Computing (2014) 96:1131–1147
DOI 10.1007/s00607-013-0370-9

Efficient and highly portable deterministic multithreading (DetLock)

Hamid Mushtaq · Zaid Al-Ars · Koen Bertels

Received: 22 March 2013 / Accepted: 5 November 2013 / Published online: 19 November 2013

Abstract In this paper, we present *DetLock*, a runtime system to ensure deterministic execution of multithreaded programs running on multicore systems. *DetLock* does not rely on any hardware support or kernel modification to ensure determinism. For tracking the progress of the threads, logical clocks are used. Unlike previous approaches, which rely on non-portable hardware to update the logical clocks, *DetLock* employs a compiler pass to insert code for updating these clocks, thus increasing portability. For 4 cores, the average overhead of these clocks on tested benchmarks is brought down from 16 to 2 % by applying several optimizations. Moreover, the average overall overhead, including deterministic execution, is 14 %.

Keywords Multicore · Multithreading · Determinism

Mathematics Subject Classification 68N01 · 68W10 · 68N20

1 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 is

In this paper, we extended DetLock, whose paper was published in MuCoCos 2012 affiliated with SC12. In this journal paper, we further improve the performance of DetLock by applying several more optimizations. Furthermore, we evaluated the performance with several more benchmarks.

H. Mushtaq (✉) · Z. Al-Ars · K. Bertels
TU Delft, Delft, Netherlands
e-mail: h.mushtaq@tudelft.nl

Z. Al-Ars
e-mail: z.al-ars@tudelft.nl

K. Bertels
e-mail: k.l.m.bertels@tudelft.nl

interrupts or signals, which are rare. On the other hand, multithreaded programs have a more 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 tolerant 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, but just makes sure that the same results can be replicated.

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 [15]. 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 [7] and DMP [4]. 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 [13] 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 [12], *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 [16]. 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 [8] 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 techniques to reduce the overhead of the code used to update the logical clock and improve the performance of deterministic execution.

This paper is an extension on our previous work on this topic [11]. In this paper, we apply several more optimizations to improve the performance. This paper is organized as follows. In Sect. 2, we discuss the background and related work, while in Sect. 3, we give an overview of DetLock's architecture. This is followed by Sect. 4 where we present the optimization methods used to improve the performance of *DetLock*. In Sect. 5, we evaluate the performance of our scheme, and we finally conclude the paper with Sect. 6.

2 Background and related work

In this section, first we will discuss the state of the art for deterministic execution and then discuss our contribution.

2.1 State of the art

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 [14] and SHIM [5]. 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 [10].

Deterministic execution at runtime can be done either through hardware or software. Calvin [7] 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 to be maintained. DMP [4] 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 [15]. 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 [13] 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 [16]. Moreover, *Kendo* requires modification of the kernel to read from such hardware performance counters. A technique related to deterministic multithreading is record/replay. Examples of systems using this technique are Rerun [6], Karma [1] and Respec [9].

2.2 Our contribution

As discussed in the previous section, we already have tools such as *Kendo* to execute multithreaded programs deterministically on multicore platforms. However, one main bottleneck of using *Kendo* is that it requires deterministic hardware performance counters, which are not available on many platforms. For evaluation of their tool, the authors of *Kendo* had to specifically use the Core 2 processor, which had deterministic retired stores counters available on it. As we can see from Fig. 1, which shows the retired stores difference compared to the expected value, none of the listed processor besides Core 2, has a deterministic retired stores counter. Moreover, *Kendo*, also requires modification of the kernel to access these performance counters.

