DESIGN CONSTRAINTS ON EMBEDDED REAL TIME CONTROL SYSTEMS

Philip J. Koopman Jr.
Senior Scientist
Harris Semiconductor
2525A Wexford Run Rd.
Wexford, PA 15090

ABSTRACT

Today's embedded real time control applications place extraordinary demands on microprocessors. They have stringent constraints on size, weight, power, cooling, computing performance, reliability, and cost. Unfortunately for the designers of such systems, modern processor architectures usually incorporate design features that are valuable for general-purpose computing, but violate constraints for hard real time problems. These undesirable architectural features include: cache memory, instructions with low semantic content, write buffers, branch target buffers, prefetch queues, pipelines, pure load/store architectures, scoreboards, superscaler instruction issue, dependence on sophisticated compiler technology, large register files, virtual memory, autonomous I/O processors, and the use of special DRAM access modes.

INTRODUCTION

The traditional approach for low performance controller applications has been to use an integrated microcontroller chip. The "system-on-a-chip" solution has many obvious benefits, and makes possible embedding an entire computing system in an application. Unfortunately, these microcontrollers have traditionally been based on modest 8- or 16-bit general purpose micro-processor designs, leading to a serious lack of computing power for real time applications.

A recent trend has been to place more functionality and power in embedded systems. Usually this leads to using a 16- or even 32-bit general purpose microprocessor. Using a more powerful general purpose processor provides significant benefits in terms of speed and capability to solve more complex problems. But this capability is not without cost; a conventional microprocessor can require a substantial amount of off-chip support hardware, memory, and often a complex operating system. This leads to increases in required board space, power, weight, and cost. Furthermore, the design tradeoffs that are made by most processors favor desktop computing environments, and are exactly the *wrong* tradeoffs for embedded real time control.

A CHARACTERIZATION OF EMBEDDED REAL TIME CONTROL

Embedded real time control processors are computers that are integral portions of equipment such as cars, airplanes, computer peripherals, audio electronics, and military equipment. A computing task is considered to be a real time application if the processor must respond to external events within a relatively short time period. A processor is considered to be embedded if it is built into a piece of equipment that is not itself considered a computer.

Real Time Control

There are several different types of real time control, characterized by the predictability of service requests and the strictness of the deadline for responding to those events. One can think of real time control tasks as being divided into three categories: simple events, events with hard deadlines, and aperiodic events with hard deadlines.

Real time control of simple events is what most people think of when discussing real time control. This category of real time control requires reasonably quick response to an external event. Examples include responding to a mouse click quickly enough to avoid irritating a workstation user, or changing a traffic light in response to a car activating a sensor at an intersection.

The main characteristics of this kind of real time processing are there is some notion of how quick a response is desired, but the time scale involved is usually no stricter than tenths of seconds, and there is considerable leeway in meeting requirements. A single slow mouse cursor update is not very noticeable, and most drivers will wait for a few seconds before deciding that a traffic light controller is broken. Since deadlines tend to be soft, the consequences for missing the desired response time are very slight, and tend to be proportional to the amount of time by which the deadline was missed.

A somewhat more challenging real time application category is *real time control with hard deadlines*. A hard deadline is typically a firm time imposed by external events which can't be missed by any amount without a significant penalty. A common hard deadline task is periodic sampling hardware with a single-element buffer (such as a serial communications port). If the interrupt caused by reception of a data element is not responded to by the time the next data element is ready, data will be lost. The cost may involve retransmission of the data if the sender is a computer, or even permanent loss of the data if the sender is a sensor without its own local memory.

The time frame for real time control with hard deadlines ranges from microseconds to seconds. There is often enough time be sure of accomplishing a task simply by specifying a speed safety factor into the design and then computing average performance. Also, it is relatively straightforward to predict the worst case situation if the full range of external events and conditions can be predicted when the software is being designed. If outside events occur at regular intervals, a static schedule can be devised which will guarantee correct operation. The consequences of missing a deadline can be severe, but a missed service response can usually be recovered from with some cost.

