed functions. Observe that this procedure cannot be done in one pass because if we were to replace the calls to the functions with calls to the transformed functions, and were to transform the functions at a later point in time, then during compilation of the code for the critical section, the virtual machine will be unable to find the transformed methods since they would still not have been produced.

Offset Reconciliation for Branches

All of the above mentioned transformations replace an instruction with a set of instructions. These transformations potentially change the length of the bytecode stream. The way jumps are handled in bytecode is by using offsets. In particular, when we encounter a `goto` or `if` instruction, then the next two bytes in the stream evaluate to an integer which tells how many instructions are to be skipped before the jump target is reached. However, when we insert instructions to do the transformations for CriSPE, these offsets need no longer point to the correct jump target. Hence, a re-computation of these offsets is required. The way we do this is by maintaining a map between the old bytecode stream, and the new bytecode stream. For each instruction in the old bytecode stream, we store its location in the new bytecode stream. Along with this information, we also store the old value of the offset for each jump instruction in the old bytecode stream. This information is sufficient to compute the new offsets, and rewrite the offsets to their correct values in the new bytecode stream.

5.4 Proof

In this section, we prove the correctness of the CriSPE transformation.
Claim: The transformation induced by CriSPE always produces serializable schedules.
Proof: To prove the above claim, we use the well studied notion of a serializability graph. Suppose T_1, T_2,..., T_n are the threads participating in the critical section. The serializability graph consists of a node for each participating thread. A thread T_i is said to have a *Read After Write* conflict with a thread T_j if and only if it reads a value after T_j writes to it. Similarly T_i has a *Write After Write* conflict with T_j if and only if it writes a value after T_j writes to it, and a *Write After Read* conflict

is exists if and only if T_j writes a value after T_i reads from it. An edge is drawn in the serializability graph from T_i to T_j if and only if T_i has one of the mentioned conflicts with T_j.

A schedule is defined to be serializable if and only if there are no cycles in the serializability graph.

We shall now consider two participating threads T_1 and T_2, and show that if T_1 commits before T_2 commits, then the edges, if any, between T_1 and T_2 will be directed from T_1 to T_2. Suppose T_1 commits before T_2 commits, then we have the following:

1. There cannot exist a *Read After Write* edge from T_2 to T_1. Since T_1 commits before T_2 does, and CriSPE ensures that the writes performed by T_2 are not visible to T_1, T_1 can never read a value that was previously written into by T_2.

2. There cannot exist a *Write After Write* edge from T_2 to T_1 since T_2 performs all its writes at commit time, by which time T_1 has already committed, and written its values to memory.

3. *Write After Read* edges from T_2 to T_1: Suppose T_2 writes to a variable A after T_1 reads from it. Since we are given that T_1 commits before T_2 commits, we are sure that the write to A did not corrupt architectural state (otherwise T_1 would have rolled back according to the CriSPE protocol). Hence the write performed by T_2 takes place only after T_1 commits, in which case there is no dependency between T_1 and T_2 at all.

Thus, if T_1 commits before T_2, then any edges between T_1 and T_2 are directed from T_1 to T_2. Since we ensure that exactly one thread commits at a given point in time, this property ensures that the serializability graph is cycle free, which proves the claim that serializability is maintained.

5.5 Results

Micro Benchmark Results

To test the potential performance gain and overhead of CriSPE, we have evaluated our implementation on a micro benchmark. In this micro benchmark we simulate a coarse-grained lock behavior commonly seen in hash table update. A global array is locked on each entry update. Twenty threads are started for updating the array. In the critical section, only two global variable accesses are performed, and 10240 local variable accesses are performed. We ran the benchmark on a 18 processor PowerPC multiprocessor system with 1-18 processors used. Figure 7 shows the relative speedup with respect to the 1 processor normal execution.

In the figure, the lower line shows the performance results of the microbenchmark without the CriSPE transformation. The upper line shows the performance with the CriSPE transformation. Clearly, when no parallelism is available (as on a uniprocessor system), CriSPE has a high overhead, resulting in 40% slowdown. However with enough processors, CriSPE improves performance, achieving as high as 140% speedup. The speedup performance however is not very stable, which can be attributed to the other processors running on the system. The trend however is clear, and this clearly demonstrates the potential of CriSPE.

TPCW Results

We have not yet performed experiments on TPC-W with CriSPE. We will include these result in the final version of the report.

