Presented at 1999 IEEE Real-Time Applications Symposium — Work-in-Progress Session, Vancounver, Canada, June 1999.

Hardware/Software Co-Design of I/O Interfacing Hardware and Real-Time Device Drivers for Embedded Systems

David B. Stewart[1,2] and Bruce L. Jacob[1]

[1]Dept. of Electrical and Computer Engineering, and
[2]Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742
Email: *dstewart@eng.umd.edu, blj@eng.umd.edu*

***Abstract**: We have conceptualized a hardware-software co-design strategy for creating I/O interfacing hardware and device drivers for a real-time operating system that executes on microcontrollers, enabling hardware independent access to I/O devices at near-zero overhead. We achieve this low overhead through the addition of a hardware mechanism to the microcontroller architecture that we call nanoprocessors. The architecture extensions are orthogonal to the underlying microarchitecture and can be implemented inexpensively, and are thus suitable for use in low-cost microcontrollers. Our current research is to validate this concept through extensive testing on a simulated processor, and to measure the cost-effectiveness of the hardware architecture extensions over a wide range of design choices.*

1. Overview

We have conceptualized a hardware-software co-design strategy for creating I/O interfacing hardware and real-time operating system (RTOS) device drivers for microcontrollers, enabling hardware independent access to I/O devices at near-zero overhead. We achieve this low overhead through the addition of a hardware mechanism to the microcontroller architecture that we call *nanoprocessors*. The architecture extensions are orthogonal to the underlying microarchitecture and can be implemented inexpensively, and are thus suitable for use in low-cost microcontrollers. Our current research is to validate this concept through extensive testing on a simulated processor, and to measure the cost-effectiveness of the hardware architecture extensions over a wide range of design choices.

In Section 2, we discuss the general objective of our research into developing an RTOS that can support component-based software. In Section 3, we provide examples of the high overhead incurred by some input/output (I/O) devices. In Section 4, we present one of the microarchitecture enhancements that we are designing especially for use by an RTOS. How the hardware is used to reduce the overhead for I/O devices is described in Section 5. Finally, in Section 6, we summarize the current status of this work in progress.

The work-in-progress described in this paper is part of a new project that was recently funded by NSF's Experimental Software Systems Program, *http://www.nsf.gov/cgi-bin/showaward?award=9806645*.

2. Real-Time Operating Systems

The operating system creates a virtual machine, which is a layer of software that isolates the application from the hardware [5] . Our objective is to achieve the same level of abstractions in an RTOS, without sacrificing real-time performance or predictability. The current generation of RTOS do not sufficiently isolate the programmer from the hardware; for example, most real-time programs still directly access I/O registers. Although applications should not deal directly with hardware, this does occur in many embedded applications. The RTOS is bypassed in order to achieve maximum performance, as the RTOS overhead is not acceptable. For example, a device driver that reads an analog-to-digital converter (ADC) may need to first start the conversion, busy-wait for 20 μsec, then retrieve the result. If the code is in a device driver, then the 20 μsec is wasted. A skilled embedded designer will tweak the system, filling in those 20 μsec with other useful instructions, even though they are unrelated to the ADC read. Although this achieves better performance, it causes software maintenance headaches, as there is no modularity, and the code cannot easily be modified, reconfigured, or ported to another processor. Furthermore, the mapping of the hardware device to the device driver interface may be unnatural, and overhead of accessing multiple layers of the driver may be unacceptable.

To support component-based software at the application layer [6], a combination of efficient I/O hardware and real-time software is needed so that designers are encouraged to use standard interfaces, rather than bypassing them to obtain better performance. Our goal is to eliminate major operating system overhead with the aid of special hardware features, and abstract the hardware so that component-based software can execute with the same or better performance as hand-tweaked code. In this paper, we focus specifically on the high overhead of device drivers for real-time input/output.

3. High Overhead Input/Output

I/O in many embedded systems uses a large percentage of processor time, sometimes even nearing a full 100% when there are multiple I/O devices. Nanoprocessor technology can be used to reduce the overhead of device drivers and software I/O operations to almost 0%.

