DetLock: Portable and Efficient Deterministic Execution for Shared Memory Multicore Systems

Hamid Mushtaq, Zaid Al-Ars, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
Delft, the Netherlands
{*H.Mushtaq, Z.Al-Ars, K.L.M.Bertels*}*@tudelft.nl*

***Abstract*—Multicore systems are not only hard to program but also hard to test, debug and maintain. This is because the traditional way of accessing shared memory in multithreaded applications is to use lock-based synchronization, which is inherently non-deterministic and can cause a multithreaded application to have many different possible execution paths for the same input. This problem can be avoided however by forcing a multithreaded application to have the same lock acquisition order for the same input.**

In this paper, we present *DetLock*, which is able to run multithreaded programs deterministically without relying on any hardware support or kernel modification. The logical clocks used for performing deterministic execution are inserted by the compiler. For 4 cores, the average overhead of these clocks on tested benchmarks is brought down from 20% to 8% by applying several optimizations. Moreover, the overall overhead, including deterministic execution, is comparable to state of the art systems such as *Kendo*, even surpassing it for some applications, while providing more portability.

I. Introduction

Single threaded programs are much easier to test, debug and maintain than their multithreaded counterparts. This is because the only source of non-determinism in them are interrupts or signals, which are rare. On the other hand, multithreaded programs have a frequent source of non-determinism in the form of shared memory accesses. Due to this, multithreaded programs suffer from repeatability problems, which means that running the same program with the same input can result different outputs. This repeatability problem makes multithreaded programs hard to test and debug. Furthermore, it is also difficult to build fault tolerant versions of these programs. This is because fault tolerance systems usually depend upon replicas (identical copies of redundant processes) to detect errors.

If access to shared data is not protected by synchronization objects in a multithreaded program, we can have race conditions, which may produce unexpected results. Running a program with race conditions deterministically does not avoid the problem of having unexpected results with those race conditions, but just makes sure that we get the same output with the same input.

The ideal situation would be to make a multithreaded program deterministic even in the presence of race conditions. This is not possible to do efficiently with software alone though. One can use a relaxed memory model where every thread writes to its own private memory, while data to shared memory is committed only at intervals. However, stopping threads regularly for committing to shared memory degrades performance as demonstrated by CoreDet [2], which has a maximum overhead of 11x for 8 cores. We can reduce the amount of committing to the shared memory by only committing at synchronization points such as locks, barriers or thread creation. This approach is taken by DTHREADS [11]. Here one can still imagine the slowdown in case of applications with high lock frequencies. Moreover, since in this case committing to the shared memory is done less frequently, more data has to be committed, thus also making it slow for applications with high memory usage. This is why hardware approaches have been proposed to increase efficiency of deterministic execution. Two such approaches are Calvin [4] and DMP [14]. They use the same concept as *CoreDet* for deterministic execution but make use of a special hardware for that purpose.

Since performing deterministic execution in software alone is inefficient, we can relax the requirements to improve efficiency. For example, Kendo [9] does this by only supporting deterministic execution for well written programs that protect every shared memory access through locks. In other words, it supports deterministic execution only for programs without race conditions. The authors of *Kendo* call it *Weak Determinism*. Considering the fact that most well written programs are race free and there exist tools to detect race conditions, such as Valgrind [8], *Weak Determinism* is sufficient for most well written multithreaded programs. Therefore, *DetLock* also only supports *Weak Determinism*.

The basic idea of *Kendo* is that it uses logical clocks for each thread to determine when a thread will acquire a lock. The thread with the least value of logical clock gets the lock. Though being quite efficient, *Kendo* still suffers from portability problems. First of all, it requires deterministic hardware performance counters for counting logical clocks. Many popular platforms (including many x86 platforms) do not have any hardware performance counter that is deterministic [12]. Secondly, *Kendo* needs modification of

the kernel to allow reading from the hardware performance counters for deterministic execution.

To overcome portability issues faced by *Kendo*, our tool *DetLock* has a completely software-based approach of updating the logical clocks. The code for updating the clocks is inserted through an LLVM [5] compiler pass. Since, LLVM is a popular open source compiler framework available on many platforms, our approach is portable across a wide range of platforms. Moreover, it requires no modification of the kernel. We can sum up the contribution of this paper as follows.

- A portable mechanism to update logical clocks for *Weak Deterministic* execution that depends upon the compiler rather than using hardware performance counters, since many platforms have no such deterministic counters available.
- A User-space approach to update the logical clocks that does not require modifying the kernel.
- A number of optimization steps to reduce the overhead of the code used to update the logical clock and improve the performance of deterministic execution.

This paper is organized as follows. In Section II, we discuss the background and related work, while in Section III, we give an overview of DetLock's architecture. This is followed by Section IV where we present the optimization methods used to improve the performance of *DetLock*. In Section V, we evaluate the performance of our scheme. We finally conclude the paper with Section VI.

II. Background and Related Work

Single threaded programs are mostly deterministic in behavior. We say mostly because interrupts and signals can introduce non-determinism even in single threaded programs. However, these non-deterministic events are rare. On the other hand, in multithreaded programs running on multicore processors, shared memory accesses are a frequent source of non-determinism.

One way to ensure determinism of multithreaded programs is to write code for them in a deterministic parallel language. Examples of such languages are StreamIt [10], SHIM [3] and Deterministic Parallel Java [1]. The disadvantage of this approach is that porting programs written in traditional languages to deterministic languages is difficult as the learning curve is high for programmers used to programming in traditional languages. Moreover, in languages which are based on the Kahn Process Network Model, such as SHIM, it is difficult to write programs without introducing deadlocks [7].