5.6 System Calls or Native Calls

System calls or native method calls require special consideration, since their effects cannot always be undone. Our solution is to abort CriSPE execution whenever we find a that critical section is about to perform a syscall or native call. Since the native call or the syscall can occur at an arbitrary level

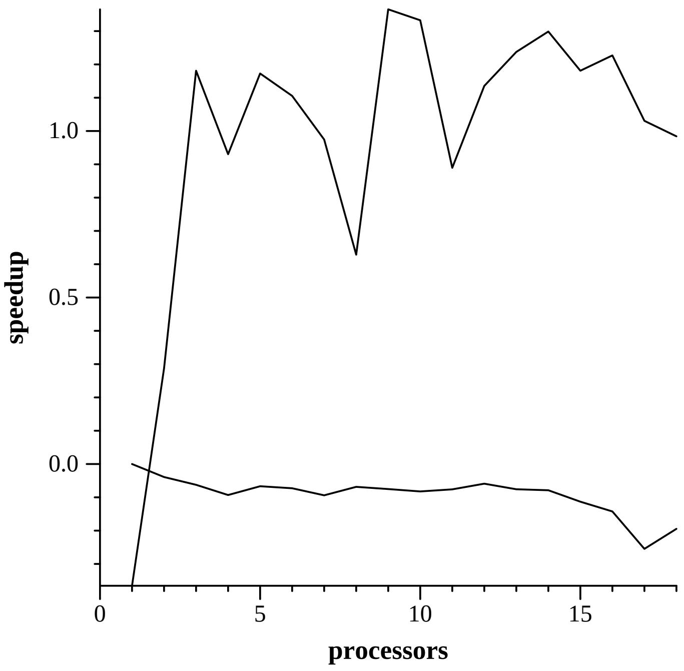


Figure 7: The CriSPE micro benchmark results.

of nesting with respect to the top level function, it is very hard to rewrite the method body to abort CriSPE execution from deep inside the call graph. Our solution to this is however rather simple: we propose to use Java's exception mechanism. During the CriSPE transformation we can insert an exeception throwing instruction before any system call. This exception can be caught at the top level function of the critical section.

Moreover, we also have to keep a copy of the original critical section code in case CriSPE aborts due to a syscall or native call. Thus, when we catch the exception caused by syscall or native call we could call the original critical section code.

The current version of CriSPE does not support a transformation for functions with system calls. We plan to implement the solution that we have proposed in this section in time for the final report. We plan to do this in time for the final report.

6 Related Work

In this section, we present a literature survey of work related to our ideas. Our ideas to make critical sections run in parallel are most closely related to some classic papers which talk about concurrency control in Database systems.

Gray et.al. [1] in their seminal paper, describe the granularity of locks, and the concept of degrees of consistency. This paper describes how placing locks at different levels of granularity in a database can affect concurrency. This paper defines various kinds of locks that a transaction in a Database system can acquire, and also the compatibilty relation between these locks. This translates in quite a straightforward fashion to programmers setting coarse and fine grained locks. Setting conservative coarse grained locks leads to poor concurrency.

Kung and Robinson describe an optimistic concurrency control algorithm in [2]. Maurice Herlihy describes his work on wait free data structures in [3]. We see CriSPE as being closely parallel to the methods discussed in these two papers. In [2] the authors allow for transactions in a database system to execute concurrently even when accessing shared data. They split up a transaction into a read phase, a validate phase, and a write phase. In the read phase, the transaction performs all its operations, and buffers the writes that it performs to the shared variables. In the validate phase, the authors use some checks to see whether committing the transaction leads to loss of serializability. If serializability could be lost, then the transaction is rolled back and restarted. Otherwise, the buffered writes are committed to memory in the write phase. However, the checks performed in the validation phase in this scheme are very conservative, and rely on the knowledge of the read and write sets of the transactions. One could imagine a similar scheme being implemented for Java, however the problem is more complicated here. Variables could be aliased, and hence one would have to look at a "lower" level than the variables, i.e., the memory locations that they actually refer to. CriSPE gets over this problem by working at this "lower" level. While committing, we check whether the local values stored in the table are the values that are present in memory. Moreover, the CriSPE transformation is not done on all the critical sections. Instead, we focus on the highly contested critical sections.

