Constant Execution Time Recording for Replay of Sporadic Real-Time Systems

Joel Huselius and Henrik Thane
Mälardalen Real-Time Research Centre
Mälardalen University, Västerås, Sweden
joel.huselius@mdh.se

Abstract

Due to intrinsic and non-deterministic interactions, executions of multi-tasking real-time systems cannot be reproduced without additional mechanisms – record/replay is a technique that uses recording of executions to facilitate off-line reproduction of executions, but recording of an execution produces large amounts of data. In this paper, we present a dynamic algorithm that can organize the memory during recording so that the amount of data that needs to be stored is reduced.

1 Introduction

Debugging is a time-consuming and therefore costly activity [14], but the research interest in this topic does not stand in proportion to the economical cost – the field has seen little interest until recently. It is our thesis that one reason for this high cost is the lack of useable tools to facilitate debugging; debugging multi-tasking systems in general, and multi-tasking real-time systems in particular is not made possible just by the presence of debuggers – more tools are needed.

The process of debugging a system using an ordinary debugger (e.g., `gdb`) is iterative in that it restarts the system over and over again (with the same input) to pinpoint the bug. Hence, a requirement for its success is a repeatable execution – a requirement that multi-tasking systems, distributed systems, real-time systems, etc, cannot fulfill – which is one of the reasons for why new tools are needed.

Deterministic reproduction of an execution through *record/replay* has been presented as plausible means for to remedy this problem [8, 15]. We have shown the applicability of record/replay in a recent case-study [10].

The basic idea is similar to a video recorder used to tape a television broadcast: After first having inserted *probes* into the system, a *recording* of an observed *reference execution* is performed. The resulting *log* is then used to create a *replay execution* that is intended to be identical to the reference execution. Non-deterministic choices encountered in the replay execution are resolved according to the log.

The main drawback of record/replay lies in the fact that the reference execution must be recorded – the cost in memory and execution time overhead is potentially considerable – which is one of the reasons why new research results are needed.

It may be possible to reduce the cost of recording by letting the replay execution start in the middle of the reference execution. Previous work has shown [10, 15] that this can be performed by regularly taking checkpoints of the system from where the replay execution can start. These checkpoints can even be *memory excluding* [4, 6], meaning that checkpoints are in-complete – deterministic parts of the data is left out as it can be derived offline.

However, in sporadic real-time systems (such as fuel injection control in a combustion engine that has a periodicity proportional to the engine revolutions [1]) distribution of resources to achieve the "best" replay may be difficult. The control algorithms realized in such systems places the additional constraint of known and constant overhead on the system; the functionality of the system will be severely degraded if the system is non-deterministic in the temporal sense. Thus, resource distribution algorithms such as that presented here should have a constant execution time.

In this paper, we present a constant execution time logging algorithm named ECETES (Extended Constant Execution Time Eviction Scheduler) that can serve to reduce the space required to keep logs. We define a notion of "best" replay, and present an evaluation comparing the algorithm with the – in the debugging context – commonly used [8] FIFO-algorithm (e.g. a circular queue).

The remainder of this paper is organized as follows: Section 2 defines a notion of "best" replay. Section 3 presents a system model, related work, and our logging algorithm ECETES. Following, Section 4 presents the evaluation of ECETES according to that notion. The paper is concluded in Section 5.

2 Logging algorithm requirements

In this section, we present a requirement for logging algorithms that will later be used in our evaluation of the presented ECETES.

2.1 Industrial debugging of real-time systems

As motivated in the previous section, debugging real-time systems can be facilitated through the use of record/replay (a solution similar to a video recorder). The terminology used is as follows: *Probes* perform a *recording* of a *reference execution* by *monitoring* data and events and *logging* that information for post-mortem analysis. The *log* is then used to create a *replay execution* that is identical to the reference execution.

In our previous work [10], we have implemented this for an industrial robot application where we use *checkpoints* of task state to allow the replay to start from a state other then the starting state of the system. These checkpoints are *memory excluding* [4, 6], meaning that they are in-complete – deterministic parts of the data is left out as it can be derived offline. These checkpoints require us to make use of a special algorithm to initiate replay [4].

