Time Accurate Simulation: Making a PC Behave Like a 8-Bit Embedded CPU ⋄

Jakob Engblom*
Dept. of Information Technology
Uppsala University
P.O. Box 325
SE-751 05 Uppsala
Sweden
jakob@docs.uu.se
http://www.docs.uu.se/~jakob

Magnus Nilsson
CC Systems AB
Fyrisborgsgatan 5
754 50 Uppsala
Sweden
magnus.nilsson@cc-systems.se
http://www.cc-systems.se

Abstract

When developing embedded systems, developers often use simulation techniques to allow development to proceed without access to the target hardware. To make use of the high quality development tools available on the PC platform, one popular simulation method is to compile the code intended for the target system to run on the PC, allowing the development of the software to proceed on the PC without use of the final target system. For a distributed system, each target node is given its own process on the host PC, with software on the PC simulating the communications network.

We have extended one such simulation environment to include the aspect of relative and absolute processing speed of the target systems, allowing for a more accurate simulation where not only functional but also timing-related bugs can be found and diagnosed. The absolute time mode makes the software on the PC run at the same speed as the real target, thus allowing the mixing of simulated nodes with real target hardware in the same system setup. The method is applicable to any embedded processor, as long as it is significantly slower than the PC.

The system has been implemented and tested on standard PCs running the Windows NT operating system, and is currently being used in industrial projects.

Keywords: Embedded Systems, Simulation, Case Study, Real-Time Systems, Computer Architecture, Hardware Modeling

1. Introduction

Developing software for embedded systems usually involves simulation on the host development system. The target system might be unavailable since hardware development is taking place in parallel to software development, the hardware might be too big or expensive to equip each engineer with her own test bench (forestry machines or cars), or the hardware is too dangerous to test early software on (weapons systems and medical equipment).

Furthermore, the productivity of a programmer is much lower when using the hardware of the target system, since setting up the system, downloading code to multiple nodes, and running it is time consuming, and the debugging tools available cannot match the convenience of desktop tools (the communications links limit the observability). Thus, simulation technology is almost necessary to allow embedded programmers to achieve any kind of productivity for complex systems.

As illustrated in Figure 1, the systems we are targeting typically include a CAN-bus connecting an operator panel and a number of control nodes containing processors running the software of the system. Each control node contains a processor that runs a single program, and has a number of I/O modules attached to them using CAN or some simpler serial links or direct

* This work was performed within the Advanced Software Technology competence center (ASTEC, http://www.docs.uu.se/astec), supported by the Swedish National Innovation Systems' Administration (VINNOVA, http://www.vinnova.se). Jakob is an industrial PhD student at IAR Systems (http://www.iar.com) and Uppsala university, sharing his time between research and development work.

⋄ Uppsala University, Dept. of Information Technology, Technical Report 2002-024. Submitted to RTAS 2002 and accepted as a poster, but none of the authors was able to attend and present the poster, thus not published.

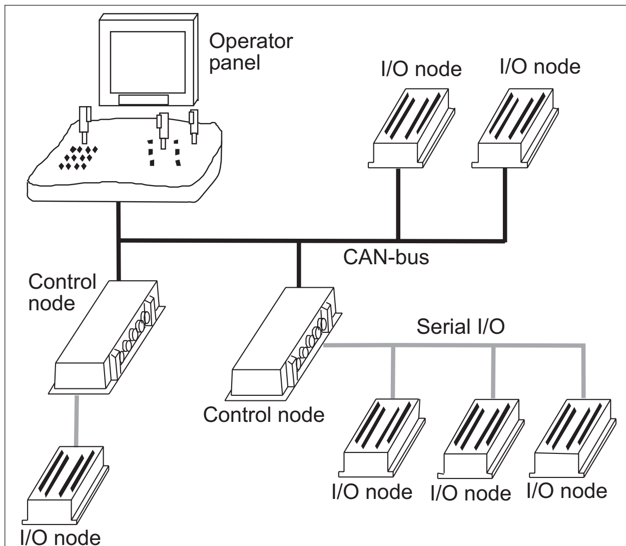

Figure 1. Example target system

electrical connections. To support the development of such systems, we need to simulate multiple nodes in the PC of the development engineer.