Deterministic execution at runtime can be done either through hardware or software. Calvin [4] is a hardware approach that executes instructions in the form of chunks and later commits them at barrier points. It uses a relaxed memory model, where instructions are committed in such a way that only the total store order (TSO) of the program has

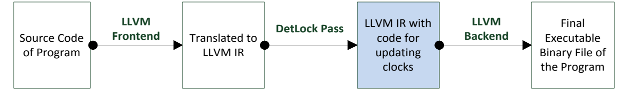

Figure 1: *DetLock* modifies the LLVM IR code by inserting code for updating logical clocks

to be maintained. DMP [14] uses a similar relaxed memory approach. The disadvantage of hardware approaches is that they are restricted to the platforms they were developed for.

Besides hardware methods, software only methods for deterministic execution also exist. One such method is CoreDet [2] that uses bulk synchronous quantas along with store buffers and relaxed memory model to achieve determinism. Therefore, it is similar to Calvin, but implemented in software. Logical clocks are used for deterministic execution. Since CoreDet is implemented in software, it has a very high overhead, possibly upto 11x for 8 cores, as compared to the maximum 2x for Calvin. Another similar approach is DTHREADS [11]. It runs threads as separate processes, so that memories which are modified can be tracked down through the memory management unit. Only at synchronization points such as locks, barriers and thread creation for example, it updates the shared memory from the local memories of the threads. Therefore, it avoids the overhead of using bulk synchronous quantas like CoreDet and also does not have the need to maintain logical clocks like CoreDet. However, the overhead for programs with high lock frequency or large memory usage is still very high.

Since performing deterministic execution in software alone is inefficient, Kendo [9] relaxes the requirements by only working for programs without race conditions (*Weak Determinism*). It does not use any hardware besides deterministic hardware performance counters found in some processors. It executes threads deterministically and performs load balancing by only allowing a thread to complete a synchronization operation when its clock becomes less than those of the other threads, with ties broken with thread IDs. Clock is calculated from retired stores, is paused when waiting for a lock and resumed after the lock is acquired. Kendo still suffers from portability problems as it requires hardware performance counters which are deterministic. Many platforms, including many x86 platforms, do not have any deterministic hardware performance counter [12]. Moreover, *Kendo* requires modification of the kernel to read from such hardware performance counters.

One technique related to deterministic multithreading is record/replay. In this method, all interleaving of shared memory accesses by different cores/processors are recorded in a log, which can be replayed to have a replica which follows the original execution. Examples of schemes using this method are Rerun [16] and Karma [15]. These schemes intercept cache coherence protocols to record inter-processor data dependencies, so that they can be replayed later on, in the same order. While Rerun only optimizes recording,

Karma optimizes both recording and replaying, thus making it suitable for online fault tolerance. It shows good scalability as well. The disadvantage of record/replay approaches as compared to deterministic multithreading is that they require a large memory for recording. Moreover, when used for fault tolerance, the redundant processes need to communicate with each other, as one replica records the log while the other reads from it.

Respec [6] is a record/replay software approach that only logs synchronization objects rather than every shared memory access. If divergence is found between the replicas, it rolls-back and re-executes from a previous checkpoint. However, if divergence is found again on re-execution, a race condition is assumed. At that point, a stricter deterministic execution is performed, which can induce a large overhead.

III. Overview of the Architecture

In this section, we discuss the architecture of *DetLock* and the application programming interface (API) that it provides to the programmer.

A. Architecture

We use Kendo's algorithm to perform deterministic execution. However, unlike *Kendo* which requires deterministic hardware performance counters, which are not available on many platforms, we insert code to update logical clocks at compile time. This also means we do not need to modify the kernel which is required by *Kendo* to read from performance counters. Figure 1 shows the point of compilation where the *DetLock* pass executes, which is between the point where the LLVM IR (Intermediate Representation) code is translated to the final binary code by the LLVM backend.

The unit of our logical clock is one instruction. For instructions which take more than one clock cycles, the logical clock is updated according to the approximate number of clock cycles they take. However, to keep our discussion simple, in this paper, for *DetLock* one instruction equals one logical clock count.

The Kendo's method of acquiring locks deterministically is illustrated in Figure 2. In this figure, an example is given for a process with two threads. If Thread 1 is trying to acquire a lock when its logical clock is 1029, it will not be able to do if Thread 2's clock is at 329, because of being less than 1029. But, as soon as Thread 2's clock get past 1029, Thread 1 will acquire the lock.

So basically our purpose is not only to reduce the code that updates the clocks but also to update the clocks as soon as possible. In fact, at compile time it is possible to increment the clock even before instructions are executed. For example, if we know that a leaf function (a function with no function calls) executes fixed amount of instructions, we can increment the logical clock before executing any instruction of that function. So for example, if Thread 2 in Figure 2 has logical clock of 329 and is about to execute a leaf function with 701 instructions, we can add 701 right away to its logical clock, making it 1030 from 329. In this way, Thread 1, whose clock is at 1029, can acquire the lock without waiting for Thread 2 to actually have executed that amount of instructions.

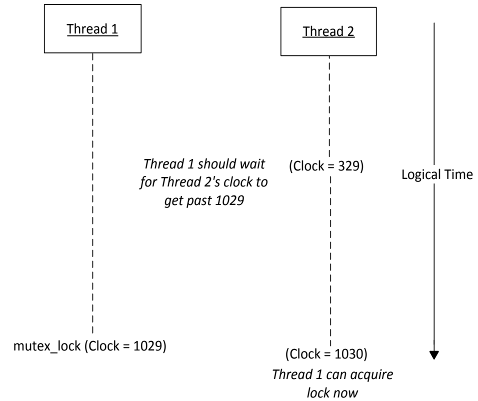

Figure 2: Kendo's method of acquiring locks for deterministic execution

Therefore in all optimizations we apply, besides trying to reduce the clock update overhead, we also try to increment the clock as soon as possible. Without any optimization, we update the clock at start of each of the basic block of LLVM IR. If there is a function call inside that block, we split that block, such that each block either contains no function call or starts and end with a function call. Then we update the clock at the top of each block if that block contains no function calls, otherwise we update the clocks in between the function calls. By splitting blocks in such a way, we can more easily apply optimizations.