2.1.1 Starting replay

The method (presented in a previous publication [4]) stipulates that, for each task in the system that is to be replayed, *potential starting points* are identified in the code offline by an expert. During the reference execution, memory excluding checkpoints (see above) are taken of the task-state at the potential starting points.

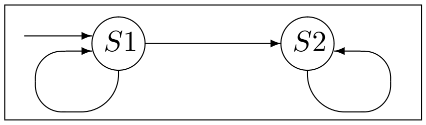
$$\begin{tikzcd} \arrow[r] & S1 \arrow[loop below] \arrow[r] & S2 \arrow[loop below] \end{tikzcd}$$

Figure 1. Two potential starting points.

The replay is started by restarting the individual tasks with the same input as during the reference execution and replacing the contexts of tasks with a checkpoint from the log as they reach their first potential local starting points. Thus, that particular starting point is a *used starting point*. When all tasks have encountered used starting points, the replay can continue [4].

Note that this method requires the execution of the individual task, up until it reaches its used starting point, to be deterministic [5].

However, depicting the task as a finite state-machine (FSM), where potential local starting points are vertices and the possible execution paths between them are edges, a problem named *multiple consecutive starting points* is illustrated by the example in Figure 1: In preparation for the replay, the task will be restarted and come to $S1$ from where it should be induced with a checkpoint (the execution is non-deterministic past this point as it can either continue in $S1$ or leave for $S2$). But without insurance from the logging algorithm, it is not guaranteed that a checkpoint remains that can describe a task-state valid in $S1$. In order to clarify, we provide two examples:

Example 1: Using one fixed-size queue per starting point would lead to external fragmentation as some records will be unused because the execution pattern between states is non-deterministic – $S2$ may never be reached, in which case it serves no purpose to allocate resources to it.

Example 2: Using one global fixed-size queue per task may lead to that only $S2$ is described – as the information about $S2$ cannot be used until the replay execution has been shepherded there from $S1$, that would result in a situation where the replay cannot be performed.

We formulate the following requirement: *The logging algorithm must ensure that the reference execution can reproduce also the most recently monitored entry.*

2.2 Testability of the end-product

Thane and Hansson [12] determines that execution time jitter, or differences in execution time, is bad for testability: An instance of a multi-tasking system can be seen as one sequential program – it is a serialization (compare to database transactions etc.) of a multi-tasking system into a single-tasking system. Thus, a multi-tasking system can be seen as a set of single-tasking systems of which all members must be tested – a larger set requires more testing. The size of the set is increasing with the jitter, wherefore jitter will increase the required testing effort. Thus, the logging algorithm should not add any jitter to the system. One way to ensure this is to show that the logging algorithm has a constant execution time.

Regarding implementations with constant execution times, Puschner and Burns [7] proposed WCET-oriented programming to facilitate WCET analysis, thus making scheduling easier and more efficient. Due to the effects of hardware, this should be supported by a platform such as SPEAR [2]. However, the negative effect of jitter has exponential characteristics [13], wherefore any reduction will have significant impact.

Thus, the requirement is extended to read: *The logging algorithm must, while maintaining testability, ensure that the reference execution can reproduce also the most recently monitored entry.*

2.3 Comparison by replay length

As the method to start a replay will treat each task independently, this will lead to a situation where tasks may start their replay at different points in the virtual time of the replay execution. However, the replay cannot be considered to be correct until all tasks are started – from there on, bugs can be found. We name the interval in which all tasks are concurrently replayed: the *shortest interval of replay* (SIR).

Theoretically, the SIR must cover the period of time from the infection of the system (the execution of a bug) to the failure of the system (when the infection is observed) [11]. We label this period the *incubation period* of the system. In practice however, it is the amount of memory assigned to house the log, and the efficiency of its use, that limit the SIR.

2.4 Logging algorithm requirement

Thus, the final requirement reads:

The logging algorithm must, while maintaining testability, ensure that the reference execution can reproduce also the most recently monitored entry and then extend the SIR as far as possible. Hence, we can use the length of the SIR as a measure of the logging efficiency. Below, we show simulations suggesting that ECETES delivers a better SIR than FIFO in sporadic real-time systems.

