Replay Debugging for Multi-threaded Embedded Software

Yann-Hang Lee[1], Young Wn Song[1], Rohit Girme[1], Sagar Zaveri[1], Yan Chen[2]

[1]Computer Science and Engineering Department
Arizona State University, U.S.A.

[2] The School of Information Science and Technology,
Xiamen University, P.R. China

Abstract

The non-deterministic behavior of multi-threaded embedded software makes cyclic debugging difficult. Even with the same input data, consecutive runs may result in different executions and reproducing the same bug is itself a challenge. Despite the fact that several approaches have been proposed for deterministic replay, none of them attends to the capabilities and functionalities that replay can comprise for better debugging. This paper introduces a practical replay mechanism for multi-threaded embedded software. The Replay Debugger, based on Lamport clock, offers a user controlled debugging environment in which the program execution follows the identical partially ordered happened-before dependency among threads and IO events as that of the recorded run. With the order of thread synchronizations assured, users can focus their debugging effort in the program behavior of any threads while having a comprehension of thread-level concurrency. Using a set of benchmark programs, experiment results of a prototyped implementation show that, in average, the software based approach incurs a small probe effect of 3.3% in its record stage.

Keywords – embedded system, replay debugging, multi-thread, partial order, Lamport clock.

1 Introduction

Debugging has been an imperative step to ensure the correctness of software programs. However, one of the survey data, in a 2002 NIST report [1], suggests that an average bug found in post-product release takes 15.3 hours to fix. This is quite costly in terms of engineer's time as well as the impact to the software products and users.

Often the very first step in debugging is to reproduce the failed execution. In the cyclic debugging process, breakpoints are set in the program and the application is re-run. Thus, the cause of the failure can be observed. For sequential programs the process is pretty straightforward – as long as each execution is deterministic and repeatable, the cause of the observed failure can be reproduced and identified. On the contrary, it is challenging to debug embedded software which is often structured with multiple threads to process concurrent IO events and application tasks. There are significant interactions between these threads as they need to pass control signals, application status, intermediate computation results, and to share resources. This reproduction of the identical execution behavior under this software structure and with thread interaction becomes extremely difficult since

- Different ordering of thread interactions and timing dependency may introduce non-determinism in the execution of concurrent threads, and
- The system interacts with, and is dependent of, an external real-time context.

It may be claimed that the existing debugging tools, such as GDB [2], TotalView [3], and IDB [4], can be used to debug multi-threaded programs. Although the tools include the functions of setting up breakpoint, single stepping and monitoring at thread level, they don't ensure the reproduction of execution behavior that is required in cyclic debugging. For instance, when a thread reaches a breakpoint while other threads are running, we may choose the "stop the world/all threads run" process control model in Intel's IDB or the "non-stop" mode in the release of GDB 7.1 in which only a subset of threads are suspended. However, there is no support to coordinate the arrivals of external events or the execution progress of running threads. It is the developers' responsibility to feed external signals synchronously with thread execution during the debugging process. This may become extremely troublesome, if not intractable, when dealing with real-time control signals, video/audio streaming data, and internet packets.

Due to the impediment of using cyclic debugging for multi-threaded embedded software, engineers are forced to look into the trace data from a failed run and to understand thread dependency. To pinpoint any specific problems, they manually create execution sequences to mimic the behavior of the failed run. This tactic is certainly problematical and time consuming. To deal with the inherent complexity of debugging such applications, approaches have been developed and published in last few years [5]. Out of those mechanisms, one of the most well known approaches is deterministic replay or trace based replay [8], [9], [12]-[28]. It is based on recording information that can be played again in a deterministic way during replay phase. However so far none of them have talked about the capabilities and functionalities that replay phase can provide to users.

In this paper, a Replay Debugger based on Lamport clocks [6] is designed for multi-threaded embedded software. The debugger considers a multithreaded execution as a partially ordered sequence of interactions and reproduces the sequence in cyclic debugging process. Hence, the developers can inspect the execution steps of each thread as well as the interactions among threads and external world. They will not be distracted by context switches or the physical time instants at which events and interactions occur. As a consequence, the debugging process can be focused in any single thread while the execution of other threads is coordinated precisely to reproduce identical software behavior.

The Replay Debugger is implemented as a record/replay framework layered between multithreaded applications and operating system kernel. The design criteria is to minimize any instrumentation overhead (i.e., probe effect) in record stage and to maximize observability and analyzability in replay stage. The record framework is a wrapper for IO and IPC (inter-process communication) library calls. During recording, the framework intercepts any IPC and device IO function calls and traces the returned code, data, and error

code into a log before returning to the application. Then, the happened-before relationships among the execution of multiple threads and IO events can be constructed. During replaying, the concurrent execution can be repeated and thread execution can be controlled individually following the partial order of thread interactions. Moreover, to make the Replay Debugger more practical, the proposed replay mechanism has been incorporated into GDB with an extended debugging interface for thread control and observation. It provides a GUI to user that shows the threads and various events that happened during the record phase. The users are allowed to use all GDB commands such as single stepping, breakpoints, tracepoints, and status views. They can also step over the individual thread interaction events and control the execution order of concurrent threads.