The time accurate simulation technique described in this paper is based on separation of the target code into a hardware-dependent and a hardware-independent part, with a well-defined programming interface to the hardware-dependent part. The hardware-independent code is compiled and run on a PC for simulation, with the hardware-dependent code replaced by code that simulates the hardware and hardware-dependent code in the PC environment.

Several nodes can be simulated, communicating over a simulated CAN network or simulated direct I/O links. The basic framework only simulates the functionality of the code and not the timing, since all code is run at full speed on the PC, with very different timing from the target system. The timing is different both in *absolute* terms (i.e. the code requires a different amount of time to run on the PC compared to the target system) and *relative* terms (i.e. the relative speed of different nodes is not reflected in the speed of the simulation).

To perform time accurate simulation, breakpoints are added to the hardware-independent source code. The breakpoints contain information about the execution time for a piece of code, when executed on the target system. By comparing the total amount of target-system time that each simulated node has accumulated, a scheduler is able to synchronize their execution on the PC, giving a relative timing behavior consistent with the speed of the target systems (slower nodes will be slower in the simulation). The scheduler can also slow down the execution so it runs at the speed given by the times in the breakpoints, thus making the absolute timing identical to the target systems (all nodes run at a speed consistent with the target system).

We have implementated a prototype system for Windows NT with the scheduler implemented as a DLL, and have performed tests on the system verifying its functionality.

2. Related Approaches

The most common method to obtain simulation with some kind of timing aspect for an embedded target on a PC host is the use of *cycle-accurate simulators* like the ARMulator from ARM [2]. They are CPU simulators that execute code using a very detailed simulation of the target processor, allowing them to give very exact time measurements. Unfortunately, such simulators typically incur slow-downs of a factor 10000 or more, making them too slow for daily usage (even if the development host is much faster than the target system). Some simulators can boot operating systems and run a whole system cycle-accurately, meaning that there is no need to adapt the code for simulation, giving a very precise simulation [4].

Hardware emulators are special versions of processors that are used in a development setting to provide full-speed execution of code with increased visibility compared to the regular processors. Emulators are very expensive, not available for all processors, and still require a download process before running and debugging can begin. They are invaluable for tracking down subtle timing-related bugs in the hardware-processor interface, but are not as useful as a replacement for the real target system during development.

Instruction set simulators like SimICS from Virtutech [10] simulate code with a slowdown between 10 and 100, but do not attempt to simulate the execution speed of the processor, making them suitable for finding functional bugs but not timing bugs. The SimICS extension called SimICS Central [9] builds on an idea similar to our Time Accurate Simulation. SimICS Central synchronizes several target nodes running on multiple host computers, while we synchronize within a single simulating host computer (and run the programs compiled for the host, not inside a simulator).

Hardware-software co-simulation, where CPU simulators interact with low-level simulation of the VHDL code used for custom FPGAs or ASICs is a very powerful approach. However, the hardware simulators are very very slow (factor of 1000000), and the approach is mostly intended for systems containing large amounts of application-specific logic. Thus, this does not help much with simple systems built around standard microcontrollers.

ENEA OSE offers a *simulation environment providing the API of a Real-Time Operating System (RTOS)* for use on host machines [14]. Applications written for the OSE API can thus be run on the host PC, without any change to the code, and also communicate with other real or simulated OSE nodes. This is similar in concept to our basic simulation system, but it is particular to OSE, and there is no effort to provide timing correctness.

A common problem with all those solutions is that they require the use of special debuggers that are able to talk to the simulators or hardware. One purpose of the PC-based simulation on which we based the time-accurate simulation was to enable to use of PC programming tools like the Visual C++ Debugger from Microsoft or Purify from Rational. The goal of our work is a simple and useful system that is very cheap to implement.

