Grasp: Visualizing the Behavior of Hierarchical Multiprocessor Real-Time Systems

Mike Holenderski, Reinder J. Bril, Johan J. Lukkien

Eindhoven University of Technology, Den Dolech 2, 5600 AZ Eindhoven, The Netherlands

Abstract

Trace visualization is a viable approach for gaining insight into the behavior of complex distributed real-time systems. Grasp is a versatile trace visualization toolset. Its flexible plugin infrastructure allows for easy extension with custom visualization and analysis techniques for automatic trace verification. This paper presents its visualization capabilities for hierarchical multiprocessor systems, including partitioned and global multiprocessor scheduling with migrating tasks and jobs, communication between jobs via shared memory and message passing, and hierarchical scheduling in combination with multiprocessor scheduling. For tracing distributed systems with asynchronous local clocks Grasp also supports the synchronization of traces from different processors during the visualization and analysis.

Keywords: trace visualization, hierarchical scheduling, multiprocessors, real-time systems

1. Introduction

Modern real-time systems are becoming increasingly more complex, with many tasks executing concurrently on many processors, making it difficult to understand the system behavior. A popular trend in coping with the vast number of tasks and the resulting interferences between them is to hide tasks inside components and to integrate the system from those components. This approach requires hierarchical scheduling, which has been covered extensively in the literature for uniprocessor systems. Recently, the real-time literature has been investigating applying hierarchical scheduling to multiprocessor platforms. In this paper we address the problem of how to provide insight into complex interaction patterns between jobs executing in a hierarchical multiprocessor system.

Several approaches are available for tackling the complexity of modern software systems. Ideally, every system would be meticulously documented, providing a formal yet concise description of the emergent system behavior. However, this is a long and costly process without immediate effects (such as additional functionality) and is therefore not common in practice. Examples of poorly documented code and system designs are abundant. The description of the dynamic system behavior therefore needs to be extracted from existing systems. There are modeling and verification tools available, which rely on the developers analyzing the implementation and constructing its model. These tools then employ formal methods to verify the behavior of the extracted model against an abstract model. The state of the art modeling and verification techniques, however, are not scalable and therefore can be applied to verify only a small portion of the entire system.

Visualization tools offer an interesting alternative. Existing systems can be instrumented to generate runtime traces, which can then be analyzed by engineers and researchers, leveraging their expertise and human capacity to recognize patterns, to gain insight into the system behavior. The challenge here ***lies in representing and presenting*** the information in an intuitive way, enabling the user to extract the essential properties of the analyzed system. ***While trace visualization on its own is insufficient for the verification of timing constraints of a real-time system, it is well suited for early design of such systems before the formal validation stage is reached.***

Grasp is a toolset for tracing and visualizing the

[☆] The work presented in this paper was supported in part by the European ITEA2-CANTATA project. The Grasp toolset together with example traces is available for Linux, Mac and Windows at `http://www.win.tue.nl/~mholende/grasp`.

behavior of complex real-time systems. Its main strength lies in providing many different visualizations for various real-time primitives and scheduling techniques in a consistent and intuitive way. Its flexible architecture ***allows one to extend it easily*** with new visualization and analysis techniques.

We have been using Grasp extensively within our group during our ***research on embedded*** real-time systems and the development of various extensions of a commercial real-time operating system μC/OS-II, including a hierarchical scheduling framework and slot shifting. ***The use of Grasp*** has also been reported in [1, 2, 22] where it was used to gain insights into new approaches for hierarchical scheduling in Linux and VxWorks operating systems. Recently Grasp was used in the context of the SOFIA project to visualize the communication patterns between sensor nodes in a smart home environment.

In this paper we focus on ***visualizing traces of*** multiprocessor systems. The challenge here ***lies in representing and presenting*** the execution and communication between jobs running on different processors in an intuitive and compact way. Moreover, if the timestamps of events occurring on a processor are recorded in its local time, special care must be taken to synchronize the individual traces. While some custom-built clusters such as IBM Blue Gene offer sufficiently accurate global clocks, most distributed systems can only rely on local clocks [3]. If the local clocks drift too far apart it may lead to inaccuracies or even errors during the trace analysis or visualization, as the causality between events may appear to be broken (e.g. messages arriving before they were sent).

Contributions

In this paper we build on top of our previous work presented in [12]. We focus on visualizing the timing of job execution and communication in the context of multiprocessor systems. In particular, we demonstrate Grasp's ability to visualize

- partitioned and global multiprocessor scheduling,
- migrating tasks and jobs,
- communication between jobs via shared memory and message passing,
- hierarchical scheduling in combination with multiprocessor scheduling.

We also describe Grasp's interface for synchronizing timestamps in traces generated in a distributed system.

