Trace Recording for Embedded Systems: Lessons Learned from Five Industrial Projects

Johan Kraft[1], Anders Wall[2] and Holger Kienle[1]

[1] Mälardalen University, Box 883, 72123, Västerås, Sweden
{johan.kraft, holger.kienle}@mdh.se,
[2] ABB AB, Corporate Research, Västerås, Sweden
anders.wall@se.abb.com

Abstract. This paper presents experiences from five industry collaboration projects performed between 2004 – 2009 where solutions for embedded systems trace recording have been developed and evaluated; in four cases for specific industrial systems and in the last case as a generic solution for a commercial real-time operating system, in collaboration with the RTOS company. The experiences includes technical solutions regarding efficient instrumentation and logging, technology transfer issues and evaluation results regarding CPU and RAM overhead. A brief overview of the *Tracealyzer* tool is also presented, a result of the first project (2004) which still is used by ABB Robotics and now in commercialization.

Keywords: embedded-systems, scheduling, tracing, trace-recording, monitoring, experiences, case-studies, overhead

1 Introduction

Trace recording, or tracing, is a commonly used technique useful in debugging and performance analysis. Concretely, trace recording implies *detection* and *storage* of relevant events during run-time, for later off-line analysis. This work targets *embedded* computer systems, i.e., specialized control systems used in many industrial products, for instance cars, trains, robotics and telecom systems. Embedded systems come in all sizes, from single-chip 8-bit computers with a few KB of RAM to 32-bit computers with features and performance comparable to PCs. Embedded systems are often *real-time* systems, meaning that the correctness also depends on response time, i.e., the latency from an input to the corresponding output. This must not exceed a specified requirement, the *deadline*. Embedded systems are typically implemented on multi-tasking real-time operating systems, where *tasks* (threads) share the CPU using *fixed-priority scheduling* [3, 5].

Trace recording for embedded systems can be performed at different abstraction levels and can be accomplished using software solutions, hardware solutions, such as Lauterbach Trace32[3], or hybrid hardware/software solutions such as the

[3] www.lauterbach.com

RTBx product of Rapita Systems[4]. A software-based approach means to add code instrumentation which logs the desired information in a software recorder module. This is typically performed without changing the application code but implies an overhead on CPU and RAM usage which for embedded systems can be of significance. Hardware solutions however require large, expensive equipment, mainly intended for lab use, while software solutions can remain active also in post-release use. This can be very valuable for reproducing customer problems, such as transient timing problems which only occur under rare circumstances.

The type of trace recording discussed in this paper is software-based trace recording for embedded systems, focusing on scheduling events, inter-process communication (IPC) events and relevant operating system calls. This is a higher abstraction level compared to, e.g., the work by Thane et al. [8] on replay debugging. However, our approach often gives sufficient information to pinpoint the cause of an error. If more information is necessary, this facilitates a detailed analysis using a debugger. In return, such recording is easy to integrate in existing systems since no application code instrumentation is required and the run-time overhead is very low, which allows for having the recording active also post-release. Many RTOS developers, including Wind River[5], ENEA[6] and Green Hills Software[7], provide tracing tools for their specific platform, but they typically never reveal any details or overhead measurements regarding their solutions. The main contribution of this paper is a synthesis of our experiences from five industry collaboration projects where trace recording solutions have been developed, including technical solutions used as well as results from recording overhead measurements.

2 Software Trace Recording

Software trace recorders typically operate by storing relevant events in a circular RAM buffer, as binary data in fixed-size records. In this manner, the recorder always holds the most recent history. In all implementations presented in this paper, a single ring-buffer is used for storing all types of events.

It is possible to detect scheduling events on most real-time operating systems, either by registering callbacks (hooks) on system events like task-switches, task creation and termination, or by modifying the kernel source code. The callback approach is possible on at least VxWorks (from Wind River) and OSE (from ENEA). Operating systems with available kernel source code, e.g., Linux and RTXC Quadros[8], can be modified to call the trace recorder module on relevant events. Åsberg et al. [2] has shown that for Linux (2.6 kernel), the only kernel modification required is to remove the "const" keyword from a specific func-

[4] `www.rapitasystems.com`
[5] `www.windriver.com`
[6] `www.enea.com`
[7] `www.ghs.com`
[8] `www.quadros.com`

tion pointer declaration. It is however possible to realize Linux trace recording without kernel modifications, if using a custom scheduler like RESCH [1].