Real time control of aperiodic events with hard deadlines is the most challenging of the categories we are discussing. The difference between this category and the last is that the external events demanding service can not be scheduled in advance, and therefore can not be analyzed when writing the software. To make matters worse, the response time demanded by these aperiodic events is often much quicker than for other real time control tasks.

An example of an aperiodic event with hard deadlines is receiving pulses from an interferometer used to measure the position of an object. If there are several channels of measurement data, a processor must be very careful to service all inputs quickly enough to avoid missing a pulse.

This category of real time control is very difficult. In situations where microprocessors are able to provide a solution, extreme measures must often be taken by the system designers. For example, it is common for programmers to program exclusively in assembly language, and count clock cycles of instructions in their program so as to budget time down to the level of a single clock cycle. Predictability and determinism of execution speed are vital to ensure meeting tight deadlines.

Characterization of Embedded Systems

Embedded real time control applications not only require real time control processing, but are also characterized by having significant external restrictions to the system design. Often the fact that a computer is present in an embedded system is completely invisible to the user, such as in an automobile anti-skid braking system. At other times, the fact a computer is present may be obvious, such as in an aircraft autopilot.

An embedded processor may be added to a product to meet basic requirements for functionality that require a computer, to reduce system cost by replacing special purpose hardware with an inexpensive general purpose processor, or sometimes, simply to add marketing features. In

all cases, the processor and its requirements must be subservient to the needs of the system in which it is embedded.

SYSTEM LIMITATIONS

Most embedded systems place severe constraints on the processor in the terms of requirements for size, weight, power, cooling, performance, reliability, and cost. This is because the processor is just a component of a larger system, which has its own operating requirements and manufacturing constraints.

Size and Weight

Size and weight restrictions can greatly limit the complexity of the hardware which can be brought to bear on solving a particular embedded control problem. Platforms such as aircraft, spacecraft, and portable equipment may have strict weight limitations. Most other applications also have size limitations of some sort. A typical embedded controller may have a size budget of a few cubic inches, and a weight budget of a few pounds (including the power supply requirements).

The problem with size and weight restrictions is that high performance processing systems tend to be larger and heavier than slower systems. At the CPU level, a processor that has a large number of pins takes up a large printed circuit board area. At the system level, a design that needs cache memory controller chips and large amounts of memory takes even more printed circuit board area.

The key to the size and weight issue is keeping component count small. This is best accomplished by using the highest integration level possible. But, even with custom VLSI chips and surface-mount technology, the fact remains that in embedded system, high complexity means problems with size and weight budgets. The smallest systems are those which have on-chip memory for some or all of the system requirements, do not require external support chips (especially cache memory support), and which allow small, efficient encodings of programs.

Power and Cooling

The processor complexity can affect the power consumption of the system. High power consumption increases the size and weight requirements of the power supply. Since essentially all power consumed by a computer is eventually given off as heat, increased power consumption results in increased cooling requirements.

The amount of power used by the processor is related to the number of transistors and pins on the processor chip. Processors that rely on exotic process technology for speed usually consume too much power to be useful. Processors that need huge numbers of power consuming high speed memory devices likewise can exceed a power budget.

Consequently, CMOS components are usually used whenever possible. Newer CMOS technology runs at high speed, yet dissipates little power in comparison to other technologies. Static CMOS is often selected because it can operate with a slow or stopped clock to save power when the system is idle.

Computing Performance

Computing performance in a real time embedded control environment is not simply an instructions-per-second rating. While raw computational performance is important, other factors which are vital to system performance include interrupt response characteristics, context switching overhead, and I/O performance. Since real time tasks are characterized by frequent, often unpredictable events, the performance of a processor at handling interrupts is crucial to its suitability for real time processing. Since a control application usually involves a reasonable amount of communication with the outside world (such as reading sensors and activating control circuits), good I/O performance is also important.

Two key considerations for more difficult real time control applications are *predictability* and *determinacy*. Predictable systems have behavior at run-time that is easily understandable from examination of the original source code program;

in other words, there are no surprises to a user of average sophistication. Deterministic systems have instructions whose behavior does not vary; in other words, the execution speed of an instruction does not depend on execution history of the program. For real time control with hard deadlines, a designer must be able to predict with absolute certainty the running time of the piece of code responding to an external event. If a system has variable performance elements, such as cache memory, the designer must be extremely pessimistic about the performance of these features, and plan on the worst case.

