Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems

Mike Holenderski, Martijn M.H.P. van den Heuvel, Reinder J. Bril and Johan J. Lukkien
Department of Mathematics and Computer Science
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands

***Abstract*—Understanding and validating the timing behavior of real-time systems is not trivial. Many real-time operating systems and their development environments do not provide tracing support, and provide only limited visualization, measurements and analysis tools. This paper presents Grasp, a tool for tracing, visualizing and measuring the behavior of real-time systems. Grasp provides a simple plugin infrastructure for extending it with custom visualization and measurement methods. The functionality of Grasp is demonstrated based on experiences during the development of various real-time extensions for the commercially available μC/OS-II real-time operating system. All the tools presented in this paper are open source and freely available on the web[1].**

I. Introduction

A real-time system is usually comprised of a real-time application running on top of a real-time operating system. One such operating system is μC/OS-II [Labrosse, 1998]. It is maintained and supported by Micrium, and is applied in many application domains, e.g. avionics, automotive, medical and consumer electronics. Our choice of μC/OS-II is in line with our industrial and academic partners.

Based on the requirements posed by the real-time applications we are researching, we have set out on extending μC/OS-II with several real-time primitives. In [Holenderski et al., 2008] we presented an application of our industrial partner in the surveillance domain and pointed out a problem with their current system. We proposed a solution and identified several extensions required from the underlying real-time operating system. These include processor reservations based on deferrable servers, and support for resource sharing based on the Stack Resource Policy (SRP) [Baker, 1991].

A. Problem Description

During the development of these μC/OS-II extensions we needed to validate the behavior of the introduced primitives. Many commercial off-the-shelf real-time operating systems and their development environments, including μC/OS-II, do not support tracing, and provide only limited visualization and analysis support. Moreover, current visualization tools only allow to visualize single level scheduled systems. Finally, most commercial tools are not easily extensible.

[1]The work presented in this paper is supported in part by the European ITEA2-CANTATA project and the Dutch HTAS-VERIFIED project. The Grasp player together with two demo traces is available at http://www.win.tue.nl/~mholende/grasp

B. Contributions

In this paper we address the problem of tracing, visualizing and measuring the behavior of real-time systems and present Grasp, which is a set of tools addressing this problem. It comes with a powerful set of features out of the box, such as visualization of servers in hierarchical scheduling and buffer usage for tasks communicating via shared buffers. It also provides a simple infrastructure for extending it with custom visualization and measurement plugins. We used Grasp extensively during the development of several extensions of μC/OS-II. The target systems were executed in the cycle accurate OpenRISC simulator [OpenCores, 2009].

C. Outline

The remainder of this paper is structured as follows. In Section II we summarize the related work, followed by a description of our execution environment in Section III. Section IV is the main contribution of this paper. It presents Grasp, illustrated with examples from extending μC/OS-II with additional real-time primitives. Section V concludes the paper and outlines the future work.

II. Related work

In this section we outline the existing work related to the real-time operating system under consideration, followed by a discussion of the support for tracing, visualization, and measurements provided by existing tools.

A. μC/OS-II and its tools

Micrium provides the full μC/OS-II source code accompanied by an extensive documentation [Labrosse, 1998]. The μC/OS-II kernel provides preemptive multitasking, and the kernel size is configurable at compile time, e.g. services like mailboxes and semaphores can be disabled. It is well suited for proprietary extensions and experimentation.

A μC/OS-II application can enable a built-in statistics task, which collects information about the processor usage of all tasks in the system. Micrium also provides a powerful monitoring tool called μC/Probe, allowing to inspect the state of any variable, memory location, and I/O port in a μC/OS-II enabled application during runtime. However, there is no tracing support for μC/OS-II.

B. Tracing

There are basically two approaches to tracing: instrumentation and sampling [Mughal and Javed, 2008]. With instrumentation, code is inserted in key places of the system (such as the top of particular method calls). This code then records the events at runtime for later offline analysis. With sampling, the system remains unmodified and is instead analyzed periodically by a profiler during runtime, allowing inspection of metrics such as the amount of CPU time used by processes and/or functions. Grasp and all other tools presented in this section take the instrumentation approach, where a recorder component generates a trace file which later serves as input for the visualization application.