3 Logging algorithms

In this section, we present the technical contribution of the paper: the logging algorithm.

3.1 System model and terminology

We assume an instrumented, preemptive, sporadic, online scheduled, multi-tasking real-time system where probes are implemented in software and are allowed to have their execution protected from interrupts. *Tasks* can communicate with each other and with the environment. Each task emits a single new instance, a new *job*, at a time. The periodicity (p) of the jobs varies in the interval described by the task basic deadline (bdl) and *sporadicity* (sp) such that $p = [bdl, bdl + sp]$. A new job of a task cannot be released prior to the completion of a previous.

It is assumed that execution time jitter (differences in execution time) has a negative effect on the functionality of the end system (e.g. fuel injection control in a combustion engine that has a periodicity proportional to the engine revolutions [1]).

The recording effort probes *control-flow* events (i.e. context-switches) and *data-flow* events (i.e. communication and checkpoints of task data-state). Each *event* observed by a probe results in an *entry* that is logged in memory in one or more *records*. The recording effort is assigned a memory-space on which both the implementation of the logging algorithm and the logs are to be kept (thus, smaller implementations will have larger space to keep logs).

3.2 Related work

Surveying the field of available logging algorithms, one finds that only a few alternatives are known: As noted by Stewart Gentleman [8], the commonly used solution is a First-In-First-Out (FIFO) logging algorithm. There is also a method called *adaptive logging*, presented by Zambonelli and Netzer [15], but the online decision-maker of that method has substantial jitter-properties. Sultan et al. presented a lazy garbage collection for distributed systems [9]. As the checkpointing process does not require a distributed solution to facilitate replay [4], and the checkpointing and garbage collection of their proposed method is transparent and unrelated to the application (thus inferring jitter in to the system), their method is not adaptable to the situation described here.

3.3 FIFO logging algorithms

Three approaches seem feasible for fixed-size First-In-First-Out logging algorithms:

Global FIFO (GFIFO, FIFO within the system): all memory available for logging is used in one single FIFO-queue. (May be difficult to implement as several different entry-sizes must be accommodated.)

Local FIFO (LFIFO, FIFO within tasks): each task will log entries to dedicated queues, one separate queue is reserved for entries relevant for all tasks.

Starting point FIFO (SFIFO, FIFO within starting points), each starting point will log entries to dedicated queues, one separate queue is reserved for entries relevant for all starting points.

3.4 ECETES

Due to the dynamic properties of the underlying system, there are significant difficulties with using non-dynamic logging algorithms to solve this problem. As the distribution of required resources is constantly changing during the operation of the system, a static assignment will often be found to have an non-appropriate distribution that wastes limited memory resources. The difficulty of distributing the resources so that the distribution is always the best-fit constitutes the motivation for using dynamic logging algorithms.

The basic functionality of the dynamic logging algorithm ECETES is as follows: Similar to SFIFO, one queue per starting point is maintained. Logging an entry of a number of records is preceded by the act of evicting the same number of records from a single queue. The output from the eviction process is a chain of emptied records that are then filled with the new entry and inserted to the designated queue.

3.5 The ECETES algorithm

Our algorithm-proposition, ECETES, for the process of inserting a new entry of n records to queue J functions as follows:

Let the label E describe the queue to evict from, there is a default assignment to the label. One-by-one, each queue in the system is examined. For each examined queue I, the following items are respected: properties of I, the time stamp of the $(n+1)$:th record counted from the back end of queue I, and the time stamp of the $(n+1)$:th record counted from the back end of queue E. If I is found to be a better candidate than E (if the properties of I allows the eviction, and if the record examined in I is older than the record examined in E) the label E is modified to indicate I, otherwise E is modified to indicate E.[1] Thereafter, n records are removed from the back end of E, filled with the new entry, and inserted into the front end of J.

The properties that can be posted to queues are: *temporal span* that ensures that a queue has coverage during an interval of time, *spatial size* that ensures the queue to hold a minimum number of records, and *priority*. A queue cannot be forced to evict any entries if it means violating the properties of the queue.