For example, a serial I/O interrupt handler may take 100 μsec[3] of processing time on a microcontroller each time it is called. Most of the overhead is to handle the interrupt and save and restore registers. Serial I/O is usually implemented

through a UART or DUART chip. A DUART, for example, has four channels: two for transmitting and two for receiving. When an interrupt occurs, all four of the channels need to be checked, to see which one is generating the interrupt. There is then the transfer of one byte from memory to DUART register, or vice versa. Software polling of registers may be required if more than one of the four channels is enabled. If this is the last byte transmitted or received, then there is additional overhead of copying the data from the interrupt handler's reserved memory to user memory, so that the program can access it. The process waiting for the serial transmission to end must also be signalled.

Suppose the serial port is set to 19,200 baud. That means there are 19,200 interrupts per second. At 100 μsec each, this represents 20% CPU overhead for a single channel on the DUART. If all four channels are operating independently, up to 80% of the CPU might be used to handle serial I/O.

A common approach to solving this problem is to use an intelligent serial I/O card. Such a card has an 8-bit microprocessor on it, such that 100% of the CPU is dedicated to the above actions. Data transmitted and received over the port is transferred to the main processor through shared memory. The overhead of the main processor is then reduced to a block copy of data and a signal between the 8-bit and main processors.

The main constraints on this method include the more expensive cost of an intelligent I/O card as opposed to just a DUART, the extra board space needed to accommodate the additional hardware, the greater power consumption when a processor needs to access off-chip data, and the limitation that this hardware is to be used strictly for serial I/O. If for any length of time the application does not call for serial I/O, the bandwidth of the 8-bit processor cannot be reclaimed for other duties.

Parallel I/O ports can incur significant overhead, but in a much different form than serial I/O. For example, in one robotics application, two parallel I/O 8-bit ports are used to acquire data from a force/torque (F/T) sensor. The data contains a vector of eight 16-bit values, plus a 16-bit parity check. One 8-bit port is used for data, the second port is used for handshaking. Every 2.5 msec, an interrupt is generated from the F/T sensor to the PIO board. When the interrupt occurs, the handler sends a ready signal to the F/T sensor. It then waits 10 μsec, reads the value from the input port, sends a 5 μsec pulse to the output port, waits another 10 μsec, and repeats. Speeding up a CPU will not reduce the percent utilization of the software handshaking over parallel I/O, because the timed pulses must remain the same regardless of CPU performance. Typical overhead for this handler was approximately 300 μsec. At a rate of 400 times per second, this uses 12% of the CPU time.

ADCs and digital-to-analog converters (DAC) incur overhead due to the multiplexing of several channels onto a single chip. ADC and DAC are generally used to interface to analog sensors and actuators. In robotics, polling rates typically vary between 30 and 1000 Hz. Although an ADC board may have 8 channels, a low-cost implementation would multiplex the 8 channels. The processor selects a channel, starts the conversion, waits, then reads the value, before moving onto the next channel. If all channels are to be read, this sequence is repeated once for each channel. The waiting period for typical ADC chips is between 5 μsec and 100 μsec. I/O boards with the lower conversion times are usually very expensive. When a cheaper, slower ADC is used then busy waiting for the conversion to complete wastes CPU cycles. Since the context switch overhead on most embedded processors is at least 50 μsec, it is usually not worthwhile to preempt the process. An ADC with 50 μsec conversion time, 8 ports, 10 μsec overhead for selecting a port and reading values, and reading all eight channels once per millisecond, yields a CPU utilization of 48% for this device alone. Reducing utilization requires more expensive ADC hardware, either by using a board with a faster conversion time, obtaining a board with a separate chip per channel, allowing conversions to occur in parallel, or as done with the serial I/O, using an intelligent ADC card that has a dedicated processor to perform these functions.

The overhead for accessing each individual device is significant. If a microcontroller has several devices, the overhead can easily consume most or all of the processor bandwidth, thus forcing the need to use more expensive or greater power consuming processors. It is thus desirable to reduce this overhead. We now present *nanoprocessors*, a microarchitecture enhancement that can perform that function.

