Dynamic Analysis of Embedded Software using Execution Replay

Young Wn Song and Yann-Hang Lee
Computer Science and Engineering
Arizona State University
Tempe, AZ, 85281
ywsong@asu.edu, yhlee@asu.edu

***Abstract*—For program optimization and debugging, dynamic analysis tools, e.g., profiler, data race detector, are widely used. To gather execution information, software instrumentation is often employed for its portability and convenience. Unfortunately, instrumentation overhead may change the execution of a program and lead to distorted analysis results, i.e., probe effect. In embedded software which usually consists of multiple threads and external inputs, program executions are determined by the timing of external inputs and the order of thread executions. Hence, probe effect incurred in an analysis of embedded software will be more prominent than in desktop software. This paper presents a reliable dynamic analysis method for embedded software using deterministic replay. The idea is to record thread executions and I/O with minimal record overhead and to apply dynamic analysis tools in replayed execution. For this end, we have developed a record/replay framework called P-Replayer, based on Lamport's happens-before relation. Our experimental results show that dynamic analyses can be managed in the replay execution enabled by P-Replayer as if there is no instrumentation on the program.**

Keywords – program dynamic analysis; probe effect; deterministic replay; profiling, embedded software

I. INTRODUCTION

Dynamic program analysis is widely used to collect execution information while programs are running. The approach is widely applied to aid optimization and debugging of the programs. For the collection and analysis of program execution, software instrumentation is inevitable unless hardware-assisted approaches are available. For instance, dynamic binary instrumentation tools such as INTEL PIN [17] and Valgrind [22] are most commonly used since they do not require source code and recompilation of program. However, instrumentation overhead from the tools is very high no matter how trivial the analysis is.

The instrumentation overhead can perturb the execution of a program and lead to different execution paths, and consequently misrepresent analysis results. This is so called the probe effect [10]. One example is the delayed executions of input events caused by the instrumentation overhead. Consequently, arriving external inputs may be lost or the deadlines for real-time applications may be missed. The instrumentation overhead may also lead to different order of thread operations on a shared resource and produce different execution results.

In embedded systems, applications run with multiple threads interacting with each other as they share resources. The execution of threads is often triggered by external inputs. Thus, program behavior will depend upon the time that input events arrive as well as the input value received. Also, it is a common practice to use asynchronous functions to increase response time. Hence, embedded software will be sensitive to probe effect caused by instrumentation overhead as the timing of input events and the order of thread execution can be easily deformed. On the other hand, the probe effect is a less concern in desktop applications which usually perform predefined work load (e.g., file input) with fixed loop counts and synchronous functions. Even if there were execution changes due to instrumentation overhead from analysis tools, analysis results would be amortized with repeated executions of the same code, e.g., consistent data races and call-path analysis results.

In Tables 1 and 2, the probe effect is illustrated in the execution of two embedded programs. The two programs are based on the class projects of Embedded Systems Programming in Arizona State University [5]. Each result is from the average of 10 runs. The first program is a QT [24] application which draws lines following the inputs of mouse movement. The program consists of three threads. The first thread receives mouse movement packets and sends them to a POSIX message queue (MQ). The second thread receives the input packets from the MQ and draws lines on a display device. The last thread performs a line detection algorithm with the received mouse inputs. We collected mouse movement at a normal sampling rate of 300 inputs per second and then fed the inputs to the application with variant speeds. If the first thread was delayed and was not ready to receive an input, we counted it as a missed input. The program is instrumented using two dynamic analysis tools, i.e., the cache simulator [12] using PIN and our implementation of the FastTrack data race detector [8]. The workload of the program is very light as it only spends less than 10% of CPU time. However, the instrumented execution may miss up to 45% of the inputs. The impact of probe effect caused by the instrumentation is obvious since analysis results may be misleading when the input data are missed.

The second program shown in Table 2 is a MQ test program with six threads and two MQs. The two sender threads send items to the first MQ and the two router threads receive the items from the first MQ and send them to the second MQ with timestamps. Finally, two receiver threads receive the items from the second MQ. We used asynchronous functions for queue operations and, if a queue is empty or full, the thread sleeps a fixed time interval and retries. We count the numbers of occurrences that the queues become empty or full as a way of measuring different program behaviors. In

Table 1. QT application with mouse inputs
(% of inputs missed out of 4445 mouse movement inputs)

inputs/sec	Native execution	PIN Cache	Race detector
150	0.0%	16.8%	0.3%
300	0.0%	36.1%	1.2%
450	0.0%	45.5%	1.9%

Table 2. POSIX Message Queue application
(# of Queue full/# of Queue empty)

Queue Length	Native execution	PIN Cache	Race detector
5	1.3/7.5	8.3/191.9	5.5/56.3
10	0.5/8	2.5/146.8	2.4/37.9

this program, there is no external environment affecting the program execution, but the execution is determined by order of thread executions on the shared MQs. As the results show, instrumentation overhead from the tools has changed the relative ordering of thread operations on the shared MQs which, in turn, leads to different status of the message queues.