To illustrate the effect of our optimizations, we are going to show how the optimizations change the example function shown in Figure 3. This function is taken from the *Radiosity* benchmark of SPLASH 2 [13]. The clocks associated with each block are shown at the right of the assignment operators.

B. Application Programming Interface

We provide our own functions for locks, barriers and thread creation for deterministic execution. They internally use the *pthread* library. However, it is not necessary for the programmer to modify the code to use them. A header file is provided by us that replaces the definition of these functions with ours. The header file can be specified in the makefile, thus making it unnecessary to modify source code files. Moreover, the code to initialize the clock for the main thread is inserted by the compiler.

It has to be noted that since our method depends upon the compiler to insert clocks for deterministic execution, it is not possible to increment the clocks in functions which are implemented in a library (Since they have not been compiled with our pass). This problem also exists for functions which

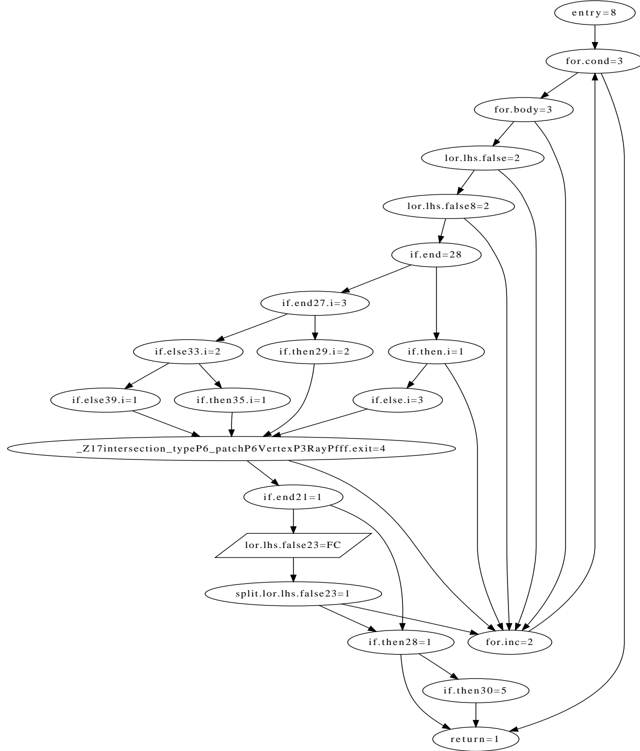

Figure 3: Example function for discussing the optimizations

are built in the compiler, as LLVM generates no code for them at IR level. For many built-in functions such as *memset* and math functions, we just keep an estimate of the instructions they take and increment the clock accordingly. For *memset* and other functions which depend upon the size parameter, we increment the clock considering the size parameter. Since most built-in functions are simple, we can use an estimate for them. We provide a text file (instructions estimate file) for such purpose, where these functions can be defined with the approximate number of instructions they take along with their dependency on input parameters. However, this is not always possible for functions in shared libraries. One way is to ignore them and the other way is to add them in the *instructions estimate file* if possible (If the instructions count for them can be approximated satisfactorily).

Another concern are functions which internally use locks, such as *malloc*. For such functions, we provide our own implementation which replaces the locks with our own deterministic locks.

IV. Performance Optimizations

We apply several optimizations to reduce the clock updating overhead. Moreover, we try to increment clocks as soon as possible so that waiting time for threads who are waiting for other thread's clocks to go past them is reduced. Clock updating code is removed from the blocks whose clocks are made zero by our optimizations. In this paper, we highlight such blocks with gray color. The optimizations are discussed below.

```
 1: function isClockable(out Int avg, ref Function f)
 2:     if hasLoops(f) or hasUnclockedFunctions(f) then
 3:         return false
 4:     end if
 5:     clocks = getClocksOfAllPaths(f)
 6:     avg = mean(clocks)
 7:     s = std(clocks)
 8:     r = range(clocks)
 9:     if r > (m / 2.5) or s > (m / 5) then
10:         return false
11:     end if
12:     return true
13: end function

14: function updateClockableFuncList
15:     modified = true
16:     while modfied do
17:         modified = false
18:         for all f in Program do
19:             if (not clockableList.find(f)) and isClockable(avg, f) then
20:                 removeClockFromFunction(f)
21:                 clockableList.insert(f, avg)
22:                 modified = true
23:             end if
24:         end for
25:     end while
26: end function
```

Figure 4: Pseudocode for Optimization 1 (Function Clocking)

A. Optimization 1 (Function Clocking)

As discussed in Section III-A, the sooner the clocks are updated, the better, and leaf functions with only one basic block are perfect candidates for such an optimization. Clocks can be removed from such functions and instead be added to the basic blocks calling such functions. Besides functions with only one blocks, our method also considers leaf functions with multiple blocks, given that there are no loops in such functions. If our pass sees that all possible paths taken by such a function do not differ by much, we calculate the mean value for all possible paths and use that mean value to update the clock. The criteria we have set is that the minimum and maximum clock difference of all possible paths should not be more than the mean value divided by 2.5. Moreover the standard deviation between all the different paths should not be greater than one fifth of the mean value. This is checked by calling the *isClockable* function shown in Figure 4.