The format of the trace file has several implications. A standard textual file format (e.g. XML used by VET [McGavin et al., 2006]) can be used as input for other tools with relatively little effort. A binary file format (e.g. used by the Tracealyzer [Mughal and Javed, 2008]) results in smaller trace files, which can be important when tracing is a part of the target system after deployment in the field. A Grasp trace is a Tcl [Welch et al., 2003] script. It is less verbose than XML, but not as compact as a binary representation. However, its main advantage is the flexibility it offers for the Grasp player, as explained in Section IV.

C. Trace visualization

Traces contain huge amounts of data, which may be of no interest for a particular investigation. Several visualization tools therefore provide filtering mechanisms, which allow the user to display only those events he is interested in. Tracealyzer [Mughal and Javed, 2008] offers predefined filters which can be changed during the visualization, allowing to hide certain events, such as locking of a semaphore. VET [McGavin et al., 2006] provides a plugin mechanism allowing an expert user to implement custom filters, which then can then be reused by regular users. The Grasp player presented in this paper allows to filter trace events referring to certain tasks.

A trace can be visualized in different ways, e.g. one may want to show the task execution on a timeline, or how each task contributes to the current processor load. Tracealyzer provides a timeline and load view. VET provides a message sequence and class association diagrams, but also supports tracing of custom events, and an API which can be used to implement custom visualizations. This API, however is limited to a fixed event structure. The file format for Grasp traces allows to easily extend the trace with arbitrary custom events and visualizations.

To the best of our knowledge, all existing tracing tools can visualize only single level scheduling. Grasp on the other hand, also allows to visualize hierarchical systems by means of illustrating the budget consumption of servers. In case tasks communicate via shared buffers, Grasp can also provide insight into the contents of the buffers at any moment during the traced system execution.

D. Trace measurements

Tracealyzer measures the execution time, response time and number of fragments for each job and the corresponding worst case and average case values of all jobs of a task. Grasp measures the execution and response time of jobs and provides a summary of the average, best case and worst case times of all jobs of a task. It also allows to easily implement custom measurement tools, as illustrated in Section IV-D.

III. Simulation platform

To run our target systems we needed an execution environment supporting μC/OS-II. To avoid the inconveniences of running the target systems directly on hardware, we chose to run them inside the cycle-accurate OpenRISC simulator.

The OpenCores project [OpenCores, 2009] provides an open source platform architecture, including software development tools. The hardware architecture comprises a scalar processor and basic peripherals to provide basic functionality [Bolado et al., 2004]. The small and predictable processor architecture makes the OpenRISC processor suitable for real-time computing [Whitham and Audsley, 2006]. The accompanying Software Development Kit (SDK) is based on GNU tools. It includes a C/C++ compiler, linker, debugger and architectural simulator. The OpenRISC simulator allows simple code analysis and system performance evaluation. Recently, we created a port for μC/OS-II to the OpenRISC platform[2].

Unlike other μC/OS-II simulators, such as the Windows and Linux ports [uco, 2007], the OpenRISC port provides a cycle-accurate simulation: it is independent of the system load due to other applications on the host operating system and therefore provides predictable timing behavior for timed events.

It is important to note that the only interface to the simulator during runtime is via the standard input and standard output. In particular, there is no support for reading from or writing to files on the host operating system.

IV. Grasp

Grasp is composed of three entities, shown in Figure 1. The recorder is responsible for generating the trace of the target system. The generated trace file contains the raw data from a particular run, which is not comprehensible in its raw form. The player reads in a trace and displays it in an intuitive way.

Note that the recorder and the player are independent of each other, as long as the trace follows the predefined format. In this paper we demonstrate a Grasp recorder for μC/OS-II. Porting Grasp to other operating systems requires only implementing a Grasp recorder.