4. Microarchitecture Enhancements for Real-time Hardware

Nanoprocessors are small software-configurable on-chip state machines used to off-load processing from the operating system, so that it occurs in parallel with normal computation. For example, these state machines can be configured to handle interrupts without intervention from the operating system. They can also be configured for high-frequency polling of I/O devices, used in place of timers, dedicated to running a real-time scheduler, used to sort delayed events by wakeup time and priority, or to autonomously and unobtrusively measure the execution time of real-time code.

A nanoprocessor is directly analogous to executive assistants in the corporate world, as illustrated in Figure 1. Corporate executives do not personally answer all of their mail; they do not personally answer all of their telephone calls; they do not personally meet with every client that asks to see them. They employ assistants who are given a well-defined level of autonomy to handle the more mundane chores. If the assistants do their jobs properly, the executive (whose time is ostensibly more valuable) is only interrupted for tasks that truly require his or her attention. Most mail can be handled with a form letter that an assistant can sign and mail without the intervention of the executive; most telephone calls can be handled similarly; and most inter-personal communications can be summarized by the assistant for the executive to digest later. Assistants are semi-autonomous in that they are

[3] Times shown are approximations based on prior actual measurements on a 25 MHz Motorola MC68030 processor in the Chimera Real-Time Operating System [7] .

empowered to act on the authority of the executive; each assistant acts as a direct representative of the executive, and each has a well-defined set of rules to determine whether a given request is important enough to bring to the immediate attention of the executive.

Nanoprocessors are equivalent to the executive assistants; while the main processor is the executive. A nanoprocessor executes RTOS-defined functions so that the main processor is not interrupted unnecessarily. Mundane and repetitive RTOS tasks, such as scheduling, interrupt handling, and input/output, are offloaded to the nanoprocessors. Nanoprocessors are on-chip, and implement finite state machines that are software-configurable by the RTOS. They are configured by changing a *nanoprogram* that defines the machine. Programming a nanoprocessor is dynamic, and is simply a matter of transferring data into the nanoprocessor's memory space; reconfiguration time is therefore proportional to the size of the nanoprogram (which we expect to be on the order of 128 to 1024 bytes). It does *not* require restructuring the datapath as in an FPGA.

Each nanoprocessor reports to the main processor, and is abstracted by the RTOS such that it is transparent to the user. Each has a well-defined set of rules to determine whether a given task is important enough to bring to the immediate or delayed attention of the RTOS. The nanoprocessor is not an independent processor—it is the rudimentary processor core and a subcomponent of a microprocessor. Every microprocessor could have several of these nanoprocessors on-chip.

5. Eliminating I/O Overhead using Nanoprocessors

Most I/O overhead can be eliminated by implementing the interrupt handler, handshaking overhead, and busy waiting time on a nanoprocessor, as well as reducing subroutine call overhead typically associated with calling a device driver through a standard interface. The nanoprocessor instead transfers data to and from the main processor through shared memory as state data. Furthermore, the nanoprocessor can filter the data, to perform any scaling and add offsets to convert 8-bit or 16-bit raw data without standard units to more meaningful information such as position in meters or temperature in Celsius. A comparison of a traditional device driver and off-loading that code to a nanoprocessor driver is shown in Figure 2.

As we discuss in [6] , performing communication through states instead of messages is desirable in a component-based software environment, to preserve the automaton characteristic of each control module. For instance, in a message-based system, a command such as "turn on the brake" may be sent between processes. On the other hand, in a state-based system one object may set the state "the brake should be on." The driver that is responsible for the brake recognizes this new state, and in turn actuates the brake to correspond to the desired state.

For the main processor, the overhead of the I/O driver becomes a memory copy operation with shared memory and the time needed to lock and unlock the shared data. As we have previously shown, locking a state table only takes a few microseconds when using either the port-based-object state variable table mechanism described in [6] or the triple-buffer mechanism described in [8] .

Nanoprocessor technology reduces the need for expensive or power-hungry hardware add-ons, reduces I/O usage of the main processor to almost 0%, and is flexible to accommodate most I/O types found in embedded control applications. If there is no I/O, the nanoprocessor can be configured for other functions, such as scheduling, profiling, and error detection. These other functions will be investigated in the future.