3. Basic Simulation System

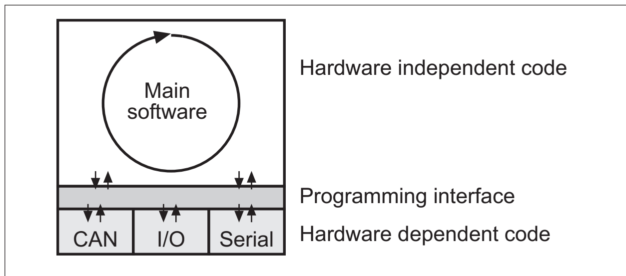

Figure 2. Programming model for simulation

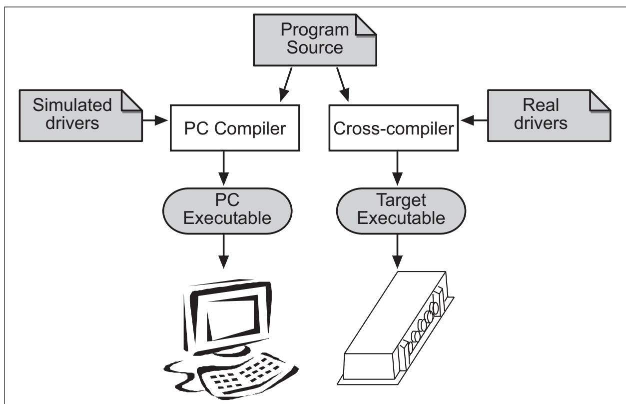

Figure 3. Compilation chain for simulation

The time accurate simulation is an extension to an existing simulation technique that allows simulation of target code on a PC host and the use of PC programming tools to debug the simulated code.

The simulation technique is based on writing applications with strict separation between the hardware-independent application code and the hardware-dependent code, as illustrated in Figure 2. This allows the hardware-dependent code to be replaced to allow simulation on a PC with the same program source code, as shown in Figure 3. Thus, the target programs execute as native programs on the host PC, enabling very fast execution and the use of any programming tool available on the PC platform for program inspection and debugging.

The interface between the hardware-dependent and hardware-independent parts is a standard API (application programming interface) that has been developed during and used in a number of embeddded systems projects. It is a set of functions like `CanSend(channel, message, &result)`, `IoRead(port, &value, &result)`, and `SerialSend(channel, buffer, bytecount, &result)`.

In the simplest simulation model, shown in Figure 4, all control nodes are simulated on the PC, communicating with each other and a simulation of the surrounding world and the controlled machine. Using a model of the world allows the software to test interaction with hardware that is not available. Each control node and world simulation node is given its own process on the host operating system, since this simplifies the structure of the software.

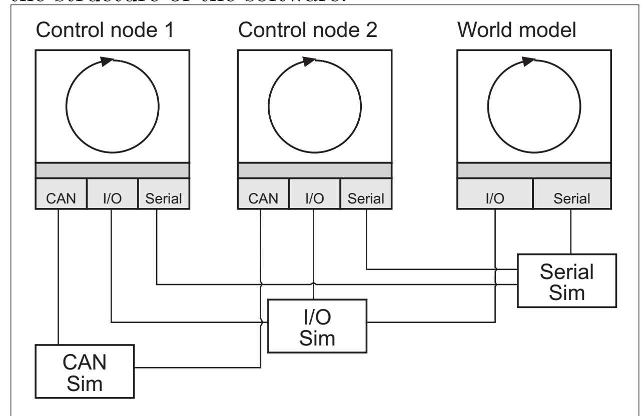

Figure 4. Simulation within a PC

As shown in Figure 5, by forwarding the signals on the simulated buses to external interface cards, it is possible to mix simulation of control nodes and a world model with real control nodes and real-world systems.

This simulation system is very powerful and allows for a gradual transition from purely simulated systems to a system running completely on the real target hardware. This methodology has proven very effective in real use, with most software errors found in simulation, before the software is first run and tested on the actual target hardware.

However, timing related bugs cannot be diagnosed in this simulation system, since there is no notion of target time. Each simulated control node runs at full speed in the time slice allocated to it by the PC operating