We call such leaf functions as clocked functions. By intuition, we can judge that it is also possible to clock functions which call only clocked functions. In this way, we can even clock functions which are not necessarily leaf functions. The *UpdateClockableFunctList* function shown in Figure 4 shows how we do this. Our algorithm greedily searches for all such functions. This is done by first checking for all the functions in the program to see if they can be

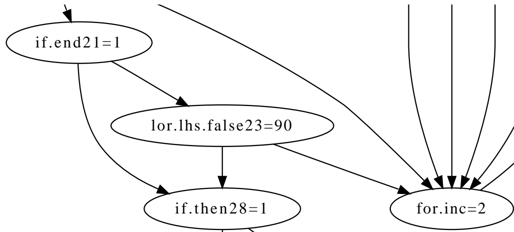

Figure 5: Part of example function after applying Optimization 1 (Function Clocking)

clocked and making them clocked functions if possible. If at the end, we see that one or more functions were added to the clocked functions list, which is signaled by the *modified* flag, we iterate over all the functions once again to search for more clockable functions. We keep on repeating this process until no more function is added to clockable functions' list in an iteration.

Part of example function after applying this optimization is shown in Figure 5. Originally, the block **lor.lhs.false23** had a function call at the start, therefore it was split in such a way that **lor.lhs.false23** contained the function call and **split.lor.lhs.false23** the remaining instructions in that block. However, this optimization notices that the function called in **lor.lhs.false23** is clockable, thus no splitting of the block is done and the mean number of instructions from all paths of that function are added to the clock of **lor.lhs.false23**. Moreover, clocks from all the blocks of the called function are removed.

B. Optimization 2 (For Conditional Blocks)

This optimization deals with if-else and switch statements and consists of two parts, **a** and **b**. The part **a** is a precise optimization, meaning that no estimation of clocks is used. They are just rearranged, so as to remove clocks from blocks if possible and incrementing the clock as soon as possible. On the other hand, part **b** is not necessarily precise, but we make sure that the clock does not diverge significantly after that pass.

1) Part a: This optimization is based on the principle that if a block has two or more successors, we can make the successor with the least clock zero and subtract its original value from all its siblings, while also adding its original clock to the parent block. Another principle of this optimization is that if all predecessors of a merge block have that merge block as their only successor, the clocks could be shifted from the merge node to them. The pseudocode of this optimization is shown in Figure 6. The *meetsOpt2aCondNodeRequirements* call on line 7 checks if a node meets the first principle, while *meetsOpt2aMergeNodeRequirements* call on line 15 checks for the second principle. Note that for the first principle, *meetsOpt2aCondNodeRequirements* also makes sure that no unclocked function call exists in the parent block and its successors. Moreover, it makes sure

```
1: function UPDATEOPT2ACLOCKS(ref bool modified, ref BasicBlock bb)
   ▷ When this function is called Entry block of a function is passed as bb
2:     if visitedList.find(bb) then
3:         return
4:     end if
5:     visited.insert(bb)
6:     modified = false
7:     if meetsOpt2aCondNodeRequirements(bb) then
8:         if allSuccessorsHaveNonZeroClock(bb) then
9:             modified = true
10:        end if
11:        min = minimumOfSucessors(bb)
12:        setClock(bb, GetClock(bb) + min)
13:        subtractFromAllSuccessors(bb, min)
14:    else
15:        if meetsOpt2aMergeNodeRequirements(bb) then
16:            pushClockUp(bb)
17:        end if
18:    end if
19:    succList = getAllSuccessors(bb)
20:    for all succ in succList do
21:        updateOpt2aClocks(modified, succ)
22:    end for
23: end function

24: function PUSHCLOCKUP(ref BasicBlock mergeBlock)
25:     clock = getClock(mergeBlock)
26:     removeClock(mergeBlock)
27:     predList = getAllPredecessors(mergeBlock)
28:     for all pred in predList do
29:         setClock(pred, GetClock(pred) + clock)
30:         if meetsOpt2aMergeNodeReq(pred) then
31:             pushClockUp(pred)
32:         end if
33:     end for
34: end function

35: function APPLYOPT2A
36:     for all f in Program do
37:         modified = true
38:         while modfied do
39:             visitedList.clear()
40:             updateOpt2aClocks(modified, f.entry())
41:         end while
42:     end for
43: end function
```

Figure 6: Pseudocode for Optimization 2a

that the parent block is dominating the successors, that is, the successors are not merge blocks. Similarly *meetsOpt2aMergeNodeRequirements* also makes sure that none of the blocks in consideration have unclocked function calls. It also makes sure that the merge block is not a loop header.

It should be noted that after having parsed all the blocks of a function and applying this optimization, if it is still possible to apply this optimization once more to reduce clock updating code, it is applied. This is done by checking the *modified* flag.

The example function after applying one pass of this optimization is shown in Figure 7. The sequence of events that will happen are given below (Refer to Figure 3 for original clock values).

- **if.then.i** is made 0 by **if.end**, which itself becomes 29 and makes **if.end27** equal to 2.
- **if.then.i** reaches the merge node

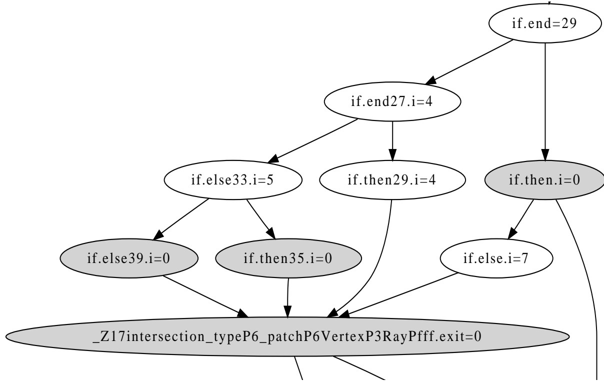

Figure 7: Part of example function after applying first iteration of Optimization 2a

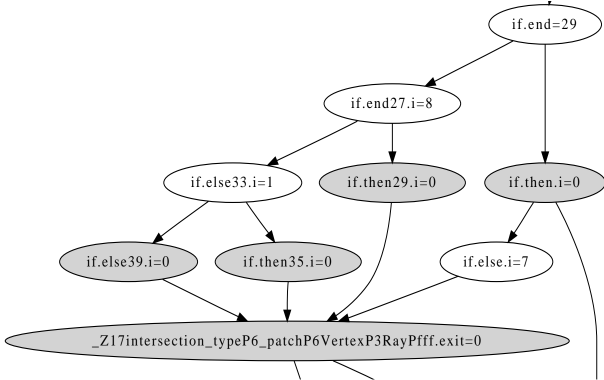

Figure 8: Part of example function after second and final iteration of Optimization 2a

_Z17intersection_typeP6_patchP6... through **if.else.i**.