Outline

Section 2 summarizes the related work, followed by an overview of the Grasp toolset in Section 3. Grasp's support for multiprocessor scheduling is presented in Section 4 and its support for hierarchical multiprocessor scheduling is presented in Section 5. Section 6 describes synchronization of distributed traces. Concluding remarks and future work are presented in Section 7.

2. Related work

Existing visualization tools for real-time systems are specialized to visualize a fixed set of behaviors. For example, the Tracealyzer [17] and TimeDoctor [21] are targeting only non-hierarchical uniprocessor systems. Making a step towards distributed systems is not trivial. Grasp, on the other hand, supports multiprocessor systems with two level virtualization.

There are several trace visualization tools which support the development of parallel programs on uniform parallel-processor platforms, such as VAMPIR [18], Paje [14], Jedule [13], or Scalasca [11]. They illustrate the execution of parallel jobs and communication between them, but they are limited to flat systems. To the best of our knowledge no visualization tools currently support the visualization of hierarchical scheduling in a uniprocessor or multiprocessor setting.

Traces accepted by most tools are lists of timed events, often in a binary format. Grasp, on the other hand, has adopted the idea of treating the trace as a script. On the one hand, Grasp traces are more verbose and require more storage space, compared to the binary format. On the other hand, they allow for large degree of flexibility, making it easy to add new events to the event model without changing the core implementation of the visualization and analysis components. This makes Grasp ideal for rapid prototyping of new visualization tehcniques. We have exploited this flexibility during the development of Grasp's various visualization and analysis features.

There are several tracing tools available, mainly for the Linux platform, which generate traces. Examples include the Data Stream Kernel Interface (DSKI) [6], Ftrace [10], and Dtrace [9]. DSKI is a platform independent interface standard to support collection of a variety of performance data from the operating system internals. It has been implemented on Linux. Ftrace and Dtrace are integrated in many Linux distributions. They exhibit low performance overhead and low memory footprint. In order to leverage their popularity, we have implemented a converter from the sched_switch tracer output of Ftrace, allowing to use Grasp in many Linux and Unix environments.

When tracing is used to analyze the behavior of distributed systems, then there is the additional problem of synchronizing the times of events occurring on different nodes. Time differences among distributed clocks can be characterized in terms of their relative offset and drift. If we assume constant drift, then the local time can be mapped onto the global (or master) time via linear interpolation. Existing time synchronization algorithms compute the interpolation based on the timestamps of messages exchanged back and forth with a master node [8, 15] or with other nodes [16, 5, 4]. The timestamp synchronization in Grasp is based on [15].

The authors of [4, 3] observe that while linear offset interpolation might prove satisfactory for short runs, measurement errors and time-dependent drifts may create inaccuracies and violate causality relations during longer runs (e.g. a message is received before it was sent). They propose a method for fixing these errors by postponing certain events in the trace. However, while maintaining the causality relations in traces, their methods change the timing of the events. As Grasp is targeting real-time systems, it relies on linear interpolation for synchronizing traces and if it detects inconsistencies in the ordering of events then it notifies the user that the constant drift assumption was violated.

3. Grasp overview

The Grasp toolset is composed of three entities: the Grasp Recorder, the Grasp Trace and the Grasp Player, as shown in Figure 1.

The Grasp Recorder is embedded in the target system and is responsible for generating a trace. The generated Grasp Trace contains the raw data from a particular system run. The Grasp Player reads in a trace and displays it in an intuitive way.

3.1. Grasp Recorder

The Grasp Recorder is implemented as a library providing functions to initialize the recorder, log

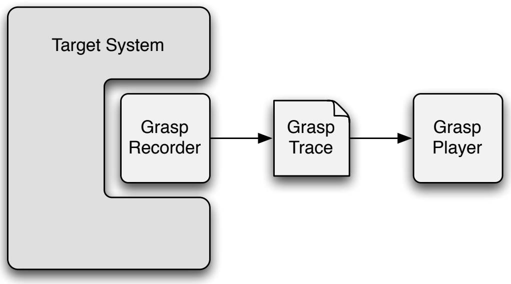

Figure 1: Overview of the Grasp archictecture.

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, arrival of tasks, or server replenishment. The recorder also provides a function to log custom events, which programmers may call inside their applications.

Designing and implementing an instrumentation infrastructure which exhibits low performance and memory overheads can be a daunting task. Therefore, rather than designing a custom Grasp Recorder and integrating it within the target system, one can implement a converter ***for an existing trace format***, leveraging existing instrumentation and tracing tools, as we have done for the sched_switch tracer output of Ftrace.