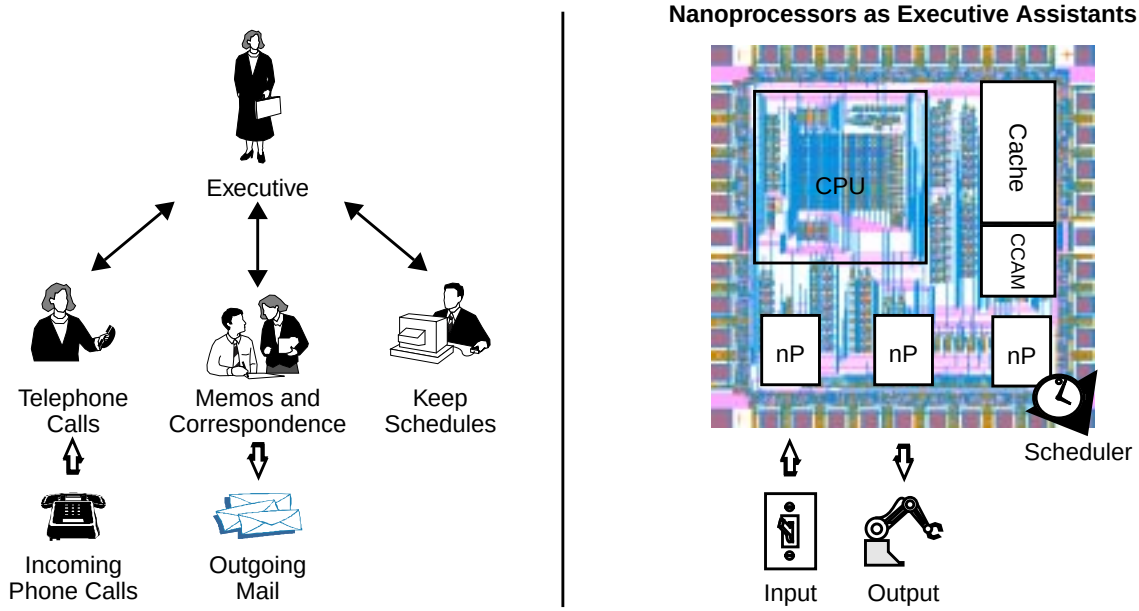

Figure 1: Diagrammatic representation of the analogy between nanoprocessors and executive assistants. Note the CPU shown on right is for display only; it is not the actual architecture we are designing.

6. Current Project Status

The project began in September 1998. The Motorola M•CORE was selected as the main processor that we would simulate, as it is representative of a modern microcontroller. We have built a simulator for the processor, and are in the process of experimentally validating the simulator. A series of test programs ranging from a few instructions to a full microkernel core have been implemented on an M•CORE evaluation module (donated to us for this project by Motorola). Execution time was measured for each program, and for different segments of each program. The same programs are executed on the simulator. This experimental testbed is being used to evaluate our hardware assists for real-time processing, including the nanoprocessors described in this paper, as well as other assists such as software-managed memory systems [2,3,4] and DRAM architectures [1] .

On the software side, we are first focussing on I/O devices that are integrated into control systems that use the port-based object model of reconfigurable software [6]. Several device drivers for the basic types of I/O ports, including serial, parallel, ADC, and DAC, have already been implemented without nanoprocessors. We are currently benchmarking these drivers on the M•CORE processor, so that we can perform quantifiable comparisons between systems equipped with nanoprocessors and those without.

7. References Cited

[1] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures," in *Proc. 26th ACM/IEEE Int'l Symp. on Computer Architecture (ISCA-26)*, May 1999.

[2] B. Jacob, "Software-managed caches: Architectural support for real-time embedded systems," *CASES98: Workshop on Compiler and Architecture Support for Embedded Systems*, December 1998.

[3] B. Jacob and T. Mudge, "A look at several memory management units, TLB-refill mechanisms, and page table organizations," in *Proc. Eighth ACM/IEEE Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-8)*, pp. 295-306, Oct. 1998.

[4] B. Jacob and T. Mudge, "Software-managed address translation," in *Proc. Third IEEE Int'l Symp. on High Performance Computer Architecture (HPCA-3)*, pp. 156-167, February 1997.