In some systems, it is important to exactly determine the time for a sequence of instructions to execute, with no possibility for variation allowed. Processors with extremely consistent execution speeds can use software to reduce system complexity by replacing hardware such as UART chips, disk controller chips, or video memory sequencers with programmed transfers using carefully timed code.

Since program memory space may be extremely limited, programs may be highly compacted by using subroutine calls to reuse common code sequences. In fact, many embedded applications use threaded languages such as Forth because they produce exceedingly compact code. This suggests that an embedded processor should have efficient support for subroutine calls as a means of conserving program memory.

Reliability

Embedded processing applications are notorious for extreme operating conditions, especially in automotive and military equipment. The processing system must deal with vibration, shock, extreme heat and cold, and perhaps radiation. In remotely installed applications, such as spacecraft and undersea applications, the system must be able to survive without field service technicians to make repairs.

The general techniques used for avoiding problems caused by operating environments is to keep the component count and number of pins as low as possible. It is also helpful to keep the system as cool as possible to inhibit aging of the system's components.

Cost

The cost of the processor itself may be very important to low- and medium-performance systems, especially consumer electronics products. Since the cost of a chip is related to the number of transistors and to the number of pins on the chip, low complexity processors have an inherent cost advantage.

In high-performance systems, the cost of the processor may be overwhelmed by the cost of the multi-layered printed circuit boards, support chips, and high-speed memory chips. In these cases, overall system complexity must be reduced to keep system costs down.

ARCHITECTURAL FEATURES

Many of the computer architect's techniques to improve system performance, such as cache, branch target buffers, and prefetch queues, work on statistical principles. They assume things such as temporal and spatial locality, which tend to be true on large workstation programs when run for long periods of time. They give relatively uniform performance when used in a CAD or office automation environment.

However, the constraints of hard real time control applications have requirements which make standard, statistically good speedup techniques inappropriate. Real time control of aperiodic events requires absolute predictability, often down to a single clock cycle. Deadlines must be guaranteed to be met 100% of the time. Having a situation where an unfortunate interaction between two different routines contending for cache memory causes an increase in cache misses is simply unacceptable. In a tightly scheduled environment, even the time taken by a single cache miss can cause a missed deadline.

Inappropriate Architectural Features

There are many reasons that an architectural feature is included in a processor. Today, usually the

feature is added in order to support "general purpose" computing, which usually means engineering workstation application programs. The problem is that this definition of "general purpose" is often at odds with requirements for the systems we have just described. The list of architectural features that are inappropriate for hard real time embedded control systems seems similar to a catalogue of modern processor design techniques. They are listed here along with a brief explanation of the problems they can cause. The focus here is on determinism and predictability, although the reader should not forget that most of these features also contribute substantially to system complexity as well.

Cache memory. Data and instruction cache memories are the biggest sources of unpredictability and indeterminacy. Modern compiler technology is just beginning to address the issue of scheduling instruction cache misses. Furthermore, it is impossible to characterize data cache misses for programs of interesting complexity. Because the time required to process a cache miss is approximately an order of magnitude slower than the time required for a cache hit, significant execution speed degradation takes place with even a small percentage of cache misses. Figure 1 shows relative execution time variations that may take place with varying numbers of cache misses. In this five-instruction example, assuming that a cache miss costs five clock cycles, execution time may vary between 5 and 25 clock cycles for the same code. In hard real-time systems, often the worst case of all cache misses must be assumed for time-critical sections, resulting in designing hardware with only fast static memory chips that render the cache management hardware superfluous. On-chip caches add to system debugging concerns because they are often not visible outside the chip.

Low semantic content of instructions. Low semantic content of instructions greatly increases the number of instructions that must be executed to accomplish a given task. This, in turn, increases demands for high-speed memory. A requirement for large amounts of high speed memory in turn either requires the use of cache memory or a significant expense in making *all* program memory out of cache-grade memory chips for high guaranteed system performance.