- Merge node **_Z17intersection_typeP6_patchP6...** becomes 0 while propagating its clock to all of its 4 predecessors, which are **if.else39**, **if.then35.i**, **if.then29.i** and **if.else.i**, whose values now become 5, 5, 6 and 7 respectively.
- **if.end27** subtracts 2 from **if.else33** and **if.then29.i** to make them 0 and 4 respectively while itself becoming 4.
- **if.else33.i** takes value of 5 from **if.else39.i** and **if.then35.i** after making them 0.

Note that after applying this one pass, further optimization is still possible, but after the second pass (shown by Figure 8), no further optimization is possible.

2) Part b: The part **b** of this optimization deals with if conditions, such as those made by the blocks **if.end21**, **lor.lhs.false23** and **if.then28** in Figure 10. The pseudocode for this optimization is shown in Figure 9. The variable *swSucc* in Figure 9 represents the block in the middle, which is **lor.lhs.false23** in this example, while *endSucc* represents the merge node, which is **if.then28** for this example. The *meetsOpt2bRequirements* function call at line 6 checks if a pattern like the one shown in Figure 10 is formed.

If the block **lor.lhs.false23** was not jumping to **for.inc**, that

```
 1: function UPDATEOPT2BCLOCKS(ref BasicBlock bb)      ▷ When this
    function is called, Entry block of a function is passed as bb
 2:     if visitedList.find(bb) then
 3:         return
 4:     end if
 5:     visited.insert(bb)
 6:     swSucc, endSucc, meetsReq = meetsOpt2bRequirements(bb)
 7:     if meetsReq then
 8:         modifyClocks(bb, swSucc, endSucc)
 9:         updateOpt2bClocks(endSucc)
10:         swSuccList = getAllSuccessors(swSucc)
11:         for all succ in swSuccList do
12:             if succ != endSucc then
13:                 updateOpt2bClocks(endSucc)
14:             end if
15:         end for
16:     else
17:         succList = getAllSuccessors(bb)
18:         for all succ in succList do
19:             updateOpt2bClocks(succ)
20:         end for
21:     end if
22: end function

23: function APPLYOPT2B
24:     for all f in Program do
25:         visitedList.clear()
26:         updateOpt2bClocks(f.entry())
27:     end for
28: end function
```

Figure 9: Pseudocode for Optimization 2b

is, it had no successor other than **if.then28**, we could have straight away removed clock updating code from **if.end21** and added its clock value to **if.then28** to make it 2. That optimization, like part **a** would have been precise. However, since **lor.lhs.false23** has one more successor, our algorithm checks to see how much clock divergence we will get by removing clock from **if.end21**. The criteria we keep is that if the divergence is less than one tenth, we proceed with the optimization. In this case, by removing clock from **if.end21**, if we jumped to **for.inc** from **lor.lhs.false23**, the divergence would be 1/93, which is well below one tenth. Therefore, we proceed with it. The example function now becomes what is shown in Figure 10.

Note that this pass also determines if clock has to be removed from the upper block (**if.end21** in this case) or the lower block (**if.then28** in this case). We prefer to remove it from the lower block (and add it to upper block) so that clock is incremented ahead of time. However, in certain cases, we remove it from the upper block (and add it to lower block). One such case is when the upper block is at a higher loop depth than the lower block. Removing clock from upper clock is beneficial here since it is in a more critical path and therefore we save clock updating overhead. Another case where we remove clock from the upper block (and add to the lower block) is when the lower block has a higher clock than the upper block and middle block has more than one successors. This is because shifting clock to the upper block in this case will cause a larger divergence

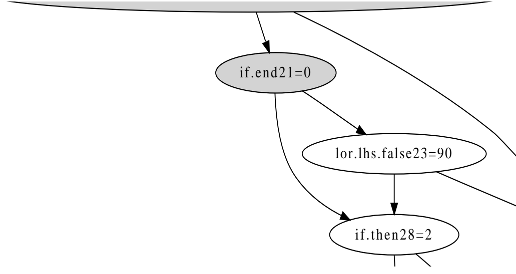

Figure 10: Part of example function after applying Optimization 2b

```
1:  function UPDATEOPT3CLOCKS(ref BasicBlock bb)          ▷ When this
    function is called, Entry block of a function is passed as bb
2:      if visitedList.find(bb) then
3:          return
4:      end if
5:      visited.insert(bb)
6:      if meetsOpt3Requirements(bb) then
7:          clocks, touchedBlocksList = getClocksOfAllOpt3Paths(bb)
8:          if isClockable(avg, clocks) then
9:              setClock(bb, avg)
10:             for all tb in touchedBlocksList do
11:                 removeClock(tb)
12:             end for
13:             tbSuccList = getAllSuccessorsOfTB(touchedBlocksList)
14:             for all succ in tbSuccList do
15:                 updateOpt3Clocks(succ)
16:             end for
17:             return
18:         end if
19:     end if
20:     succList = getAllSucessors(bb)
21:     for all succ in succList do
22:         updateOpt3Clocks(succ)
23:     end for
24: end function

25: function APPLYOPT3
26:     for all f in Program do
27:         visitedList.clear()
28:         updateOpt3Clocks(f.entry())
29:     end for
30: end function
```

Figure 11: Pseudocode for Optimization 3

in clock. In this example, since **if.end21** is at higher loop depth than **if.then28**, we remove the clock from **if.end21** and add it to **if.then28**.