There are three contributions of our Replay Debugger:

1. Both IPC and IO events are considered during the record and replay.
2. It provides two different debugging modes, named Replay Minima and Replay Maxima, for users to control over a debugging process with the deterministic order of execution.
3. It is practical and has been incorporated into GDB with an extended debugging interface for thread control and observation.

The rest of the paper is organized as follows. In Section 2, a concise survey of related works is described. Section 3 presents the design of our Replay Debugger for multi-threaded embedded software. The implementation details are explained in Section 4. In Section 5, performance evaluation is presented, followed by a conclusion and the future work.

2 Related Works

With respect to related work in the field of replay debugging of concurrent and real-time systems, the main issue with record/replay approaches is what information should be traced during the initial record phase? On the one hand, enough information about the execution has to be generated for a faithful re-execution of the program. On the other hand, the amount of information traced and the computational overhead should be limited as much as possible in time and in space in order to avoid any probe effect [7].

On general topic of replay, some researches have been relying on special hardware to reproduce interrupts and task switches [8][9]. The software instruction counter approach [10] records the backward branches and has been used to identify the exact location at which an event of interest (e.g. interrupt) occurs. The approach was adopted for log-based recovery to force the replay of asynchronous events at the same execution points [11]. In addition, Hill et al. present their Flight Data Recorder (FDR) [12] as an intrusive hardware that facilitates debugging of data races in multiprocessor systems.

On software based replay mechanisms, Carver et al. provided an approach to reproduce the rendezvous of Ada tasks [13]. The approach can only replay concurrent program execution events like rendezvous, but not real-time specific events like scheduled preemptions, asynchronous interrupts or mutual exclusion operations [14]. LeBlanc and Mellor-Crummey proposed a method for recording information during run-time and using this information to reproduce the exact behavior of the execution off-line [15]. This method, called Instant Replay, allows cyclic debugging techniques to be used for non-deterministic systems. It is targeted at coarse grained operations and traces all these operations. It does not use any technique to either reduce the size of the trace files or limit the perturbation introduced. Replay mechanism for message passing systems is discussed in ROLT [16]. For shared memory computers, Netzer [17] introduced an optimization technique based on vector clocks. As the order of all memory accesses is traced, both synchronization and data races will be replayed. However, no implementation was ever proposed (of course, the overhead would be huge if all memory accesses are traced). A more elaborate discussion on debugging parallel programs can be found in [18].

Montesinos et al. introduced a software-hardware interface for deterministic replay of multiprocessors [19]. This approach, called Capo, separates the responsibilities of the hardware and software components of the replay system. During the recording phase, the software logs all replay sphere inputs into a *Sphere Input Log*, while the hardware records the interleaving of the replay sphere threads into an *Interleaving Log*. During the replay phase, the hardware enforces the execution interleaving encoded in the *Interleaving Log*, while the software provides the entries in the *Sphere Input Log* back into the replay sphere and squashes all outputs from the replay sphere. The approach combines the performance of hardware-only schemes with the flexibility of the software-only ones.

Park et al. [20] proposed a probabilistic replay via execution sketching to help reproduce concurrency bugs on multi-processors. This approach recorded only partial execution information during the production run, and relied on an intelligent replayer during diagnosis time to systematically explore the unrecorded non-deterministic space and reproduce the bug. With only partial information, the approach has low recording overhead, but it may require more than one coordinated replay run to reproduce a bug. Moreover, as the approach exhaustively search the unrecorded non-determinism space to find a bug-producing combination, it might be difficult to reproduce an occurred bug within an acceptable time limit.

The concept of deterministic replay was applied for debugging real-time systems. In [21], a general-purpose processor is dedicated to monitoring on each multiprocessor. The monitor can observe the target processors via shared memory. The target systems software is instrumented with monitoring routine, by means of modifying system service calls and interrupt service routines. Their basic assumption about having a distributed system consisting of multiprocessor nodes makes their approach less general.

A similar technique of deterministic replay to debug distributed real-time systems is discussed in [22]. The method starts by identifying and recording significant events, together with the time at which they occur, in the execution of the sequential program. To reproduce the run-time behavior during debugging, all inputs and outputs are replaced with recorded values. All transfers of control, accesses to critical regions, releases of higher priority tasks, and preemptions by interrupts are dictated by the recorded

timestamps. A similar replay debugging using Time machines is discussed in [23]. As the authors claim, it is the first method for deterministic replay of single tasking and multitasking real-time systems using standard off-the shelf debuggers and real-time operating systems and without instruction counters support for replay thus making it compiler and RTOS independent.

Probably the most similar approach to the replay mechanism of the proposed debugger is RecPlay [24][25] which replays the synchronization operations, while detecting data races. It is also based on Lamport's happens-before relation [6]. By checking the ordering of all events and monitoring all memory accesses, data races can be detected for one particular program execution. Some prior work based on Lamport clock, i.e., ordering based replay method for shared memory programs is discussed in [26]. This mechanism lists the advantages of using Lamport clock to generate partial ordering. It produces small trace files and is less intrusive. Moreover, the method allows one to use a simple compression scheme [27] which can further reduce the trace files. Different from the RecPlay, our Replay Debugger focuses on debugging techniques for multi-threaded embedded software based on the record/replay framework. It is integrated with GDB and supplies debugging functionalities for users to control over a debugging process with the deterministic order of execution.

