Hardware-Software Run-Time Systems and Robotics: A Case Study

Vincent John Mooney III[1], Diego Ruspini[2], Oussama Khatib[2] and Giovanni De Micheli[1]
[1]Computer Systems Laboratory, Stanford University
[2]Computer Science Robotics Laboratory, Stanford University
Stanford, CA 94305
{mooney,ruspini,khatib,nanni}@cs.stanford.edu

Abstract

We present a sample implementation of a run-time scheduler, split between hardware and software, controlling a real-time robotics application. The hardware part of the run-time scheduler, implemented as a Finite State Machine (FSM), schedules the tasks for the application and can be readily extended to include additional tasks in hardware or in software. The software part executes tasks based on which tasks are ready to execute as indicated by the FSM. We have successfully implemented the scheduler on a working prototype which shows the feasibility of our approach.

1 Introduction and Motivation

Hardware-software co-design[6] is becoming more important as the performance of applications increasingly depends on the interaction between hardware and software. Designers would like to have some routines implemented in software, other routines in hardware, and yet others split between them. One of the important problems faced is the synchronization and scheduling the routines (tasks) in software and in hardware. A clear and easy solution is to put the run-time system in software and suitably design the hardware such that it can be controlled from the software [4]. Another solution is to design the system as a set of communicating Finite-State Machines (FSMs) and design the software run-time system to react to input events [1]. Yet a third approach keeps track of software routines ready to execute next in a FIFO [3]. A fourth way is to model all communication as *Remote Procedure Calls* where one process (in hardware or software) can trigger the execution of a thread in another process [11]. In this paper we consider the mixed implementation of a *run-time scheduler* in hardware and software as first proposed in [7].

We look at a real design case where a designer has to make a system work within timing constraints. We wanted to experiment with our co-design methodology in the context of a robotics application where we can test our implementation on a prototype. By looking closely at the requirements for this case, we can then characterize the CAD requirements from the designer's perspective. This helps us identify problem areas both in the design flow and in the CAD tools used or proposed for use. The specific co-design methodology we wanted to test was the *run-time scheduler* of [7]. We were able to implement a simple version of the scheduler. Thus, this paper presents a "proof of concept" example of a run-time scheduler split between hardware and software.