Considerable research has also been done on removing unwanted synchronization statements from Java programs [5, 6, 7, 8]. These papers propose static analysis techniques which fall under the domain called *Escape Analysis* that determine which objects are accessed by exactly one thread during their lifetime. Synchronization on such objects can safely be removed since the analysis has discovered that no more than one thread will ever access these objects. The papers report significant improvements in performance, for eg.[8] shows performance improvement of upto 36% on some benchmarks which perform frequent synchronization operations.

We must contrast CriSPE with these analyses: the analyses presented in the above papers aims at removing synchronization to achieve speedup, our goal is to improve throughput in the presence of locks by allowing more parallelism in the execution of critical sections. Moreover our analysis compromises on the performance of induvidual threads to improve the overall throughput of the system. We note that CriSPE could be made to work in conjunction with escape analysis. Escape analysis would first remove the unnecessary synchronization operations, and CriSPE would work on this transformed program. The overall throughput of the system can be expected to be better.

Another line of research that was pursued was to reduce the number of machine instructions required to obtain a lock, thereby reducing the locking overhead. These have been proposed in [9] where the notion of a *Thin* lock and *Inflated* lock are introduced. The paper observes that the synchronization provided by java is heavy-weight because of a complicated implementation of the locking scheme. The paper proposes the thin locking scheme which does a few bit manipulation operations to acquire the lock. The inflated locks are the pre-existing Java synchronization primitives. The implementation platform for this project was the Jikes RVM, and the current release of Jikes includes this locking scheme. Thus our implementation can build on the gains reported in this paper.

Finally, we compare our methods to Speculative lock elision [11]. In speculative lock elision, as in optimistic concurrency control the locks are elided, and the shared data structure is open to access by all the threads contending for it. Cache coherence is used to detect if there was a consistency problem when the threads accessed the shared data structure. CriSPE performs at the software level, and hence the flavour of the checks used to detect memory inconsistency are different.

7 Conclusions and Scope for Future work

Our implementation of CriSPE is still very preliminary. While the transformations that are carried out by CriSPE are fully automated, the profiling information that determines which critical sections are highly contested is still a manual phase. One could think of a system where the critical sections that are to undergo the transformation are automatically chosen by the virtual machine as it "learns" from the runs of the programs as to which the highly contested locks are.

We have learnt a great deal about the internals of the Java Virtual Machine during the implementation of the CriSPE system. We have demonstrated that binary rewriting is a viable technique that can be implemented in the virtual machine. By doing so the semantics of original program semantics can be without the support of an interpreter. Additionally, CriSPE has shown that speculative execution is possible in a high performance Java virtual machine. However, since the speculative execution/recovery is implemented in software, tradeoff between performance gain and runtime overhead must be carefully analyzed.

References

[1] Gray J., Lorie R., Putzolu F., and Traiger I. *Granularity of Locks and Degrees of Consistency in a Shared Data Base*: Modeling in DataBase Management Systems, 1976.

[2] Kung H. and Robinson J. *On Optimistic methods for concurrency control*: ACM TODS, June 1981.

[3] Herlihy M. *A Methodology for implementing highly concurrent data objects*: ACM TOPLAS, 1993

[4] Agrawal R., Carey M., and Livny M. *Concurrency Control Performance Modeling: Alternatives and Implications*: ACM TODS, December 1987.

[5] Salcianu A. and Rinard M. *Pointer and Escape Analysis for Multithreaded Programs*, PPoPP 2001.

[6] Aldrich J., Chambers C., Sirer E., Eggers S. *Static Analyses for Eliminating Unnecessary Synchronization from Java Programs*: 6th Static Analysis Symposium, 1999

[7] Choi J-D., Gupta M., Serrano M., Sreedhar V., Midkiff S. *Escape Analysis for Java*: OOPLSA 1999.

[8] Bodga J., Holzle U. *Removing Unnecessary synchronization in Java*: OOPSLA 1999.

[9] Bacon D., Konuru R., Murthy C., Serrano M. *Thin Locks: Featherweight Synchronization for Java*: PLDI 1998.

[10] Bacon D., Strom R., Tarafdar A. *Guava: A Dialect of Java without Data Races*: OOPSLA 2000

[11] Rajwar R. and Goodman J. *Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution*: IEEE Micro, 2001

[12] Lindholm T. and Yellin F. *The Java^{TM} Virtual Machine Specification*: Addison Wesley Publishers