C. Optimization 3 (Averaging of Clocks)

This optimization is based on the fact that paths emanating from a block in a function could be matching close together in total clock values. One can imagine it as a specialized case of the Optimization 1 (Function Clocking). For Function Clocking, we just considered the paths emanating from the entry block, but here we also check for paths besides the entry block. When forming paths for a block, we only consider blocks dominated by it (execution must pass through the dominating block to reach its dominated blocks).

The pseudocode for this optimization is shown in Figure 11. When finding paths for a block, we stop when we see backedges or when we see blocks with unclocked function calls. Moreover, we stop at a merge node if any of its successor is not dominated by the block in question. Like Optimizations 2a and 2b, we start to search for this optimization from the entry block of a function. If we find a block whose paths can be averaged, then after removing the clocks from the blocks in its path, we start to look for other blocks in the function. For this we consider the successors of the blocks in the path (from which the clocks were removed), given that those successors are not within that path. This is done by using the code from line 13 to 16 in Figure 11.

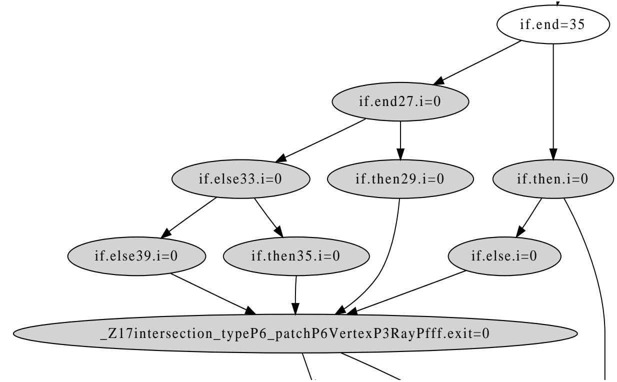

Figure 12: Part of example function after applying Optimization 3

The example function after applying this optimization is shown in Figure 12. In this figure, accumulated clocks for all different four paths emanating from **if.end** were 37, 38, 38 and 29, with a mean value of 35.5 and standard deviation of 4.36. Since the range here is 8 (37-29) and is less than $mean/2.5$ as well as the standard deviation is 4.36, which is less than $mean/5$, we assign a clock of 35 to **if.end**, while removing clocks from all the blocks in the path. Note that we did not consider nodes below the merge node **_Z17intersection_typeP6_patchP6...** because it has **for.inc** as its successor, which is not dominated by **if.end**.

D. Optimization 4 (Loops)

This optimization considers the fact that loops are often executed multiple times. So for example, if you have a *for* loop, the increment operation will take place just before the next iteration. Therefore we check for back edges and if we see that the clock of the block from which the backedge is originating is less that a certain threshold value and is also less than the clock of the block it is jumping to, we merge its clock value to that block's clock and remove clock updating code from it. In this example, the clock of **for.inc** is merged with **for.cond**.

Figure 13 shows the example function after applying this final optimization.

V. Performance Evaluation

We selected only those benchmarks from SPLASH-2 [13] which only have locks and barriers as synchronization opera-

Table I: Performance results of our scheme for the selected benchmarks

Benchmark	**Ocean**	**Raytrace**	**Water-nsq**	**Radiosity**	**Volrend**	**Average**
Original Exec Time	2903	670	1451	496	1340	-
Locks/sec	343	227835	126034	2211621	443070	-
Clockable Functions	1	33	1	39	35	-
After Inserting Clocks						
With No Optimization	2918 (1%)	718 (7%)	2082 (43%)	698 (41%)	1446 (8%)	20%
With Function Clocking Only (O1)	2901 (0%)	706 (5%)	2072 (43%)	644 (30%)	1445 (8%)	17%
With Conditional Blocks Optimization Only (O2)	2889 (0%)	715 (7%)	1779 (23%)	643 (30%)	1392 (4%)	13%
With Averaging of Clocks Only (O3)	2898 (0%)	702 (5%)	2072 (43%)	675 (36%)	1445 (8%)	18%
With Loops Optimization Only (O4)	2903 (0%)	707 (6%)	1752 (21%)	677 (36%)	1442 (8%)	14%
With All Optimizations	**2895 (0%)**	**695 (4%)**	**1748 (20%)**	**562 (13%)**	**1386 (3%)**	**8%**
After Inserting Clocks and Performing Deterministic Execution						
With No Optimization	2918 (1%)	768 (15%)	2096 (44%)	855 (72%)	1451 (8%)	28%
With Function Clocking Only (O1)	2924 (1%)	758 (13%)	2085 (44%)	711 (43%)	1448 (8%)	22%
With Conditional Blocks Optimization Only (O2)	2918 (1%)	766 (14%)	1785 (23%)	788 (57%)	1398 (4%)	20%
With Averaging of Clocks Only (O3)	2916 (0%)	742 (11%)	2090 (44%)	807 (63%)	1450 (8%)	25%
With Loops Optimization Only (O4)	2904 (0%)	760 (13%)	1761 (21%)	837 (69%)	1448 (8%)	22%
With All Optimizations	**2915 (0%)**	**742 (11%)**	**1758 (21%)**	**683 (38%)**	**1395 (4%)**	**15%**