The other concern, followed by the data shown in Tables 1 and 2, is that it will be very hard to know if there exists any probe effect on an instrumented program execution. If we take any serious measurement to find possible probe effect, the measurement itself would incur instrumentation overhead that can lead to execution divergence. Even if we know that there were some changes in program execution, we still would not be able to know how the changes affect the results of the analysis.

To make dynamic analysis tools as non-intrusive as possible, hardware-based dynamic analyses can be used. Intel Vtune Amplifier XE [13] exploits on-chip hardware for application profiling. ARM RealView Profiler [25] supports non-intrusive profiling using dedicated hardware or simulator. However, they are specific for performance profiling and do not support the analyses that require significant processing capabilities. Sampling based analyses [2, 9, 13, 19, 28] can also be used, but the analysis accuracy decreases when sampling rate is reduced to limit any measurement overhead.

In this paper, we present a dynamic program analysis method for embedded software using deterministic replay. The idea is to record thread executions and I/O events with a minimal recording overhead and to apply dynamic analysis tools on replayed executions. With the record/replay, dynamic analyses that were not feasible due to probe effect can be performed accurately. For this end we have developed a record/replay mechanism, called P-Replayer. P-Replayer is implemented as a record/replay framework based on Lamport's happens-before relation [14]. The design goal is to minimize recording overhead to prevent probe effect during recording and to have minimal disturbance on analyzed program executions during replaying. The contributions of this paper are:

1. We present a dynamic analysis method that makes analyses of a program feasible and faithful, and expedites the debugging and optimization process for embedded programs.
2. We present a record/replay framework that can be incorporated with dynamic analysis tools.
3. We show that for thread profilers the real execution time after removing measurement overhead can be accurately recovered with the record/replay framework.

The rest of paper is organized as follows. In the following section, the dynamic analysis method with deterministic replay is described. Section III presents the design and implementation of P-Replayer. In section IV, the performance evaluation of P-Replayer and the accuracy of dynamic analysis with P-Replayer are presented. A concise survey of related works is described in section V and we conclude the paper in section VI.

II. Dynamic Analysis with Execution Replay

For dynamic analysis of a program, instrumentation overhead is not avoidable whatever optimization techniques are used. The overhead can result in different execution behavior. It is possible to repeat the same analysis again and again hoping that eventually we see the true program behavior without the overhead. This repeated running is not even feasible if the program execution depends on the timing of external inputs.

Consider the idea of using a record/replay framework for dynamic analysis as shown in Figure 1. First, an execution of a program is recorded. If the overhead of recording is small enough to avoid probe effect, we can assume that the recorded execution is as same as the original program execution. Second, we apply dynamic analyses on the replayed execution which has the same thread executions and I/O inputs as the recorded one. Thus, the analyzed program will be executed as if there is no probe effect.

Moreover, the analyses of a program can be expedited with reproducible execution. When we find an unexpected execution during testing a program, the first step will be to locate the cause of the problem. We may need to run various analysis tools to locate the cause. This will be very time consuming and multiple trials may be needed since it can be hard to reproduce the execution given the significant overhead of the analysis tools. Instead, we record the program execution during the test run. As the execution is reproducible in replayed execution, analysis runs can be performed in the same program execution.

During a deterministic replay, measurement overhead from profilers can also be calculated to obtain accurate execution time measurement. In thread profiling, execution times of a program's activities such as function execution time can be measured for each thread. Since the measurements can incur probe effect, several execution time estimation algorithms [18, 27] have been proposed to recover the real execution time. In the approaches, the real execution time can be recovered with the considerations of three factors: 1) thread local measurement overhead, 2) event reordering, and 3) execution time difference due to the event reordering. As an example of the factors 2) and 3), consider the take and give operations performed on a semaphore. Assume