Now imagine a scenario, where a company wants to reduce the cost of testing for its software as well as ease maintainability of it by making it deterministic. If they go for the *Kendo* technique, it would make their software non-portable as it would be unable to run on processors which do not have any deterministic hardware performance

Machine	Before Adjustment	Adjusted
Core2	0±0	0±0
Atom	—	—
Nehalem	411,408±4	9±1
Nehalem-EX	411,914±6	9±1
Pentium D	163,402,604±185	11,776±175
Phenom	—	—
Istanbul	—	—

Fig. 1 Determinism of retired stores performance counter of various processors [16]

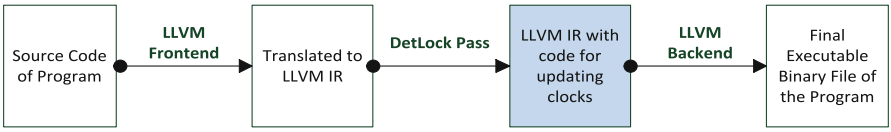

Fig. 2 *DetLock* modifies the LLVM IR code by inserting code for updating logical clocks

counter. Moreover, you can expect users not wanting to modify the kernels of their operating systems. This is where our technique is useful, as it would allow a program to run on every machine, without requiring to modify the kernel.

This work is an extension of our previous work [11] on this topic. By applying new optimizations, we were able to further reduce the overhead of clock updating code inserted by our compiler pass, and improve performance of deterministic execution.

3 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.

3.1 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 that we do not need to modify the kernel which is required by *Kendo* to read from performance counters. Figure 2 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 Fig. 3. 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 1,029, it will not be able to do so if Thread

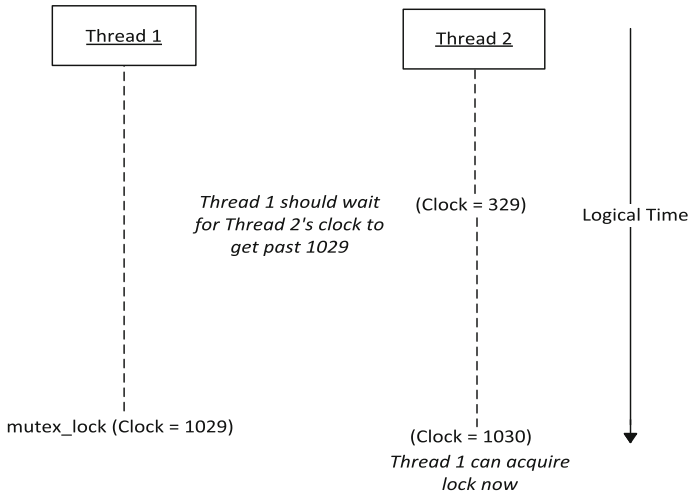

Fig. 3 Kendo's method of acquiring locks for deterministic execution

2's clock is at 329, because of being less than 1,029. But, as soon as Thread 2's clock get past 1,029, 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.

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 ends 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.

3.2 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 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.

4 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 threads' 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. To illustrate the effect of our optimizations, we are going to show how the optimizations change example functions. The clocks associated with each block are shown at the right of the assignment operators. Moreover, a block in parallelogram shape implies that it contains one or more function calls. The optimizations are discussed below.

4.1 Optimization 1 (function clocking)

As discussed in Sect. 3.1, 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.

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. More detail on this optimization can be found in [11].

Previously, in [11], we did not consider the possibility of clocked functions being called indirectly through functions pointers. In that case, since we remove the clocks

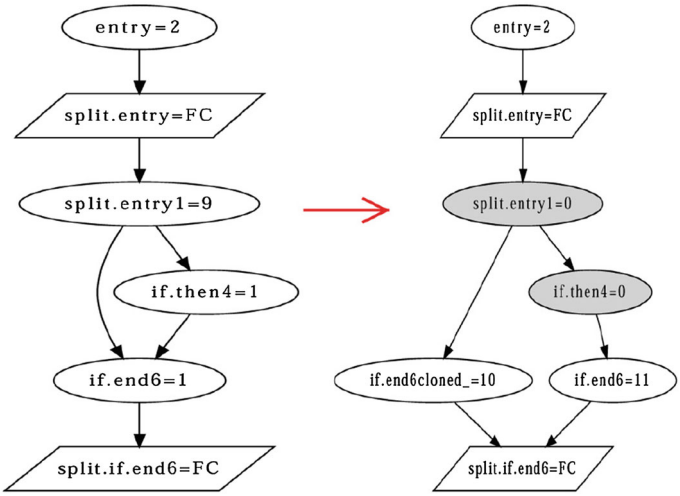

Fig. 4 Example function before and after applying optimization 2b

of clocked functions, the clock does not get updated. To correct this problem, we create clones of clocked functions. The clock from cloned clock functions is removed but clock updating code is inserted in the original clocked functions. Wherever our pass finds a direct call of a clocked function, it replaces it with call to the cloned version of that clocked function. However, since indirect calls still call the original clocked function, the clock does get updated properly even with indirect calls.