3.2. Grasp Trace

The Grasp Trace is a Tcl [23] script. The decision for treating the Grasp Trace as a script results in large degree of flexibility. The Grasp Player basically provides a set of commands which can be called from within a Grasp Trace. A trace can therefore be a simple list of commands, but it can also be a complete system simulator, or anything in between. This ***allows one to embed*** various extensions (or plugins) inside a trace, resulting in a self-contained trace which can be visualized by any Grasp Player, independent of the plugins it provides. It can also be used to reduce the size of very large traces, by automatically generating or factoring out common or repeating parts. Also, a trace may call methods in the player's public API to override its default settings, making sure that the trace is visualized as intended by its creator. The greatest benefit of the trace being a script, however, is the simple plugin infrastructure discussed in the next section.

A typical Grasp Trace event has the following structure:

```
plot time event arguments
```

which means that *event* has occurred at time *time*. The *arguments* parameter is a list and describes the instance of the *event*. Every *event* defines its own signature, i.e. the number and the semantics of the *arguments* which it accepts. Usually an event accepts a list of required arguments followed by a list of `-key` *value* pairs for optional arguments. In the remainder of this paper we will often ignore the `plot` *time* part, as it is common for most events. Also, we will ignore optional arguments for customizing the trace visualization, such as assigning names or colors to tasks.

There are several basic events for tracing job execution:

- `newTask` *task* creates a new task, where *task* is a new identifier used in later events.
- `jobArrived` *job task* indicates that *job* belonging to *task* has arrived, where *job* is a new identifier used in later events, and *task* is the identifier of a task created previously with `newTask`.
- `jobStarted` *job* indicates that *job* has started.
- `jobPreempted` *job* indicates that *job* has been preempted.
- `jobBlocked` *job* indicates that *job* has been blocked (e.g. trying to access a locked shared resource).
- `jobResumed` *job* indicates that *job* has been resumed.
- `jobCompleted` *job* indicates that *job* has completed.

An example trace is shown in Figure 2.

```
newTask task1 -priority 7 -name "Task 1"
newTask task2 -priority 8 -name "Task 2"
plot 5 jobArrived job2.1 task2
plot 5 jobResumed job2.1
plot 20 jobArrived job1.1 task1
plot 20 jobPreempted job2.1 -target job1.1
plot 20 jobResumed job1.1
plot 35 jobCompleted job1.1 -target job2.1
plot 35 jobResumed job2.1
plot 50 jobCompleted job2.1
```

Figure 2: Example of a Grasp Trace.

3.3. Grasp Player

The Grasp Player is the main contribution of Grasp. It basically provides an execution environment for the script inside of a Grasp Trace. As the Grasp Player is also written in Tcl, its operation is very simple: it loads the definitions of all methods which can be called inside a trace, and then evaluates the trace script. Figure 3 shows an example of a trace of a video processing algorithm. The visualization correlates the contents of the frame buffers with the system execution, allowing to inspect their content at different times in relation to the dynamic events occurring during runtime. ***As the mouse cursor moves across the trace, the contents of the buffers changes.***

The Grasp Player comes with a powerful set of features, including the visualization of task execution in flat and hierarchical systems, uni- and multiprocessor scheduling, intervals in slot shifting, measurement of execution and response times, automatic verification of certain trace properties, command line interface, and exporting to postscript (useful for creating high quality figures for research articles, e.g. Figures 4,5,6, and 8).

Plugins

The Grasp Player provides a simple yet versatile infrastructure for extending it with custom visualization and analysis plugins. For example, the Grasp Recorder extension and Grasp Player visualization plugin for intervals in slot shifting was implemented by a student within two hours, extending the budget visualization for servers in hierarchical scheduling.

A plugin has three interfaces at its disposal:

(i) A plugin can define and implement its own methods which can be called within a trace. The Buffer visualization in Figure 3 is an example of such a plugin. It defines methods for tracing the content of buffers via events for adding and removing messages from a buffer:

- `newBuffer` *buffer* creates a new buffer, where *buffer* is a new identifier used in later events.
- `bufferplot` *time* `write` *buffer message* indicates that *message* was added at *buffer*'s tail at time *time*.