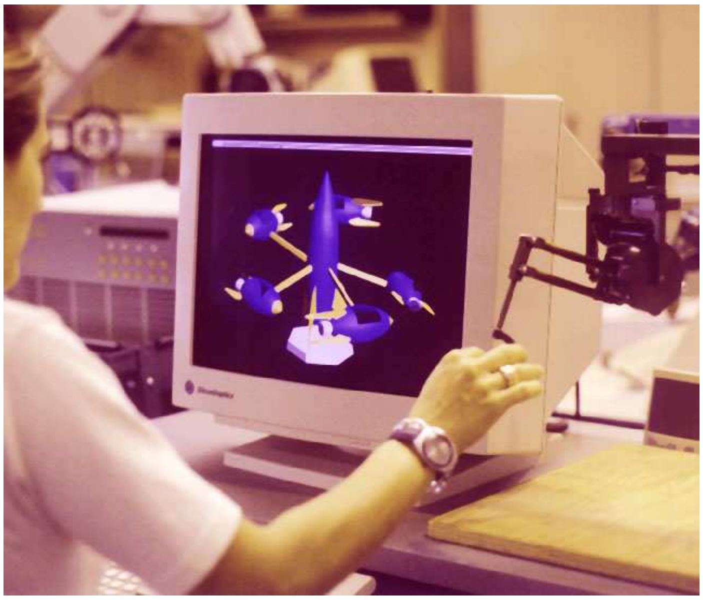

Figure 1: Haptic Robot With Graphics

2 Design Case Study

For our case study we considered the design of the following real-time robotics application: a *haptic* robot implementing force-feedback based on interaction through a graphics display [9]. The haptic robotics device contains a thimble where the user places his or her finger. The thimble is connected to the end of a small robot arm which can exert force on the thimble in any direction. The object in the graphics display is represented by a collection of polygons, usually in the range of 10,000 to 20,000 polygons. Figure 1 shows a user interacting with a graphic display where the haptic device gives feedback based on the position of a small point (called a *proxy*) on the screen. In particular, whenever the proxy collides with a graphical object, a force is generated and the user's finger in the haptic device is stopped from continuing penetration in that direction. In fact, the feedback is quite complex: the tactile interaction includes contact constraints, surface shading, friction, and texture [9]. Such a system has wide-ranging application possibilities, from helping surgeons operate on patients to training pilots with flight simulation. This application is a good case study because there are some tasks which are poorly implemented in software, e.g. collision detection, which could potentially run much faster in hardware. The first step towards integrating a hardware implementation of such a task into the system is to have a scheduler for the application.

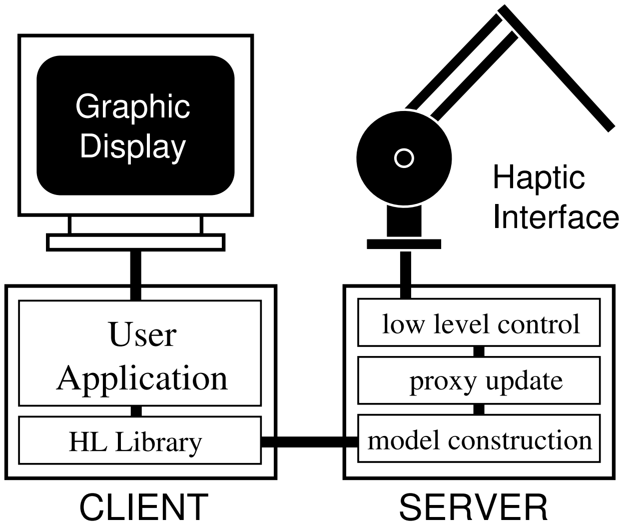

Figure 2: System Architecture

2.1 Original Design

The original design consists of an all software solution running on a Silicon Graphics Indigo (SGI) workstation and an IBM compatible PC. The SGI client contains the graphics routines which update the display, and the PC server runs the low level routines for controlling the haptic device. Our system architecture can be seen in Figure 2.

2.1.1 Collision Detection

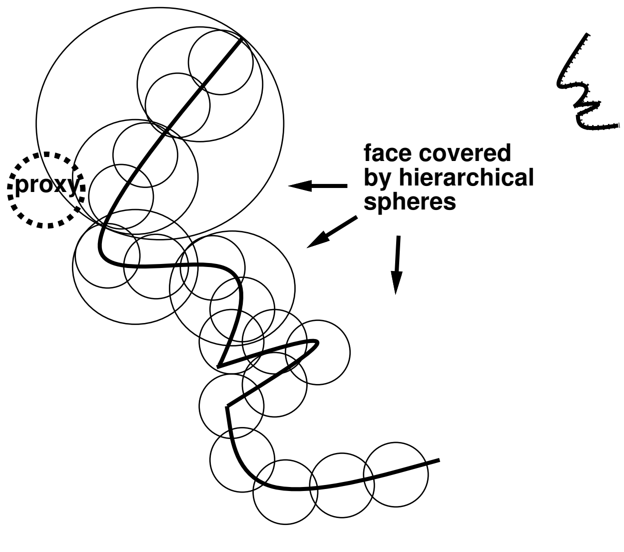

Figure 3: Sphere Characterization

From measurement, we observed that approximately 50% of the CPU time is spent in detecting when the proxy collides with an object in the graphics display. Collision detection is achieved by an algorithmic approach first described in [8]. The basic idea is to take a polygonal surface and cover each polygon with a small sphere. Then, from this initial set of spheres, they are hierarchically covered. Figure 3 shows the beginnings of covering a face using this method (the actual algorithm was written for three dimensions). At the end, we have a root sphere with covers the entire graphical object and all subspheres. The resulting tree data structure of hierarchical spheres has height $O(lg\ n)$. Since the collision detection algorithm checks the sphere hierarchy to see if collision has occured, $O(lg\ n)$ checks are needed.

