Runtime performance evaluation of embedded software

Anders Ive
Dept of Computer Science, Lund University
Box 118, S-221 00 Lund, Sweden
e-mail: Ive@dna.lth.se

Abstract. When developing real-time system software it is often desired to study the execution timing of processes and programs. Worst-case execution times, location of bottlenecks, processor utilization could be found if the programmer could analyze programs at runtime. The system software described in this paper provides a way to measure the execution times. The system makes minor changes to the performance and enables flexibility to the evaluation method. The system, and the changes made in the real-time kernel in order to implement the system, are described. It was experienced during evaluation of a real-time garbage collector that the system was a valuable debugging and verification tool.

1 Introduction

The purpose of this project was to develop support to evaluate time aspects of implemented code. The primary focus was to measure execution times of real-time processes in a runtime system. It can be difficult to calculate or predict worst-case execution time by static analysis alone. A way to analyze execution times is to run an application, and measure the times with a clock. The goal was to provide software support to do these kinds of measurements with an external measuring device, for instance, a logic analyzer. Another interesting aspect is to follow the scheduling process, and study a system's runtime behavior. The programmer should also be able to insert signals to keep track of when a specific code sequence is executing.

The process supervision system has been implemented, and is described in the report. Firstly, the new functionality is covered, along with the changes that have been made in the real-time kernel. A case study of a real-time garbage-collector is presented, with focus on the supervision system. The concluding sections describe future development and future usages of the supervision system.

2 Background

The initial purpose of this project was to study the execution times of a real-time garbage collector. The calculated execution times of the garbage collector had to be physically verified. With the aid of the supervision system this has been done. The case study is covered in Section 5. This section relates the supervision system with related techniques. A brief introduction of the hardware used in the project is described. An example of the output from the supervision system is presented and described.

2.1 Related techniques

Visualization or measurement of execution times and context switches has been done in different ways ever since development of multitasking software started. At our university this started 25 years ago on PDP-11 computers. One analog output was used to track the current running concurrent process. The information of the output was then used to visualize the different processes.

Numerous vendors of development tools nowadays provide various types of hardware and software support for this. However, these available tools can be classified into two categories, each with certain drawbacks.

1. Hardware-based tools like in-circuit emulators and logic analyzers, which can be aware of the running program with symbol tables loaded into the analyzer, are powerful tools but they are costly, complicated to learn, and require physical reconfiguration of the system.
2. Software-based tools are quite flexible and avoid most of the drawbacks with the hardware-based tools, but they interfere with the real-time properties of the system, since they use system resources. An example of a software-based tool can be found in [RTI94].

Thus, the available tools are most suitable when debugging includes finding hardware faults (case 1 above), or when focus is on real-time performance at the user or application level (case 2 above). At the system level of software development, however, there is a need for a solution that combines the benefits of the just mentioned techniques.

The principle presented in this paper is believed to provide a unique and appropriate combination of:

- Timing measurements down to the level of microseconds.
- Insignificant changes of real-time properties.
- Process-level timing analysis without software changes.
- Possibilities to add specific checkpoints in the application.
- Flexible configuration of timing result output (to shared memory, to log files, to digital IO, etc.).
- Possibilities to let additional processors analyze and adjust real-time properties on line.

We think our solution is particularly useful when optimized real-time software needs to be developed using a minimum of resources.

2.2 System description

The real-time kernel that was modified, was developed at the Institute of Automatic Control, Lund University. More information about the kernel can be found in [AnB91]. The kernel is used in many research projects, and in the education. For instance, the robot-laboratory computers controls robots with the aid of the kernel. To host the supervision system, the real-time kernel had to be modified. The kernel in question is implemented in Modula-2. Time-critical parts, like the process switch, are written in assembler. Changes in the kernel are discussed in Section 4.

The underlying system is a robot control computer which controls a robot's motion. The kernel runs on a Motorola 68040, 25MHz-processor. A logic analyzer was added to keep track of the output of the supervision system. Figure 1 shows the different parts of the system.