4.2 Optimization 2 (conditional blocks)

This optimization deals with if-else and switch statements and consists of four parts, which are described below.

4.2.1 Opt 2a: pushing clocks upwards

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. More details about this optimization can be found in [11].

4.2.2 Opt 2b: cloning blocks

In this optimization, we clone blocks where possible to reduce the number of clock updates. For example, for the example function shown in Fig. 4, block *if.end6* is cloned, so that for the paths formed by blocks *split.entry1*, *if.then4* and *if.end6*, clock needs to be updated only once, rather than twice or thrice.

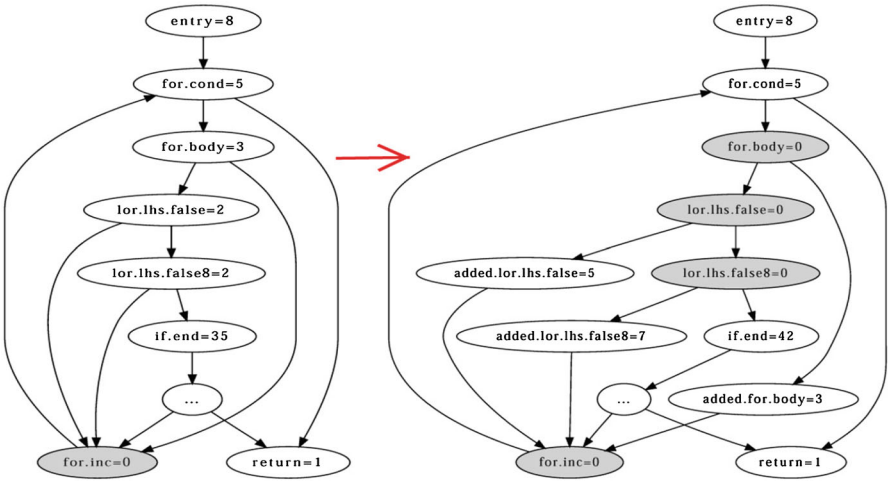

Fig. 5 Example function before and after applying optimization 2c

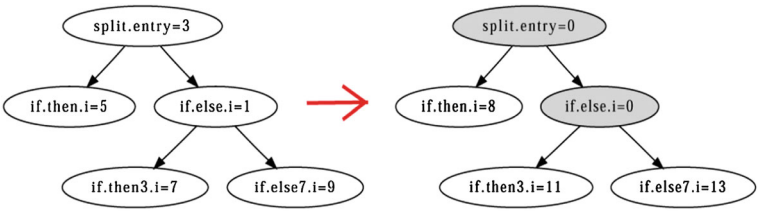

Fig. 6 Part of an example function before and after applying optimization 2d

4.2.3 Opt 2c: adding additional blocks

In this optimization, we add new blocks where necessary to reduce the clock updating code. To illustrate this, an example function before and after applying this optimization is shown in Fig. 5. Three new blocks are added to update the clocks removed from the three blocks shown in gray in the optimized version. The accumulated clock of those three blocks is also added to the clock of block *if.end*. With this optimization, for the path from *for.cond5* to *if.end*, clock is updated only twice, while in the unoptimized version it is updated 5 times. Note that the block (. . .) here represents a bundle of basic blocks, which we do not show due to space limitations.