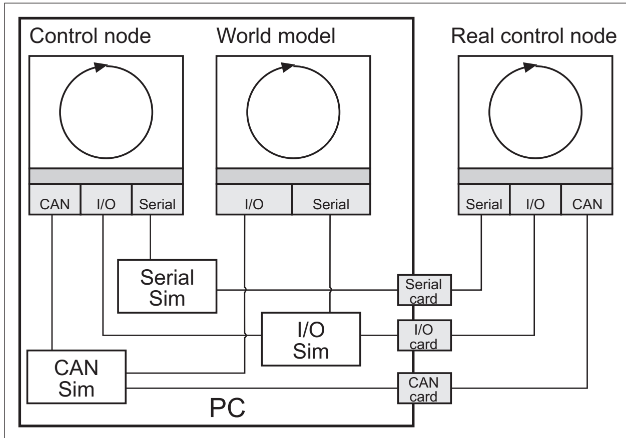

Figure 5. Mixed simulated-real environment

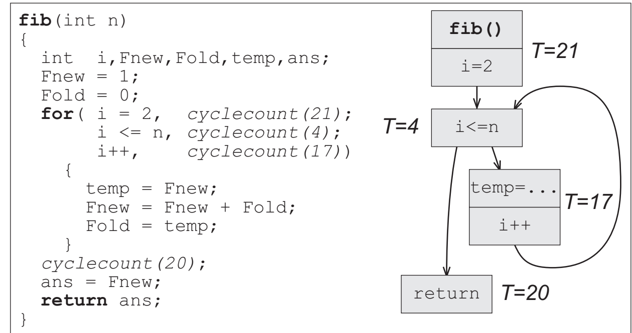

Figure 6. Basic blocks and time annotations

system, making the relative speed between nodes and the absolute speed of execution completely unrelated to that of the target system.

4. Time Accurate Simulation

We have extended the simulation system described above to support simulation with correct *relative timing* and *absolute timing*. To achieve this, we *synchronize* the simulated nodes based on the amount of target time they have accumulated.

To support this synchronization, *breakpoints* are inserted into code running on the PC. Each breakpoint indicates the time it would take the target system to execute the code between the breakpoint and the last breakpoint. Each simulated node has its own local time counter which is incremented when the code reaches a breakpoint.

4.1. Placing Breakpoints

The placement and density of breakpoints is a critical point in our simulation method. Many implementations of multithreaded languages like Erlang [3] and Java [1] use *yield points*, points at where threads can be interrupted. Yield points are inserted in such a way that a program can never loop without encountering a yield point. In essence, our breakpoints fill the same function, but we might want a higher density in order to allow a smooth simulation.

From a timing point of view, the best and most precise model is to place a breakpoint at the end of each *basic block* in the code. A basic block is a piece of code that is always executed as a unit; typically, basic blocks start at jump targets and end at branches or jumps [11].

For each basic block, we ascribe an execution time on the target platform. The time is then entered into the source code of the program by `cyclecount` annotations, as shown in Figure 6. The `cyclecount` annotations are then converted into breakpoints by appropriate `#define`s when compiling on the PC (for programs written in C/C++).

The execution time for a basic block can be generated by static analysis techniques like those used in worst-case execution time (WCET) analysis [6], or by measurement. For simple target machines, the assumption of a fixed execution time for a block is reasonable.

Placing breakpoints after each basic blocks obviously gives a very large overhead, since each breakpoint requires a comparison of times with all other processes in the simulation – for what might be the execution of a single target instruction. To reduce the overhead, the breakpoints can be implemented in such a way that the global time comparison is postponed until the local execution has accumulated a certain amount of execution time. This allows for a trade-off between the precision in the simulation and the simulation overhead.

Breakpoints can also be placed less densely in the code, in order to reduce the overhead or because timing information on the basic block level is not available. To keep one node from running away from the others, and to keep the time to be recorded by a breakpoint constant, it is necessary to place one breakpoint inside each loop. A loop which always executes the same number of iterations and contains no conditionals will have a fixed execution time, and could conceivably be handled by a single breakpoint.

Placing the breakpoints less densely (or implementing more efficient breakpoints as detailed above) means that the simulation will progress in a less smooth manner, in that the difference in time between nodes will be greater. However, over time, we will still maintain the correct relative and absolute timing.

Currently, we use measurements or manual estimates to generate times for larger chunks of code (sometimes including loops with fixed iteration counts). This works quite well in practice, and has been used in commercial projects with good resuls. Actually, even a small error in the timing for a breakpoint

will not be fatal in practice, as long as the overall execution time is approximately correct. An advantage of using less dense breakpoint placement is that the sensitivity to error in breakpoint timing is reduced, since each breakpoint will be counted fewer times and thus an error will have a smaller overall impact (assuming that the absolute magnitude of the timing error is on the same level for dense and sparse breakpoints).

For many practical applications, breakpoints on the basic block level might not actually be necessary to obtain good results. Considering the experience from multiprocessor simulators like SimICS [9], allowing each node to run a few thousand clock cycles before switching to another node is considered quite reasonable, and there is no reason why the same logic shouldn't apply to our system.