Another approach based on Lamport clock is taken by a race detector Eraser [28]. It goes slightly beyond the works based on the happened-before relation. Eraser checks that a locking discipline is used to access shared variables: for each variable it keeps a list of locks that were hold while accessing the variable. Each time a variable is accessed, the list attached to the variable is intersected with the list of locks currently held and the intersection is attached to the variable. If this list becomes empty, the locking discipline is violated, meaning that a data race occurred. The most important problem with Eraser is that its practical applicability is limited in that it can only process mutex synchronization operations and in that the tool fails when other synchronization primitives are built on top of these lock operations.

The record/replay approaches have become acceptable for debugging in practice. Recently, VMware Workstation 6.5 includes the feature of replay debugging for C/C++ developers using Microsoft Visual Studio. The tool records program execution via VMWare's virtualization layer and the replay is guaranteed to have instruction-by-instruction identical behavior. It also includes a feature of simulating reverse execution of the program, making it easier to pin point the origin of a bug.

GDB record patch, available at [29], enables a reversible debugging function in GDB. By disassembling the instruction that will be executed, the record patch saves the memory and register content before they are modified. The recorded information is then used to restore the processor state during reverse execution.

In conclusion, although there are many related works on the record/replay of concurrent and real-time systems, there still has no practical and convenient mechanism or tool for users to debug the multi-threaded embedded software. In this paper, we propose a practical Replay Debugger which offers a user controlled debugging environment. With the Replay Debugger, users can focus their debugging effort in the program behavior of any threads while having a comprehension of thread-level concurrency.

3 Design of the Replay Debugger

To facilitate execution replay, we will need to record the execution paths and input events of the original run. Hence, instrumentation must take place in a record phase and must have a minimal probe effect. To replay single threaded programs, we will need to supply identical input values read from device drivers and system calls (e.g. timer function). The execution can then be reproduced given the sequential nature of the programs. On the other hand, for multi-threaded programs, threads can interact with each other by invoking inter-process communication operations. To reproduce the execution, these invocations must take place in the identical order as the original run. In this paper, we assume these interaction and input events are invoked via system calls, including inter-process communication primitives and device drivers. Shared resources are protected by synchronization primitives such as locks or semaphores and there is no data race condition. If a shared resource is not properly protected and causes a race condition during record phase, deterministic replaying the program will reveal the cause of errors.

3.1 Definitions

As mentioned above, for a reproducible replay, we need to record thread interaction and input read events (or simply "thread interaction events") during the original thread execution. In the following, several related definitions are given.

Definition 1. A thread interaction event is a 4-tuple:

$$E = <type, T_{id}, O_{id}, t> \quad (1)$$

where *type* is an event type of E, e.g., read(), sem_wait(), sem_post(), etc.; T_{id} is the thread invoking event E; O_{id}, identifies the synchronization and communication object which E operates on, e.g., semaphore, message queue, mutex lock, IO file, etc.; and t is the Lamport clock timestamp [6] of E.

Definition 2. The thread interaction events operated on the same synchronization and communication object are collected into an event set. The set is called *object event set* and is denoted as OES_{id} for object $O_{id.}$

Definition 3. The thread interaction events in the same thread are collected into an event set. The set is called *thread event set* and is denoted as TES_{id} for thread T_{id}.

We use the happened-before relation [6] to maintain a partial order between thread interaction events. The happened-before relation between events a and b is denoted by "$\rightarrow$", i.e., if a happened before b, it is represented as $a \rightarrow b$. The Lamport clock [6] is used to identify events in reconstructing a partial order graph in the replay phase. A Lamport clock is maintained for each O_{id} or T_{id} and is equal to the Lamport clock of the most recent event happened on O_{id} or invoked by T_{id}. $LC(a)$ is a function that returns a Lamport clock timestamp taking a parameter as E_i, O_{id} or T_{id}.

In addition to the happened-before relation we define an *immediate happened-before* relation, denoted by "$\mapsto$", for two successive thread interaction events as follows:

Definition 4. For two thread interaction events E_b and E_a, there is an immediate happened-before relation $E_b \mapsto E_a$, if:

1) $E_b, E_a \in Set_{id}$ where Set_{id} is OES_{id} or TES_{id}, and

2) $\forall E_i \in Set_{id}$ $(i \neq b)$ that satisfies

$$LC(E_a) - LC(E_i) > LC(E_a) - LC(E_b) > 0 \qquad (2)$$

According to the expression (2), E_b is the last event before E_a in the Set_{id}.

In the subsequence discussion, a *timestamp* for a thread interaction event is denoted as an instant of Lamport clock when the thread interaction event occurs and an *event* is denoted as a thread interaction event.