4.2.4 Opt 2d: pushing clocks downwards

In this optimization, we update the clock from top to bottom. This optimization can remove clocks more efficiently in some cases as compared to *Opt 2a*. However, Since we try to update clocks as soon as possible, we apply this optimization only for paths where the accumulated clock is less than a certain value. Part of an example function, before and after applying this optimization is shown in Fig. 6. We can see that with

this optimization, for the part of the example function, clock is only updated once, rather than twice or thrice in the original version of that function.

4.3 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). More details about this optimization can be found in [11].

4.4 Optimization 4 (loops)

This optimization deals with loops. The different types of optimization we applied on loops are discussed next.

4.4.1 Opt 4a: forwarding clocks from blocks with backedges

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 than 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.

4.4.2 Opt 4b: incrementing clocks before for loops

This optimization is based on the fact that for many *for* loops, the number of iterations can be checked at compile time. So for example, if at compile time, we can see that a *for* loop is executed n number of times, we can update the clock ahead of time. First our pass figures out the least number of instructions an iteration in a loop will execute. This number multiplied with n is incremented before execution of the *for* loop. Inside the *for* loop, we update the clock only where it is necessary.

An example function before and after applying this optimization is shown in Fig. 7. Here, the minimum number of instructions executed by the *for* loop for an iteration is 21. Therefore it is multiplied by the number of iterations N of the *for* loop.

The pseudocode for optimization 4b is shown in Fig. 8. For each block in a function, it checks if its a loop header. This optimization is only applied for inner most loops, as they are usually the most compute intensive types of loops. The *meetsOpt4bRequirements* checks if all blocks inside the loop are at the same level, as well as checks other things, such as, no block has unclocked functions. If all the requirements are met, the optimization is applied. *preds* here are the predecessor blocks of the loop header. For example, in the example function shown in Fig. 7, block