In our approach we abstract from the context-switch overhead posed by the operating system and consider the task-switches as instantaneous actions. Only a single time-stamp is stored for each task-switch event and the OS overhead is instead accounted to the execution time of the tasks. Each task-switch event corresponds to exactly one *execution fragment*, i.e., the interval of uninterrupted execution until the next task-switch event. The rest of this section will discuss the information necessary for task-switch recording, the "what", "when" and "why". Due to space constraints, we focus of techniques for recording of task-switch events. Recording of IPC and operating system calls are however very similar.

2.1 Task Identity (the "What")

Most operating systems use 32-bit IDs for tasks, even though many embedded system only contain a handful of tasks. It is therefore often a good idea to introduce a short task ID, *STID*, using only 8 bits or 16 bits in order to make the task-switch events less memory consuming.

The STIDs needs to be allocated on task creation and quickly retrieved when storing task-switch events. This can be implemented by storing the STIDs in a data area associated with the task, for instance the task control block (TCB) in VxWorks, where there are unused "spare" field. In OSE there is a "user area" associated with each process, which can be used for this purpose.

Complex embedded systems with event-triggered behavior, such as telecom systems, often create and terminate tasks dynamically. In that case it is important to recycle the STIDs to avoid that they run out. This means that the termination of tasks must be registered in order to mark the particular STID as no longer in use. An STID may however not be reused for newly created tasks as long as there are references to a particular STID in the event ring-buffer.

2.2 Time-stamping (the "When")

Obtaining a time-stamp is normally a trivial operation, but standard libraries typically only allow for getting clock readings with a resolution of maximum 1 or even 10 milliseconds, depending on the tick rate of the OS. This is too coarse-grained for embedded systems timing analysis, since many tasks, and especially interrupt routines, have execution times measured in microseconds. Fortunately, embedded systems usually have hardware features for getting more accurate time-stamps, such as real-time clocks (*RTC*). In other cases, if the CPU frequency is constant, it is possible to use a CPU instruction counter register.

In order to reduce the memory usage when storing the events, a good method is to encode the time-stamps in a relative manner, i.e., to only store the time passed since the previously stored event, i.e., the durations of the execution fragments. If the absolute time of the last stored event is kept, it is possible to

recreate absolute time-stamps during off-line analysis. This allows for correlating the trace recording with other time-stamped logs created by the system.

The relative time-stamp encoding allows for using fewer bits for storing time-stamps, typically between 8 – 16 bits per event. A problem however occurs in cases where the duration of an execution fragment exceeds the capacity of the time-stamp field, i.e., 255 or 65535 time units. Handling the overflow issue for relative time-stamps introduces a tradeoff between memory usage and recorder-induced jitter (i.e., predictability). The most reliable but least efficient solution is to use enough bits for this purpose so that the overflow does not occur. A more efficient solution is to reduce the number of time-stamp bits to better fit the typical fragment duration, and instead introduce an alternative handling of the few cases where the number of time-stamp bits are insufficient. In this case, an extra "XTS" event (eXtended Time-Stamp) is inserted before the original event, carrying the time-stamp using enough (32) bits. This however introduces a control branch in the task switch probe, which might cause timing jitter in the recorder overhead and thereby additional timing jitter in the system as a whole, which can be bad for testability and predictability. We however believe that this jitter is negligible compared to other sources of jitter, such as execution time variations. The XTS approach is used in all five recorder implementations presented in this paper. Storing time-stamps of higher resolution (e.g., nanoseconds instead of microseconds) results in higher RAM usage due to either a wider time-stamp field or more frequent XTS events. However, if using a too low time-stamp resolution (e.g., milliseconds), some execution fragments may get a zero duration and thus becomes "invisible" in off-line visualization and analysis.

2.3 Task-switch Cause (the "Why")

In preemptive fixed-priority scheduling [3, 5] a task-switch may occur for several reasons: the running task might have been blocked by a locked resource, it might have suspended itself, terminated, or a task of higher priority might have preempted the task. This information is necessary to record in order to allow grouping of execution fragments into task *instances*, also known as task jobs. A task instance corresponds to one logical execution of the task, i.e., the processing of one work-package. The end of an instance is referred to as the *instance finish*, and corresponds to the termination of the task, i.e., exit from main function, or for non-terminating tasks when the task has performed one iteration of the main loop and enters a blocked or waiting state awaiting the next task activation, i.e., the start of the next instance.

From a trace perspective, a task instance corresponds to one or several consecutive execution fragments of the same task, possibly interleaved by execution fragments of other tasks, where the last fragment is ended by the instance finish, and where any previous fragments of the same instance is ended by preemption or blocking. The concepts of instances and execution fragments are illustrated by Figure 1, using an example with three tasks, where task H has the most significant priority and task L the least significant priority. Each execution fragment is labeled $T_{i,f}$, where T is the task name, i the instance number and f the

execution fragment number within the instance. The upper row indicates the task-switch cause: preemption (P) or termination (T) (i.e., instance finish).