A. Grasp recorder

The Grasp recorder is implemented as a library providing functions to initialize the recorder, log events, and finalize the recorder. Calls to the event logging methods are inserted at several places inside the kernel to log common events, such as context switches and the arrival of periodic tasks. The

[2] A precompiled OpenRISC tool chain for Linux (Ubuntu 8.10) is available at http://www.win.tue.nl/~mholende/ucos

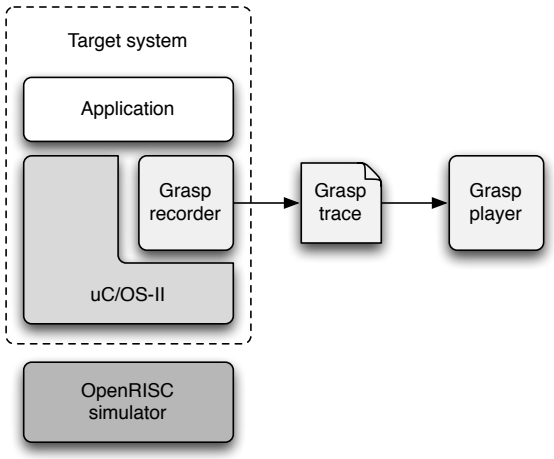


Fig. 1. The Grasp architecture

recorder also provides a function to log custom events, which the programmer may call inside his application.

To limit the interference with the target system, during runtime the μC/OS-II Grasp recorder stores the event information in a binary format in an array in the memory.

At the end of a simulation the log array is traversed, a trace is generated and written to a file in text format. This way the I/O overhead associated with writing the trace to a file is postponed until the very end and limits the interference with the target system during a simulation run. Note that the OpenRISC simulator has no file system support, but does allow to print to the standard output via the `printf()` method. The recorder therefore prints the contents of the trace file to the standard output, which is then redirected to a file on the host operating system.

B. Grasp trace

The Grasp trace file is actually a Tcl [Welch et al., 2003] script. An excerpt from an example trace is shown in Figure 2.

```
1   ...
2   plot 50 taskArrived Task0x11da50
3   plot 50 jobPreempted Job0x11d948_1 \\
4             -target Job0x11da50_1
5   [Job Job0x11da50_2] initWithTask Task0x11da50 \\
6             -name "Task1 2"
7   plot 50 jobResumed Job0x11da50_2
8   plot 60 jobCompleted Job0x11da50_2
9   plot 60 jobPreempted Job0x11da50_2 \\
10            -target Job0x11d948_1
11  plot 60 jobResumed Job0x11d948_1
12  ...
```

Fig. 2. An excerpt from a trace file.

Each line in the trace is a Tcl command. A Tcl command has a very simple syntax: method name followed by a possibly empty list of arguments separated by spaces. Let us take a closer look at the first line in Figure 2, which indicates the arrival of task `Task0x11da50`. The meaning of this command is the following:

- `plot` is the method name responsible for handling this trace event. In Section IV-D we will see how this dispatching mechanism is used to implement plugins. The method determines the semantics of the following arguments. In this case, the `plot` method expects at least two arguments:
- `50` is the event time. Time is measured in ticks. In our simulations 1 tick corresponds to 1ms.
- `taskArrived` is the event kind.
- `Task0x11da50` is an additional argument. In this case it identifies the task which has arrived. In Grasp each trace object such as a task or a job has a unique identifier.

Figure 2 shows also several other events. For example at time 50 the job with id `Job0x11d948_1` is preempted by the job with id `Job0x11da50_1`. In the following sections we will describe other events supported by Grasp.

Every task or job referred to by an event needs to be created first. Line 5 in Figure 2 creates a job with id `Job0x11da50_2` for a task with id `Task0x11da50`. The optional parameter `-name` specifies a custom job name, referring to the second job of task `Task1`.

Note that since a trace is a Tcl script it may contain any Tcl code, in particular it may define its own methods, include loops, etc. While we do not exploit this feature in this paper, we have used it during the development of Grasp itself.

C. Grasp player

The Grasp player is written in the Tcl scripting language[1]. The job of the Grasp player is basically to provide an execution environment for the script inside a Grasp trace, by implementing all methods called in a trace file.

The Grasp player goes through the following stages:

1) Load the default methods, e.g. the `plot` method in Section IV-B.
2) Load any plugins, which define additional methods.
3) Read in and execute the trace script.
4) Do any post processing, e.g. export a postscript file.