3.2 Record

In the record phase of Replay Debugger, the happened-before relations between events are captured by chains of immediate happened-before relations. The program execution is reproduced by following the execution orders of events represented in the immediate happened-before relations during the replay.

A wrapper function is provided for each event type, and recoding of an event E_i is invoked a thread T_{id} and operates on an object O_{id}, the computation of their timestampts follows the Lamport clock algorithm [6] as

$$\begin{aligned} LC(E_i) &= max(LC(T_{tid}), LC(O_{oid})) + 1 \\ LC(T_{tid}) &= LC(E_i) \\ LC(O_{oid}) &= LC(E_i) \end{aligned} \qquad (3)$$

Then, the 4-tuple <*type*, T_{id}, O_{id}, *t*>, as the unique representation of E_i, is saved into a log file. If E_i is an input read event, the input data is saved too.

Note that $LC(O_{oid})$ is initialized to 0 when the object is created, e.g., open() for files, sem_open() for semaphores, and mq_open() for message queues, in Linux. Also, when a new thread is created, we add a thread creation event E_c as the first event to the new thread and the timestamp of the new thread is initialized to $LC(E_c)$. If E_i is a thread exit event of a joinable thread, E_i is added as the last event in T_{tid} and $LC(T_{id})$ is kept until the join action is invoked.

After the recording stage, a partial ordering of events is constructed using the 4-tuple event information collected. The partial ordering is represented by a graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{L})$, where $\boldsymbol{V}$ is a set of events recorded, $\boldsymbol{L}$ is a set of edges, and each $l=(E_1, E_2) \in L$ denotes the immediate happened-before relation between E_1 and E_2. The algorithm to calculate $\boldsymbol{L}$ is shown in Figure 1. Note that the steps 3 and 4 can be done easily once the events are sorted according to their timestamps.

```
1: CalculateE()
2:   For each event E_i =<type, T_id, O_id, t> in V,
3:     Find E_i1 such that E_i1 ∈ TES_tid and E_i1 ↦ E_i
4:     Find E_i2 such that E_i2 ∈ OES_oid and E_i2 ↦ E_i
5:     L = L ∪ (E_i1, E_i)
6:     L = L ∪ (E_i2, E_i)
```

Figure 1 Algorithm of Edge Calculation

Figure 2 shows an example program, and the program has few possible execution paths. Figure 3 shows one possible program execution with a partial order graph calculated from timestamps.

```
1:  int shared = 1;
2:  void thread2(int *arg){
3:      sem_wait(&sem);
4:      shared = shared + (*arg);
5:      sem_post(&sem);
6:  }
7:  void thread3(int *arg){
8:      sem_wait(&sem);
9:      shared = shared * (*arg);
10:     sem_post(&sem);
11: }
12: int main(void){
13:     pthread_t t2, t3;
14:     char buf[10];
15:     int arg;
16:
17:     sem_init(&sem, 0, 0);
18:     read(0, buf, 10);
19:     arg = atoi(buf);
20:
21:     pthread_create(&t2, NULL, (void *)thread2, &arg);
22:     pthread_create(&t3, NULL, (void *)thread3, &arg);
23:     sem_post(&sem);
24:
25:     pthread_join(t2, NULL);
26:     pthread_join(t3, NULL);
27:     return 0;
28: }
```

Figure 2 An example program

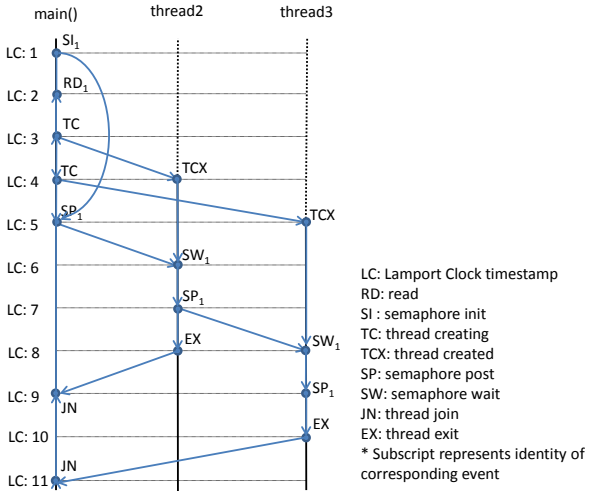

Figure 3 A partial order graph for one possible execution path

3.3 Replay

The deterministic replay is achieved by following an order represented in the partial order graph $\boldsymbol{G}$. To execute an event, we should execute all events that are happened before it. This is done by following edges of the partial order graph backward and executing all the events happened before the target event.