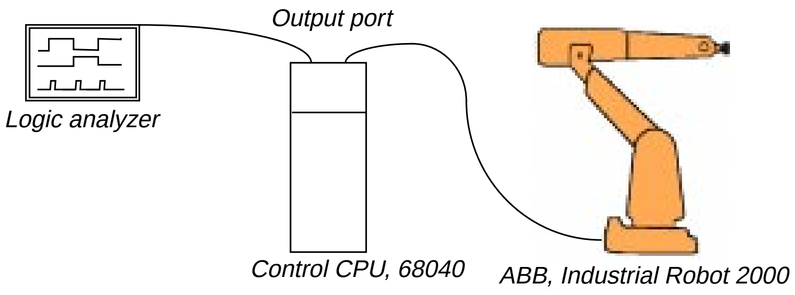

Fig. 1. The parts of the supervision system.

2.3 Supervision output

The output of the supervision system is dependent on the *output function*, which duty is to perform the supervision. The function is specified by the application programmer. Thus, the supervision system does not provide an output function. The responsibility of the real-time kernel is to provide the output function with correct information at proper situations. For instance, the kernel automatically calls the output function every time a process switch takes place. In the case study (see Section 5), the output function consisted of a single assembler instruction, transferring the input argument to the output port. The port was connected to an external logic analyzer.

Every process is given an identification number. This number is used as argument when the kernel calls the output function. As the process is running, its number is transferred to the output port by the output function, thus the execution can be traced on the logic analyzer. An example of the system's output can be viewed in Figure 2. The top line is set and reset by the application, thus the source code must contain explicit calls to use these lines. There are no limitations on the implementation of the output function, since it is specified by the programmer. An alternative output function could, for example, buffer the identification number with timestamps, for later analysis.

Figure 2 shows the scheduling of four processes, named A, B, C and D. They are given one line each. The horizontal axis displays the time. A routine that handles the clock interrupt, is displayed at the bottom. The top line show when a specific calculation is made in process A. As the calculation starts, the line is raised, and when the calculation is finished, the line is lowered. Another interesting line, represents the idle process. It is active when the processor is idle. The logic analyzer can measure times. These measurements can be used to calculate processor utilization, mean execution time of a process, worst-case execution time and the like.

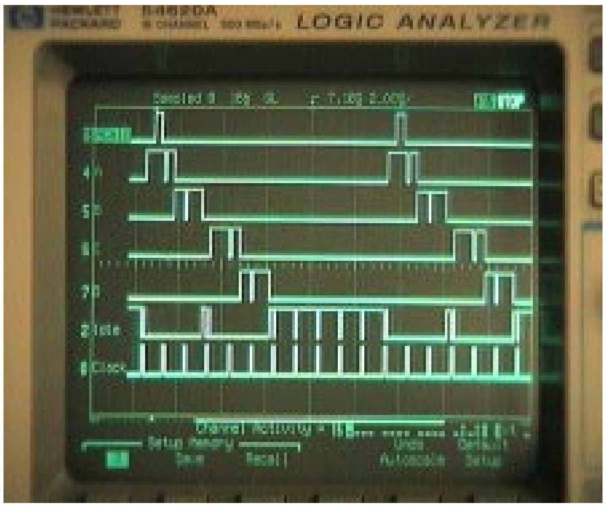

Fig. 2. Activities on the output port, presented by a logic analyzer.

3 Functionality description

To keep track of the processes in an application, they must have a unique identification number. The identification is used as an argument to the programmer-specified output function. The output function is called every time there is a change in the execution of a process, e.g. another process is scheduled to run. To host the new functionality, changes in the real-time kernel have to be made. New calls to the output function must be added in the kernel. The task of the output function is to register a change of executing process, in a programmer-specified way. The identification number enables the output function to keep track of the different processes. The process identification numbers are explicitly set in the application program; only the processes that are interesting from the programmer's view are given a unique identification. Since the output function is specified by the application programmer, changes in the output function can be made without recompilation of the kernel.

Other situations that have to be supervised are interrupts. In particular, the clock interrupt has to be modified to call the output function each time it runs. Other interrupt handlers can easily be modified to perform the necessary calls to the output function.