4.2. Relative Timing

Correct relative timing means that if two target nodes run their software at different speed, the same difference in speed should be manifest in the PC simulation. At each breakpoint, the local times of all simulated nodes are compared, and the node with lowest time is allowed to execute.

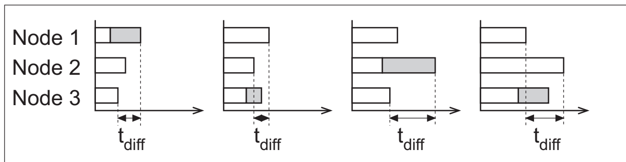

Figure 7. Time progresses in steps

As illustrated in Figure 7, at each instance the node with the lowest total execution time (shortest white horizontal bar) is allowed to execute up to its next breakpoint, adding a certain amount of execution time to that node (the grey bar). t_{diff} is the difference in execution time between the node that has executed the least target time and the node that has executed the most target time, and it changes as time goes on.

This method does not give a completely smooth progress of time on all simulated nodes simultaneously (which is impossible given the fact we only use a single processor), but rather stepwise progress where the nodes are approximately synchronized. Note that the maximal difference in time between two nodes, t_{diff}, is never greater than the greatest time between two breakpoints: $t_{\text{diff}} \leq \texttt{max}(t_{\text{break}})$, so the smoothness of simulation can be controlled by the spacing of breakpoints.

4.2.1. Unit of Time

To allow the comparison of time between different nodes, a common time base is needed. The execution time of code for each target is usually reported in clock cycles, but as different targets will have different clock speeds, this is not possible. Since clock speeds are often odd, like 6.33 Mhz or 1.57 Mhz, it is not feasible to use some common "base clock". Our solution is to use 64 bit integers to count time in units of picoseconds ($10^{-12}s$); with this resolution, we can represent execution times up to about 5100 hours (210 days), which is clearly sufficient. The error in converting to this unit from a clock speed of 6.33 Mhz is $1.17 \cdot 10^{-13}s$, which is about 0.000074 % of a clock cycle. All times are converted to this common unit as soon as they enter the system.

4.2.2. Hardware Interface Calls

The time consumed by calls to the hardware interface has to be accounted for. This can be done in two ways. Either a breakpoint is added at the call to the programming interface in the application code, or the hardware simulation code is instrumented with breakpoints.

Instrumenting the application code is simple and corresponds to the approach of giving each hardware interface call a certain execution time. Instrumenting the hardware simulation has the advantage of allowing a more fine-grained simulation and potentially yielding more precise times (since variations in the execution time can be modeled), but it is quite difficult to map the execution times for pieces of the target hardware dependent code to the simulation code on the PC.

4.2.3. Multiple Threads in one Node

When a target node contains several programs that execute in a multithreaded environment (without an RTOS), they all need to have the same target system time. This is solved by having all threads executing on the same node share a single local time. Determining which thread to run is based on a simple priority scheme where the thread with the highest priority is allowed to run; this scheme is mainly used to handle interrupts, since it could lead to starvation for low priority tasks.

Note that if an RTOS was used to schedule tasks on a node, the scheduler of this RTOS would have to be simulated to provide a correct simulation. However, the projects we have looked at (and for which Time Accurate Simulation was designed) have not used an RTOS.[1]

4.2.4. Interrupts

Interrupts complicate the picture, since they belong in the hardware-dependent layer and are invisi-

[1] It is still very common that embedded real-time systems do not use an RTOS, since in many cases they are not needed [5].

ble to the hardware-indepedent code (unlike the interface calls). Since the functionality provided by the interrupts is accounted for by the interface simulation, we only need to account for the time consumed by the interrupts. For the time being, we do not allow interrupts to activate functions in the hardware-independent code. The basic simulation framework and supporting code was designed in such a way that interrupts are confined to the target-dependent code.

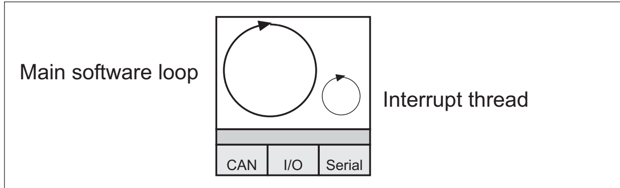

Figure 8. Interrupt thread inside a node

To account for the time spent in interrupts, a special thread is started inside a simulated node, as shown in Figure 8. This thread occasionally wakes up, queing itself for execution at the next breakpoint. The interrupt thread is given the highest priority, which means that when the node gets to execute the next time, the interrupt thread will execute and increment the local time by an amount corresponding to the interrupt handler execution time on the target. After each activation, the interrupt thread sleeps until it is time to trigger another interrupt.