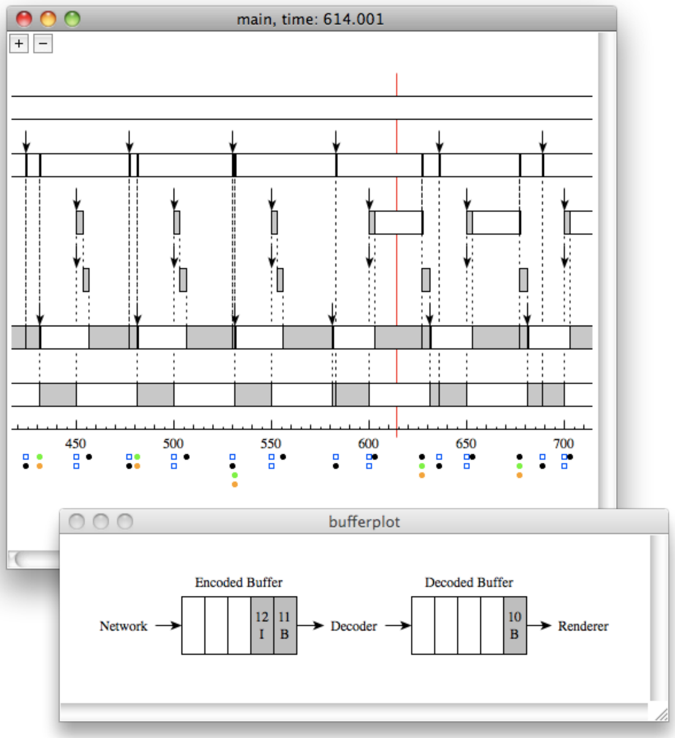

Figure 3: Example illustrating a video processing application comprised of several tasks (including Network, Decoder and Renderer tasks) executing on a single processor and communicating individual frames of an MPEG video via two shared buffers. The figure shows the contents of the buffers at time 614, including the sequence number and the kind of the video frames.

- `bufferplot` *time* `read` *buffer* indicates that a message was removed from the *buffer*'s head at time *time*.

(ii) Alternatively, a plugin can register handlers for a set of virtual events, which are generated when the traced events are processed. The Grasp Player provides a method allowing a plugin to register a script which will be evaluated whenever a particular event occurs. For example, the Measurement plugin registers a handler for the `jobArrived` and `jobCompleted` events, to compute the response time of jobs.

(iii) The Grasp Player also provides a set of player events. For example, a plugin can register a script which will be called upon the `TimeChanged` event, which is generated when the mouse cursor is moved across the trace. This player event is used by the Buffer plugin to illustrate the buffer content at the time pointed to by the mouse cursor (e.g in Figure 3).

The simple plugin infrastructure is made possible by the Grasp Trace being a script. Other visualization tools rely on a "dispatch" method which is called for each event in the trace to dispatch the corresponding event handler. Extending such tools with new events requires to modify the dispatch method (or to limit the syntax of traced events). As ***the*** Grasp Trace simply calls methods provided by the Grasp Player, there is no need for a dispatch method. Extending the Grasp Player with a plugin requires simply to place the plugin script inside of the plugins directory (which is automatically included when the player starts).

Automatic verification

The plugin infrastructure can be leveraged to implement various verification tools for automatically analyzing the system behavior in a trace. For example, the BudgetCheck plugin shows a warning when a server exceeds its budget, and the MutexCheck plugin verifies proper nesting of mutex locking events inside a trace.

For any given target system, if a particular behavior is expected, then a "test-suite" plugin may be implemented to verify that for a specific scenario the target system satisfies the desired properties, e.g. after a maintenance activity.

4. Multiprocessor visualization

In this section we present Grasps support for multiprocessor systems. The multiprocessor support is implemented by extending a subset of events for tracing job execution with an optional `-processor` argument.

Our goal was to support various concepts commonly found in multiprocessor scheduling. Grasp supports partitioned as well as global multiprocessor scheduling with task and job migration, and communication between jobs on shared and distributed memory platforms. In this section we discuss each of these features in more detail.

4.1. Creating a processor

Similar to other objects in a trace, such as tasks or servers, a processor needs to be created before it can be referred to in other trace events.

- `newProcessor` *processor* creates a new processor, where *processor* is an identifier which can be added to other trace events to support multiprocessor visualization.

4.2. Partitioned and global scheduling

In partitioned scheduling, each task is assigned to a particular processor and during runtime all of its jobs execute on that processor. In global scheduling, different jobs of the same task may execute on different processors.

Partitioned scheduling

When a task is created, it can be assigned to a particular processor:

- `newTask` *task* `-processor` *processor* creates a new *task* and assigns it to the *processor*.

All subsequent job events will be mapped to the *processor* (unless the processor argument is overridden, as discussed in the next section). Figure 4 shows an example of a trace on partitioned multiprocessor platform.

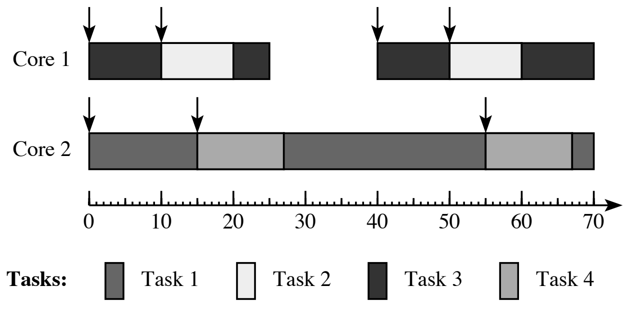

Figure 4: Example showing the execution of five tasks on a partitioned multiprocessor platform consisting of two cores.