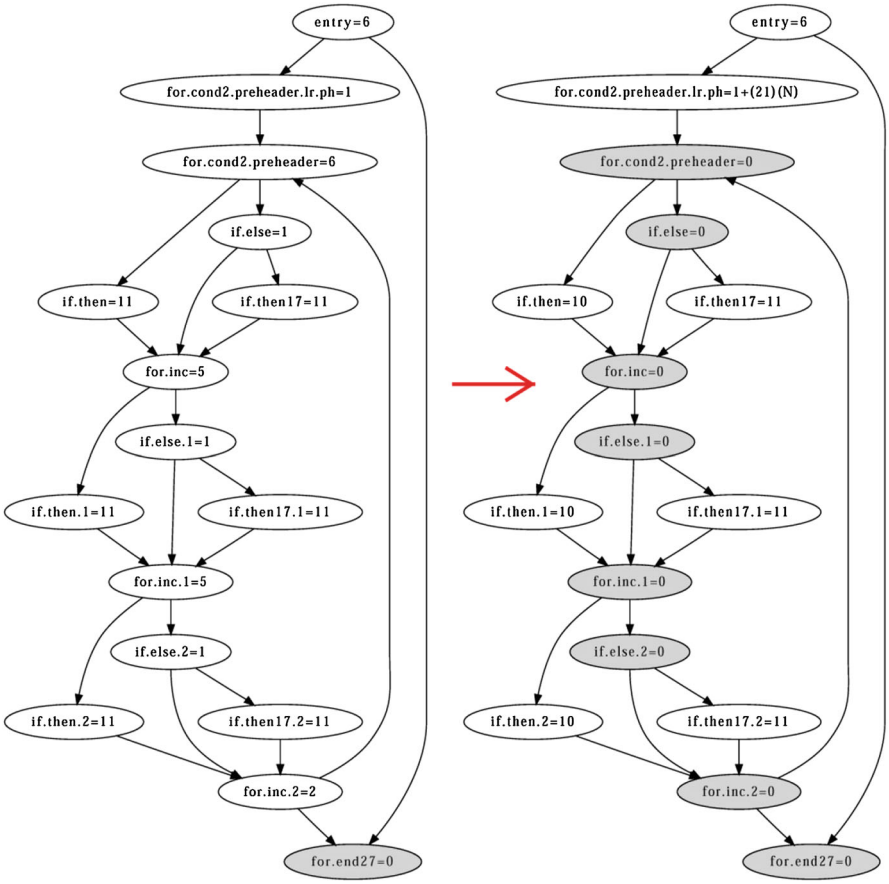

Fig. 7 Example function before and after applying optimization 4b

for.cond2.preheader.lr.ph is the predecessor. If there is only one predecessor, which have no other successor than the loop header, the clock is added to that predecessor block, as shown by the code on line 14 to 19. Otherwise, a block is added in between the loop header and its predecessors to update the clock. The *be* block returned by *meetsOpt4bRequirements* is the block that contains the backedge, which is *for.inc.2* in this case. This is passed as a parameter to the *updateClocksInLoop* function, which shifts the constant clock value of the loop, that is, the least number of instructions the loop will always execute in an iteration, to the header block. The value in the header block is then shifted to the predecessor block. In this case, the *updateClocksInLoop* function added 15 to the original clock of *for.cond2.preheader*, which is later shifted to the predecessor block *for.cond2.preheader.lr.ph*.

UpdateClocksInLoop works by first concentrating clock to all merge nodes which are guaranteed to be executed in an iteration. Such blocks are *for.inc*, *for.inc1* and *for.inc2* in the example function. The list of all blocks preceding such a merge node is checked to see which one has minimum value, and that minimum value is added

```
1: function updateOpt4bClocks(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:     preds, be, loopIterInfo, meetsReq = meetsOpt4bRequirements(bb)
8:     if meetsReq then
9:         pathsList, blocksInLoop = getLoopPaths(bb, be)
10:        blocksPresentInAllPaths = getBlocksPresentInAllPaths(pathsLists, bb)
11:        if blocksInLoop.size() > 1 then
12:            updateClocksInLoop(blocksPresentInAllPaths, bb, be, blocksInLoop)
13:        end if
14:        loopIterInfo.constLoopClock = getClock(bb)
15:        setClock( bb, 0 )
16:        if preds.size() > 1 or numSuccessors(preds[0]) > 1 then
17:            addedBlocks = addBlockInBetween(preds, bb)
18:            setloopIterInfo(addedBlocks, loopIterInfo)
19:        else
20:            setloopIterInfo(preds[0], loopIterInfo)
21:        end if
22:    end if
23:    succList = getSuccList(bb)
24:    for all succ in succList do
25:        updateOpt4bClocks(modified, succ)
26:    end for
27: end function

28: function ApplyOpt4b
29:    for all f in Program do
30:        modified = true
31:        while modfied do
32:            visitedList.clear()
33:            updateOpt4bClocks(modified, f.entry())
34:        end while
35:    end for
36: end function
```

Fig. 8 Pseudocode for optimization 4b

to such merge node. For example, *if.then* and *if.else* are blocks preceding the merge node *for.inc*. *if.then17* is not included in the list since its being dominated by *if.else*, which is already preceding the merge node in question. Since *if.else* has the minimum clock of the two, its clock (1) is added to *for.inc*, making clock of *for.inc* 6, while making clock of *if.else* 0, and also subtracting 1 from *if.then*. The same is done with the other two merge nodes. After doing this step, *for.inc* and *for.inc1* contain clock values of 6 each while *for.inc2* contain clock value of 3. So, overall these three merge nodes contain clock value of 15. This clock value is removed from all these nodes and shifted to *for.cond2.preheader* to make its clock 21 from 6, while making the clocks of these merge nodes 0. Later on, clock is removed from *for.cond2.preheader* and updated before executing the loop in *for.cond2.preheader.lr.ph* by multiplying it with the number of times the loop will execute.