Interrupts can thus only hit a node at breakpoints, which is not entirely true to their actual behavior on the target system. However, our solution does reflect the important effect of unpredictably adding execution time to a program, which is often good enough to find bugs. Note that the invocation of the interrupts in the simulated environment is not tied to the causes of events in the real system, like messages arriving on the CAN bus. This implicitly assumes that the interrupts can be modeled as a random time disturbance.

4.3. Absolute Timing

To obtain the correct *absolute* timing relative to the surrounding world, the simulation is simply synchronized to the clock on the host system. Only if the simulation node that is the furthest behind is also behind the real-time clock of the host is it allowed to execute. Thus, the simulation will sometimes be completely idle to let the real world catch up.

4.4. Taking Control of the Scheduling

Since the synchronization approach is based on determining which of the simulated tasks is allowed to run, we need to take control over the scheduling of the simulation tasks from the operating system scheduler.

In our solution for Windows NT, we use a `mutex` symbol that is shared between all the Windows NT processes involved in the simulation. All processes except the one that is currently running are blocked and queued on the `mutex`. To enable access to the shared `mutex`, we use a dynamically linked library (DLL) that all processes load on startup. This is the best way to obtain shared code and shared data under Windows NT, since each DLL can have a data area visible to all processes loading it. Actually, all the shared functionality in the simulation, like communications buses, are implemented by DLLs as shown in Figure 9.

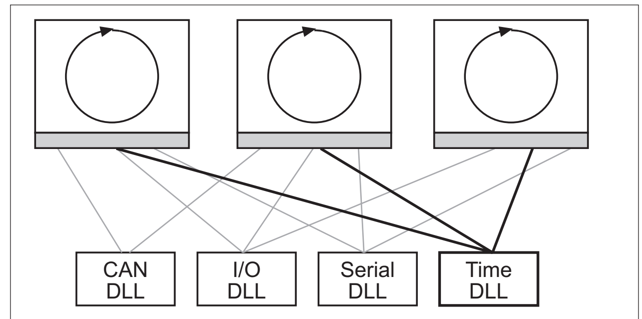

Figure 9. Dynamically linked libraries

In the Time DLL, there is a function that checks the execution times of the nodes in the simulation and gives the `mutex` to the node with the lowest execution time. This function is invoked by the breakpoints inserted into the code of each simulating node.

An alternative approach to scheduling control that used operating-system priorities to slow down nodes that were running ahead was ruled out since the control over execution was not strong enough to guarantee good synchronization.

Another alternative, using multiple threads instead of multiple processes, was discarded since it would complicate the compilation of the system. The code for each node is written separately, and integrating them into one program would thus entail possible name collisions. It would also require other changes to the code, and on balance, the performance increase would not be worth the trouble.

During our experiments, we found one problem with the Windows NT scheduler: when releasing a process from a `mutex` symbol, Windows NT increases its priority temporarily, so that it will run quickly after its release. This caused a problem since it meant that the process releasing the `mutex` was switched out before it could queue itself on the `mutex`, giving very bad performance for the system overall. This was solved by making the process that starts to run (gets the `mutex`) do a short sleep before beginning its work, giving the releasing task time to queue itself on the `mutex`.

5. Prototype and evaluation

We have completed a prototype implementation of the time-accurate simulation system, running on Windows NT. To provide some insight into the working of the system, a control panel application (shown in Figure 10) was implemented, as well as a tool to read and diagnose the event logs generated by the system.