For an event $E_i \in TES_{tid}$, let *ProceedToEvent*(E_i) be a schedule algorithm to run the thread T_{id} from its current program counter to the completion of an event E_i. The algorithm can be realized by recursive calls as depicted in Figure 4, where the Lamport clock for threads and events are managed following the same steps in Eq. (3).

```
1:  ProceedToEvent(E_i)
2:    // assume that E_i ∈ TES_tid, i.e. to be invoked by T_id
3:
4:    while LC(T_id)<LC(E_i) do
5:          run thread T_id until the next event E_k
6:          find the event E_j where E_j ↦ E_k
7:                  and E_i ∉ TES_tid
8:          ProceedToEvent(E_j)
9:          execute E_k
10:   return
```

Figure 4 Algorithm of event execution

Given a partially ordered happened-before graph, there are many different execution orders. In addition to user controlled execution ordering, Replay Debugger provides two debugging modes: Replay Minima and Replay Maxima. In Replay Minima mode, the events of the selected thread will proceed first. The events of other threads will be carried on only if they are happened before the executing event of the selected thread. On the other hand, for Replay Maxima, the events of all other thread can proceed first until there is happened-before dependency upon the current execution of the selected thread. In this case, the events of the selected thread will be advanced in order. The two replay modes are useful to illustrate the dependency between threads. Let's assume a breakpoint BR_i is set along a thread T_i's execution path. When the thread execution is suspended after reaching the breakpoint under the replay modes:

1) *Replay Minima:* Only a limited subset of events in all other threads is executed. This subset represents the minimal amount of execution that must be done for T_i to reach BR_i. Any causes that may result in an incorrect state status at BR_i would have been triggered by the execution of the events in the subset.
2) *Replay Maxima:* A maximal set of events has been carried out as thread T_i is suspended at BR_i. This set represents how far the impact of T_i's current execution can stretch to if there is no further execution in T_i. So, if thread T_i's execution is bug-free up to BR_i and no bugs in other threads, then the execution of this maximal set of events should be error-free.

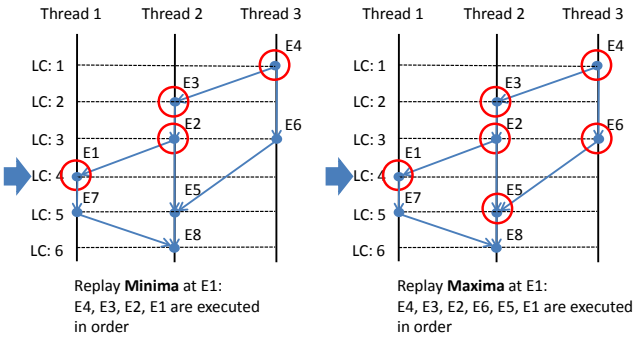

Figure 5 Replay mode example

Two different debugging scenarios by Replay Minima and Replay Maxima are shown in Figure 5. The replay is implemented by differentiating event statements that we record from other normal statements. The corresponding replay algorithm of the current debugging mode is performed at the event statements. The *ProceedToEvent* algorithm, shown in Figure 4, implements the Repaly Minima mode. Other program statements that we do not record are preceded just as normal debugging statements without applying the algorithm.

The external IO data saved during record phase is also replayed. This guarantees that the program execution during replaying receives the same input data as in the recording stage. Note that in user's point of view all the details about deterministic replay are hided and he/she can concentrate on debugging without worrying about switching threads and feeding IO data.

4 Implemenation

The current implementation of Replay Debugger is built on POSIX APIs in Linux environment. The architecture of Replay Debugger is shown in Figure 6. It includes a library of wrapper functions. The actual system calls are substituted with the wrapper functions during compile time. In the record phase, the wrapper functions log information needed in building a partial order graph including Lamport clocks and save external input contents into files. The wrapper functions in the replay phase read external input contents from the files saved in the record phase. In the current implementation, it is the GDB thread module that performs a deterministic replay using the partial order graph built in the record phase. A selected subset of system calls are implemented at this stage. Further extension will include mmap device, signal handling, time function, etc.

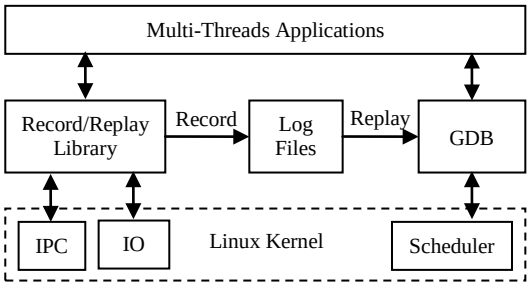

Figure 6 The architecture of Replay Debugger

4.1 Replay Scheduling Using GDB

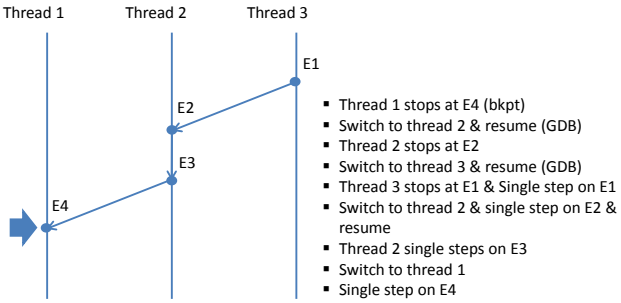

Figure 7 A scheduling example using GDB

The GDB thread module is modified to schedule thread executions according recorded partial order. On start of a GDB session, Linux is prevented from scheduling threads using the GDB command *"set scheduler-locking on"* [33] and a breakpoint is set at each event for checking happened-before relations. The GDB command *"thread thread-num"* [33] is used for switching to a thread numbered *"thread-num"*. When GDB stops at an event *E1*, threads are scheduled as follows,

- If there is no immediate happened-before event before *E1*, then single step on *E1* and resume;
- Else find *E2* such that $E2 \mapsto E1$, switch to the thread that *E2* belongs to and resume the thread.