Apart from the automatic runtime supervision behavior, there is a manual way to use the output function. The programmer can explicitly call the output function with any argument, called event identification number. These numbers are solely controlled by the programmer; the real-time kernel does not affect them. This enables detailed studies of parts of processes. Execution time of a calculation can, for example, be measured. The internal states of a process, could be monitored. An event can last over several process switches. Event identification numbers in manual supervision calls are superimposed on the automatic calls. The programmer is thus responsible of choosing relevant identifications that do not conflict with the identifications of the processes.

Another process, created by the system, is the idle process. This process is activated each time the processor has nothing to do. The idle process is never activated then the processor is fully utilized. The utilization of the processor can thus be calculated when execution times are measured for the idle process, and other processes. The supervision system gives the possibility to adapt the scheduling of the processes to an optimal performance. More information on how to use the supervision system can be found in Section 6.

The supervision system supports many interesting areas of use. The scheduling can be tracked in detail. Execution times can be measured. Worst-case execution times can be estimated, although a more exhaustive examination of the worst-case execution time has to be performed before the definite time limit can be determined.

The functionality of the output function has to be specified by the programmer. This makes the supervision flexible, since the output function can be adapted to any specific hardware. The function should be short, because it is executed many times. A possible algorithm is a direct transfer of the identification to an output port. Another version can provide logging of processes and switch times. Trigger functionality can be implemented to start the logging at a desired situation. At runtime, the output function can easily be removed, or the output function can be changed. This increases the flexibility of the supervision system. For example, one process could use a specific output function to handle an event occurring in the process, and other processes could use a default output function.

4 Implementation issues

One primary aspect of the supervision system is to give each process an identification number. We have coded the identification number as an integer. To store the identification, each *process record*, containing essential information to the runtime system about the process, was expanded with an extra slot for the integer. This identification number is used by the runtime system. In addition, there is a global slot for an event identification number which the programmer handles manually.

Automatically, when a process switch takes place, the identification is “ored” with the number of the event identification. The result is placed in another global slot, the current identification, and sent as an argument to the output function. Figure 3 shows an overview of this proceeding.

The reason to store the current identification number is to give the programmer the ability to read it during execution, thus gaining access to the state of the application. Thus, the application programmer should divide the range of identification numbers so

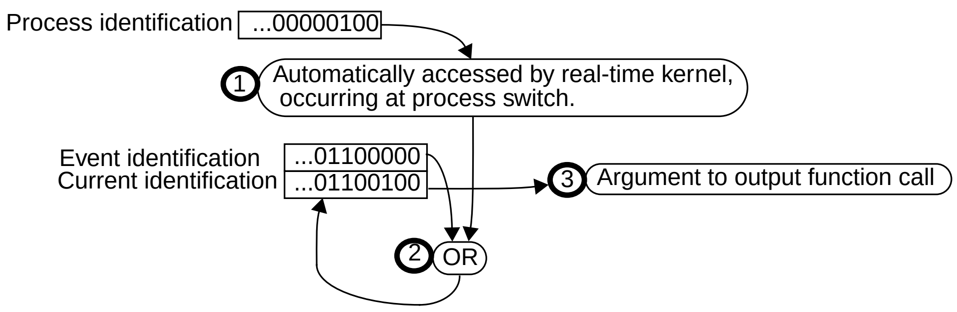
Process identification
...00000100
1
Automatically accessed by real-time kernel, occurring at process switch.
Event identification
Current identification
...01100000
...01100100
3
Argument to output function call
2
OR

Fig. 3. The supervision routine of the runtime system, at a process switch.

that e.g. the lower X bits are used for process identification numbers and the higher Y bits for event identification numbers.

If there is no output function specified then the work described in Figure 3, does not have to be performed. To ensure this behavior, there are checks to ensure the existence of an output function. If it does not exist, the runtime supervision functionality is skipped. The kernel will then behave like the original kernel, except for the extra checks. The cost for the extra checks is presented in Table 2.