Variable length execution times of instructions. Many instructions (usually on CISC systems) can take a variable number of clock cycles to execute, often depending on the data input to the instruction. An example of such an instruction is a multiply routine that has an "early-out" feature.

Write buffers. Write buffers allow the CPU to perform a write to memory without pausing for an actual memory cycle to take place. The problem is that this write must then wait for spare bus cycles.

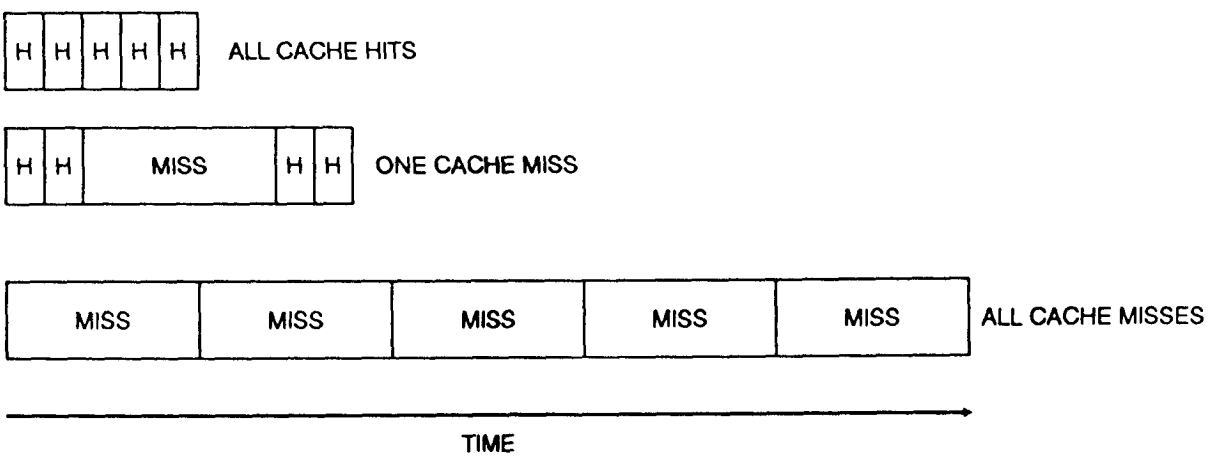


Figure 1.
Example of cache misses causing variable execution time.

If no spare bus cycles are forthcoming, the processor must be stalled when the write buffer overflows. Additional stalls can be caused if a memory read could possibly correspond to a memory location that has yet to be updated by the write buffer. Interaction between the write buffer and cache misses for instruction and data fetches can cause indeterminacy in program execution.

Branch target buffers. This is a special case of an instruction cache, in which past program execution history is used to cache instructions at branch targets for quick access. This type of instruction cache operates in a manner dependent on input data, and so is beyond the ability of compilers to manage effectively. Branch target buffers are sometimes used in conjunction with branch prediction strategies, in which the compiler encodes a guess as to whether a particular branch is likely to be taken into the instruction itself, causing either the branch target or the next consecutive instruction to be fetched before the outcome of the branch is resolved. The actual time to perform a branch then depends on whether the compiler guessed the branch outcome correctly, and whether the branch target buffer value corresponds to the guess made by the compiler.

Prefetch queues. Prefetch queues greatly affect the predictability of execution because the execution time for an instruction depends on whether or not the preceding instructions were slow enough to allow the prefetch queue to accumulate new instructions. Thus, determining whether a particular instruction will execute quickly from the prefetch queue or slowly from program memory requires cycle counting of several preceding instructions to see if spare memory cycles would have been available for the prefetch queue to fill. This cycle counting is subtly affected by data-dependent execution paths through the program as well as the number of wait states or cache misses in program memory. For example, if there is a latency for filling an empty prefetch queue, adding a one-cycle wait state that causes the prefetch queue to be emptied may add *more than one* clock cycle to program execution time.