4.4.3 Opt 4c: cloning while loops

It is not possible to apply optimization 4b, where the number of iterations of a loop could not be determined at compile time, which is usually the case with *while* loops. If

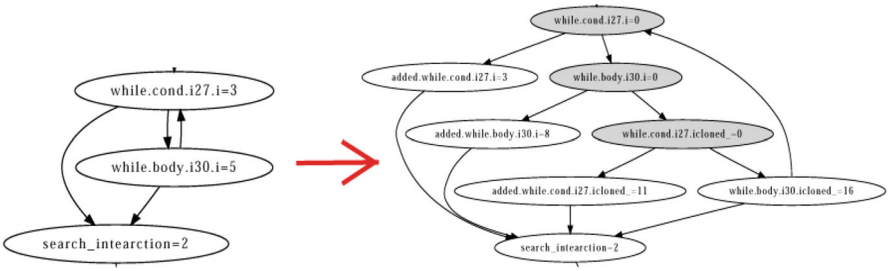

Fig. 9 Part of an example function before and after applying optimization 4c

a *while* loop executes constant number of instructions in each iteration, we clone that while loop, so that clock is updated at every other iteration rather than each iteration. A part of an example function before and after applying this optimization is shown in Fig. 9. Note that even if we applied optimization *4a* here, the clock would have been updated after 8 instructions. But by cloning these loops, it is executed after 16 instructions. Note that we also add blocks to update the clock if the *while* loop exits on an odd iteration.

5 Performance evaluation

We tested performance of *DetLock* with 8 benchmarks, 6 from SPLASH-2 [17] and 2 from PARSEC [3]. 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.

5.1 Results

Table 1 shows the performance overheads with different optimizations and Fig. 10 gives a pictorial view of that overhead. In Table 1, along with the results with different optimizations, we also show the original execution times, locks per second and number of clocked functions for each benchmark. Note that all the times are in milliseconds. The left bars in Fig. 10 show the performance overhead without applying optimizations, while the bars in the middle show performance after applying optimizations of [11] only. The bars on the right show the overhead after applying all the optimizations, including those mentioned in this paper. 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 Fig. 10, we can see that new optimizations introduced in this paper improved performance for several benchmarks as shown by the decrease in size of the right bars. The improvement relative to [11] is most significant for *barnes*, *water* and *swaptions*. For *water* for example, the overhead of deterministic execution is brought down from 20 to 0 %. Table 1 show performance with different optimizations. The optimization 4

Table 1 Performance results of our scheme for the selected benchmarks

BM	Barnes	Ocean	Volrend	Water	Swaption	Fluidanim	Raytrace	Radiosity	Avg
Orig	1,191	2,656	949	1,333	892	1,447	2,165	928	–
Locks/sec	462,101	375	125,293	141,067	689,283	2,472,882	241,757	3,170,433	–
Clk Funcs	9	0	34	0	2	0	32	39	–
After inserting clocks									
No Opt	1,436 (21 %)	2,681 (1 %)	975 (3 %)	1,890 (42 %)	959 (8 %)	1,701 (18 %)	2,307 (7 %)	1,225 (32 %)	1,647 (16 %)
Only [11]	1261 (6 %)	2,669 (0 %)	950 (0 %)	1,603 (20 %)	952 (7 %)	1,530 (6 %)	2,305 (6 %)	969 (4 %)	1,530 (6 %)
Opt 1	1,361 (14 %)	2,681 (1 %)	973 (3 %)	1,890 (42 %)	949 (6 %)	1,701 (18 %)	2,306 (7 %)	1,148 (24 %)	1,626 (14 %)
Opt 2	1,242 (4 %)	2,667 (0 %)	950 (0 %)	1,387 (4 %)	954 (7 %)	1,638 (13 %)	2,306 (7 %)	1,013 (9 %)	1,520 (6 %)
Opt 3	1,257 (6 %)	2,672 (1 %)	950 (0 %)	1,880 (41 %)	953 (7 %)	1,701 (18 %)	2,307 (7 %)	1,089 (17 %)	1,601 (12 %)
Opt 4+1	1,271 (7 %)	2,657 (0 %)	951 (0 %)	1,333 (0 %)	917 (3 %)	1,688 (17 %)	2,296 (6 %)	1,109 (20 %)	1,528 (6 %)
All Opts	1,199 (1 %)	2,656 (0 %)	949 (0 %)	1,333 (0 %)	913 (2 %)	1,527 (6 %)	2,247 (4 %)	968 (4 %)	1,474 (2 %)
After inserting clocks and performing deterministic execution									
No Opt	1,547 (30 %)	2,698 (2 %)	1,036 (9 %)	1,890 (42 %)	1,154 (29 %)	2,411 (67 %)	2,412 (11 %)	1,557 (68 %)	1,838 (32 %)
Only [11]	1,403 (18 %)	2,675 (1 %)	951 (0 %)	1,604 (20 %)	1,076 (21 %)	2,220 (53 %)	2,403 (11 %)	1,249 (35 %)	1,698 (20 %)
Opt 1	1,403 (18 %)	2,698 (2 %)	1,034 (9 %)	1,890 (42 %)	1,151 (29 %)	2,411 (67 %)	2,411 (11 %)	1,354 (46 %)	1,794(28 %)
Opt 2	1,449 (22 %)	2,682 (1 %)	951 (0 %)	1,388 (4 %)	1,148 (29 %)	2,315 (60 %)	2,412 (11 %)	1,454 (57 %)	1,725 (23 %)
Opt 3	1,436 (21 %)	2,676 (1 %)	972 (2 %)	1,880 (41 %)	1,112 (25 %)	2,364 (63 %)	2,411 (11 %)	1,437 (55 %)	1,786 (27 %)
Opt 4+1	1,369 (15 %)	2,668 (0 %)	984 (4 %)	1,334 (0 %)	1,008 (13 %)	2,396 (66 %)	2,390 (10 %)	1,353 (46 %)	1,688(19 %)
All Opts	1,211 (2 %)	2,667 (0 %)	950 (0 %)	1,333 (0 %)	1,005 (13 %)	2,220 (53 %)	2,385 (10 %)	1,247 (34 %)	1,627 (14 %)