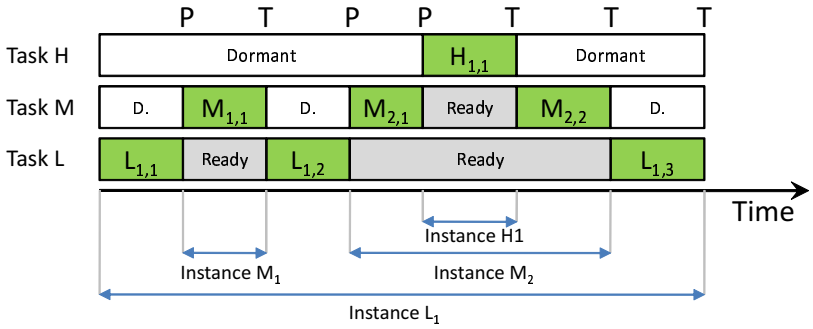

Fig. 1. Execution fragments and task instances

What counts as an instance finish for non-terminating tasks is system specific and depends on the software architecture. For non-terminating tasks there are two options for detecting instance finish: using the scheduling status or using code instrumentation. If a certain scheduling status can be unambiguously associated with the inactive state of a task, a task-switch due to this scheduling status can be regarded as the instance finish. The next execution fragment of this task is thereby the start of the next instance. This approach is however difficult if the task may be blocked for other reasons (other semaphore or message queues), since the scheduling status at best tells the type of resource causing the blocking, but not the identity of the specific resource. A pragmatic solution is to add code instrumentation in the task main loop, immediately before the operating system call corresponding to the instance finish. A problem with code instrumentation in the application code is that the application developer has to be aware of the recorder solution, maintain the instrumentation points properly and also adding such instrumentation when adding new tasks to the system.

3 The Tracealyzer Tool

The Tracealyzer is a visualization tool with analysis capabilities for various timing and resource usage properties. The first version of the Tracealyzer was developed in 2004, in the project described in Section 4.1 in collaboration with ABB Robotics.

The main view of the tool displays a task trace using an novel visualization technique. Other trace visualization tools, such as the Wind River WindView, uses a trace visualization technique similar to a logic analyzer or Gantt-style charts, where the status of every task is displayed at all times, with one row or

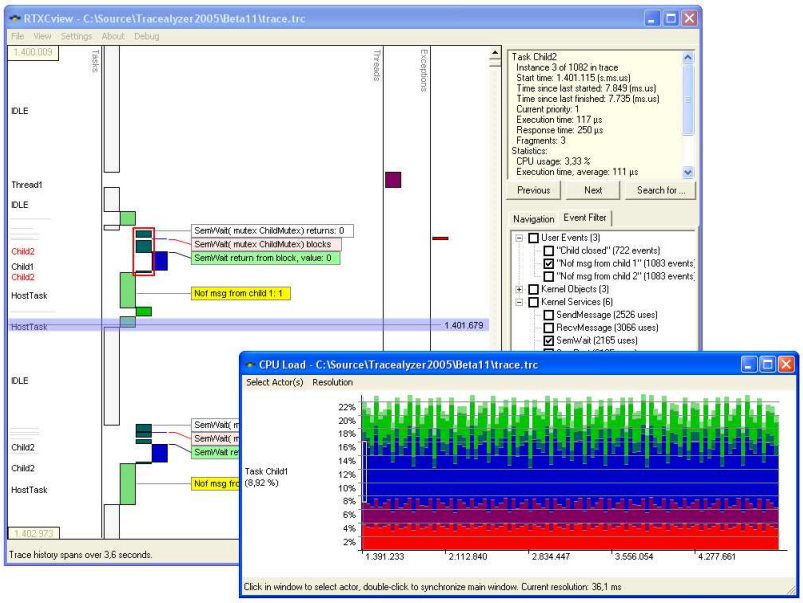

Fig. 2. The Tracealyzer (version 2.0)

column per task. Such visualizations become hard to comprehend when zooming out to overview a longer scenario and the user may need to scroll in two dimensions.

In contrast, the visualization used by the Tracealyzer focuses on the task preemption nesting and only shows the currently active tasks, as depicted by Figure 2. This makes the trace easier to overview, especially long and complex scenarios with many tasks involved. The tool also provides a CPU load view over the entire trace. The two views are synchronized; the time window display in the main window is indicated in the CPU load overview and by clicking in the CPU load overview the trace view displays the corresponding time window. The tool has advanced features for searching, with several filters, and can also generate a report with detailed timing statistics for each task. The tool also allows for exporting timing data regarding tasks and other events to text format. More information about the tool is available at `www.percepio.se` where a demo version can be downloaded.