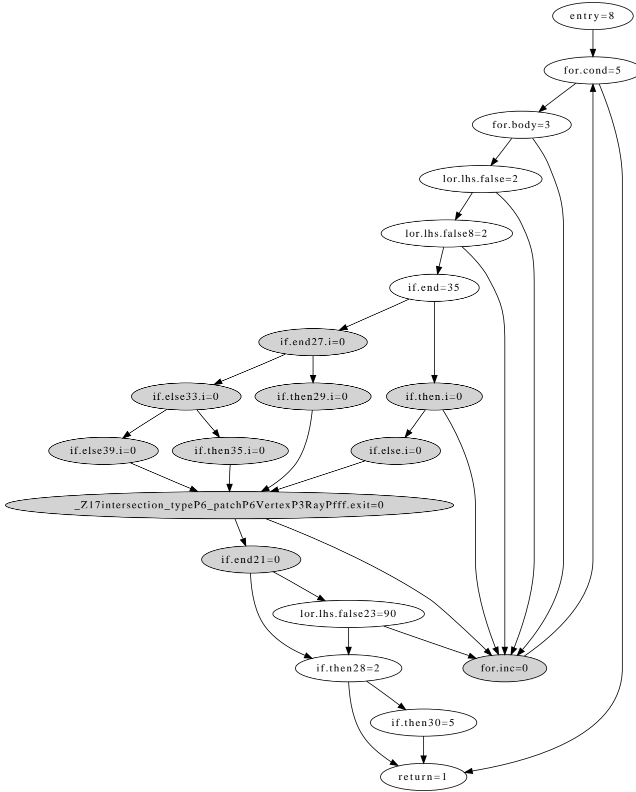

Figure 13: Example function after applying all optimizations

tions, as we have not yet implemented other synchronization operations, such as *condition variables* for example. All benchmarks were run on a 2.66 GHz quad core machine and compiled with maximum optimization enabled (level -O4 for clang/llvm). We first discuss the results. Afterwards, we show how clocking instructions ahead of time improves the deterministic execution. Lastly, we compare our results with those from *Kendo*.

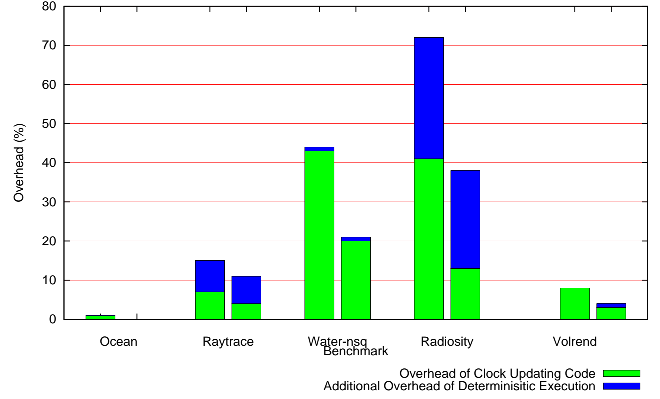

Figure 14: Overhead of inserting clocks and deterministic execution

A. Results

Table I shows the performance overheads with different optimizations and Figure 14 gives a pictorial view of that overhead. The left bars in Figure 14 show the performance overhead without applying optimizations while the bars on the right show the overhead after applying all the optimizations. The lower portion of the bar is the overhead of the inserted clocks updating code only, while the upper portion shows the additional overhead for deterministic execution.

From Table I, we can see that different optimization affect different benchmarks differently. For example, Optimization 4 (Loops Optimization) has a significant impact on the performance of *Water-nsq* while not having that much effect on other benchmarks. This is because *Water-nsq* frequently executes a loop with a small body. The optimization that had the most impact on performance overall is Optimization 2 (Conditional Blocks Optimization). This is because conditional paths are frequently found in programs and this optimization efficiently reduces clock update for such paths. The Optimization 3 had the least impact. This is because it is unlikely for a program to have all paths originating from

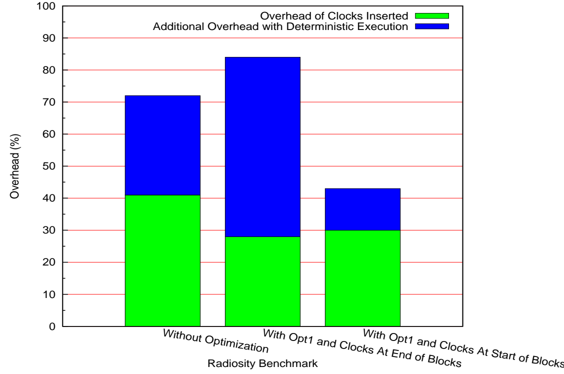

Figure 15: Improvement of the *Radiosity* benchmark from updating clocks ahead of time

a node to have similar clock values.

As far as Optimization 1 (Function Clocking) goes, *Radiosity* is one benchmark where this optimization significantly improved the performance. This is because this benchmark has such functions which are compute intensive and execute frequently. One interesting result of this optimization is that it significantly reduces the overhead of deterministic execution in addition to the reduction in clock updating overhead. This is discussed in more detail in the next section. Overall, we see that the average overhead of inserting clocks is at 7%, whereas the average overhead for deterministic execution is at 14%, with the overhead not exceeding 38% even for *Radiosity*, which has a very high lock frequency.

B. Effect of Updating Clocks Ahead of Time

There is a great benefit in updating the clock as soon as possible, so that threads waiting at a lock acquisition have to wait less. This effect is more evident for a benchmark like *Radiosity* which has high a lock frequency. From Table I, we can see that for *Radiosity*, although Optimization 2 (Conditional Blocks Optimization) reduces the clock overhead by the same amount as Optimization 1 (Function Clocking), Optimization 1 adds far less additional overhead for deterministic execution at 13% (43% - 30%) as compared to 27% (57% - 30%) in the case of Optimization 2. This is because Optimization 1 is able to increment the clock more aggressively ahead of time as it works for whole functions.