2.1.2 Timing Constraints

Standard solutions are used for the low level hardware interactions that might otherwise involve strict timing constraints. For writing torque values to the haptic device, we use a device driver; for reading in the joint positions, we utilize the same device driver to read values from the port.

Model information about the graphics objects and the proxy are communicated between the SGI workstation and the PC by sending and receiving packets using the TCP/IP protocol. In the actual code on the PC, we never perform a blocking wait: instead, we check to see if a packet has arrived, and if so we accept the packet and continue.

The overriding timing constraint we have is a rate constraint: the tasks of the following section must complete before a hard real-time deadline is reached. Any delay in updating the torques could damage the haptic device or the user.

2.1.3 Haptic Library

The original code (called the "haptic library") for controlling the haptic device was written in C. Some of the most time-consuming tasks, such as that of communicating the polygons composing the graphics objects and then building a sphere hierarchy, are performed during the initialization and sphere building phases. Once a particular graphical display is up and running, the following tasks are executed in each iteration of a core loop called the *servo* loop:

- wait for next millisecond clock tick
- write torques to haptic device
- read joint angles of haptic device
- convert joint angles to x,y,z coordinates
- collision detect
- calculate new proxy position based on collision or not
- compute new torques for haptic device
- if ready, send/receive network packets (new proxy position, etc.)

For example, consider a user interacting with a graphical display of a teapot. When the proxy is in space not near the teapot, the user can move the proxy freely. However, as soon as the proxy comes close to the teapot, penetrating the sphere hierarchy (an example penetration in two dimensions is shown in Figure 3), collision detection is used to check if the user's proxy on the screen has hit the teapot. The haptic device provides force-feedback control to simulate the interaction of the proxy with the graphical object, e.g. when sliding along the curved surface of the teapot. Figure 1 shows a user utilizing the proxy to push around a spaceship merry-go-round. An execution of the *servo* loop for controlling the robot must complete once every millisecond.

2.2 New Design

The new design contains a slightly altered scheduler for the *servo* loop. We divide the loop into tasks in order to control their execution from a hardware FSM. Before entering the loop, we kick off execution of the FSM. Within the loop, we execute tasks as directed by the FSM.

2.2.1 Task Execution

We divided the tasks of Section 2.1 into three coarse grained groupings as follows:

- "Phantom" routines:
  - –wait for next millisecond clock tick
  - –write torques to haptic device
  - –read joint angles of haptic device
  - –convert joint angles to x,y,z coordinates
- "Proxy" routines:
  - –collision detect
  - –calculate new proxy position based on collision or not
  - –compute new torques for haptic device
- "Network" routines, executed only if there are network packets ready to send/receive:
  - –send new proxy position to graphics over network
  - –receive new graphics info over network

We implemented an FSM in hardware to sequence the above three course granularity software threads. For the sake of experimentation, we use an FPGA-based board (the PCI Pamette[10]) for the hardware implementation. This hardware FSM portion of the run-time scheduler is specified in Verilog and synthesized using the Synopsys-Xilinx interface; the tool flow is shown in Figure 4. The Synopsys tools used are the Behavioral Compiler^{TM} (BC)[5], Design Compiler^{TM} (DC) and FGPA Compiler^{TM} (FPGA).

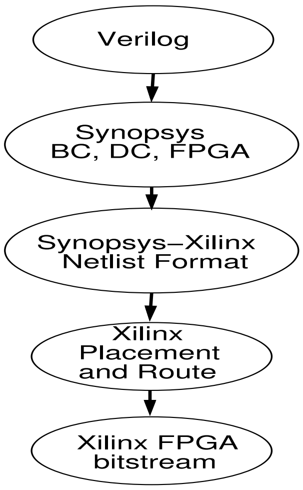

Figure 4: Synopsys-Xilinx Tool Flow

Task execution is described in [7]. Briefly, we associate a *start* and a *done* event with each software task (thread). In software, we have a *start* vector and a *done* vector which encapsulate the *start* and *done* events for each software-task. Since there are less than 32 distinct software-tasks, each vector is contained in a single word with a simple one-hot encoding.