The Tracealyzer tool is since 2009 in commercialization by Percepio AB in collaboration with Quadros Systems, Inc. who develops the real-time operating system RTXC Quadros. A Quadros version will soon be marketed by Quadros Systems, under the name RTXCview.

4 Five Industrial Trace Recorder Projects

Starting 2004, five industry collaboration projects have been performed by the main author where trace recorders have been implemented for different existing systems. Four of these projects have included evaluation with respect to CPU and RAM usage. Three of the projects have lead to industrial deployment of the results, in one case as the coming official tracing tool for a commercial real-time operating system. The purpose of these projects have varied slightly, but all have included trace recording and visualization using the Tracealyzer, described in Section 3. The research motivation for these projects have been to verify the applicability of custom (third party) trace recording on common platforms for embedded systems. The motivation of the industrial partners where mainly to investigate the suitability of the Tracealyzer tool, which served as a "low-hanging fruit" for collaboration.

4.1 The RBT Project

ABB develops a control system for industrial robots, IRC 5. This is a large and complex embedded software system, consisting of around 3 million lines code. The operating system used is VxWorks, and the hardware platform is an Intel-based Industry PC. At the time of the evaluation, this system used an Intel Pentium III CPU and had 256 MB of RAM. It moreover has a flash-based hard drive, a network connection and an onboard FTP server.

Since VxWorks has features for registering callbacks on task-switch, task creation and task deletion, these events could be captured without kernel modifications. The task-switch callback function receives pointers to the task control blocks (TCBs) of both the previously executing task and for the task that is about to start. The developed recorder uses 8-bit STIDs, stored in an available "spare" field in the TCB by the task create callback routine. The task names are stored at creation time in a list of tasks, indexed by the STID.

All types of events are stored in a single ring buffer, using a fixed event size of 6 bytes. This required the use of bit-wise encoding in order to fit the desired information into the 48 bits available. The two first bytes are used to store two pieces of information in an asymmetric manner, where 2 bits are used for the event code and 14 bits for a relative time-stamp, obtained from an instruction counter of the Intel CPU used by this system. Since the time-stamp resolution used in this recorder is 1 μs, this solution allows for a execution fragment duration up to 2^{14} μs (16.4 ms). This is typically more than enough for this system; there are usually several task-switch events every millisecond. However, in some system modes, such as during system startup, the task-switch rate is much lower and the 14 bits may then be insufficient. As a precaution, an additional "XTS" event (eXtended Time-Stamp) is stored if the relative time-stamp does not fit in 14 bits. The XTS event stores the relative time-stamp using 32 bits and overrides the time-stamp field of the associated (following) event.

Recording inter-process communication events was considered important and this was accomplished by adding code instrumentation in the OS isolation layer.

Semaphore operations are however not instrumented; they are very frequent in this system and it was feared that monitoring these would cause a major additional recording overhead. The event rate of the ABB system when recording task scheduling and IPC operations was found to be around 10 KHz. A ring buffer capacity of 100 000 events (600 000 bytes) therefore gives a trace history of around 10 seconds. The runtime of a recorder probe was found to be on average 0.8 μs, which at the typical event-rate of 10 KHz translates into a CPU overhead of 0.8 %.

As mentioned, ABB Robotics personnel decided after this project to integrate the recorder in their control system IRC 5 and to keep it active by default, also in the production version. The Tracealyzer is today used systematically at ABB Robotics for troubleshooting and for performance measurements. The recorder is triggered by the central error handling system, so whenever a serious problem occur a trace file is automatically stored to the system's hard drive. A trace file is in this case only about 600 KB and can therefore easily be sent by e-mail for quick analysis, e.g., if a customer experiences a problem.

4.2 The ECU project

The system in focus of this project was the software of an ECU, i.e., a computer node in a vehicular distributed system developed by Bombardier Transportation[9]. Since also this system used VxWorks a similar recorder design could be used as in the RBT project. The company developers were mainly interested in the CPU usage per task, as well as for interrupt routines, during long-term operation of the vehicle. The hardware platform was a Motorola[10] PowerPC 603 running at 80 MHz.