[5] A. Silberschatz and J. L. Peterson, *Operating System Concepts*, Alternate Ed.,(Addison-Wesley) 1989.

[6] D.B. Stewart, R.A. Volpe, and P.K. Khosla, "Design of Dynamically Reconfigurable Real-Time Software using Port-based Objects," *IEEE Trans. on Software Engineering*, v.23, n.12, December 1997.

[7] D.B. Stewart, D.E. Schmitz, P.K. Khosla, "The Chimera II real-time operating system for advanced sensor-based control applications," *IEEE Trans. on Systems, Man, and Cybernetics*, v.22, n.6, pp. 1282-1295, Nov./Dec. 1992.

[8] D. B. Stewart, *Real-Time Software Design and Analysis of Reconfigurable Multi-Sensor Based Systems*, Ph.D. Thesis, Carnegie Mellon University, April 1994. (Available at *http://www.ece.umd.edu/serts/bib/thesis/dstewart.html*)

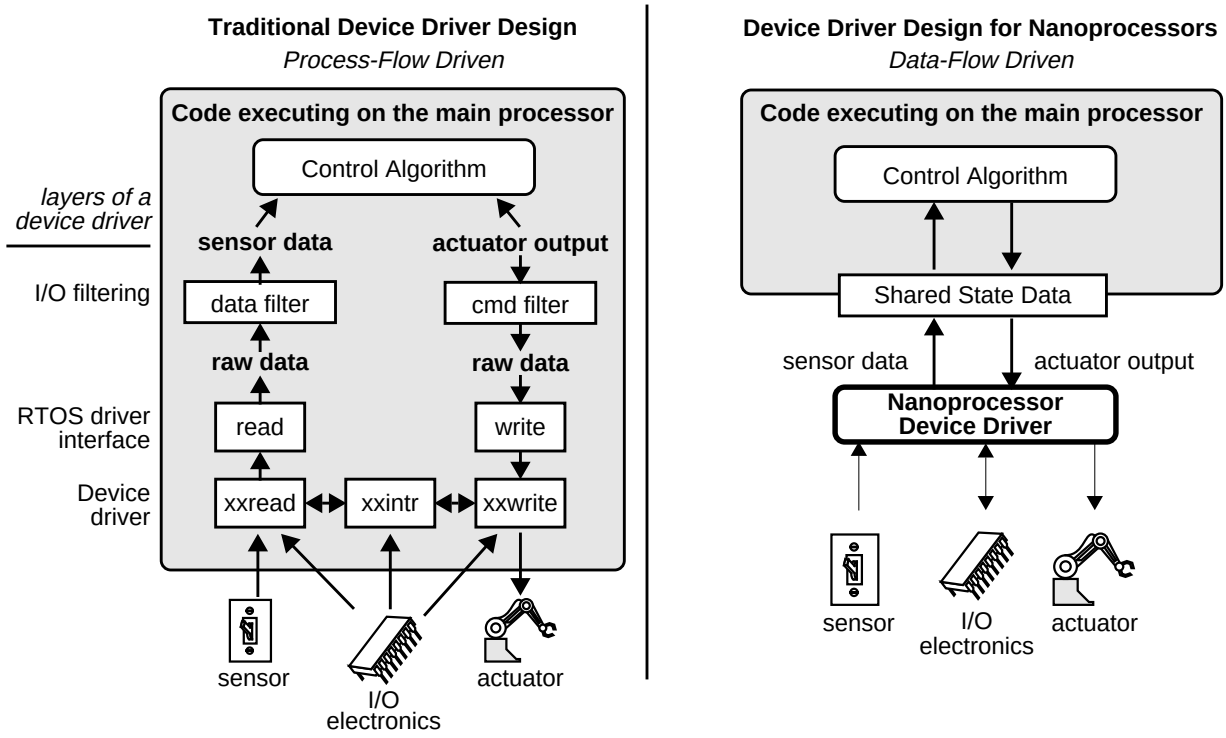

Figure 2: Comparison of traditional device drivers and a nanoprocessor device driver.