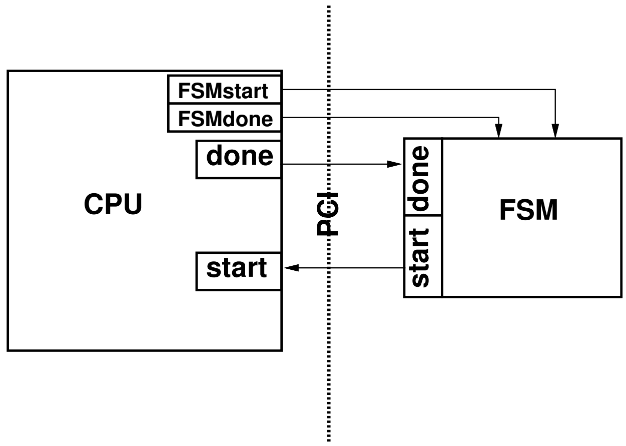

Figure 5: Run-Time Scheduler Control Communication

The run-time scheduler hardware FSM, synthesized to implement the control-flow of task invocations, updates the *start* vector in software as follows. First, it updates a local register containing the *start* vector. Then the CPU reads in the new value on a polling loop. When a software-task is finished executing, it updates the *done* vector by writing the value out with memory mapped I/O. Thus, the the *done* vector in the run-time scheduler in hardware is updated.

Note that we wanted to be able to turn the hardware FSM on and off from software, since the system initialization is directed by software. Thus, we added $FSMstart$ and $FSMdone$ signals to kick off and terminate, respectively, FSM execution. Figure 5 shows the communication of the $FSMstart$, $FSMdone$, *start* and *done* vectors.

Therefore we split the run-time scheduler into two parts:

- An executive manager in hardware with cycle-based semantics that can satisfy hard real-time constraints.
- A polling scheduler that executes different threads based on eligible software-tasks as indicated by the *start* vector.

The haptic library code was altered to accommodate this new split. In particular, a polling scheduler was written as the inner core loop implementing the three course-grained tasks as described here.

3 System Implementation

The original system in the CS Robotics Lab at Stanford was successfully ported to the NT environment all in software. Then we successfully implemented the split run-time scheduler in the actual design.

3.1 System Architecture

Our system architecture consists of an SGI workstation for the graphics, a PC with a Pentium^{TM} processor, and a haptic device connected to the PC.

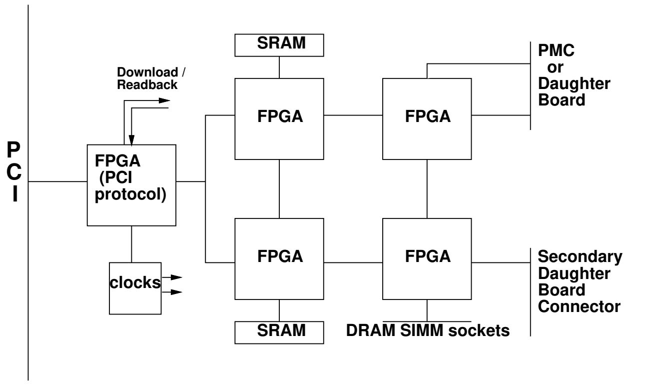

Figure 6: PCI Pamette Version 1 – Architecture

The PC has a PCI Pamette[10] board connected to one of its slots. The PCI Pamette, shown in Figure 6, has one FPGA dedicated to talking to the PC using the 32 or 64 bit PCI protocol, with four more Xilinx 4020E FPGAs configurable by the user. The two 128KB SRAMs are essentially scratchpad memories which the nearest FPGA can use. 16 bits of memory can be written to or read from each SRAM every cycle.

For communication with the FPGAs, we use the PCI protocol as implemented by the PCI Pamette software library for Visual C++ and the FPGA on the PCI Pamette. From the point of view of the software code, this appears as a memory-mapped read or write. However, there are timing constraints which must be observed by the two FPGAs that can read data from the 32-bit bus coming out from the FPGA implementing the PCI protocol: once an address appears on the bus, the data corresponding to that address must be read in the following cycle. Similarly, for writing to the bus (in which case the software is executing a read from memory-mapped IO), the data read must be driven to the bus on the following cycle and held there for six cycles. There are many more constraints explained in the PCI documentation [10].