Figure 4 shows the system behavior in a collapsed view, where the execution of all tasks is collapsed on a single timeline. Alternatively, the Grasp Player also supports an expanded view, where each processor is shown in a separate window illustrating the interactions between the local tasks, as shown for a single processor in Figure 3.

Note that the Grasp Player provides many details upon a mouse click. For example, when the mouse is clicked on top of a downward pointing arrow, a message is shown telling which task has arrived. These features are difficult to visualize in a paper.

Global scheduling

In global scheduling we can distinguish between task and job migration (also referred to as restricted- and full-migration scheduling, respectively [7]). When only task migration is allowed, then tasks are allowed to migrate between processors, however, each job must execute on one processor. When job migration is allowed, then jobs may migrate between processors, i.e. they can halt on one processor and resume on another. Grasp supports both task and job migration by having the `jobArrived`, `jobStarted`, and `jobResumed` events accept an optional `-processor` argument. In a trace containing only task migration only the `jobArrived` event will specify the `-processor` argument. In a trace containing job migration also the `jobStarted` and `jobResumed` events will specify the `-processor` argument. Figure 5 illustrates job migration by having the first job of task 1 arrive at time 15 on core 2 and later at time 22 migrate to core 1 ***(indicated by the dashed arrow)***.

```
newProcessor core1 -name "Core 1"
newProcessor core2 -name "Core 2"
newTask task1 -name "Task 1"
...
plot 15 jobArrived job1.1 task1 -processor core2
plot 15 jobPreempted job4.1
plot 15 jobResumed job1.1
...
plot 22 jobPreempted job1.1 -processor core1
plot 22 jobPreempted job3.1
plot 22 jobResumed job1.1 -processor core1
plot 22 jobResumed job4.1
...
```

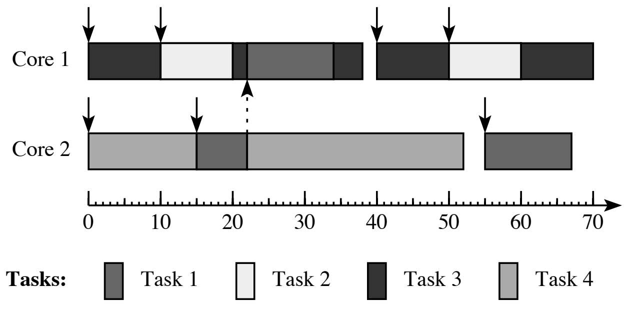

Figure 5: Example showing a partial trace and the corresponding visualization, illustrating the migration of a job.

4.3. Communication between jobs

Depending on the memory architecture in a multiprocessor system, jobs can communicate via shared memory or via message passing.

Shared memory

When jobs executing on different processors communicate via shared memory, it is critical to maintain the data consistency of the shared data structures. A common approach is using mutexes. Grasp provides events for acquiring and releasing a mutex, as shown in the example in Figure 6. The relevant events are:

- `jobAcquiredMutex` *job mutex* indicates that *job* has acquired *mutex*.
- `jobReleasedMutex` *job mutex* indicates that *job* has released *mutex*.

The arguments *job* and *mutex* are identifiers for a previously created job and mutex, respectively.

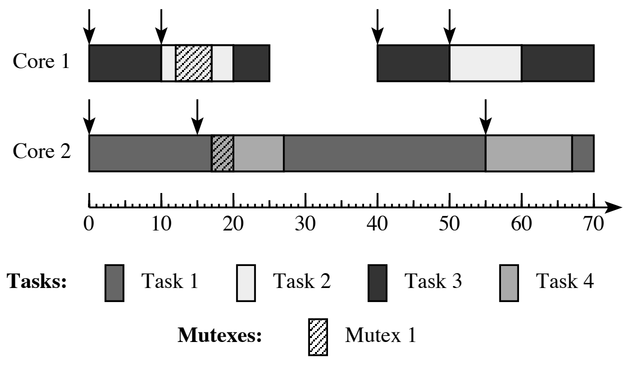

Figure 6: Example showing tasks 2 and 4 using a mutex to communicate via shared memory.

Figure 6 shows an example of two tasks communicating via shared memory. At time 12 task 2 locks Mutex 1 guarding a shared memory location. When task 4 arrives at time 15 it finds the shared mutex in a locked state and is suspended. At time 17, when task 2 unlocks the mutex, task 4 is able to resume and lock Mutex 1 to read the data communicated from task 2.

Message passing

On a distributed memory platform jobs communicate via message passing. A popular example is the Message Passing Interface (MPI) [19]. We reuse the Buffer plugin [12] for this purpose. Depending on the communication paradigm (one to one, broadcast, multicast), we create the appropriate message buffers.