3.6 Example

With the intent to clarify, we provide an example of the operation: See Figure 2, where we have a system of

[1]This assignment is a patch to ensure the constant execution time.

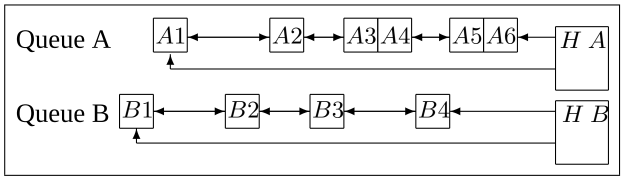

Figure 2. ECETES example.

two ECETES-queues (A and B). Each queue receives entries to log into records, and has a vector of logged records associated to it. The further left in the figure the record is placed, the older the record is. There is a header for each of the queues ($H\ A$ and $H\ B$), these contain the properties posted on the queue, and pointers to the first and last element in its vector of records. Also, each record contains some information (i.e. a time stamp and the identities of the neighbors to the record).

To store a new entry consisting of two records in queue A, we must identify a queue from where to evict two records. This identification is performed by examining the two queue-headers, and one record from each queue ($A3$ and $B3$). These records are picked as they are the oldest records that will remain in the respective queue if two records are evicted from it. Essentially, provided that it will not lead to a violation of the queue-constraints, the queue to which the oldest of the two records belong will the subjected to the eviction. Here, $B3$ is the oldest, so B will have to surrender its two oldest records ($B1$ and $B2$). The two new records ($A7$ and $A8$) can then be inserted, and the new records are then linked with their designated queue.

Thus, the ECETES algorithm results in a SIR with the length of the difference between the timestamps of $B3$ and $A8$.

An LFIFO algorithm in the corresponding situation would evict $A1$ and $A2$ resulting in a SIR no longer than the difference between the timestamps of $A3$ and $A8$.

A feasible solution would be to let an algorithm evict $A1$ and $B1$, resulting in a SIR equal to the difference between the timestamps of $B2$ and $A8$ which is obviously the longest SIR possible. However, our efforts to construct an algorithm that would make that choice, while working efficiently, has failed due to the constraints on execution time (see Section 2.2).

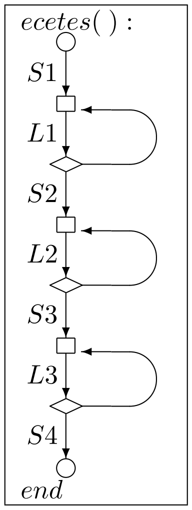

Figure 3. ECETES instruction-flow: S-segments represents sequential code executed once per function call, L-segments represents sequential code executed as a loop.

3.7 Implementation

Figure 3 describes the assembler instruction-flow of the implementation, the source code is available in [3], each loop in the function is performed a fixed number of times for an entry of a given size. Assuming that an entry produced by a given probe will produce an entry of a given size, this is sufficient ensuring a constant execution time.

4 Evaluation

We have evaluated ECETES with respect to the requirement formulated in Section 2.4, the result of that evaluation is presented in this section.

Concerning the maintenance of system testability, as shown in Figure 3, this is accomplished by enforcing a constant execution time. As the program flow consists of either sequential code or loops with constant iterations, the same instructions will be executed each function call – assuming that an instruction always takes the same time to execute, this will lead to a constant execution time. It is true that the underlying hardware may use heuristics (e.g. branch prediction or caches) that may cause the execution time of

Properties	**Task A**	**Task B**
Deadline [time units (tu)]	2000	8000
Execution time [tu]	990 - 999	3960 - 3999
Data-state [bytes]	1100	1100

Table 1. Simulator taskset 1.

Properties	**Task A**	**Task B**	**Task C**
Deadline	2000	4000	8000
Execution time	990 - 999	990 - 999	990 - 999
Data-state	1100	1100	1100

Table 2. Simulator taskset 2.

a given instruction to vary, but there are architectures that avoid this [2]. Also, as the negative effect of jitter has exponential characteristics [13], any jitter reduction (such as that of using ECETES in the place of an algorithm with jitter) will have significant impact.

Concerning the ability to reproduce also the most recently monitored entry, this is possible as ECETES maintains one queue per starting point.