An overview of the programming interface of the supervision system, is found in Table 1. The usual way to set the identification of the process is by assigning an identification when the process is created.

Procedure	*Argument*	*Remark*
Set process identification	32-bit integer	The currently running process's identification number is set.
Set one bit of the identification	32-bit integer	Only one bit is set in the identification number.
Set event identification	32-bit integer	The output function is also called.
Read current identification		Return the argument given to the output function
Set output function	Function pointer	
Reset output function		Clear the function pointer
Call output function	32-bit integer	Explicit call form the application program.
Is there a output function?		Returns boolean

Table 1 Programming interface of the supervision system.

The supervision kernel brings extra overhead, compared to the original kernel. Two expenses are obvious, extra memory and performance loss. Extra memory, for the identification, is allocated in every process that is created. Another memory expense is in the runtime system itself, to keep track of the event identification number, and the current identification number. These memory costs are negligible.

The performance loss depends on the extra overhead due to the supervision system. To measure the overhead, four different measurements on the process switch routine in the kernel, have been made. First, the original kernel's execution time is measured. Three other measurements are conducted on the supervision-modified kernel: when there exists an implementation of the output function, when the output function exists but does nothing, and when there is no output function. In the case where the output function is implemented it transfers the argument to an output port. In this case the output function consists of a single line of assembler; it can represent an common implementation of the output function. More advanced implementations could be made, but their execution time must not exceed the interval of time of the clock interrupt. Otherwise the execution of the system cannot be guaranteed to work properly, because the kernel will call the output function before the previous call is finished executing. The results are presented in Table 2.

Kernel	*Clock interrupt*	*Process switch*
Original (μs)	54.90 - 110.9	29.20 - 30.90
No output function (μs)	56.90 - 112.9	29.80 - 31.45
Output function (no functionality) (μs)	63.75 - 121.7	32.45 - 34.75
Output function (Transfer argument to output) (μs)	66.75 - 123.7	33.55 - 36.45

Table 2 Comparison of execution times of the original real-time kernel, and the supervision-modified kernel.

To measure the execution time of the process switch routine and the clock interrupt handler, a line of code is added before and after the routines. These two extra lines produce a signal to an output port, which is connected to a logic analyzer. The execution times can thus be monitored. The extra time these lines take has also been measured and presented in Table 3. The benefit of inlining the output port code is to exclude extra overhead that would result from a function call.

Time to set and reset output	1.500 μs

Table 3 Execution time to set and reset an output port.

4.1 Supervision example

To give a concrete form to the use of the supervision system, we show a simple example. Two processes, named Luke and Ben, are created. There is also an idle process created implicitly by the real-time kernel. The clock interrupt handler is also shown. The process Ben has higher priority than the process Luke. Luke does three loops and then rests for 100 milliseconds. The ability to manually trace the execution is also used in the Luke process, after each loop. Ben, on the other hand, only executes one loop

```
Process Ben {
    SetPriority(HighPrio );
    SetProcessIdentification(BenID );
    LoopForever {
        loop 3000 times;
        WaitTime(100 ms );
    }
}
```

```
Process Luke {
    SetPriority (LowPrio );
    SetProcessIdentification(LukeID );
    LoopForever {
        SetEventIdentification(High );
        loop 1500 times;
        SetEventIdentification(Middle );
        loop 1000 times;
        SetEventIdentification(Low );
        loop 500 times;
        WaitTime(100 ms );
    }
}
```

Fig. 4. Pseudo-code for the processes Luke and Ben.

before resting for 100 milliseconds. Figure 4 shows pseudo code for the two processes. The output function is only transferring its argument to an output port, which is connected to a logic analyzer. The output of the analyzer is shown in Figure 5. The picture is taken when the system is running. Notice that Ben interrupts Luke, due to its higher priority.