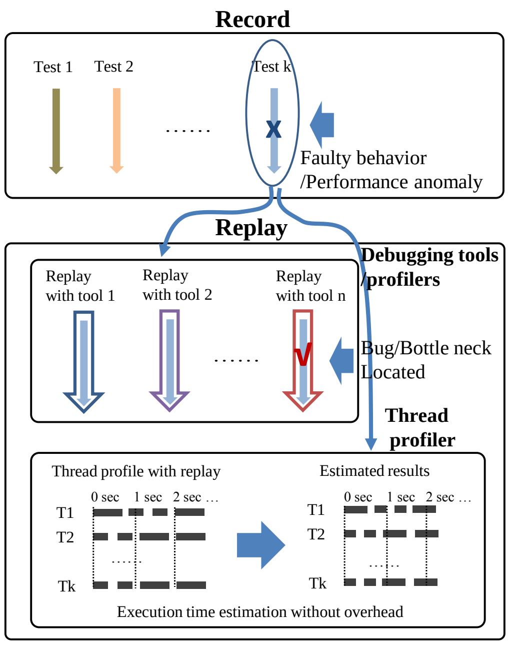

Figure 1. Dynamic analyses with execution replay

that in the real execution, the semaphore is given first and is taken afterward. Hence there is no blocking. In the instrumented execution, if the semaphore give operation had started late due to the delay of instrumentation overhead, then the thread taking the semaphore would have been blocked. The estimation algorithms [18, 27] can account for the blocking time and then reorder the events. However, if there is a change of program execution paths, then there will be no way to recover the real execution.

To avoid the above-mentioned problem, thread profiling can be applied in a replayed execution. Note that the execution with the profiling is deterministic as the event reordering is handled by the replay scheduling. The overhead compensation for the reordering events is no longer needed. Therefore, as long as we can identify the overhead caused by the replay mechanism, the total overhead from the thread profiling tool on a replayed execution is simply the sum of the replay overhead and the thread local measurement overhead from the profiler.

III. P-REPLAYER

In this section, we describe a record/replay framework, called P-Replayer. The framework is based on our Replay Debugger [16], and is optimized and generalized for dynamic analysis tools. The framework is designed to have minimal record and replay overheads. To enable execution replay, we consider the happens-before relation [14] among events which are execution instances of synchronization or IO function calls. The record/replay operations are implemented in the wrapper functions for events. The happens-before relation over a set of events in a program's execution is logged during recorded execution and are used to guide the thread execution sequence in execution replay. The happens-before relation between two events, denoted "→", is the smallest relation satisfying,

- **Program order:** If *a* and *b* are in the same thread and *a* occurs before *b*, then $a \rightarrow b$.
- **Locking:** If *a* is a lock release and *b* is the successive lock acquire for the same lock, then $a \rightarrow b$.
- **Communication:** If *a* is a message send and *b* is the receipt of the same message, then $a \rightarrow b$.
- **Transitivity:** If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

In P-Replayer, a data race detector is included as an analysis tool and for managing execution replay in the presence of data races. It also provides an approach for recovering real execution time without measurement overhead of thread profilers.

A. *Record/Replay Operations*

During recording operation, happens-before relations for all events in a program execution are traced and saved into a log file, and the same happens-before relations are to be enforced during execution replay. A general approach [26] to obtain the happens-before relation is to use Lamport clock [14]. In the approach, a Lamport clock is maintained for each thread and the clock is incremented and synchronized by the happens-before relation. A timestamp (Lamport clock value) is attached to each recorded event. During the subsequent replay operations, the corresponding event is delayed until all other events having smaller clock values are executed. This can enforce a stronger condition than necessary for replaying the partially ordered events. In our approach, we use a global sequence number to order the events traced in recording operation. This sequence represents a total order of events and is used to index the set of events during execution replay.

To identify the happens-before ordering, the event log of an event consists of the two sequence numbers for the event itself and the event immediately before it, plus thread id, the function type and arguments for the event. So, for an event *b*, let *a* be the event happened before *b* immediately. For the execution of the event *b*, the sequence numbers of both events *a* and *b* are recorded in the event log of the event *b*. For an input event, the received input is saved into the log file as well. All logging operations are performed in a dedicated logging thread to avoid possible jitters caused by file operations.

In a subsequent replay operation, an event table is constructed from the recorded log. The event table contains a list of events for each thread. Using the sequence number in the record log, events are indexed to represent the happened-before relations. Thus, based on the table, the next event for each thread that is eligible for execution can be identified and the same happens-before relations as the recorded ones are used to schedule thread execution. The replay scheduling is performed inside replay wrapper functions, thus the replay operation is implemented purely in application level.

A replay wrapper function consists of three parts. Firstly, a thread looks up the event table for the events happened before its current event. If any of the events that should be happened before are not executed, the thread is suspended waiting for a conditional variable. In the second part, when a thread can proceed with its current event, the thread carries out the original function. If the function is for an input event, the thread reads input value from the log file instead of actual reading from an I/O device. Lastly, the event for this wrapper function is marked as executed and the thread wakes up (signal) other threads waiting for the execution of this event. Note that the total order of events based on the sequence numbering is used only for indexing events and the replay operation follows the partial order of happens-before relations.