Figure 7 shows an example of thread scheduling.

4.2 GUI Support

We have also implemented a graphical environment for our Replay Debugger in the form of pluggable components called Eclipse [30] plug-ins. The current implementation is based on Eclipse for Linux. Eclipse plug-ins are components that provide certain types of services within the context of the Eclipse workbench. They add to the C/C++ Development Tools (CDT) that provides a fully functional C and C++ Integrated Development Environment (IDE) for the Eclipse platform.

Support for our Replay Debugger functionality in Eclipse can be categorized into three parts.

4.2.1 Support for the Record

To enable recording, applications are built using the record library so that we can record its behavior. Our plug-in provides a new project type "RecordReplay" which is shown in Figure 8. It has two new configurations "Record" and "Replay". The user can avail of the Record and Replay services in GDB by selecting this project type.

Since our plug-in is a part of Eclipse, the new project wizard includes the project-type and configuration definitions from our manifest files along with other manifest files to populate the list of choices presented to the user. Contents of the build property page for a project are created by examining the tool-chains, tools, option categories, and options defined for the current configuration. Therefore, if the user wants to use the Replay Debugger functionality, the "RecordReplay" type of project should be created from the list of project types. After selecting the project type, the user then selects "Record" and "Replay" configurations for his new project.

4.2.2 Support for the Replay

In Eclipse framework, the C/C++ Debugger Interface (CDI) was created by Eclipse/CDT developers, so that CDT can access external debuggers. Similarly, the Machine Interface (MI) [30] was created by GDB developers, so that external applications/software can access the GDB.

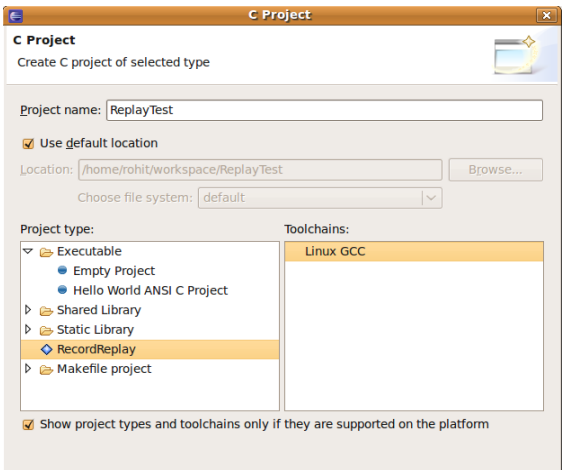

Figure 8 A new project type "RecordReplay" in the plug-in

For our purpose, we added MI commands to the GDB/MI interface which are used by our code in CDT to call any particular replay commands. So far replay commands of Start Replay, Thread Information, Set Minima Mode, Set Maxima Mode and Set LC Breakpoint have been added to the CDT. Start Replay initializes the replay module in GDB; Set Minima Mode sets minima mode for execution of events in GDB; similarly Set Maxima Mode configures the maxima mode for event execution in GDB. Set LC Breakpoint initiates a breakpoint at the thread clock entered by user. Thus, debugging can follow the order of Lamport clock. Thread Information displays a table similar to the Replay Graph in the Eclipse Console. This table shows the different events for corresponding threads which have not yet been executed. To call the commands added to CDT, we contributed menu/buttons to the workbench using commands in Eclipse, which is shown in Figure 9. A command in Eclipse is a declarative description of a component. The behavior of a command is defined via handlers. Where and how should the command be included in the UI is defined using a location Uniform Resource Identifier.

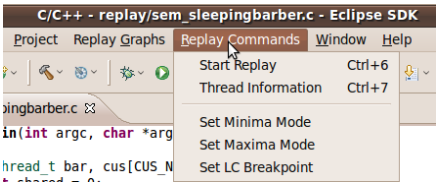

Figure 9 Menu of replay commands

4.2.3 Support for displaying threads and events

Displaying the different threads and events of an application being debugged gives the user a clear picture of what is going on. We have used Swing and Abstract Window Toolkit provided by Java for our purposes.

The memory log obtained after recording the application contains the information about the threads and events of the program. Information from this log is used to draw the display.