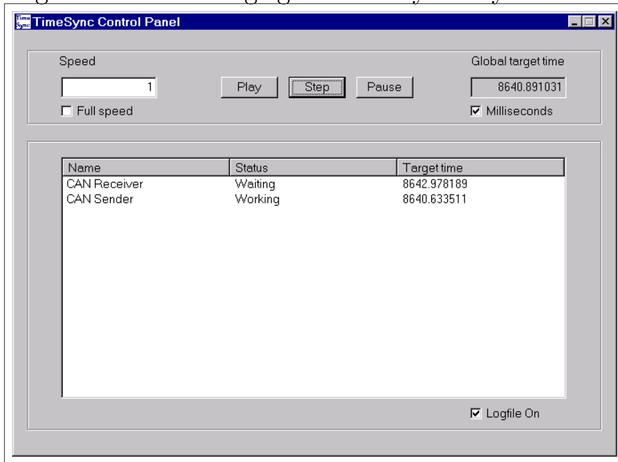

Figure 10. System control panel

The control panel also indicates some interesting and useful side effects of using the time-accurate simulation system. First, when running in absolute time mode, it is possible to vary the speed of the system by multiplying the real time by a constant before comparing it to the target systems' times (setting the "speed" value greater than one makes the simulation faster, and setting it to less than one makes it slower). Second, the entire simulated system can be stopped at the press of a button, which is very useful to allow the investigation of the state of multiple processes without the risk of any one of them running away.

5.1. Overheads observed

The basic functionality of the system was tested by implementing two tasks that send messages to each other. We extended the experiment by adding an interrupt thread to one of the processes, and using multiple threads within the same node. All the functionality tested correctly.

The overheads observed for switching between tasks was about 40 microseconds in the worst case. The experiments were performed with a median time of 5 ms (target time) between breakpoints, which is rather coarse. At this granularity, the overhead for the time simulation, running at absolute speed, was less than 2% of the total host execution time.

The main limitation to the number and speed of nodes that can be simulated on a single PC is the density of breakpoints in the simulated code rather than the absolute speed of the intended target. Putting breakpoints every 1 ms will increase the overhead per node to about 10 %, which should still make it possible to execute at least five nodes simultaneously in absolute time. This means that breakpoints on the basic block level are infeasible for simulation with correct absolute time. Basic-block level breakpoints can still be used to get a high-resolution simulation with correct relative time, however.

Note that even if breakpoints are placed rather sparsely, the system still provides a useful service in that over time, each node will run at the right rate. This does allow us to detect errors like one node feeding another node data at a too high rate. We can still detect errors in time-dependent algorithm, since the time taken to compute results can be determined. Simulating multiple nodes on a single PC will always entail a certain coarseness in the simulation, and experience points out that even rather coarse time-accurate simulation systems can be very useful in a practical development environment.

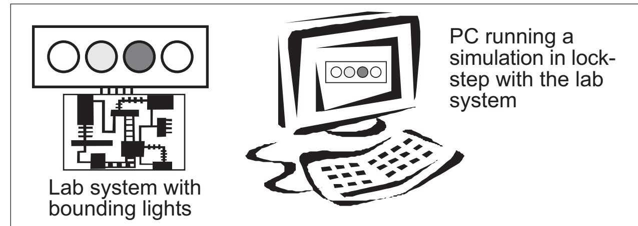

Figure 11. Demonstration system

Figure 11 shows the setup of another experiment, where a single task was written that simulated the LED output of a simple application running on a lab board. The PC simulation was then instrumented with the timing measured from the lab board, and the two programs started simultaneously. The PC simulation was executed in absolute time, and maintained synchronization with the lab system, providing a nice demonstration that the absolute time simulation worked.

5.2. Limitations

The system as presented is not perfect or complete in any sense, and we would like to point some of the current limitations of our approach.

The usability of this approach depends on the absolute speed of the target systems being much less than that of the simulating system, since we want to simulate multiple nodes and the communications between them. For the intended target systems, 8-bit and 16-bit processors running at speeds between 1 Mhz and 40 Mhz, this is definitely the case[2].

[2]Note that the host systems we using, PCs, are among the

The assumption of a single execution time for a basic block (or greater unit between breakpoints) depends on using predictable and deterministic hardware. For advanced processors like the Pentium 4 or PowerPC G4, this is not the case, but most of the CPUs used in embedded systems are typically simple and deterministic [6, 7]. For machines with pipelines, the model could be expanded with times for the edges in the basic block graph, in the same fashion as done in WCET research [13], but this would entail an incredible overhead. More work is needed on how to optimally place breakpoints considering the allowable overhead, and then how to account for pipeline effects within this breakpoint structure.

There is no support for a real-time operating system (RTOS) being used on a node. If an RTOS was used, the RTOS scheduling mechanism would need to be simulated and RTOS overheads accounted for.