B. *Handling Data Races*

For efficient record and replay operations, only synchronization and I/O events are considered. However, one drawback is that a replay may not be correct in the presence of data races. A data race occurs when a shared variable is accessed by two different threads that are not ordered by any happens-before relation and at least one of the accesses is a write. Assume that, in the recorded execution, there is a data race on a variable that is used to decide alternative execution paths. Since there is no synchronization operation to enforce any happened-before relation, the order of accessing the variable is not recorded. This implies that, if an alternate accessing order from the record one is chosen in replayed execution, the replayed program may take a different execution path.

We use a similar approach as RecPlay [26] to handle the presence of data races in replaying operations. RecPlay records an execution of a program as if there is no data race and any occurrences of data races are checked during replay operation using a data race detection tool. If data races are found, the replay operation is repeated after removing the races. The approach is correct since a replayed execution is correct up to the point where the first race occurs as shown in [4]. However, most data race detection tools incur substantial overheads. It may not be practical to fix every data race during replaying operations.

Instead, P-Replayer detects the occurrence of an unexpected event (a different event from the recorded one) during replay process and stops the process at the location of the unexpected event. Then, the race can be detected with a race detection tool and fixed. The detection of an unexpected event is done in the replay wrapper by comparing the current events with the events in the record log. After fixing the race that results in different execution path, the replay can be safely used with various analysis tools including a race detection tool for detecting races that cause errors other than a change of execution paths.

We have implemented a data race detection tool based on FastTrack [8] and the tool is modified to be integrated with P-Replayer. FastTrack algorithm detects data races based on happens-before relations in a program execution. However, the replay scheduler invokes additional locking operations which appear as extra happens-before relations for the FastTrack detector. As a consequence, some data races may not be detected in the replayed execution. To correct the missed detections, we alter the FastTrack approach to discount the synchronization operation introduced by the replay scheduler.

C. *Execution Time Estimation*

In this subsection, we present an approximation approach to estimate execution time that can account for the overheads of replay operation and thread profilers. First, we present the algorithm that can estimate the real execution time without replay at program level. Second, the algorithm is refined to estimate the real execution time at thread level without the overheads of profilers and replay operation. The replay overhead is approximated based on the assumptions that 1) threads run on multiple cores, 2) events are evenly distributed to each core, 3) the record overheads for all event types are same, and 4) the number of worker threads is greater than or equal to the number of cores.

1) *Execution Time Estimation at Program Level*

The estimated execution time of a program execution, $C_{estimate}$, can be calculated by subtracting the replay overhead O_{replay} from the replayed execution time C_{replay}, such that,

$$C_{estimate} = C_{replay} - O_{replay} \qquad (1)$$

The replay overhead O_{replay} is the sum of 1) replay initialization time C_{init}, 2) extra execution time, C_e, spent in replay wrapper functions, and 3) blocking time, B_u, by the replay scheduling that leads to extra delay of replay execution, i.e.,

$$O_{replay} = C_{init} + C_e + B_u \qquad (2)$$

The extra execution time C_e is classified into two parts. Note that for two events a and b with $a \rightarrow b$, the execution of event b can be *delayed* until event a is executed or can be executed with *no-delay* if event a has already been executed. The two possible ways of event executions contribute to various amount of overheads, thus we account them differently. Let n_{nd} and n_d be the numbers of events of *no-delay* and *delayed* execution, respectively. Let c_{nd} and c_d be the execution times for *no-delay* and *delayed* events, respectively. Then, C_e can be expressed as,

$$C_e = n_{nd}*c_{nd} + n_d*c_d \qquad (3)$$

Threads can be blocked by the replay scheduling through *global locking* and by *delayed events*. The blocking of threads leads to additional execution time only when the number of ready threads becomes less than the number of cores, i.e., when the cores are underutilized. Let n_g be the number of occurrences that threads are blocked by the global locking and let b_g be the average delay caused by each global locking. Also, let b_d be the total execution delay caused by the delayed events due to the replay scheduling at the end of a replay execution. Then, B_u can be expressed as,

$$B_u = n_g*b_g + b_d \qquad (4)$$

Combining Equation (3) and (4) into Equation (2) gives,

$$O_{replay} = C_{init} + n_{nd}*c_{nd} + n_d*c_d + n_g*b_g + b_d \qquad (5)$$

2) *Overhead Measurements*

The replay initialization time C_{init}, and the counter values, n_{nd}, n_d, and n_g, can be measured during a replay execution. The blocking delay, b_d, is calculated using the algorithm in Figure 2 based on the utilization of the cores. On the other hand, the per event overheads, c_{nd}, c_d, and b_g, are hard to

```
n = number of runnable threads
M = number of cores

//At start of every delayed event
if ( n==M ) //start measurement
   t_s = get_timestamp();
else if (n < M)
   tmp =  get_timestamp();
   b_d += (tmp- t_s )*((M-n)/M);
   t_s = tmp;
n--;

//At end of every delayed event
if ( n < M )
   tmp =  get_timestamp();
   b_d += (tmp- t_s )*((M-n)/M);
   t_s = tmp;
n++;
```

```
//At start of event delay in thread T
t_s-T = get_timestamp();

//At end of event delay in T
t_e-T = get_timestamp();
b_T += (t_e-T - t_s-T);

int blocking_overhead(thread T)
{
    if T is not blocking from replay
        return b_T;
    else
        return \
          (b_T + get_timestamp()-t_s-T);
}
```