Pipelines. A deep instruction pipeline increases interrupt response latency. Even if the first instruction of an interrupt service routine can be inserted into the pipeline within one clock of the interrupt being asserted, it still takes several clock cycles for the instruction to pass through the pipeline and actually do its job. A pipeline also requires some sort of handling of data dependencies and delays for memory access, which results either in compiler-generated nops, instruction rearrangement, or hardware-generated pipeline stalls. All of these dependency-resolution techniques decrease predictability, determinacy, or both.

Pure load/store architectures. Load/store RISC architectures can by their very nature make no provision for atomic read/modify/write instructions. This generally required the addition of external hardware for implementing semaphores, locks, and other inter-process communication devices, increasing system complexity.

Scoreboards. Scoreboards attempt to speed up program execution by allowing several instructions to be in execution concurrently. Scoreboarding usually implies out-of-order execution, which may make it impossible to correctly determine the correct system state in the event of an exception or interrupt (the term usually used here is that the system has imprecise interrupts). Furthermore, the execution time of an instruction depends on the resources being used by the previous several instructions, and can vary considerably.

Superscaler instruction issue. Newer-generation microprocessors are beginning to implement "superscaler" instruction execution, in which multiple instructions may be issued in a single clock cycle. Unfortunately, the number of instructions that may be issued depends on the types of the instructions, the available hardware resources, and the execution history of the program. All these factors make it very difficult to determine any single instruction's execution time.

Dependence on sophisticated compiler technology. Most RISC designs depend implicitly on complex and ambitious optimizing compilers for fast operating speeds. This tradeoff between hardware and software complexity is a part of the RISC design philosophy. The problem is that the optimizing compilers make it difficult to establish

a correspondence between source code and the code that actually is executed because of the large number of program transformations performed. This complicates program debugging, as well as decouples the programmer's source code changes from effects on final program performance, decreasing predictability. A surprising side-effect of the complexity of these compilers is that they add a level of indeterminacy in the mapping of source to object code. The use of heuristics for block-level and global optimization techniques can actually cause the same exact sequence of source code statements to generate significantly different object code in two places of the same program module (Razouk *et al.* 1986).

Large register files. The use of a large number of registers allows programs to execute quickly, but greatly increases the amount of information (the *machine state*) that must be saved on context switches and interrupts. This adversely affects responsiveness to external events.

Virtual memory. The use of virtual memory implies the use of a cache memory to perform address translation (a Translation Lookaside Buffer), which has the same problems as other caches. If a disk-paging system is used, the problem of speed degradation can become much worse than for simple cache misses.

Autonomous I/O processors. The use of autonomous I/O processors and DMA hardware that steal bus cycles can cause non-determinism as the processor is stalled for bus accesses. Consequently, it is sometimes desirable to perform I/O directly from the CPU.

Use of DRAM access technology. Some processor implementations exploit special access techniques to DRAM chips in order to achieve high average throughput with a low system cost. The use of static-column, paged, and video DRAMs creates real problems in predictability because access to these DRAMs becomes much slower whenever page boundaries must be crossed. The use of DRAM memory chips also requires performing memory refresh, which can interact with other accesses to cause performance degradations that are worse than expected.

The Challenge to Processor Designers

The above list of architectural characteristics contains elements of design found in virtually every RISC or CISC processor made today. These architectural characteristics are truly valuable for speeding up average system performance, especially in workstation environments. The problem is that hard embedded real time control applications do not have the same characteristics and requirements as workstation-based engineering design applications, and so these characteristics actually hurt system performance. Therefore, what is needed are new CPU designs that make tradeoffs in favor of the requirements of real time embedded control, even at the expense of performance of workstation-type applications.