When the mouse cursor is dragged inside of the Grasp Player window, the contents of the buffers is animated, reflecting their state at the current time, indicated by the long vertical red line. Clicking on a buffer element reveals more message details (in case they were provided in the trace).

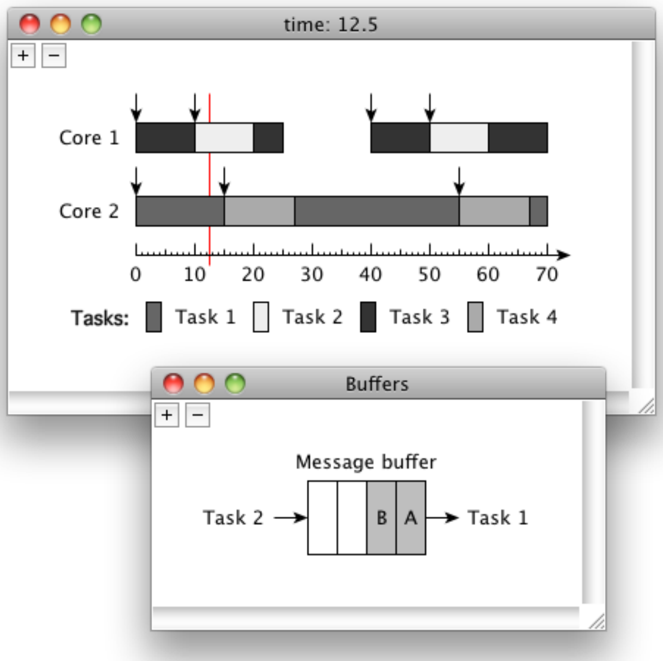

Figure 7: Example showing tasks 1 and 2 using a buffer to communicate via message passing.

Figure 7 shows an example of tasks 1 and 2 communicating via message passing. At time 12, there are 2 messages A and B from task 2 inside of a message buffer, waiting for task 1 to read them.

5. Hierarchical multiprocessor visualization

In [12] we have introduced Grasp's support for hierarchical scheduling in uniprocesor systems. The events for tracing the budget of a server are:

- `serverReplenished` *server budget* indicates that *server*'s remaining budget was replenished to *budget*.
- `serverResumed` *server* indicates that a task has started consuming *server*'s budget.
- `serverPreempted` *server* indicates that a task has stopped consuming *server*'s budget.
- `serverDepleted` *server budget* indicates that *server*'s remaining budget has been depleted.

In this section we elaborate on the combination of hierarchical and multiprocessor scheduling.

Using the standard hierarchical scheduling support, the Grasp Player is not aware of the task-to-server mapping, nor of the desired behavior of particular server types (such as periodic-idling or deferrable server). The hierarchical scheduling events pertain only to the replenishment, depletion and consumption of server's budget. The target system is responsible for generating the correct behavior. However, the Grasp Player can be easily extended with a verification plugin, making sure that the server behavior is according to its specification, e.g. that only tasks assigned to the server consume its budget, or that a periodic idling server always idles its budget away.

The fact that Grasp is not aware of the mapping between servers and tasks allows to easily trace systems where tasks consume budgets from *several* servers, and systems where a server is serving its budget to *several* tasks executing at the same time on different processors [20]. The latter is accomplished by allowing several `serverResumed` events to occur in a trace without a corresponding `serverPreempted` event in between. This provides a very simple way for tracing budget consumption in a multiprocessor setting: whenever a task assigned to a budget is resumed on any processor, the corresponding server is also resumed. Similarly, a server is preempted whenever a task consuming its budget is preempted on any processor.

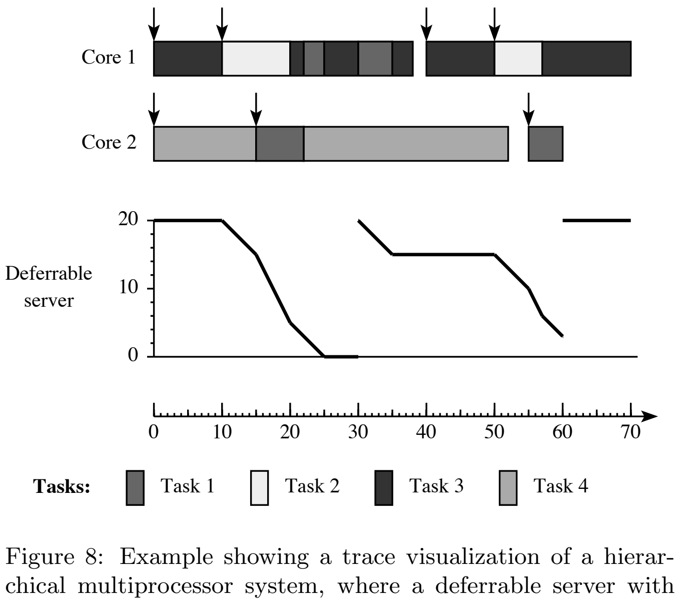

Figure 8: Example showing a trace visualization of a hierarchical multiprocessor system, where a deferrable server with period 30 and capacity 20 is serving its budget to tasks 1 and 2.