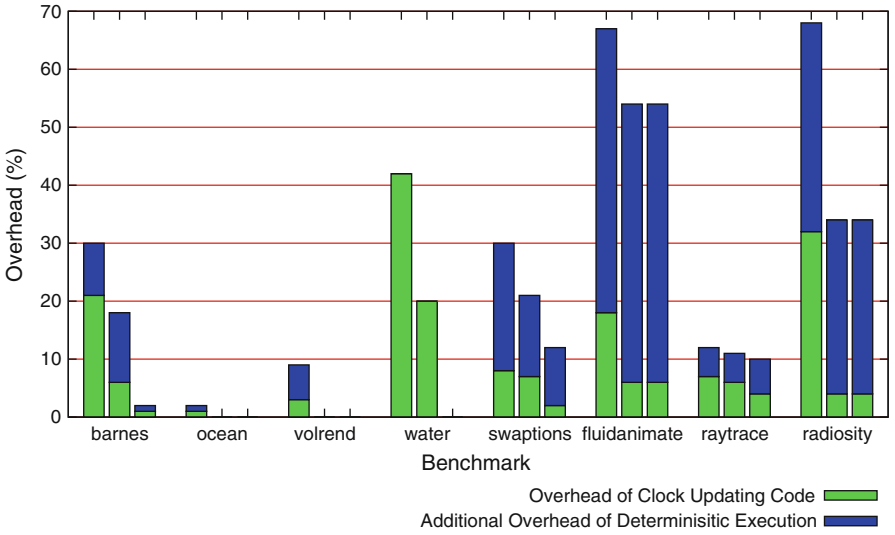

Fig. 10 Overheads

is shown combined with optimization 1 in the table, because some loops in the benchmarks consist of clocked functions, and without adding optimization 1, the impact of optimization 4 could not be seen in such cases. Overall, we see that all optimizations work to reduce the clock updating overhead as well as the deterministic execution overhead. However, from the *average* column, we can see that while optimization 2 reduced the clock update overhead more than optimization 4 and 1 combined, the later reduced the overall time, including deterministic execution more than optimization 2. The reason for this is discussed next.