To evaluate the last part of the requirement, ECETES is compared to LFIFO in a simulator – the intention is to validate the hypothesis that ECETES outperforms LFIFO in sporadic real-time systems. The validation is performed by measuring, given a certain memory budget, which of the methods that will provide the longest SIR (see Section 2.3).

4.1 Overhead

We can measure overhead in two ways; spatial size describes the amount of memory consumed by an implementation, temporal overhead describes the execution time required to execute it. We display the spatial size measured on the Intel platform, the same platform used for the simulations.

For the test cases described in tables Table 1, and Table 2, the overhead of the current ECETES implementation on an Intel platform is as follows: The temporal overhead is described by Table 3, where **It** describes the number of times a block of code is executed for each call to the function, and **Inst** describes the size in assembler code instructions of each block **Lab** as described by Figure 3. Note that the instruction count of loop $L2$ does not include the body of the function `memcopy()`, which is called from within that loop (also note that all calls to the function `memcopy()` has the same size-parameter: the maximum record size). Regarding the iterations, *queues* is a constant describing the number of queues that are defined in the monitoring activity, *records* is the number of records required to log the entry, and *max records* is a constant describing the maximum number of records required to log an entry in the system.

Lab	**Inst**	**It**	**Lab**	**Inst**	**It**
S1	26	1	L1	92	ECETES queues
S2	43	1	L2	36	ECETES records
S3	35	1	L3	34	ECETES max records
S4	7	1			

Table 3. Temporal overhead in assembler instructions of the Intel implementation (see Table 4 for constants and Figure 3 for labels).

Post	**TS 1**	**TS 2**
ECETES queues	3 pcs.	4 pcs.
ECETES records	72 pcs.	102 pcs.
ECETES max records	3 pcs.	3 pcs.
Assembler code	389	389
Queue headers	96	128
Entry index	36	48
Records	74752	105472
TOTAL ECETES	**75273**	**106037**
TOTAL LFIFO	**76214**	**106662**

Table 4. Listing of system resources, and resulting spatial size in bytes.

The spatial size of an Intel implementation is described by Table 4, where *TS* is *taskset*. In the same table, also the corresponding LFIFO overhead is shown. Note that the test was configured in favor of the LFIFO logging algorithm; the spatial LFIFO size is strictly larger than the corresponding ECETES overhead.

4.2 The simulator

The simulator used was tailor-made for the evaluation, source code is provided with the source of ECETES [3]. The task model is a fixed-priority scheduled sporadic task (priorities are set according to rate-monotonic) and complies to the system model of this paper. As the simulator has no notion of multiple consecutive starting points, LFIFO can be expected to work properly.

4.3 Simulation setup and configuration

We performed simulations with different tasksets, two of which will be examined more carefully. The properties of the two sets that we survey here are displayed in Table 1 and Table 2. As described by the system model, the periodicity of the jobs varies in the interval described by the task deadline and *sporadicity*. Each taskset was subjected to 23 suites of simulation with different values on the sporadicity of the tasks of the taskset. Each suite was simulated in $100,000$ instances.

In the setup for both ECETES and LFIFO, for each task, a queue was assigned to log the data-flow, and the monitored checkpoints. One queue was also assigned to monitor the control-flow of the entire system (i.e. context-switches).

The LFIFO logging algorithm requires some parameterization regarding individual queue-lengths, both for checkpoints and for control-flow entries. Note however that the size of a control-flow entry is likely to be much smaller than a data-flow entry [10]. In our setup, as we use the same entry size for all entries, a gained control-flow entry for ECETES will therefore reflect unrealistically in the results; if all entries are of the same size, a gained control-flow entry for ECETES can be directly transferred to a data-flow entry, but in a real system it would probably take several control-flow entries to do the same. Therefore, in our simulation setup, we have decided not to consider the memory allocated for logging the control-flow of the system as this would unjustly favor ECETES.

Thus, only the count of checkpoints from each task under an interval must be calculated. This is performed by assuming periodicities according to the worst-case sporadicity and calculating the number of jobs under a period equal to the Least Common Multiple (LCM) of these periods.