Note that step 3 is a single Tcl command, but it is also the place where the main work happens and where the trace is actually visualized.

A handy feature of the Grasp player is the option to export the trace visualization to a postscript file and an option to print a legend. These are useful for automatically creating figures for research articles. An example of such a figure is shown in Figure 3, which visualizes the complete trace from Figure 2.

Shared resources: Grasp can visualize the synchronization of tasks in case they share resources.

Figure 3 demonstrates the behavior of three fixed priority scheduled tasks, with two of them (Task2 and Task3) sharing one logical resource according to the Stack Resource Policy [Baker, 1991]. The highest priority task has the lowest index number, i.e. Task1 has the highest priority. In this example, the execution of a critical section is visualized by a dark section. Acquiring and releasing of mutexes is traced by the events `jobAcquiredMutex` and `jobReleasedMutex`, which are not visualized in the example.

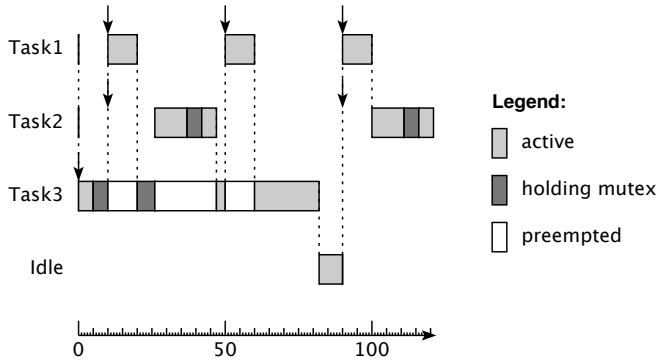


Fig. 3. Example trace visualization created by the Grasp player.

Hierarchical scheduling: An interesting and unique feature of Grasp is the built in support for visualizing behavior of servers in a hierarchical real-time system. An example is shown in Figure 4.

There are four server events:

1) `serverReplenished` sets the capacity of a server.
2) `serverDepleted` creates a depleted message for a server.
3) `serverResumed` starts consuming a server's budget at a constant rate of 1 unit per time unit.
4) `serverPreempted` stops consuming a server's budget.

These four events are sufficient to visualize the behavior of most servers in the real-time literature. We have extended μC/OS-II with polling [Lehoczky et al., 1987], periodic idling [Davis and Burns, 2005] and deferrable servers [Strosnider et al., 1995].

Figure 4 shows a Grasp player window after loading a trace file. The task execution is shown on top, with the server capacities illustrated underneath. In this particular example Task1 is assigned to the Deferrable Server, and Task2 is assigned to the Polling Server. The different shapes underneath the timeline indicate different events. For example, a triangle pointing upwards indicates a server replenishment and a square indicates the arrival of a periodic task. Clicking on a shape with the mouse reveals details about the event, e.g. the name of the server which is replenished.

The thin vertical line spanning across the tasks and servers is the *time marker*. It moves with the mouse cursor and indicates the current time in the visualization, also shown in the windows title bar.

During the development of the different servers Grasp provided useful insight into their behavior and speeded up the debugging process considerably.

D. Grasp plugins

Choosing the Tcl script format for a Grasp trace allows for a very simple interface for implementing plugins: a Grasp plugin is a set of methods which are called within a Grasp trace. There are no restrictions on the syntax of the plugin methods, as long as they do not conflict with the default Grasp player methods.