Figure 2. (Left): the time measurement in which the cores are underutilized due to the delayed events by the replay scheduling. (Right): the measurement of blocking overhead for a thread *T*.

measure online in the execution on multi-core systems. To avoid the online measurement, we assume they can be viewed as constants for a given system and can be estimated offline using measurement programs.

The measurement program for c_{nd} consists of multiple threads which invoke their own locks. Thus, threads are independent of each other and there is no overhead from delayed events. Hence, $n_d*c_d=0$ and $b_d=0$. To avoid any idle CPU core caused by global locking, extra threads running busy loops and with no synchronization and IO event are added. Hence, the blocking overhead from the global locking becomes zero ($n_g*b_g=0$) since all cores are utilized fully. The execution time without replay, C_{m1}, is measured and we can assume that $C_{m1} = C_{estimate}$. Then, with Equation (1) and (5), c_{nd} can be calculated using the following equation:

$$C_{m1} = C_{replay} – (C_{init} + n_{nd}*c_{nd})$$

A similar program is used for measuring c_d, where a lock is shared among all threads. Hence, $n_d*c_d≠0$. Using the extra threads with busy loops, the program keeps $n_g*b_g=0$ and $b_d=0$. Assuming that the execution time without replay, C_{m2}, is equal to $C_{estimate}$, c_d can be calculated using the following equation:

$$C_{m2} = C_{replay} – (C_{init} + n_{nd}*c_{nd} + n_d*c_d)$$

The remaining constant b_g is calculated using a similar measurement program for measuring c_d but without the extra threads with busy loops.

3) *Execution Time Estimation at Thread Level*

In thread profiling, the execution times of threads' activities can be measured. As presented in section II, the measurement overhead from the thread profiler consists of 1) thread local overhead and 2) execution time differences due to event ordering. When the profiler runs in a replayed execution, the latter overhead is contained in the replay overhead for delayed events ($n_d*c_d + b_d$). Therefore, the total overhead from the profiling on a replay execution is simply the sum of the replay overhead and the thread local overhead of the profiler. To estimate the real execution time without the overheads, we need per-thread measurement at a given time instant. For instance, if a function in a thread T is invoked at t_s and finishes at t_e, then the execution time of the function is measured as t_e-t_s. If $C_{estimate-T}(t)$ is the estimated execution time of a thread T up to time t, then the real execution time of the function can be estimated as,

$$C_{estimate-T}(t_e)- C_{estimate-T}(t_s)$$

We can start the measurements after the initialization of a replay execution, thus C_{init}=0. All counter values (n_{nd}, n_d, and n_g in Equation (5)) are maintained for each thread. Note that all per-event measurements (c_{nd}, c_d, and b_g in Equation (5)) are measured for concurrent executions on multiple cores. Since each thread can only be run on a single core, the per-event measurements for each thread, denoted as c_{nd}', c_d', and b_g', can be approximated as the product of M and c_{nd}, c_d, and b_g, respectively, where M is the number of cores in the system. The blocking delay (b_d) is replaced with accumulated blocking time for each thread and it can be calculated as shown in Figure 2. Then, the replay overhead in a thread T at a given instant t can be represented,

$$O_{replay-T}(t)=n_{nd-T}(t)* c_{nd}' + n_{d-T}(t)* c_d' + n_{g-T}(t)* b_g'+ b_T(t) \quad (5)'$$

Let the thread local overhead from the profiler in a thread T up to a given instant t be $O_{profile-T}(t)$. Then, the estimated execution time for a thread T at time t can be expressed as,

$$C_{estimate-T}(t) = t – (O_{replay-T}(t) + O_{profile-T}(t)) \quad (1)'$$

IV. Evaluation

In this section, we show the effectiveness of the analysis approach in replay execution through several benchmark experiments. First, we show the overheads of P-Replayer. Second, we present the evaluation results of the execution time estimation algorithm. Lastly, the accuracy of dynamic analyses using P-Replayer is presented. All experiments were performed on an Intel Core Duo processor running Ubuntu 12.04 with kernel version 3.2.0.