The length of each instance of simulation was allowed to vary randomly between high numbers (relative to the amount of memory allocated) so that the logging algorithm had to prioritize between the entries logged in every simulation.

4.4 Evaluation criteria

For each simulation performed, the shortest interval of replay (SIR) was measured based on the log contents from each of the algorithms. The result from a simulation is the relation between the ECETES and LFIFO SIR's; a result of 100% describes a tie between the subjects of the evaluation, a result below 100% indicates a win in favor of LFIFO, a result exceeding 100% is a win in favor of ECETES.

4.5 Simulation results

The results of the evaluation are displayed in: Figure 4 for taskset 1 and Figure 5 for taskset 2. According to Formula 1 below, we display the function of s, which is the behavior of ECETES as it relates to the behavior of LFIFO while the sporadicity varies.

$$s = \frac{SIR\, for\, ECETES}{SIR\, for\, LFIFO} \tag{1}$$

With increasing sporadicity, there are 23 simulation suites for each taskset, each of these suites has a column of five marks representing the four quartiles and the minimum value of s from the $100,000$ simulations (Q0, Q1, Q2, Q3, and Q4, from the bottom and up). For example, the next-to lowest mark in each column is Q1, the 25^{th} percentile. Thus, 25% of the $100,000$ calculates s's of that suite are smaller than that mark. Correspondingly, Q0 is the minimum value measured.

Hence, the graphs should be interpreted so that when Q2 (the median, the 50^{th} percentile) has crossed the 100% barrier, ECETES wins over, or out-performs, LFIFO in a majority of cases (assuming the current setting). In order to ease comprehension of the figures, we have marked the Q2 function as a continuos line.

The profile of Q0-values (i.e. the minimum SIR's observed) maintains a close-to-constant value in a given taskset-simulation, thus indicating that the number of simulations performed for each simulation setup is sufficient to provide a sound result.

The simulation of taskset 1, accounted for in Figure 4, shows that ECETES wins already at low sporadicity relative to the periodicities of the tasks in the set (Q2 breaks the 100%-barrier at a sporadicity between 700 and 800 tu).

As we can see, this result is then further confirmed by the simulation of taskset 2, the simulation is accounted for in Figure 5: When a third task is added

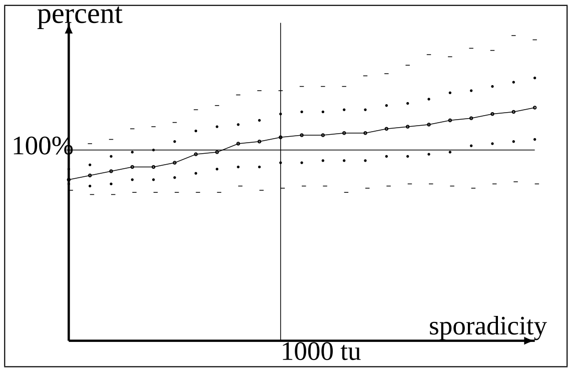

Figure 4. Taskset 1: ECETES performance in relation to LFIFO, marks in column represent quartiles of the test-result (Q0, Q1, Q2, Q3, Q4).

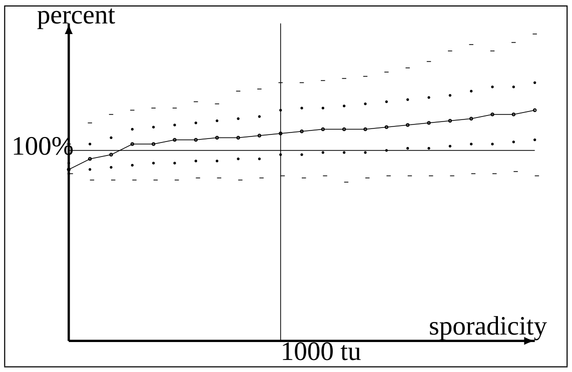

Figure 5. Taskset 2: ECETES performance in relation to LFIFO, marks in column represent quartiles of the test-result (Q0, Q1, Q2, Q3, Q4).

to the taskset, the sporadicity of the system execution is increased, and the ECETES performs even better in relation to LFIFO.