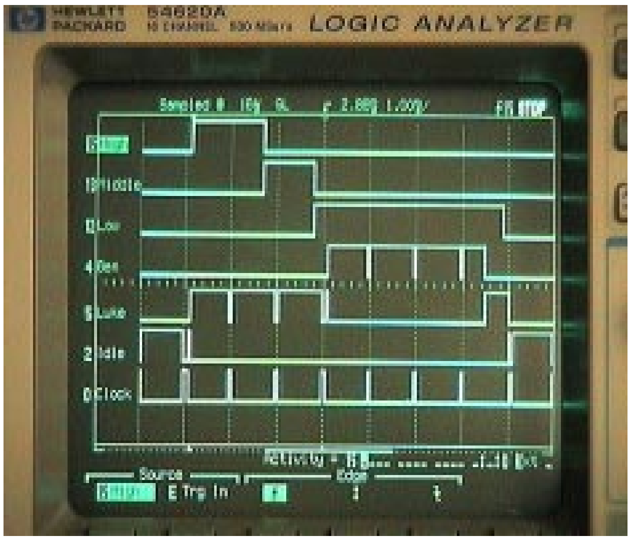

Fig. 5. Output of the logic analyzer.

Information that can be extracted from the logic analyzer are execution times of the processes, and how much time internal phases take for the Luke process. The processor utilization can be calculated. The different execution times of the processes are presented in Table 4.

Execution time, Ben	3.520 ms
Execution time, Luke	3.360 ms
Time during idle	95.04 ms
Cycle time	102.1 ms
Processor utilization	7%

Table 4 Information extracted from the output of the supervision.

5 Case study of a real-time garbage collector

The supervision system has been used in a study of the execution of a real-time garbage collector (RTGC). The purpose of this section is to emphasize the benefits of the supervision system, and not to explore the intricate execution of the RTGC. Information about the RTGC can be found in [Hen96].

The RTGC is a predictable garbage collector suitable for hard real-time systems. Its execution can be calculated beforehand, making it a design-phase problem. To verify these calculated execution times, the supervision system has been used.

The RTGC groups processes into low-priority and high-priority processes. During the execution of low-priority processes, the garbage collecting work is interleaved with the execution of the application process, i.e. each time memory is allocated the RTGC performs some garbage collecting work. On the other hand, high-priority processes are more time critical. A minimum amount of garbage collecting work is performed during their execution. Instead the work is collected, and performed by the RTGC process after a high-priority process is done executing.

The supervision system monitors the execution of the RTGC. The output function simply transfers its input argument to an output port, which is connected to a logic analyzer. A snapshot of the logic analyzer display is shown in Figure 6. The lines in Figure 6 are representing, from bottom upwards, the clock interrupt handler, the idle process, a low-priority process, the RTGC process, a high-priority process, and the time the RTGC uses to accomplish its work. In the figure, the low-priority process is aborted by the high-priority process. After the execution of the high-priority process, the RTGC process resumes the execution, tidying up the memory heap. Then the low-priority process is allowed to continue its execution. The RTGC process is not activated after the execution of the low-priority process, since the RTGC is handling the heap interleaved with the execution of the low-priority process.

The new supervision aspect of the RTGC revealed interesting information about its behavior. Its execution was verified, but new ideas on how to increase the performance of the RTGC were found. Bugs were also found and corrected. One major improvement, resulting from the study, was to make it possible to abort the RTGC during a memory copy in the heap. The supervision system showed long periods where the interrupts were deactivated. No high-priority processes could start to execute during these periods. During these long periods, the RTGC moved a large object. The solution

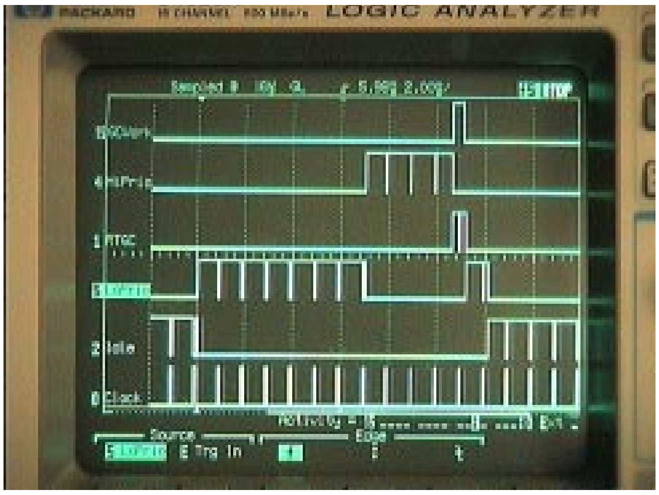

Fig. 6. Snapshot of the logic analyzer output, taken when the behavior of the RTGC is studied.

was to modify the copy-routine and make it abortable, thus allowing high-priority processes to abort the execution of the garbage collector.