In initial experiments using the Tracealyzer tool, the main problem was the endianness; the Motorola CPU uses big endian encoding, while the Tracealyzer expected little-endian encoding. In the first experiments in using the Tracealyzer for this system, the solution was a recorder design where all data is stored in little-endian format during run-time, by assigning each byte explicitly. This is far from optimal with respect to the CPU overhead of the recording and should be avoided. The latest version of the Tracealyzer assumes that the recorder writes the data to a binary file in native format and therefore detects the endianness, and converts if necessary, while reading the trace file. The endianness is detected by using a predefined 32-bit value, where the four bytes have different values, which is written to a predefined file location by the recorder, typically in the very beginning. An off-line analysis tool can then find the endianness from the order of these values.

Unlike the RBT project, this project included recording of interrupt routines. The operating system VxWorks does not have any callback functionality or similar for interrupts, but the interrupt controller of the CPU allowed for this.

[9] `www.bombardier.com`
[10] Now Freescale

Interrupt routines could thereby be recorded as high-priority tasks, by adding task-switch events to the main ring buffer in the same way as for normal tasks.

An interesting requirement from Bombardier was that the recorded information should survive a sudden restart of the system and be available for post-mortem analysis. This was accomplished by using a hardware feature of the ECU; the event buffer was stored in Non-Volatile RAM (NVRAM). During the startup of the system, the recorder recovers any trace data stored in the NVRAM and writes it to a file, thereby allowing for post-mortem analysis. The ECU was equipped with 4 MB of NVRAM which is plenty since the company only needed a 2.5 second trace history. Since it was only desired to log task-switch events in this project, i.e., no IPC events like in the RBT case, it was possible to reduce the event size from six to four bytes per event.

A recorder and a company-specific analysis tool was developed in a Master's thesis at Bombardier[4], but the Tracealyzer was not used after the initial tests leading to the thesis project. One of the Masters students was however employed by the company after the thesis project.

4.3 The WLD Project

This system is also an ECU-like computer, although not in the vehicular domain and the company is anonymous in this case. The computer system in focus is a node in a distributed system, with the overall purpose of automated welding for production of heavy industrial products. The computer in focus controls an electrical motor and is connected to a set of similar computer nodes over a field bus. The CPU used was an Infineon XC167, a 16-bit CPU running at only 20 MHz. The operating system used was RTXC Quadros.

Since the kernel source code of RTXC Quadros is available for customers, the recorder could be integrated in a custom version of the kernel. It was however not trivial to find the right location where to add the kernel instrumentation, especially for the task-switch events, since parts of the context-switch handling is written in assembly language. Time-stamps were obtained from the real-time clock (RTC) feature of the Infineon XC167 CPU and stored in a relative manner in the same way as in the previous cases.

There was no need for using short task IDs (STIDs) for reducing memory usage, since RTXC Quadros already uses 8-bit task handles. However, dynamic creation of tasks required an indirect approach, involving a lookup table, as the task handles of the operating system are reused. The lookup table contains a mapping between the RTXC task ID and the index of the task in an recorder-internal list of tasks, which is included in the generated trace file. The recorder task list contains the name and other information for up to 256 tasks. On task creation, the list is searched in order to find a matching task, so repeated dynamic creations of a single task only generates a single entry. However, there was no "garbage collection" in the recorder task list, so tasks which are no longer in the trace history still occupy an entry. This issue is however solved in the latest recorder implementation, described in Section 4.5. Interrupt routines were

recorded by adding two probes in every interrupt service routine (ISR). Task-switch events are stored in the beginning and in the end of the ISR, using the interrupt code to look up a "faked" task entry, specified in a static table containing all interrupts. Nested interrupts are supported using a special purpose stack, holding the identity of the preempted ISRs, as well as the currently executing task.

The CPU overhead of the recording was measured and found higher than in previous cases, although still acceptable. The event rate was found to be around 500 Hz, i.e., about ten times less than in the ABB system, but the slow, low-end CPU (16-bit, 20 MHz) caused relatively high probe execution times, around 60 μs. This is 75 times longer than the probe execution times in the ABB system (0.8 μs). With a 500 Hz event rate, this translates into a CPU overhead of 3 %, which is significant, but probably not a serious issue compared to the potential benefits of trace recording. However, this recorder was not optimized for CPU usage; it was rather a first prototype on this platform. Several optimizations/fixes are possible in order to reduce the CPU usage of this recorder solution, as discussed in Section 4.6.

In a first evaluation by developers at the company, the welding system recorder was used together with the Tracealyzer tool in order to pinpoint the cause of a transient error which they previously had not been able to find. By studying a recorded trace in the Tracealyzer tool they could find that the error was caused by a wrongly placed "interrupt disable" instruction, which allowed for interrupts occurring during a critical section where interrupts should have been disabled. The company did however not integrate the developed recorder solution on a permanent basis, but has used the solution later for similar purposes. On those occasions, they have created a custom build using the instrumented RTXC Quadros kernel. This can lead to probe effect [7] problems, i.e., that the activation (or deactivation) of recording changes the system behavior.