Figure 8 shows an example visualization of such a system, where a deferrable server with period 30 and capacity 20 is serving its budget to tasks 1 and 2. Tasks 3 and 4 are not bound to any server. At time 10, when task 2 arrives, it starts consuming server's budget. At time 15, when task 1 arrives, it also starts consuming server's budget. The budget is consumed at twice the rate until task 2 completes at time 20.

6. Timestamp synchronization

In Section 3.2, we mentioned that the events comprising a Grasp Trace do not have to be totally ordered by time. In a distributed multiprocessor system this allows to record traces on each processor individually, and then to simply concatenate the traces to form a single system trace, without the need for interleaving the events. This system trace file can then be loaded into the Grasp Player as any other trace file.

In the absence of a global time, i.e. if each processor records its local events using an asynchronous local clock, we need to synchronize the events which were recorded on different processors. The clock synchronization algorithm used by Grasp is based on the work presented in [15]. Unlike [15], however, Grasp does not enforce to synchronize with a single master node. It allows nodes to synchronize their time with an arbitrary node, creating clusters of synchronized nodes.

Let P_i (for $i \in \{1..p\}$) be a processor in our target system, and let $C_i(t)$ be the value of its local clock at physical time $t \in \mathbb{R}$. If we assume constant drift of each local clock, then we can express each local clock function as

$$C_i(t) = \alpha_i + \beta_i t,$$

where α_i is the initial offset at $t = 0$ and β_i is the drift with respect to the physical time t. We can then use the algorithm presented in [15] to approximate the α_i and β_i parameters, and use the $C_i(t)$ functions to map events recorded on different processors onto the same timeline.

The algorithm relies on processors exchanging synchronization messages, as illustrated in Figure 9. The tuples indicate the recorded events: $(C_i(t), S)$ indicates the transmission and $(C_i(t), R)$ the reception of a synchronization message at local time $C_i(t)$. Processor P_i sends a message to P_j and records the event $(C_i(t_1), S)$ using its local clock time. Upon reception of the message, P_j records the event $(C_j(t_2), R)$ using its own local time and immediately sends a message back to P_i, recording the sending event $(C_j(t_3), S)$.

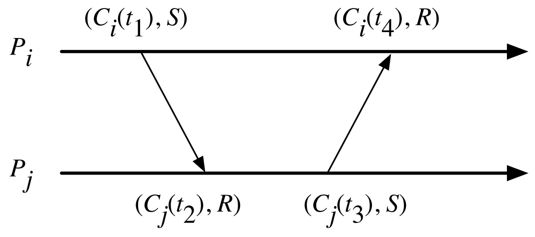

Figure 9: Two processors P_i and P_j exchange messages to synchronize their clocks.

Upon reception of the message, P_i records the event $(C_i(t_4), R)$.

Grasp provides the following events for synchronizing time between events generated on different processors:

- `syncSent` *source target id* indicates that the *source* processor has sent a message *id* to the *target* processor.
- `syncReceived` *source target id* indicates that the *target* processor has received a message *id* from the *source* processor.

The *id* parameter is used to match the transmission and reception events, in case several synchronization messages are sent between the same nodes. It also allows Grasp to identify lost synchronization messages.

Figure 10 shows an example of a system trace, which was obtained by concatenating two traces recorded on processors `core1` and `core2`. Note that

```
newProcessor core1
...
plot 15 syncSent core1 core2 m1
plot 18 syncReceived core2 core1 m2
...
newProcessor core2
...
plot 28 syncReceived core1 core2 m1
plot 30 syncSent core2 core1 m2
...
```

Figure 10: Example showing a partial trace illustrating the use of synchronization events.

each synchronization event is recorded with the local time on the processor where it is generated. When the Grasp Player loads the complete system trace, it has enough information to synchronize the events.

Grasp requires at least two synchronization messages between any two synchronizing processors. The larger the number of messages, the higher the accuracy of the approximation of the α_i and β_i parameters. Of course, the accuracy comes at the cost of additional overhead for exchanging the messages and storing the corresponding events.

Best synchronization results are achieved when the response to a synchronization message is sent immediately after its arrival. In terms of Figure 9, this means that the time interval between $(C_j(t_2), R)$ ***and*** $(C_j(t_3), S)$ ***should be as short as possible.***

If the constant drift assumption is violated, then the linear interpolation approach in [15] is no longer applicable. In our experiments we have never experienced any violation of the constant drift assumption. However, we have prepared Grasp for this eventuality by having it check for inconsistencies in the ordering of events and notify the user that the constant drift assumption was violated when an inconsistency is observed.