In order to concretize, we examine a particular simulation instance: With taskset 2, when the sporadicity is set to 1000 time units (tu), Task A has a periodicity in the closed interval $[2000, 2999]$ tu, the same closed interval for Task B is $[4000, 4999]$ tu, and $[8000, 8999]$ tu for Task C. In this particular simulation run, we measured that the Q2 SIR of ECETES was 39362 tu, and the same value for LFIFO was 37867 tu. Thus, the gain from using ECETES, in this particular simulation instance, seems clear.

The results presented here give a clear indication that ECETES will indeed outperform its competitor already at small sporadicities, it also seems that this gain will be increased as the taskset grows.

5 Conclusions

In the context of sporadic real-time systems, we have:

- shown the need for dynamic structures to organize memory during recording of executions.
- defined a measure for comparing algorithms used for this purpose based on the usefulness of their output. Note that this measure can be used also when comparing other sub-algorithms (for checkpointing, for to start replay, etc.) used in record/replay.
- introduced ECETES, the first dynamic structure that does not compromise the testability of the system.
- evaluated ECETES compared to a FIFO algorithm - the conclusion that ECETES distributes resources better in sporadic systems with limited memory.

The research presented here has served to connect many of the various results produced by our research group during the past four years.

5.1 Future work

Future improvements to ECETES include *unneeded* records: Records that are not part of a consistent checkpoint, or has an older consistent checkpoint in the queue, can be evicted without SIR-penalty, these should be considered first by ECETES. Further, the effectiveness of ECETES is dependent of the relative sizes of logged entries; if these sizes are multiples of each other, performance will be better. This issue is not reflected in the validation performed.

References

[1] K. Åström and B. Wittenmark. *Computer-Controlled Systems, Theory and Design*. Prentice-Hall International, 2^{nd} edition, 1990. ISBN 0-13-172784-2.

[2] M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor support for temporal predictability - the SPEAR example. In *Proceedings of the* 15^{th} *Euromicro Conference on Real-Time Systems*, pages 169–176, July 2003.

[3] J. Huselius. Source-code to the ECETES logging strategy. Technical Report, Mälardalen University, Department of Computer Science and Engineering, August 2003. Available at www.idt.mdh.se/˜jhi.

[4] J. Huselius, D. Sundmark, and H. Thane. Starting conditions for post-mortem debugging using deterministic replay of real-time systems. In *Proceedings of the* 15^{th} *Euromicro Conference on Real-Time Systems*, pages 177–184, July 2003.

[5] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[6] J. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software - Practice and Experience*, 29(2):125–142, 1999.

[7] P. Puschner and A. Burns. Writing temporally predictable code. In *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91. IEEE, January 2002.

[8] D. Stewart and M. Gentleman. Non-stop monitoring and debugging on shared-memory multiprocessors. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 263–269. IEEE Computer Society, May 1997.

[9] F. Sultan, T. Nguyen, and L. Iftode. Lazy garbage collection of recovery state for fault-tolerant distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 13(10):1085–1098, October 2002.

[10] D. Sundmark, H. Thane, J. Huselius, A. Pettersson, R. Mellander, I. Reiyer, and M. Kallvi. Replay debugging of complex real-time systems: Experiences from two industrial case studies. In *Proceedings of the* 5^{th} *International Workshop on Automated Debugging*, pages 211–222, September 2003.

[11] H. Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Kungliga Tekniska Högskolan, Sweden, May 2000.

[12] H. Thane and H. Hansson. Testing distributed real-time systems. *Journal of Microprocessors and Microsystems, Elsevier*, 24(9):463–478, February 2001.

[13] H. Thane, A. Pettersson, and H. Hansson. Integration testing of fixed priority scheduled real-time systems. In *IEEE/IEE Real-Time Embedded Systems Workshop*, December 2001.

[14] S. J. Vaughan-Nichols. Building better software with better tools. *Computer*, 36(9):12–14, September 2003.

[15] F. Zambonelli and R. Netzer. An efficient logging algorithm for incremental replay of message-passing applications. In *Proceedings of the 13th International and 10th Symposium on Parallel and Distributed Processing*, pages 392–398. IEEE, April 1999.