4.4 The TEL Project

This project was performed together with an anonymous company in the telecom industry, which develops products based on the operating system OSE from ENEA. The particular system studied used a high-end PowerPC CPU, running at 1 GHz and with 256 MB of RAM. This project had the goal of providing means for exact CPU load measurements. Previously they had used a tool which sampled the currently executing task at randomly selected times and in that way got an approximate picture of the CPU usage of the various tasks. This was however considered too inaccurate. A Master's thesis project was initiated in 2008 in order to develop a recorder for this system [6].

A recorder for the Tracealyzer tool was developed and evaluated using standard performance tests of the system. The recorder used the "kernel hooks" feature of OSE, which is similar to the callback features in VxWorks, and 16-bit STIDs for tasks (*processes* in OSE terminology), stored in the "user area" of the process. The main problem was that OSE did not allow direct access to the kernel memory, for reading the process control block. It was thereby not possible

to get the scheduling status of the tasks, which is necessary in order to identify task instances. A workaround was implemented, the Tracealyzer was modified for this case, so that priorities were used instead of status. This assumes that the priorities are static since the recorder cannot read them at the task-switch events, only at task creation. The resulting recorder was evaluated in the company lab using their normal test-cases for load testing. The CPU overhead of the recorder was found to be 1.1 % at an event rate of 18 KHz and a CPU load of 30 %. This result has to be considered as most acceptable, especially since the recorder was not optimized for CPU usage.

The project was successful in meeting the requirements, i.e., providing means for exact CPU load measurement, but the Tracealyzer could not be used to its full potential due to security restrictions in the OSE operating system, which prevented direct access to the process control blocks. The CPU overhead of the recorder was measured under realistic conditions and found to be relatively low despite a high event rate. The company did however not use the resulting recorder since it was not mature enough for industrial deployment, which requires a very robust solution, and since there was no obvious receiver at the company who could take over the recorder development and verification.

4.5 The RTOS Project

In 2009 the thesis author was contacted by a representative of Quadros Systems, Inc. who expressed interest in a collaboration aiming at developing a new trace tool for their operating system. This resulted in the development of the second generation Tracealyzer, along with a special version for Quadros Systems named *RTXCview*. This project also included the development of a whole new recorder design, in close collaboration with the chief engineer at Quadros Systems.

This recorder has little in common with the previous four versions. A major difference is that this recorder is designed for logging of generic operating system services without any hard-coded information in the recorder design. The recorder contains no assumptions on the operating system services that should be logged, this is configured through kernel instrumentation and using a configuration file of the Tracealyzer/RTXCview. All information needed by the off-line tool is stored in a single block of data which is statically initialized during compile-time. This eliminates the need for calling a recorder initialization routine at system startup, which was necessary in the previous versions. This design reduces the startup time of the system and makes it easy to retrieve the trace recording, e.g., if the system has stopped on a breakpoint using a debugger. This recorder does not use any bit-wise manipulations, which should reduce its CPU usage significantly. To achieve this, a larger event size was necessary, using eight bytes per event instead of four or six bytes.

In this design, there is no explicit task-list, as in other earlier recorders, but instead there is a generic symbol table which contains the names of tasks, user events, semaphores, and other named objects. A string added to this symbol table returns a 16-bit reference, the byte index of the string in the symbol table. If an identical string already exists in the symbol table, a reference to the existing

string is returned instead of creating a new entry. This is therefore memory efficient and solves the issue of repeatedly created dynamic tasks. The symbol table lookup is fast since all symbol names which share a 6-bit checksum are connected in a linked list, as depicted by Figure 3. This however requires two extra bytes per symbol name, for storing the index of the next symbol with the same checksum, and an array holding 64 16-bit values, the linked-list heads. If a longer checksum (i.e., more checksum bits) is used, the look-up time is reduced, but the amount of memory required for the array of linked-list heads doubles for every extra checksum bit. For systems with plenty of memory, an 8-bit checksum should however not be any problems, since it only requires 512 bytes.