7. Conclusions

Grasp is a visualization toolset aiming to provide insight into the behavior of complex real-time systems. Its flexible plugin infrastructure allows for easy extension with custom visualization and analysis techniques for automatic trace verification. In this paper we have presented its features for visualizing hierarchical multiprocessors scheduling. It provides various visualizations for partitioned and global multiprocessor scheduling with migrating tasks and jobs, communication between jobs via shared memory and message passing, and hierarchical scheduling in combination with multiprocessor scheduling. For tracing distributed systems with asynchronous local clocks Grasp also provides a simple interface which aids in synchronizing the individual traces during the visualization and analysis.

References

[1] M. Åsberg, T. Nolte, S. Kato, Towards hierarchical scheduling in linux/multi-core platform, in: International Conference on Emerging Technologies and Factory Automation (ETFA), 2010, pp. 1–4.

[2] M. Åsberg, T. Nolte, S. Kato, A loadable task execution recorder for hierarchical scheduling in linux, in: Embedded and Real-Time Computing Systems and Applications (RTCSA), vol. 1, 2011, pp. 380 –387.

[3] D. Becker, M. Geimer, R. Rabenseifner, F. Wolf, Extending the scope of the controlled logical clock, Cluster Computing (2011) 1–19.

[4] D. Becker, R. Rabenseifner, F. Wolf, J. C. Linford, Scalable timestamp synchronization for event traces of message-passing applications, Parallel Computing 35 (2009) 595–607.

[5] M. Biberstein, Y. Harel, A. Heilper, Clock synchronization in cell be traces, in: Euro-Par 2008 – Parallel Processing, vol. 5168 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008, pp. 3–12.

[6] B. Buchanan, D. Niehaus, S. Sheth, Y. Wijata, The data stream kernel interface, Tech. Rep. ITTC-FY98-TR11510-04, University Of Kansas (June 1998).

[7] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, S. Baruah, Handbook of Scheduling: Algorithms, Models, and Performance Analysis, chap. A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms, Chapman and Hall/CRC, 2004, pp. 30–1 – 30–19.

[8] F. Cristia, Probabilistic clock synchronization, Distributed Computing 3 (1989) 146–158.

[9] Dtrace, http://wikis.sun.com/display/dtrace/dtrace (2008).

[10] Ftrace, http://www.kernel.org/doc/Documentation/trace/ftrace.txt (2008).

[11] M. Geimer, F. Wolf, B. J. N. Wylie, D. Becker, D. Böhme, W. Frings, M.-A. Hermanns, B. Mohr, Z. Szebenyi, Recent developments in the scalasca toolset, in: International Workshop on Parallel Tools for High Performance Computing, Springer, 2010, pp. 39–51.

[12] M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, J. J. Lukkien, Grasp: Tracing, visualizing and measuring the behavior of real-time systems, in: International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), 2010, pp. 37–42.

[13] S. Hunold, R. Hoffmann, F. Suter, Jedule: A tool for visualizing schedules of parallel applications, in: International Conference on Parallel Processing Workshops (ICPPW), 2010, pp. 169–178.

[14] J. C. D. Kergommeaux, B. D. O. Stein, M. S. Martin, Paje: An extensible environment for visualizing multi-threaded program executions, LNCS 1900.

[15] E. Maillet, C. Tron, On efficiently implementing global time for performance evaluation on multiprocessor systems, Parallel Distributed Computing 28 (1995) 84–93.

[16] D. L. Mills, Network Time Protocol (Version 3), The Internet Engineering Task Force—Network Working Group, RFC 1305 (1992).

[17] M. I. Mughal, R. Javed, Recording of scheduling and communication events on telecom systems, Master's thesis, Mälardalen University (2008).

[18] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, K. Solchenbach, Vampir: Visualization and analysis of MPI resources, Supercomputer 12 (1996) 69–80.

[19] P. Pacheco, Parallel Programming with MPI, Morgan Kaufmann, 1996.

[20] I. Shin, A. Easwaran, I. Lee, Hierarchical scheduling framework for virtual clustering of multiprocessors, in: Euromicro Conference on Real-Time Systems (ECRTS), 2008, pp. 181 –190.

[21] TimeDoctor, http://sourceforge.net/projects/timedoctor/ (2011).

[22] M. van den Heuvel, R. Bril, J. Lukkien, Protocol-transparent resource sharing in hierarchically scheduled real-time systems, in: International Conference on Emerging Technologies and Factory Automation (ETFA), 2010, pp. 1–8.

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