In order to meet these exact timing constraints, we latch values going on/off chip using DC and then read the values using behavioral Verilog synthesized in cycle-accurate mode with BC.

3.2 Software Generation

The software for programming and controlling the PCI Pamette is available for Microsoft Visual C++ 4.0^{TM} with Windows NT 4.0^{TM} or for the DEC Alpha. Because we wanted to use a PC, we utilized the NT version.

The original code (called the "haptic library") for controlling the haptic device was written in 10,000 lines of C for Linux. In order to use the Pamette, we ported the haptic library to Visual C++ 4.0^{TM} with Windows NT 4.0^{TM}. This porting effort included writing a device driver in NT to control the haptic device as well as rewriting the network code for communication with the SGI workstation using TCP/IP.

Reading and writing to the SRAM on the Pamette is accomplished using memory-mapped I/O and hardware-tasks in the FPGA. The PCI interface takes an average of 5 to 9 CPU clock cycles to communicate a single 32-bit read or write.

Therefore, given a particular value of the *start* vector, the appropriate software-task(s) can be executed. The scheduler for the software is a simple polling loop. Note that for this to work we have to guarantee that after indicating that a particular software-task has completed by writing to the *done* vector, the next *start* value must be updated and ready to be read before the software polling loop next reads in the *start* vector. Otherwise, the software scheduler could read in the exact same *start* vector again and thus fail to meet the rate constraint of updating the robot's torque values every millisecond. We verified that the FSM implemented in the FPGA was fast enough by extensive simulation.

Figure 7: Teapot Graphical Object With Proxy

Figure 7 shows a graphical teapot model which we used to test the design. The proxy is shown on the teapot near the base of the spout. The teapot is composed of 3,416 triangular surfaces. The client computer was an SGI Indigo2 High Impact running IRIX 6.2 and the haptic server was a PC with a 266 Mhz Pentium Pro running Windows NT 4.0. The PC has 32 MB of main memory and a 512KB cache. Communication between the two computers was done through a standard ethernet TCP/IP connection. The haptic device used was a ground based PHANToM manipulator with 3 degrees of freedom in it force-feedback.

Table 1 shows the code space used for the various software tasks in the inner *servo* loop. The final executable took up 485KB of memory; however, the code and data used in the *servo* loop is much less and likely fit entirely in the 512KB cache on the PC (however, we did not verify this). Table 2 shows the code space used for reading and writing data from/to the bus and the SRAM, starting/terminating the hardware FSM, and the hardware FSM itself (in `hapticontrol.v`). Notice that the FSM takes only 178 lines of Verilog, while the supporting Verilog code takes

Task	Lines C
wait for next millisecond clock tick	65
write torques to haptic device	50
read joint angles of haptic device	48
convert joint angles to x,y,z coordinates	428
collision detect	2189
calculate new proxy position	664
compute new torques for haptic device	10
send/receive information over network	1328
device driver	899

Table 1: Code space for software tasks.

Task	Lines Verilog	Style of Verilog
ebusread1.v	146	behavioral
ebuswrite1.v	114	behavioral
generatecontrol.v	48	behavioral
haptic.v	242	structural
hapticcontrol.v	178	behavioral
startcontrol.v	150	behavioral
transactionmodelib.v	154	structural
writestart.v	99	behavioral

Table 2: Code space for hardware tasks.

1195 lines. Table 3 shows the various measures of utilization provided for the Xilinx 4020E which implements the Verilog code. The 4020E can fit at most around 20K logic gates. We are currently using about half of the available CLBs.

4 Conclusion and Future Directions

For future work, an ASIC implementation of the collision detection algorithm would drastically speed up the application, especially since the sphere checking is quite naturally parallelizable. The run-time scheduler described here could quite easily be augmented with such an ASIC. In fact, the inclusion of multiple hardware ASICs could be easily added to the system. The major practical design cost would be the specification and design of the collision detection ASIC.