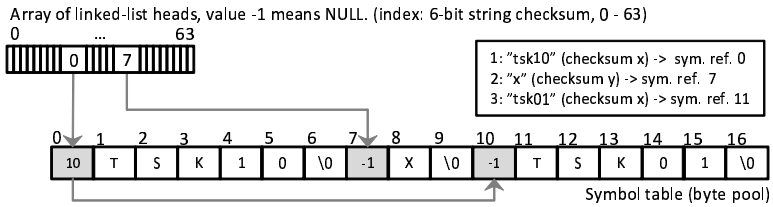

Fig. 3. The Symbol Table

On task-switch events, the 8-bit RTXC task handles are stored without bothering about possible later reuse of the handle, which then might change the meaning of the currently stored handles. This is instead resolved off-line. The names of the currently active tasks are stored in a "dynamic object" table which is updated on task creation. When a task is terminated ("closed" in Quadros terminology), the name from the dynamic object table is stored in the symbol table and the resulting reference is stored, together with the RTXC task handle, in a special "close" event, which informs the off-line analysis tool that this mapping was valid up until this point. The off-line analysis can then find the correct task names of each execution fragment by reading the event trace backwards, starting at the trace end, and for each close event update the current mapping between RTXC task handle and name.

The described approach for handling reuse of dynamic task handles is used for all types of dynamically created kernel objects in RTXC Quadros, i.e., tasks, semaphores, mailboxes, alarms, etc. Time-stamps are stored in a relative manner, using 8, 16 or 32 bits per event, depending on the number of bytes available for each event type. Like in the other projects, XTS events are inserted when the normal time-stamp field is insufficient. The time unit of the time-stamps does not have to be microseconds as the time-stamp clock rate is specified in the recorder and provided to the off-line analysis tool, which converts into microseconds. It is thereby possible to use the hardware-provided resolution directly without run-time conversion into microseconds. Another time-related aspect is that absolute time-stamps are maintained also if the recording is stopped abruptly, e.g., due to

a crash or breakpoint. The absolute time of the last stored event is kept updated in the recorder's main data structure and is thereby available for the off-line analysis. From this information and the relative time-stamps of the earlier events it is possible to recreate the absolute time-stamps of all events in the trace.

A prototype of this recorder has been implemented and delivered to Quadros Systems, who at the moment (Spring 2010) are working on integration of the recorder in their kernel. There are no big problems to solve; it is mainly a question of the limited development resources of Quadros Systems. No evaluation regarding the CPU overhead of this recorder has yet been performed. Developing and verifying a trace recorder for an operating system is much harder than for a specific embedded system, since an operating system recorder has to work for all hardware platforms supported by the operating system.

4.6 Summary of Recording Overhead Results

This section summarizes the measured recording overhead imposed by the recorders in the four cases where such measurements have been made, i.e., all cases except for the RTOS case (Section 4.5). The results are presented in Table 1.

Table 1. Measured recording overheads in four industrial cases

Case	OS	CPU	F (MHz)	ES (bytes)	ET (μs)	ER (KHz)	CPU OH (%)	RAM OH (KB/s)
RBT	VW	P. III	533	6	0.8	10.0	0.8	60.0
ECU	VW	PPC 603	80	4	2.0	0.8	0.2	3.1
WLD	RTXC	XC167	20	4	60.0	0.5	3.0	2.0
TEL	OSE	PPC 750	1000	4	0.6	18.0	1.1	72.0

In Table 1 "ES" means Event Size, i.e., the number of bytes used per event. ET means average probe execution time, ER means average event rate, in a typical recording. CPU OH means the corresponding CPU overhead and RAM OH means the corresponding number of (event buffer) bytes used per second. Note the relatively long probe execution time in the WLD case: 60 μs. The next faster ET, for ECU, was 30 times shorter even though the clock frequency was only four times higher in this case. This is probably due to the difference in CPU hardware architecture, the CPU in the WLD case is a 16-bit micro-controller, while more powerful 32-bit CPUs were used in the other cases.

Note that the four evaluated recorders were for low RAM usage, on the expense of higher CPU usage. It therefore possible to reduce the CPU overhead significantly by instead optimizing for CPU overhad, e.g., by increasing event size in order to avoid bit-wise encoding. Other possible optimizations are to move as much functionality as possible off-line (e.g., time-stamp conversion) and by using "inline" functions and macros instead of C functions. The latest recorder design, presented in Section 4.5, includes these improvements and should thereby give significantly lower CPU overhead, although not yet confirmed by experiments.

5 Lessons Learned

An important consideration is choosing an appropriate level of detail for the trace recording, e.g., should the recording include interrupt routines, or semaphore operations? This is a trade-off between the value of the information, with respect to the purpose of the recording, compared to the consequences of the associated recording overhead, such as a reduction in system performance, or increased unit cost if compensating the overhead with better but more expensive hardware. Including too little information may however also lead to increased costs if quality assurance becomes harder.