The two programs shown in section I are used to illustrate the effect of the minimized probe effect in record phase. All other experiments were performed with 11 benchmarks for desktop computing to reveal the efficiency and accuracy of the dynamic analysis methods performed in the replay phase. The benchmarks are from PARSEC-2.1 [1] and from real-world multithreaded applications: *FFmpeg* [6], a multimedia encoder/decoder; *pbzip2* [11], a data compressor; and *hmmsearch* [7], a search/alignment tool in bioinformatics.

A. *Overhead of Record/Replay Operations*

Table 3 shows the overheads of the record and replay operations in P-Replayer. "Number of events/sec" column shows the number of recorded events per second in the execution of each benchmark program, and from the column we can have a general idea of how big the overheads will be. The overheads of record and replay operations are 1.46% and 2.78% on geometric mean, respectively. The results suggest that P-Replayer will be suitable for dynamic program analysis in replay execution. One exceptional case is *fluidanimate* which incurs noticeable record/replay overheads due to the large number of events in the execution.

Table 3. Record/Replay overhead

	Benchmark Program	Base time (sec)	Record (sec)	Replay (sec)	Number of events/sec	Record Overhead	Replay Over-head
PARSEC	facesim	6.050	6.054	6.055	2,200.5	0.07%	0.08%
	ferret	5.027	5.073	5.161	2,066.2	0.92%	2.67%
	fluidanimate	2.054	3.620	4.887	2,163,267.3	76.24%	137.93%
	raytrace	9.823	9.843	9.813	9.8	0.20%	-0.10%
	x264	2.196	2.288	2.323	17,487.2	4.19%	5.78%
	canneal	6.643	6.674	6.677	1.5	0.47%	0.51%
	dedup	7.750	8.208	8.711	75,623.9	5.91%	12.40%
	streamcluster	3.781	4.071	4.092	38,417.1	7.67%	8.23%
	ffmpeg	3.053	3.052	3.157	3,560.4	-0.03%	3.41%
	pbzip2	5.297	5.396	5.381	622.2	1.87%	1.59%
	hmmsearch	26.550	26.624	26.699	3,644.7	0.28%	0.56%
Geometric mean						1.46%	2.78%
Average						9.78%	17.31%

Table 4. Revisit of Table 1 with recording operation

inputs/sec	Native execution	Record	PIN Cache	FastTrack
150	0.0%	0.0%	16.8%	0.3%
300	0.0%	0.0%	36.1%	1.2%
450	0.0%	0.0%	45.5%	1.9%

Table 5. Revisit of Table 2 with recording operation

Queue Length	Native execution	Record	PIN Cache	FastTrack
5	1.3/7.5	1.3/8.1	8.3/191.9	5.5/56.3
10	0.5/8	0/7.1	2.5/146.8	2.4/37.9

Tables 4 and 5 are the revisits of Tables 1 and 2 with the additional measures collected in record phase. For the QT application (shown in Tables 1 and 4), the QT libraries are not instrumented in all the tools and no event recording is done in the execution of the library code. The results in both tables suggest that multithreaded program executions can be recorded by P-Replayer with a minimal probe effect.

B. *Execution Time Estimation*

Based on the overhead analyses in Section III, the replay overhead for each benchmark can be measured and calculated with Equation (5) and the real execution time can be estimated using Equation (1). In Table 6, the estimated execution times of the benchmark programs are listed in column 4 where column 5 gives the estimation error. On average, the estimation error is 1.24%.

The replay overhead is classified into the 5 categories in Equation (5). In Figure 3, the relative overhead in the 5 categories is illustrated for the four benchmarks that have more than 5% of replay overhead, i.e., *fluidanimate*, *x264*, *dedup*, and *streamcluster*. In the chart, the items are correspondent to the five categories of Equation (5). The applications, *dedup* and *streamcluster*, show relatively more percentages of blocking overhead caused by delayed events (around 50%) than the other two applications. This implies that they

Table 6. Execution time estimation without replay

Benchmark Program	Base time (sec)	Replay (sec)	Estima-tion (sec)	Error
facesim	6.050	6.055	6.041	-0.2%
ferret	5.027	5.161	5.122	1.9%
fluidanimate	2.054	4.887	2.144	4.4%
raytrace	9.823	9.813	9.812	-0.1%
x264	2.196	2.323	2.247	2.3%
canneal	6.643	6.677	6.677	0.5%
dedup	7.750	8.711	7.838	1.1%
streamcluster	3.781	4.092	3.859	2.1%
ffmpeg	3.053	3.157	3.052	0.0%
pbzip2	5.297	5.381	5.378	1.5%
hmmsearch	26.550	26.699	26.595	0.2%
Average				1.24%

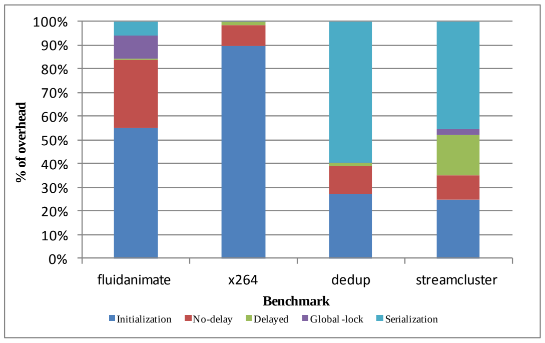

Figure 3. The decomposition of the replay overhead is shown for benchmarks that have more than 5% replay overhead

experience more per event disturbance caused by the replay scheduling than the other two applications.