5.2 Effect of updating clocks ahead of time

From Table 1, we can see that optimization 4 and 1 combined reduce the overall deterministic execution time more than optimization 2. This is even when optimization 2 reduced the clock updating overhead more than optimization 4 and 1 combined. The reason for this is because updating clocks ahead of time reduces waiting time of a thread which is in the process of acquiring a lock, as the clocks of other threads progresses more quickly in this way (even before execution of some instructions), thus allowing a waiting thread's clock to reach the minimum global more quickly. This effect is most pronounced for *swaptions* and *radiosity*. For *swaptions* for example, clock updating overhead with optimization 2 is 7 % while overall deterministic execution overhead with Optimization 2 is 29 %. On the other hand, with optimization 4 and 1 combined, the clock updating overhead is 3 % while overall deterministic overhead is only 13 %, which means an increase of only 10 % overhead over clock updating overhead as compared to an increase of 22 % with optimization 2. Similarly, for *radiosity*, there is an increase of 26 % (46–20 %) overhead over clock updating overhead with optimization

4 and 1 combined as compared to an increase of 48 % (57–9 %) with optimization 2. This is because optimizations 4 and 1 can more aggressively increment clock ahead of time as compared to optimization 2, as optimization 4 and 1 work at function and loop levels, whereas optimization 2 works only at basic blocks level.

6 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 2 %, whereas the overall overhead including deterministic execution is 14 % for selected benchmarks. This is an improvement over our previous work [11], with which on average, the overhead of inserting clock updating code is 6 %, while overall overhead including deterministic execution is 20 %.

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

References

1. Basu A, Bobba J, Hill MD (2011) Karma: scalable deterministic record-replay. In: ICS '11. ACM, New York, p 359–368
2. Bergan T, Anderson O, Devietti J, Ceze L, Grossman D (2010) Coredet: a compiler and runtime system for deterministic multithreaded execution. SIGARCH Comput Archit News 38:53–64
3. Bienia C, Kumar S, Singh JP, Li K (2008) The parsec benchmark suite: characterization and architectural implications. In: PACT '08. ACM, New York, p 72–81
4. Devietti J, Lucia B, Ceze L, Oskin M (2009) Dmp: deterministic shared memory multiprocessing. In: ASPLOS '09. ACM, New York, p 85–96
5. Edwards SA, Tardieu O (2005) Shim: a deterministic model for heterogeneous embedded systems. In: EMSOFT '05. ACM, New York, p 264–272
6. Hower DR, Hill MD (2008) Rerun: exploiting episodes for lightweight memory race recording. In: ISCA '08. IEEE Computer Society, Washington, DC, p 265–276
7. Hower D, Dudnik P, Hill M, Wood D (2011) Calvin: deterministic or not? free will to choose. In: HPCA '11. p 333–334
8. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO' 04. Palo Alto
9. Lee D, Wester B, Veeraraghavan K, Narayanasamy S, Chen PM, Flinn J (2010) Respec: Efficient online multiprocessor replay via speculation and external determinism. In: ASPLOS'10. p 77–89
10. Mushtaq H, Al-Ars Z, Bertels K (2011) Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. In: IDT '11. p 12–17
11. Mushtaq H, Al-Ars Z, Bertels K (2012) DetLock: portable and efficient deterministic execution for shared memory multicore systems. In: MuCoCoS '12, Salt Lake City
12. Nethercote N, Seward J (2007) Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 2007 programming language design and implementation conference

13. Olszewski M, Ansel J, Amarasinghe S (2009) Kendo: efficient deterministic multithreading in software. SIGPLAN Not 44:97–108
14. Thies W, Karczmarek M, Amarasinghe SP (2002) Streamit: a language for streaming applications. In: CC '02. Springer-Verlag, London, p 179–196
15. Tongping Liu EDB, Curtsinger Charlie (2011) Dthreads: efficient deterministic multithreading. In: SOSP '11
16. Weaver V, Dongarra J (2010) Can hardware performance counters produce expected, deterministic results? In: FHPM '10. Atlanta
17. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A (1995) The splash-2 programs: characterization and methodological considerations. SIGARCH Comput Archit News 23:24–36