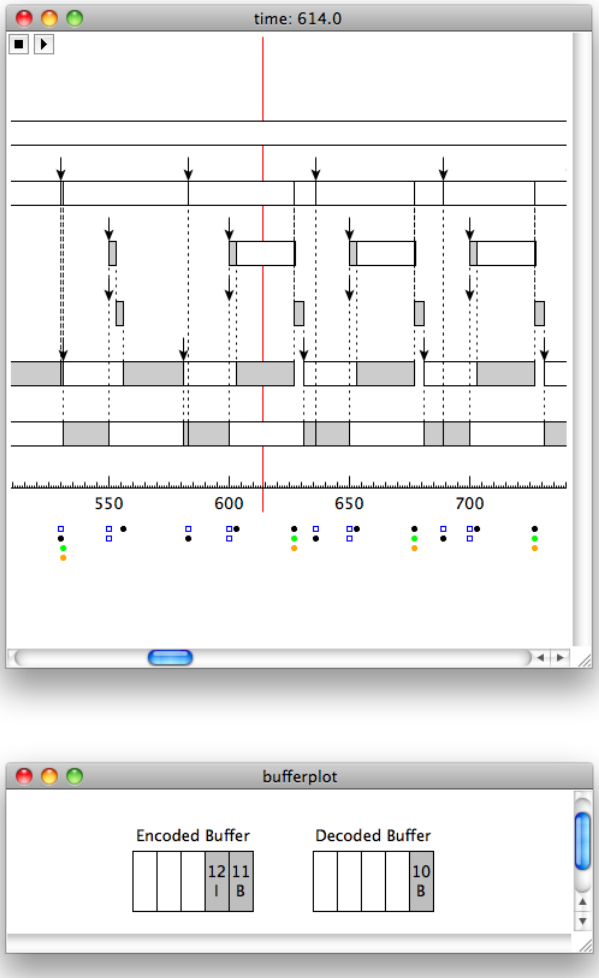


Fig. 5. Example of a trace visualization using the `BufferPlot` plugin.

To facilitate plugins which depend on the current time indicated by the time marker, Grasp provides an abstract event `<<TimeChanged>>`. A plugin can register for an abstract event using the Tcl `bind` command.

Plugins are loaded into a Grasp player by executing the player from the command line with the `-plugins` option followed by a list of paths to Tcl scripts implementing the plugins. Alternatively, the plugin scripts can be placed in the `plugins` subdirectory. All scripts residing in this directory are loaded automatically by the player.

In our recent work on mode changes in multimedia applications [Holenderski et al., 2009] we investigated an application comprised of a set of tasks communicating via shared buffers. We used Grasp to gain insight into the behavior of buffers and to measure the mode change latencies. We have implemented two plugins for this purpose.

The `BufferPlot` plugin defines four new events: `push`, `pop`, `insert` and `drop`. `BufferPlot` is implemented as a XOTcl class, which is an object oriented extension for Tcl. It follows the same structure as the `Plot` class behind the `plot` command, and implements the new events as instance methods. In the trace the new events are passed as arguments to