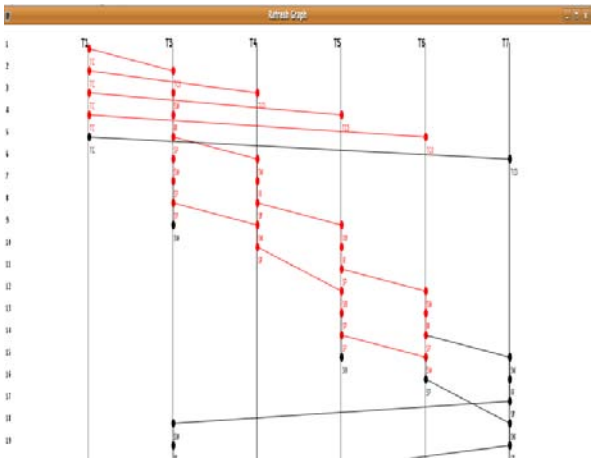

Figure 10 Displaying threads and events

Figure 10 shows an example of the diagram displaying threads and events. The display provides us with

information like number of threads, events occurred along with their types (thread created, semaphore/mutex taken or released etc), thread clock at which the event occurred and so on. An additional functionality provided is each event on the display maps to a particular line in the application program.

5 Performace Evaluation

The performance of Replay Debugger is measured on a Linux 2.6.28 system with Intel(R) Core 2 Duo CPU T6600 @ 2.20GHz and 3GB RAM. We use PAPI Library [31] to measure the instruction counts of actually system calls and the wrapper calls. The results are shown in TABLE I. "*mq_send*" and "*mq_receive*" events are measured with a message length of 40 bytes; "read" and "fread" events are measured with reading 1024 bytes data. The instruction counts of wrapper calls are larger than the original calls, as each wrapper call should do additional work: update the Lamport clock, write log in current memory buffer, and switch buffer and signal the "dump thread" if current buffer is full. Moreover, the wrapper calls for "read" and "fread" events should also create log files to save the data read, so the instruction counts of these two wrapper calls are larger than other wrapper functions.

TABLE I Instruction counts of actually system calls and the wrapper calls

System Call	Original	Wrapper
sem_wait	18	514
sem_post	16	515
mq_send (40 bytes)	81	537
mq_receive (40 bytes)	82	535
pthread_mutex_lock	40	574
pthread_mutex_unlock	42	780
pthread_create	949	1688
read (1024 bytes)	94	3940
fread (1024 bytes)	575	4446

TABLE II and TABLE III give an idea of the overhead caused by Replay Debugger during record phase for programs from the LTP benchmark suite [32]. TABLE II shows static information of benchmark programs, including the number of threads and the total number of interaction events during executions with the same input data. TABLE III shows that the overhead during the record phase for the benchmark programs ranges from 0.019% to 9.264% with the average of 3.295%. The reason of the low overhead is that Replay Debugger effectively succeeds in greatly reducing the number of synchronization operations that has to be stored on the log file. Apparently, optimizations can also be done to further reduce the overhead of wrapper function and logging operation. For instance, the code to prevent preemption during the wrapper function can be eliminated if the wrapper is moved to the kernel.

TABLE II Static data of benchmark programs

Program	Thread Number	Total Event Number
multi_con_pro	500	200200
multi_send_rev_1	80	3242
multi_send_rev_2	200	9486
sem_philosopher	5	610
sem_readerwriter	200	1002
sem_sleepingbarber	101	1208
read_send_rev	4	12011
pthread_mutex_unlock	60	3720

TABLE III The overhead during record for the benchmark programs

Program	Normal Runtime (s)	Record Runtime (s)	Overhead
multi_con_pro	75.144	81.460	8.405%
multi_send_rev_1	2.591	2.831	9.264%
multi_send_rev_2	6.181	6.392	3.414%
sem_philosopher	51.812	52.414	1.162%
sem_readerwriter	2.007	2.008	0.050%
sem_sleepingbarber	5.502	5.503	0.019%
read_send_rev	9.556	9.939	4.008%
pthread_mutex_unlock	30.130	30.142	0.040%
Average			3.295%

6 Conclusion

In this paper we have presented a practical approach to replay-based debugger for multithreaded embedded software. The proposed debugger uses the Lamport's happened-before relation to establish a partial order between thread interaction and IO events. The partial order is then applied to guide the thread execution during debugging. The enhanced replay functionalities like Replay Minima and Replay Maxima on events provide simple control over a debugging process with the deterministic order of execution. Moreover, it has been successfully integrated with GDB, and we have also wrapped our tools for the debugger in the form of pluggable components in Eclipse.

As a further step, we would like to improve our record and replay library to support most kinds of thread interaction events, asynchronous transfer (e.g. signal and watchdog timer), and mmap devices. In addition, we would like to enhance the functions of our GUI and the Eclipse plug-in based on user feedbacks. The current implementation of the proposed tool is available at *http://rts.lab.asu.edu/download*.

7 Acknowledgments

This work was supported partially by a grant from the NSF Industry/University Cooperative Research Center (I/UCRC) on Embedded Systems at Arizona State University.

References

[1] NIST report, "The economic impacts of inadequate infrastructure for software testing," Technical report, U.S. Department of Commerce, May 2002.

[2] R. Stallman, R. Pesch, S. Shebs, et al., "Debugging with GDB (ninth edition, for GDB version 7.1)," *Free Software Foundation*, 2010.

[3] TotalView Technologies, "TotalView graphic user interface reference guide: version 8.8," 2010.

[4] R. M. Albrecht, "White Paper: Intel Debugger for Linux*," *Intel Corporation*, 2009.