REFERENCE

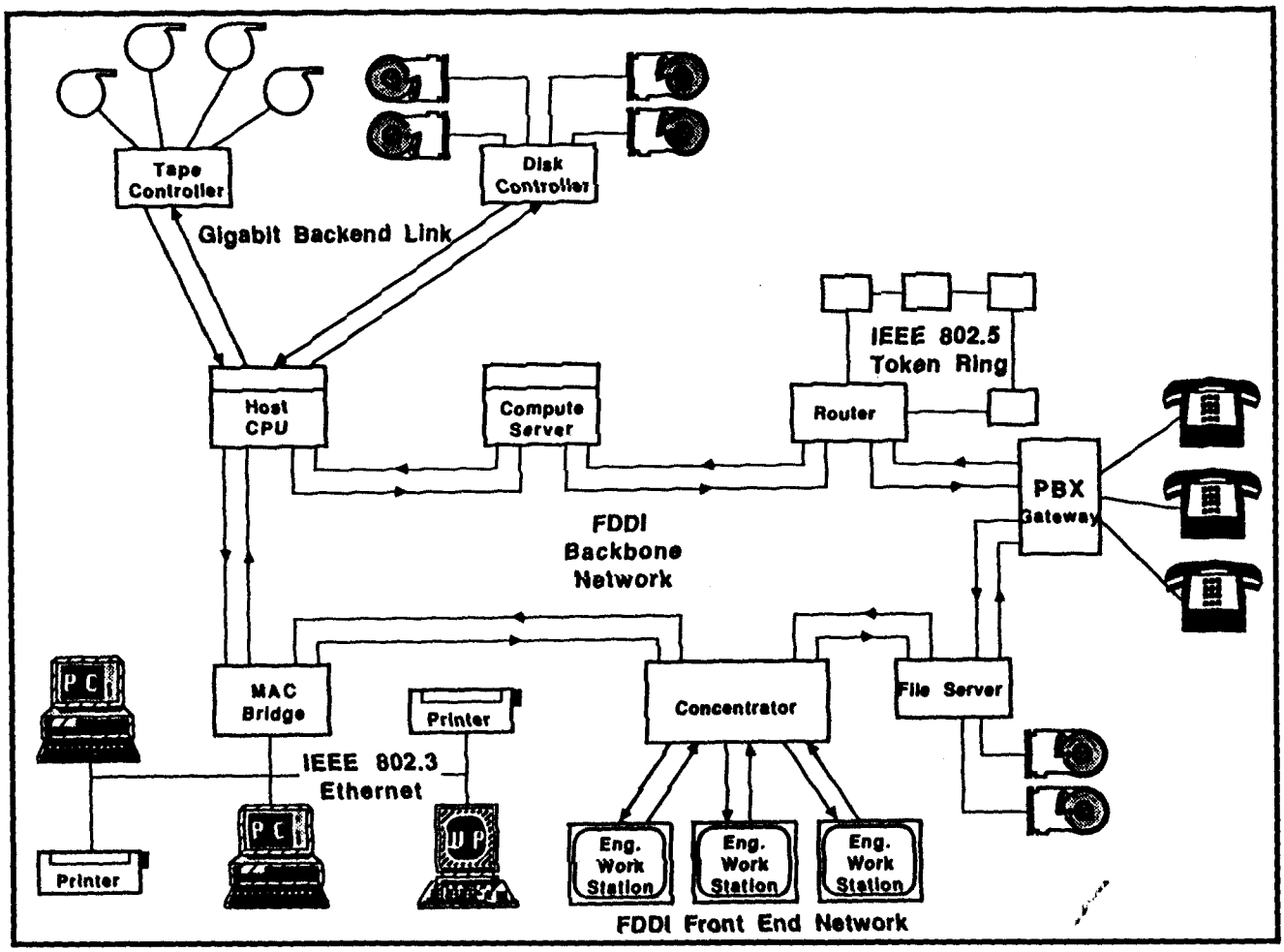
Razouk, R., Stewart, T. & Wilson, M. (1986) Measuring Operating System Performance on Modern Micro-Processors, In: *Performance 86*, ACM, NY, pp. 193-202.

Systems Design & Networks Conferenc

The Conference on Computer Systems, Peripherals and Network

Santa Clara Convention Center, Santa Clara, CA

Conference Proceedings
Microprocessor Track



EDITOR
Kenneth Majithia

MAY 8-10, 1990
SANTA CLARA, CA

Sponsored By:
EDCON &
SFBAC OF IEEE

In Cooperation with:
Electronic Design
(a Penton Publication) and ACM

MAPLE PRESS