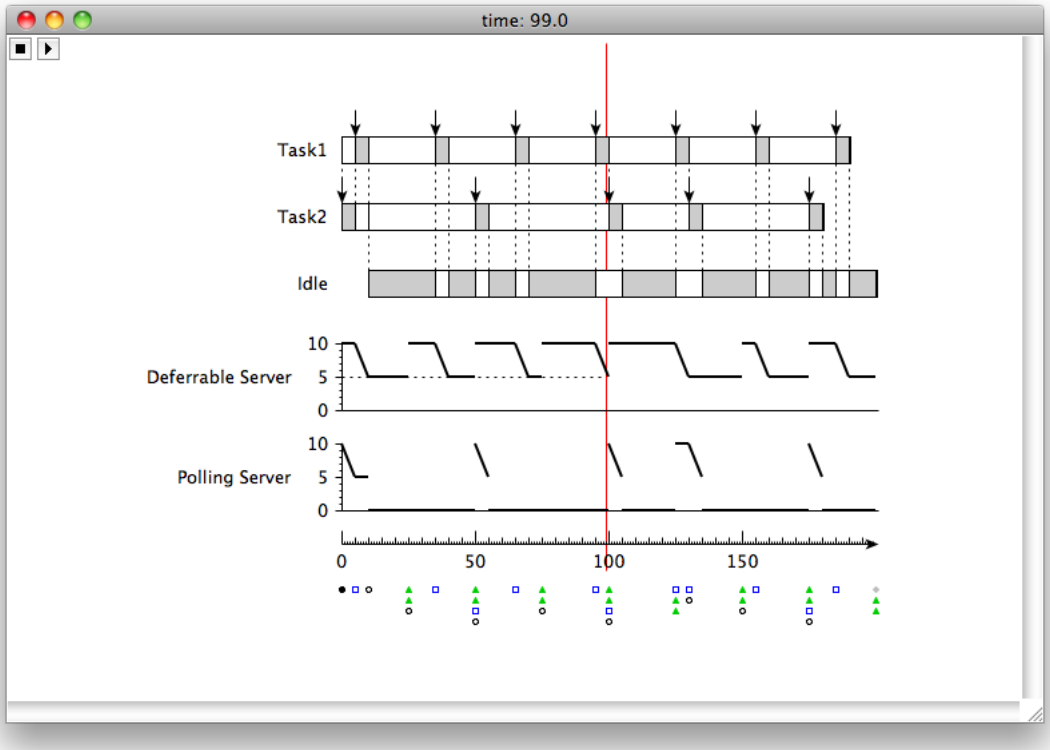


Fig. 4. Example of a trace visualization for hierarchical scheduling. The Task1 and Task2 tasks are assigned to the Deferrable Server and Polling Server, respectively.

the `bufferplot` command, rather than `plot`, which then dispatches the appropriate `BufferPlot` method. Figure 5 shows an example of how this Grasp plugin correlates the contents of the buffers with the task execution of the application.

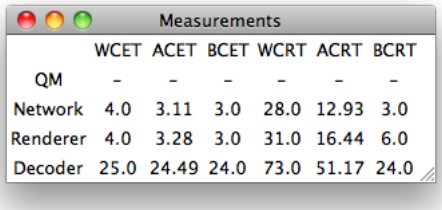
The Tcl file format of a Grasp trace makes it possible to embed plugins inside a trace file. Since a plugin is simply a definition of methods called within a trace file, inserting the plugin code at the beginning of the trace file will make sure that the necessary methods are defined before they are used. This allows to distribute a single self-contained trace file which can be visualized with any Grasp player, independent of the available plugins.

E. Grasp measurement

The Grasp player measures the execution and response time of jobs and provides a summary of the average, best case and worst case times for all jobs of a task. This information is shown on demand, by clicking on a job or a task label, or by selecting "Measurements" from the menu, shown in Figure 6.

The Grasp plugins also allow to easily implement custom measurement tools, as we did for measuring the mode change latencies for our recent work [Holenderski et al., 2009].

To measure the mode change latencies we have added a simple plugin which extended the `Plot` class with three new events: `latencyStart`, `latencyStop` and `latencySummary`. The first two events are generated throughout the simulation whenever a mode change occurs. The latter



	WCET	ACET	BCET	WCRT	ACRT	BCRT
QM	-	-	-	-	-	-
Network	4.0	3.11	3.0	28.0	12.93	3.0
Renderer	4.0	3.28	3.0	31.0	16.44	6.0
Decoder	25.0	24.49	24.0	73.0	51.17	24.0

Fig. 6. Example of trace measurements, summarizing the worst-case (WCET), average-case (ACET) and best-case (BCET) execution times, and the worst-case (WCRT), average-case (ACRT) and best-case (BCRT) response times for all application tasks.

event is generated at the end of the simulation. Its handler collects all the latencies, uses the `gnuplot` tool [gnu, 2010] to plot them on a graph, and automatically writes the graph to a postscript file.

V. Conclusions

In this paper we presented the Grasp toolset for tracing, visualizing and measuring the behavior of real-time systems. Grasp can be used to evaluate new algorithms and scheduler implementations in an operating system. We have used Grasp extensively during the development of several extensions for the μC/OS-II real-time operating system, some of which were used in this paper to illustrate the Grasp features.

Grasp is composed of three parts: (i) the recorder, (ii) the trace file and (iii) the player. The Grasp's recorder takes the

instrumentation approach to tracing, catching all events of interest. It limits the interference by storing the traced events in memory during runtime and writing the trace to a file only at the end of a run. The recorder is the only operating system specific part of the Grasp toolset. A Grasp trace is stored in the Tcl script format. The expressiveness of this format allows to easily extend Grasp functionality with visualization and measurement plugins. The Grasp player interprets the Tcl script containing the recorded trace. It visualizes task execution, task synchronization, servers in hierarchical scheduling and buffer usage for tasks communicating via shared buffers.