C. *Accuracy of Analysis in Replay Execution*

In this section, we present experimental results to show the accuracy of dynamic analysis in replay execution. Once we have a recorded execution which is as same as the original execution, the analysis result in replay execution will be as close as the analysis without any instrumentation. Since it will be very hard to know what the analysis results will be without any instrumentation, we assume that the 11 bench-

Table 7. Data race detection with FastTrack

Benchmark Program	Slowdown		Number of races detected
	Without replay	With replay	
facesim	140	139	8907
ferret	86	82	2
fluidanimate	85	116	1
raytrace	27	27	13
x264	74	77	1300
canneal	12	12	0
dedup	151	148	0
streamcluster	244	251	1053
ffmpeg	120	120	1
pbzip2	68	69	0
hmmsearch	83	86	1
Average	99	103	

Table 8. Flat profiling comparison from Callgrind

Benchmark Program	Slowdown		# of different function entries
	Without replay	With replay	
facesim	37	37	0
ferret	50	50	0
fluidanimate	59	127	7
raytrace	59	59	0
x264	77	79	0
canneal	25	24	0
dedup	41	43	0
streamcluster	58	52	4
ffmpeg	66	66	0
pbzip2	51	50	1
hmmsearch	28	29	0
Average	50	56	

Table 9 Cache simulation results from PIN Cache

Benchmark Program	Slowdown		Cache miss rate - Without replay			Cache miss rate - With replay		
	Without replay	With replay	L1-Instruction	L1-Data	L2-Unified	L1-Instruction	L1-Data	L2-Unified
facesim	472	462	0.00%	5.92%	12.02%	0.00%	5.92%	12.07%
ferret	336	341	0.02%	4.73%	16.89%	0.02%	4.74%	16.14%
fluidanimate	398	731	0.00%	**1.15%**	17.13%	0.00%	**1.58%**	18.72%
raytrace	395	387	0.00%	2.77%	2.23%	0.00%	2.77%	2.23%
x264	411	424	1.34%	4.07%	**8.10%**	1.32%	4.07%	**9.62%**
canneal	92	91	0.00%	4.06%	43.46%	0.00%	4.08%	43.12%
dedup	286	301	0.00%	4.06%	7.25%	0.01%	3.93%	7.66%
streamcluster	395	454	0.00%	6.55%	49.14%	0.00%	6.42%	48.95%
ffmpeg	478	478	0.01%	4.52%	11.56%	0.01%	4.61%	11.26%
pbzip2	377	381	0.00%	4.03%	13.54%	0.00%	4.03%	14.40%
hmmsearch	498	475	0.00%	0.78%	7.14%	0.00%	0.78%	7.10%
Average	376	411						

mark programs show no (or negligible) probe effect from the instrumentation and compare the analysis results with the ones collected from replay execution.

Table 7 compares analysis results of the FastTrack race detector in normal program execution and in replay execution. The two approaches locate the same data races. For *facesim*, there was a data race that may result in different execution paths. P-Replayer stops the replay execution after encountering the event diverging from the recorded one. Then, the race is detected with the FastTrack detector and fixed with correct synchronization. The data in the table shows the detection result after fixing the race for *facesim*.

Table 8 compares the flat profiling results from Callgrind [3]. The flat profiling lists the functions in decreasing order that have fetched most instructions. For each benchmark program, we compare the top 10 function entries from normal program execution and replay execution, and the number of different function entries is shown in the last column of Table 8. There are three cases showing different function entries. For *streamcluster*, the function entries are different from the 6th entries due to the functions invoked for replay execution. However, from the 3rd function in the list, the functions have used less than 1% of total instructions. Similarly, the 10th function entries are different in *pbzip2* which fetches a negligible percentage of instructions~~s~~. For *fluidanimate*, after removing functions used in the replay execution, the same function entries are shown in the same order.

Table 9 shows the comparisons of PIN Cache simulator running in normal program execution and in replay execution. As can be seen from the comparisons, the differences are negligible. There are only two measures of cache miss rates with a more than 10% discrepancy between normal program execution and replay execution.