A related consideration is the trade-off between CPU usage and memory usage implied by using more advanced storage techniques, such a bit-wise encoding or data compression, which are more memory efficient but also more CPU demanding. We however believe that such techniques should generally be avoided in order to prioritize lower CPU overhead, since there is often unused RAM available, for a larger recording buffer, and if not so, a shorter trace history might be acceptable. A lower CPU overhead however improves system responsiveness and also reduces the risk of probe effects. One exception could be low-end embedded systems with very little RAM where a long trace history is very important, more important than CPU overhead. No type of system matching this description is however known to the authors.

Another consideration is whether the recorder should be integrated in the system on a permanent basis, or only activated when necessary. A permanent integration means that the CPU and memory overhead of the trace recording becomes permanent and may therefore reduce the system performance as experienced by customers. We however recommend this approach for several reasons: (1) the risk for probe effects is eliminated since the recording becomes an integrated and tested part of the system, (2) a trace is always available for diagnostic purposes, (3) the availability of a trace lowers the threshold for developers to begin using the trace recorder, (4) the recording cost in terms of CPU and memory usage is typically very small and therefore well motivated by the benefits. An exception to this recommendation would be systems which are highly focused on average-case performance and where the unit cost is a major issue, such as low-end multimedia devices.

The authors recommend that all types of events are stored in a single ring-buffer with fixed-size entries. This way, the chronological order of events is maintained. More advanced solutions using multiple buffers and/or variable-sized events may reduce memory usage, but leads to higher recorder complexity, higher risk of errors in the recorder and higher CPU overhead.

A good strategy is to store the information in a single data structure, which is statically allocated and initiated. This way, the recorder does not need a special initialization routine, but is recording directly at startup. Moreover, using this approach, the data can be easily fetched, e.g., using a debugger when stopped on a breakpoint, without having to execute a special “save” routine. As file format for the off-line tool, use a binary image of the run-time data structure. Differences in endian encoding can be resolved when reading the file.

A recommendation is to design trace recorders as simple and robust as possible and instead place the "intelligence" in the off-line tool. For instance, timestamps should not be converted during run-time, bit-wise encoding should be avoided and startup initialization routines should be replaced by static initialization. A simple recorder design is also important if the recorder is to be trusted and maintained by the target system development organization. In that case, make sure there is an explicit receiver, a developer or lower level manager, which can take over the responsibility for the developed solution. This is believed to be the key success factor in the projects which led to industrial use.

6 Conclusions

This paper has presented experiences from five industry collaboration projects performed between 2004 – 2009 where solutions for embedded systems trace recording have been developed and evaluated. Several technical solutions and trade-off considerations have been presented and discussed. The CPU overhead of trace recording can be expected to be below 1 % on most systems using 32-bit CPUs, although it could reach about 3.6 % in the telecom system case if extrapolating the event rate up to 60 KHz at maximum CPU load. This is however an extreme case with respect to event rate. Implementation of trace recorder was possible as a third party developer on all three operating systems, although one required a different approach due to kernel security restrictions.

References

1. Åsberg, M., Kraft, J., Nolte, T., Kato, S.: A loadable task execution recorder for Linux. In: Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (July 2010)
2. Åsberg, M., Nolte, T., Perez, C.M.O., Kato, S.: Execution Time Monitoring in Linux. In: Proceedings of the Work-In-Progress session of 14th IEEE International Conference on Emerging Techonologies and Factory (September 2009)
3. Audsley, N.C., Burns, A., Davis, R.I., Tindell, K.W., Wellings, A.J.: Fixed priority pre-emptive scheduling: An historical perspective. Real-Time Systems Journal 8(2/3), 173–198 (1995)
4. Johansson, M., Saegebrecht, M.: Lastmtning av CPU i realtidsoperativsystem. Master's thesis, Mälardalen University, Västerås, Sweden (2007)
5. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in hard-real-time environment. Journal of the Association for Computing Machinery 20(1), 46–61 (1973)
6. Mughal, M.I., Javed, R.: Recording of Scheduling and Communication Events on Telecom Systems. Master's thesis, Mälardalen University, Västerås, Sweden (2008)
7. Schutz, W.: On the Testability of Distributed Real-Time Systems. In: Proceedings of the 10th Symposium on Reliable Distributed Systems, Pisa, Italy. Institut f. Techn. Informatik, Technical University of Vienna, A-1040, Austria (1991)
8. Thane, H., Hansson, H.: Using Deterministic Replay for Debugging of Distributed Real-Time Systems. In: 12th Euromicro Conference on Real-Time Systems (ECRTS '00). pp. 265–272. IEEE Computer Society (June 2000)