These corrections and discoveries are not easy to penetrate by looking at the source code, but they appear rather obvious when the runtime behavior is visualized.

6 Future development and use

The supervision system could be complemented, or changed in some details. One feature is to provide functionality to measure the execution time of the output function itself, or to measure the amount of overhead in the runtime system. The output function must not load the execution too much, and steal too much time from the processor. The programmer should be issued a warning if it is too large.

More flexibility could be added if more runtime functions were introduced. Different functions could be called whenever there is a change of the processes, e.g. process switch, a process waits, a process is blocked etc. Those functions should be implemented by the programmer. The changes in the kernel would be to call these functions at the proper time. Functionality like event identification, is left to the programmer to implement in the new functions.

The supervision system will be used in a newly started research project called Integrated Control and Scheduling. The project is hosted by ARTES and aimed at practical management of hard real-time demands in embedded software. ARTES stands for “A network for Real-Time research and graduation Education in Sweden” and is supported by the Swedish Foundation for Strategic Research (SSI). The project is developing, new, more dynamic, methods of scheduling hard real-time systems. The project will have two different aspects. First, the introduction of feedback from code and external devices. The information will aid the scheduler and the controllers of a system to change behavior to suit the situation. The other approach will be a timing analysis of the worst-case behavior of software. This solution incorporates attribute grammars and incremental semantic analysis. The supervision system could aid the project with verification.

7 Conclusion

In the real-time community, a program is correct if it can perform its task within a certain period of time, i.e. the program must perform its computation before a deadline. Execution times can be hard to predict, or to calculate beforehand. The calculated results tend to be pessimistic, in order to ensure the deadline to be met. The supervision system is a tool to verify execution times of processes. This report describes how the system can be used, and what it can perform. Changes in the real-time kernel have been described to host the system. Measurements on the extra overhead introduced, are done and compared to the original kernel.

The programmer's interface is described along with an example of its use. In a case study of a real-time garbage collector, the supervision system was used to verify the operation of the garbage collector. Other benefits came out as well from the use of the supervision system. Bugs were found, and new ideas on the operation of the RTGC were invented. These modifications were not easy to penetrate by looking at the source-code. Using in-circuit emulators or special profiler tools would have been costly in terms of money (hardware) and/or time (operation, setup, etc.). Furthermore, our solution's connectability to, for instance, additional CPU-boards creates new possibilities for on-line (and possibly autonomous) supervision and tuning of real-time setting (sampling periods, control performance, time-out exceptions, etc.). Even if this has not been exploited yet in our research, we claim, based on experiences so far, that the techniques presented in this paper form a powerful 'no-cost' tool for execution time analysis.

Acknowledgments

The persons I thank for their support are Roger Henriksson, Klas Nilsson, Anders Blomdell, Görel Hedin, Boris Magnusson and Patrik Persson.

References

1. [AnB91] Leif Andersson, Anders Blomdell. A Real-Time Programming Environment and a Real-Time Kernel. *National Swedish Symposium on Real-Time Systems*, Dept. of Computer Systems, Uppsala University, Uppsala, Sweden, 1991.
2. [Hen96] Roger Henriksson. *Scheduling Real-Time Garbage Collection*. Licentiate thesis, Dept. of Computer Science, Lund Institute of Technology, Lund, January 1996.
3. [RTI94] Real Time Innovations, Inc. *Stethoscope - Real-Time Graphical Monitoring and Data Collection Utility - User's Manual*. October, 1994.