Considering the use of user-level interrupts, the target systems have very short interrupt latencies, on the order of a few clock cycles. This makes it hard to provide timing-correct responses to interrupts in the simulation, since the task switch latency on Windows NT can be very long [8], making the simulation react to interrupts much slower than the target system. This is almost unavoidable, due to the different design goals of general-purpose PCs and embedded systems. However, from the point of view of the simulated programs, the interrupts will still occur at unpredictable points, which is a very useful property in testing the code.

6. Conclusions and Future Work

Compared to the basic simulation system described in Section 3, the time accurate simulation adds the ability to find and diagnose another class of bugs, timing-dependent bugs, in the simulation on a PC, thus reducing the number of bugs that have to be discovered and fixed while using the actual target hardware. Both bugs depending on the relative timing of multiple nodes in the same system, and bugs depending on the absolute speed of the code on the target system can be found. An example of the former are bugs depending on communications races between target nodes, and examples of the latter are bugs in control algorithms that depend on the real-world timing of samples and output.

We note that the system as described in this paper is being used in real development work, since it offers potential gains in programmer productivity and product quality. No data is available yet on the precise benefits or required density of breakpoints.

For the future, there are many potential directions of development. It would be interesting to investigate how the simulation could be spread across several PCs, using Ethernet to carry the signals of the CAN bus and other buses. RTOS support for in-node scheduling will be required at some point, and support for interrupt-activated code in the hardware-indepedent layer could be useful. How to place breakpoints should also be investigated, especially considering how sparse they can be placed while still retaining useful relative and absolute time in the simulation.

To make the system more precise in actual use, a development platform that includes basic-block timing analysis and back-annotation of this information to the source code is needed. This requires changes to target compilers and the inclusion of WCET analysis techniques in embedded development environments. Manual timing estimates and measurements for the execution time of pieces of the code are still good enough to make the system usable.

More details can be found in [12].

fastest uniprocessors that are available today. Workstations and servers typically rely on multiple CPUs to boost performance, but that does not help our simulation until we have extended it to use multiple CPUs.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM System Journal*, 39(1), February 2000.

[2] ARM (Advanced Risc Machines) Ltd. WWW Homepage. `www.arm.com`.

[3] Joe Armstrong, Mike Williams, Robert Virding, and Claes Wikström. *Concurrent Programming in Erlang.* Prentice-Hall, 2nd edition, 1996.

[4] Kathleen Baynes, Chris Collins, Eric Fiterman, Brinda Ganesh, Paul Kohout, Christine Smit, Tiebing Zhang, and Bruce Jacob. The Performance and Energy Consumption of Three Embedded Real-Time Operating Systems. In *Proc. 4th International Workshop on Compiler and Architecture Support for Embedded Systems (CASES 2001)*, November 2001.

[5] Jack W. Crenshaw. Mea Culpa. *Embedded Systems Programming (US edition)*, 15(3), March 2002.

[6] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis.* PhD thesis, Dept. of Information Technology, Uppsala University, April 2002. Acta Universitatis Upsaliensis, *Dissertations from the Faculty of Science and Technology* 36, `http://publications.uu.se/theses/`.

[7] Jennifer Eyre. The Digital Signal Processor Derby. *IEEE Spectrum*, 38, June 2001.

[8] Michael B. Jones and John Regehr. The Problems You're Having May Not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, June 1999.

[9] Peter S. Magnusson, Magnus Christensson, Jesper Eskilsson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2), February 2002.

[10] Peter S. Magnusson, Fredrik Dahlgren, Håkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenström, and Bengt Werner. SimICS/sun4m: A Virtual Workstation. In *Proc. of the USENIX 1998 Annual Technical Conference*, June 1998.

[11] S. S. Muchnick. *Advanced Compiler Design.* Morgan Kaufmann Publishers, 1997.

[12] Magnus Nilsson. Time Accurate Simulation. Master's thesis, Dept. of Information Technology, Uppsala University, September 2001. Thesis number: UPTEC F 01 074.

[13] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proc. 4th International Workshop on Compiler and Architecture Support for Embedded Systems (CASES 2001)*, November 2001.

[14] ENEA OSE Systems. OSE Soft Kernel and Soft Environment. Product Description, OSESE ma8082000:003 R1.0, 2000.