[5] J. Engblom, "Debugging real-time multiprocessor systems," *Embedded Systems Conference*, Silicon Valley, 2007.

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, Vol.21(7), pp: 558-565, 1978.

[7] J. Gait, "A probe effect in concurrent programs," *Software Practice and Experience*, Vol.16(3), pp: 225-233, 1986.

[8] H.-Y. Chen, J.P.P Tsai, K-Y. Fang, and D. Bi, "A noninterference monitoring and replay mechanism for real-

time software testing and debugging," *IEEE Transactions on Software Engineering*, Vol.16(8), pp: 897-916, 1990.

[9] D. R. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas, "Two hardware-based approaches for deterministic multiprocessor replay," *Commnications of the ACM*, Vol.52(6), pp: 93-100, 2009.

[10] J. Mellor-Crummey and T. LeBlanc, "A software instruction counter," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp:78-86, Boston, Massachusetts, Unite State, April, 1989.

[11] H. Slye and E.N. Elnozahy, "Supporting nondeterministic execution in fault-tolerant systems," *Proceeding of International Symposium on Fault-Tolerant Computing*, pp: 250 - 259, Sendai, Japan, June, 1996.

[12] M. D. Hill, M. Xu, and R. Bodik. "A 'flight data recorder' for enabling full-system multiprocessor deterministic replay," *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp: 122-133, San Diego, California, Unite States, June, 2003.

[13] R. H. Carver, K. C Tai, and E. E. Obaid, "Debugging concurrent ada programs by deterministic execution," *IEEE Transactions on Software Engineering*. Vol.17(1), pp: 45-63, 1991.

[14] F. Zambonelli and R. Netzer, "An efficient logging algorithm for incremental replay of message-passing applications," *Proceedings of the 13th International and 10th Symposium on Parallel and Distributed Processing*, pp: 392 - 398, San Juan, Puerto Rico, Unite States, April, 1999.

[15] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Transactions on Computers*, Vol.36(4), pp: 471-482, 1987.

[16] D. Kranzlmller and M. Ronsse, "Rolt(mp) - replay of Lamport timestamps for message passing systems," *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing*, pp: 87-93, Madrid, Spain, January, 1998.

[17] R. H. Netzer, "Optimal tracing and replay for debugging shared memory parallel programs," *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp: 1-11, San Diego, California, Unite States, May, 1993.

[18] J. Huselius, "Debugging parallel systems: a state of the art report," In MRTC Technical Report 63, Mlardalen University, Department of Computer Science and Engineering, September, 2002.

[19] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas, "Capo: a software-hardware interface for practical deterministic multiprocessor replay", *Proceedings of Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp: 73-84, Washington, DC, Unite States, March, 2009.

[20] S. Park, Y. Zhou, W. Xiong, Z. Yin, et al., "PRES: probabilistic replay with execution sketching on multiprocessors", *Proceedings of SOSP'09*, pp: 177-191, Big Sky, Montana, Unite States, October, 2009.

[21] P. Dodd and C. V. Ravishankar, "Monitoring and debugging distributed real-time programs," *Software Practice and Experience*, Vol.22(10), pp: 863-877, 1992.

[22] H. Hansson and Henrik Thane, "Using deterministic replay for debugging of distributed real-time systems," *Proceedings of 12th Euromicro Conference on Real-Time Systems*, pp: 265-272, Stockholm, Sweden, June, 2000.

[23] J. Huselius, A. Pettersson, H. Thane, and D. Sundmark, "Replay debugging of real-time systems using time machines," *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, Nice, France, April, 2003.

[24] K. D. Bosschere, M. Ronsse, and M. Christiaens, "Debugging shared memory parallel programs using record/replay," *ACM Future Generation Computer Systems*, Vol.19(5), pp: 679-687, 2003.

[25] M. Ronsse, M. Christiaens, and K.D. Bosschere, "Cyclic debugging using execution replay", *Proceedings of the International Conference on Computational Science*, pp: 851-860, San Francisco, California, Unite States, May, 2001.

[26] K. Audenaert, L. Levrouw, and J. Van Campenhout, "A new trace and replay system for shared memory programs based on Lamport clocks," *Proceedings of the Second Euromicro Workshop on Parallel and Distributed Processing*, pp: 471-478, Malaga, Spain, January, 1994.

[27] L. Levrouw, M. Ronsse, and K. Bastiaens, "Efficient coding of execution-traces of parallel programs," *Proceedings of the ProRISC / IEEE Benelux Workshop on Circuits, Systems and Signal Processing*, pp: 251-258, Utrecht, Holland, March, 1995.

[28] G. Nelson, P. Sobalvarro, T. Anderson, S. Savage, and M. Burrows, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.* Vol. 15(4), pp: 391-411, 1997.

[29] GDB record patch, http://sourceforge.net/projects/record/.

[30] Eclipse, http://www.eclipse.org/.

[31] PAPI, http://icl.cs.utk.edu/papi/.

[32] Linux Test Project, http://ltp.sourceforge.net/.

[33] GDB Manual, http://sourceware.org/gdb/current/onlinedocs/gdb/.