V. RELATED WORKS

Event-based replay has been a preferable choice for program analysis and debugging. Instant Replay [15] is one of the earliest works that record and replay a partial order of shared object accesses. RecPlay [26] has proposed a record/replay mechanism that has low record overhead and a data race detection algorithm. However, the replay overhead is too high (around 2x) for uses with dynamic analyses. Replay Debugger [16] uses a similar approach as RecPlay for record/replay operations, but it focuses on debugging techniques integrated with GDB.

As an effort to avoid probe effect, several hardware-base dynamic analysis tools have been proposed. To detect data races, CORD [23] keeps timestamps for shared data that are presented in on-chip caches. With simplified but realistic timestamp schemes, the overhead can be negligible. DAProf [21] uses separate caches for profiling of loop executions. In the approach, short backward branches are identified and profiling results are stored in the cache. Both approaches are non-intrusive with acceptable accuracies of analysis results. However, the requirement of extra hardware mechanisms may make the approaches impractical.

Moreno et.al [20] has proposed a non-intrusive debugging approach for deployed embedded systems. In the approach, the power consumption of a system is traced and matched to sections of code blocks. Thus, a faulty behavior in the execution of the code blocks can be identified only with an inexpensive sound card for measuring power consumption and a standard PC for the power trace analysis.

VI. CONCLUSIONS

In this paper, we have presented a dynamic analysis method for embedded software. In the method, the execution of a program is recorded with minimal overhead to avoid probe effect, and then the program analysis is performed in replay execution where event ordering is deterministic and is not affected by instrumentation. For this end, we have described a prototype P-Replayer and demonstrated the use of replay execution for dynamic program analyses on the benchmark programs. In addition, P-Replayer provides execution time estimation which can be integrated for thread profiling tools.

ACKNOWLEDGMENT

This work was supported in part by the NSF I/UCRC Center for Embedded Systems, and from NSF grant #0856090.

REFERENCES

[1] C. Bienia. Benchmarking Modern Multiprocessors. *Ph.D. Thesis. Princeton University,* 2011.

[2] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: proportional detection of data races. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI),* pages 255-268, 2010.

[3] Callgrind, Valgrind-3.8.1. http://valgrind.org/.

[4] J-D. Choi and S. L. Min. Race Frontier: reproducing data races in parallel-program debugging. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 145-154, 1991.

[5] CSE 438 – Embedded Systems Programming, ASU, http://rts.lab.asu.edu/web_438/CSE438_Main_page.htm.

[6] FFmpeg. http://www.ffmpeg.org/.

[7] R. D. Finn, J. Clements, and S. R. Eddy. HMMER web server: interactive sequence similarity searching. *Nucleic Acids Research Web Server* Issue 39:W29-W37, 2011.

[8] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 121-133, 2009.

[9] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proceedings of the 19th annual international conference on Supercomputing (ICS)*, pages 81-90, 2005.

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

[11] J. Gilchrist. Parallel BZIP2, http://compression.ca/pbzip2/.

[12] Intel PIN tool. http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[13] Intel Vtune Amplifier XE 2013. http://software.intel.com/en-us/intel-vtune-amplifier-xe.

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

[15] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers,* 36(4): 471-482, 1987.

[16] Y-H. Lee, Y. W. Song, R. Girme, S. Zaveri, and Y. Chen. Replay Debugging for Multi-threaded Embedded Software. In *Proceedings of IEEE/IFIP 8th Int'l. Conf. on Embedded and Ubiquitous Computing (EUC),* pages *15-22, 2010.*

[17] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G, Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI),* pages 190-200, 2005.

[18] A. D. Malony and D. A. Reed. Models for performance perturbation analysis. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging (PADD),* pages 15-25, 1991.

[19] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI),* pages 134-143, 2009.

[20] C. Moreno, S. Fischmeister and M. Anwar Hasan. Non-intrusive program tracing and debugging of deployed embedded systems through side-channel analysis. In *Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems (LCTES),* pages 77-88, 2013.

[21] A. Nair, K. Shankar, and R. Lysecky. Efficient hardware-based nonintrusive dynamic application profiling. *ACM Transactions on Embedded Computing Systems (TECS),* 10(3): Article No. 32, 2011.

[22] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 89-100, 2007.

[23] M. Prvulovic. CORD: cost-effective (and nearly overhead-free) order-recording and data race detection. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 232-243, 2006.

[24] Qt Project, http://qt-project.org/.

[25] RealView Profiler. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0412a/html/chapter_1.html.

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

[27] F. Wolf, A. D. Malony, S. Shende, and A. Morris. Trace-Based Parallel Performance Overhead Compensation. *High Performance Computing and Communications, LNCS 3726,* pages 617-628, 2005.

[28] X. Zhuang, M. J. Serrano, H. W. Cain, and J. Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI),* pages 263-271, 2006.