Xilinx Measure	No. Used	Max. Avail.	Percent Used
Occupied CLBs	401	784	51%
Bonded I/O Pins	72	160	45%
F and G Function Generators	494	1568	31%
H Function Generators	93	784	11%
CLB Flip Flops	217	1568	13%
IOB Input Flip Flops	33	224	14%
IOB Output Flip Flops	18	224	8%
3-State Buffers	0	1680	0%
3-State Half Longlines	0	112	0%
Edge Decode Inputs	0	336	0%
Edge Decode Half Longlines	0	32	0%
CLB Fast Carry Logic	8	784	1%

Table 3: Statistics for Xilinx 4020E Mapping

The PC-Pamette architecture described in the previous sections provides the basis for a modular extendable hardware-software run-time system. Since the hardware part of the run-time system is in FPGAs, it can be reconfigured quickly with the synthesis path of Figure 4. Currently we only use one of the four available FPGAs. Portions of the real-time haptic control system can be migrated to hardware, either into FPGAs, ASICs or DSPs. For example, an ASIC implementing the collision detection algorithm (which has a lot of parallelism) could be integrated quickly into the run-time system.

For the final embedded application, the hardware part of the run-time system is synthesized into hardware rapidly since it is described in behavioral Verilog and uses synthesis all the way down to the bitstream for programming the XILINX 4020E FPGAs.

In conclusion, we have shown a sample application of a run-time scheduler split between hardware and software. The scheduler has been successfully implemented on a real-time robotics system. The fact that the main loop controlling the haptic device has its sequence of tasks scheduled by an FSM in hardware is transparent from the user's perspective. Finally, this work provides the basis for an extendable run-time system in hardware and software.

Acknowledgments

This research was sponsored by ARPA under grant No. DABT63-95-C-0049, NASA/JSC under grant No. NGT-9-6, National Semiconductor, Interval Research Corporation and Boeing. We acknowledge the help of Stefan Ludwig of DEC Systems Research Center who lent us the PCI Pamette with Xilinx 4020E FPGAs.

References

[1]F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems The Polis Approach,* Kluwer Academic Publishers, Norwell, MA, 1997.

[2]C. N. Coelho Jr. and G. De Micheli, "Analysis and Synthesis of Concurrent Digital Circuits Using Control-Flow Expressions," *IEEE Transactions on CAD/ICAS,* Vol. 15, No. 8, August 1996 .

[3]R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic Publishers, Boston, MA, 1995.

[4]J. Henkel, Th. Benner, R. Ernst, W. Ye, N. Serafimov and G. Glawe, "COSYMA: A Software-Oriented Approach to Hardware/Software Co-design," *The Journal of Computer and Software Engineering,* Vol. 2, No. 3, pp. 293-314, 1994.

[5]D. Knapp, *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler*, Prentice Hall, Upper Saddle River, NJ, 1996.

[6]G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design,* Kluwer Academic Publishers, Norwell, MA, 1996.

[7]V. Mooney, T. Sakamoto and G. De Micheli, "Run-Time Scheduler Synthesis For Hardware-Software Systems and Application to Robot Control Design," *5th. Int'l Workshop on Hardware/Software Co-design,*, pp. 95-99, Braunschweig, Germany, March 1997.

[8]S. Quinlan, "Efficient Distance Computation between Non-Convex Objects," *International Conference on Robotics and Automation*, pp. 3324-3329, 1994.

[9]D. Ruspini, K. Kolarov and O. Khatib, "The Haptic Display of Complex Graphical Environments," *SIGGRAPH '96 Proceedings,* pp. 345-352, August 1997.

[10]M. Shand, "PCI Pamette V1", Digital Equipment Corporation, System Research Center, http://www.research.digital.com/SRC/pamette/.

[11]D. Verkest, K. Van Rompaey, I. Bolsens & H. De Man, "CoWare–A Design Environment for Heterogeneous Hardware/Software Systems," *Design Automation for Embedded Systems,* Vol. 1, No. 4, pp. 357-386, October 1996.