Figure 15 illustrates the effect of updating clocks ahead of time for *Radiosity*. Since *Function Clocking* optimization, where possible, increments the clock ahead of time the most (more than other optimizations), we only consider the result of Optimization 1 (Function Clocking) here. The left most bar is that without any optimization, the middle is with *Function Clocking* optimization, but clocks updated at the end of the basic blocks, whereas the right most bar is the same optimization but with clocks updated at the beginning of the basic blocks. From the figure, by looking at the upper portion of the bars, which represents the additional overhead of deterministic execution, we can see that updating clocks at the start of the block improves deterministic execution significantly as compared to updating them at the end.

C. Comparison with Kendo

In Table II, we compare our results with that of *Kendo*. Note that the purpose of our scheme is not to surpass *Kendo* in performance but to make it more portable while retaining sufficient efficiency. Since the data sets used by *Kendo* are not publicly available, neither its source code, we list the results directly from their paper. We tried to use the data sets which match the locks/sec frequency of those used by *Kendo*. For *Radiosity* and *Volrend*, we could not find matching data sets however and instead used data sets with higher lock frequencies than *Kendo*.

The only benchmark which performs worse than *Kendo* is *Water-nsq*. This is because *Water-nsq* executes a small *for* loop very frequently. The code inside that *for* loop contains an *if* statement. Although Optimization 2 (Conditional Blocks Optimization) and Optimization 4 (Loops Optimization) work to reduce overhead of clocks update in that loop, it still updates clocks frequently enough in that loop to have a relatively high overhead.

For *Radiosity*, which has a very high lock frequency, our scheme surpasses *Kendo* in performance. This is even when we used a data set which has a higher lock frequency than what *Kendo* used. This improvement in performance over *Kendo* can be explained by the fact that at compile time, we are able to update clocks before instructions are executed and thus reduce waiting time for a benchmark like *Radiosity* which has a high lock frequency. On the other hand, *Kendo* only updates the logical clocks when it receives overflow interrupts of the hardware performance counter that counts retired stores. Therefore, it cannot perform clock updates ahead of time. It also has to balance the chunk size of instructions executed between each interrupt, so as to reduce the impact of frequent interrupts while also maintaining frequent interrupts to keep the clocks incrementing. For *Radiosity*, the authors of *Kendo* had to manually adjust the chunk size to get the best performance, which is the one listed in Table II. Our scheme requires no such manual adjustments.

We even show slight improvement over *Kendo* for benchmarks which do not have very high lock frequencies, such as *Raytrace* and *Volrend*. This improvement can be explained from the fact that our scheme updates the clock more frequently than *Kendo*. Although, in case of *Kendo* there may be a less overhead of updating the clocks, threads who are in the process of acquiring a lock and thus waiting for other threads' clocks to go past them, have to wait longer due to the slow update of the clocks. Moreover, our optimizations prefer to update the clock even before instructions are executed. So even when we are updating the clocks less frequently, it is not because we are delaying their

Table II: Performance results of our scheme as compared to *Kendo*

Benchmark	Ocean	Raytrace	Water-nsq	Radiosity	Volrend
Results for Kendo					
Locks/sec	279	216979	143202	939771	79612
Overhead	1%	18%	7%	53%	7%
Results for our scheme					
Locks/sec	343	227835	126034	2211621	443070
Overhead	0%	11%	21%	38%	4%

update, but because we (most of the time) already updated them ahead of time.

VI. Conclusion

In this paper, we described our tool *DetLock*, which consists of an LLVM compiler pass to insert code for updating logical clocks for *Weak Deterministic* execution. Since our scheme does not depend on any hardware or modification of the kernel, it is very portable. Moreover, we apply several optimizations to reduce the amount of code inserted for clock updating. Furthermore, since the algorithm for *Weak Determinism* that we use gives lock to the thread with minimum logical clock, we try to increment the clocks of threads as soon as possible so that threads waiting for locks have to wait less. We increment the clocks even before instructions are executed if possible. On average, the overhead of inserting clock updating code is only 8%, whereas the overall overhead including deterministic execution is 15% for selected benchmarks. This performance is comparable to *Kendo*, while providing more portability. In fact for some applications, *DetLock* can even surpass *Kendo* in performance.

Acknowledgment

This research has been funded by the projects Smecy 100230, iFEST 100203 and REFLECT 248976.

References

[1] http://dpj.cs.uiuc.edu.

[2] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 38:53–64, March 2010.

[3] S. A. Edwards and O. Tardieu. Shim: a deterministic model for heterogeneous embedded systems. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 264–272, New York, NY, USA, 2005. ACM.

[4] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? free will to choose. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 333 –334, feb. 2011.

[5] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[6] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism, 2010.

[7] H. Mushtaq, Z. Al-Ars, and K. Bertels. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. In *Design and Test Workshop (IDT), 2011 IEEE 6th International*, pages 12 –17, dec. 2011.

[8] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *In Proceedings of the 2007 Programming Language Design and Implementation Conference*, 2007.

[9] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44:97–108, March 2009.

[10] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, 2002. Springer-Verlag.

[11] E. D. B. Tongping Liu, Charlie Curtsinger. Dthreads: Efficient deterministic multithreading. In *SOSP '11*, oct 2011.

[12] V. Weaver and S. McKee. Can hardware performance counters be trusted? In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 141 –150, sept. 2008.

[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23:24–36, May 1995.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 85–96, New York, NY, USA, 2009. ACM.

[15] A. Basu, J. Bobba, and M. D. Hill. Karma: scalable deterministic record-replay. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 359–368, New York, NY, USA, 2011. ACM.

[16] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 265–276, Washington, DC, USA, 2008. IEEE Computer Society.