Future work

We have implemented a Grasp recorder for μC/OS-II within the OpenRISC simulator. Our current research focusses on (i) deploying the Grasp recorder in an embedded environment, and (ii) investigating the applicability of the Grasp recorder in other real-time operating systems.

In this paper we have shown how to visualize a Grasp trace using the Grasp player. A trace, however, can also be used for validating the behavior of the target system by comparing its trace to a reference trace automatically. To make sure that a change in the implementation of one primitive has no impact on other parts of the system, we have setup a test suite which automatically checks whether a new μC/OS-II extension did not invalidate existing behavior. The test suite is comprised of a set of test applications with reference traces and a shell script. The script executes all the test applications and compares the new traces against the reference traces. Currently, two traces are considered to be equivalent if they exhibit the same timing behavior, modulo the unique identifiers (e.g. job identifiers) particular to every simulation run. This approach, however, may result in false positives, when a trace exhibits correct behavior, but is not equivalent to the reference trace due to different overheads of the primitives resulting in different computation times of tasks and consequently leading to a different preemption behavior, which nonetheless may be correct. As future work we would like to investigate more resilient testing methods for exploiting the Grasp traces to validate the timing behavior of a real-time system.

REFERENCES

[uco, 2007] *uCOS-II WIN32, LINUX und Freescale HCS12 PORT*. 2007. URL http://www.it.fht-esslingen.de/~zimmerma/software/.

[gnu, 2010] *Gnuplot*. 2010. URL http://www.gnuplot.info.

[Baker, 1991] T. P. Baker. *Stack-based scheduling for realtime processes*. Real-Time Systems, vol. 3(1):pp. 67–99, 1991.

[Bolado et al., 2004] M. Bolado, H. Posadas, J. Castillo, P. Huerta, P. Sánchez, C. Sánchez, H. Fouren, F. Blasco. *Platform based on open-source cores for industrial applications*. In *Conference on Design, Automation and Test in Europe (DATE)*, p. 21014. 2004.

[Davis and Burns, 2005] R. I. Davis, A. Burns. *Hierarchical fixed priority pre-emptive scheduling*. In *Real-Time Systems Symposium (RTSS)*, pp. 389–398. 2005.

[Holenderski et al., 2008] M. Holenderski, R. J. Bril, J. J. Lukkien. *Using fixed priority scheduling with deferred preemption to exploit fluctuating network bandwidth*. In *Work in Progress session of the Euromicro Conference on Real-Time Systems (ECRTS)*. 2008.

[Holenderski et al., 2009] M. Holenderski, R. J. Bril, J. J. Lukkien. *Swift mode changes in memory constrained real-time systems*. In *International Conference on Embedded and Ubiquitous Computing (EUC)*, pp. 262–269. 2009.

[Labrosse, 1998] J. J. Labrosse. *Microc/OS-II*. R & D Books, 1998.

[Lehoczky et al., 1987] J. P. Lehoczky, L. Sha, J. K. Strosnider. *Enhanced aperiodic responsiveness in hard real-time environments*. In *Real-Time Systems Symposium (RTSS)*, pp. 261–270. 1987.

[McGavin et al., 2006] M. McGavin, T. Wright, S. Marshall. *Visualisations of execution traces (vet): an interactive plugin-based visualisation tool*. In *Australasian User Interface Conference (AUIC)*, pp. 153–160. 2006.

[Mughal and Javed, 2008] M. I. Mughal, R. Javed. *Recording of Scheduling and Communication events on Telecom Systems*. Master's thesis, Mälardalen University, 2008.

[OpenCores, 2009] OpenCores. *OpenRISC overview*. 2009. URL http://www.opencores.org/project,or1k.

[Strosnider et al., 1995] J. K. Strosnider, J. P. Lehoczky, L. Sha. *The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments*. IEEE Transactions on Computers, vol. 44(1):pp. 73–91, 1995.

[Welch et al., 2003] B. Welch, K. Jones, J. Hobbs. *Practical Programming in Tcl and Tk*. Prentice Hall, 2003.

[Whitham and Audsley, 2006] J. Whitham, N. Audsley. *MCGREP–a predictable architecture for embedded real-time systems*. In *Real-Time Systems Symposium (RTSS)*, pp. 13 –24. 2006.